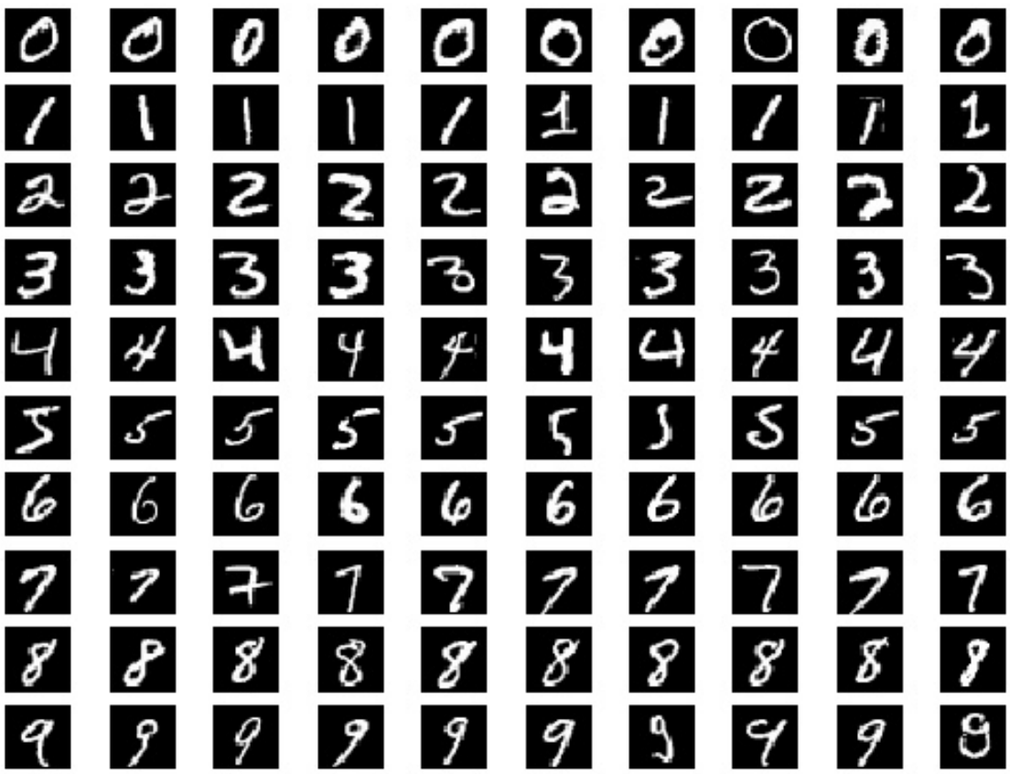
CSC411 Project 2

Zili Xie

Feb 18th 2018

Part 1

The dataset contains a large number of images representing digit from 0 to 9. Each image has a resolution of 28 by 28 and a black background with handwritten digit in white color. Most images are clear enough to recognize the digit on it, but some are not very recognizable and hard to tell which number is it. For example, the 5th image for digit 6 can be easily recognized as digit 4.



Images for each digit

Part 2

```
def part2(x, W, b):
    o = dot(W.T, x) +b
    return softmax(o)
```

The function above is used to compute neural network with no hidden layer. b is the bias. W stands for weight of neurons and x is the 28 by 28 input of handwritten image.

Part 3

(a)

The negative log-probability function can be represented as below by code. This can be also referenced from page 6 to 7 of one-hot encoding slides.

```
def cost_function(x,y_, W, b):
    return -sum(y_*log(part2(x, W, b)))
```

For M training examples, the cost functions is

$$C = -\sum_{m=1}^{M}\sum_{j} y_j log p_j$$

We also know that:

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

Computing the cost function with respect to the output:

$$\frac{\partial C}{\partial o_i} = \sum_j \frac{\partial C}{\partial p_j}\frac{\partial p_j}{\partial o_i}$$

$$= \frac{\partial C}{\partial p_i}\frac{\partial p_i}{\partial o_i} - \sum_{j\neq i}\frac{\partial C}{\partial p_j}\frac{\partial p_j}{\partial o_i}$$

$$= \sum_{j\neq i} y_j p_j - y_i(1-p_i)$$

$$= p_i \sum_{j\neq i} y_j - y_i$$

$$= p_i - y_i$$

Computing the o_i with respect to weight w_{ji}

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

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

Therefore, we can computing the cost function with respect to the weight

$$\frac{\partial C}{\partial w_{ij}} = \sum_j \frac{\partial C}{\partial o_i}\frac{\partial o_i}{\partial w_{ij}}$$

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

Part 3

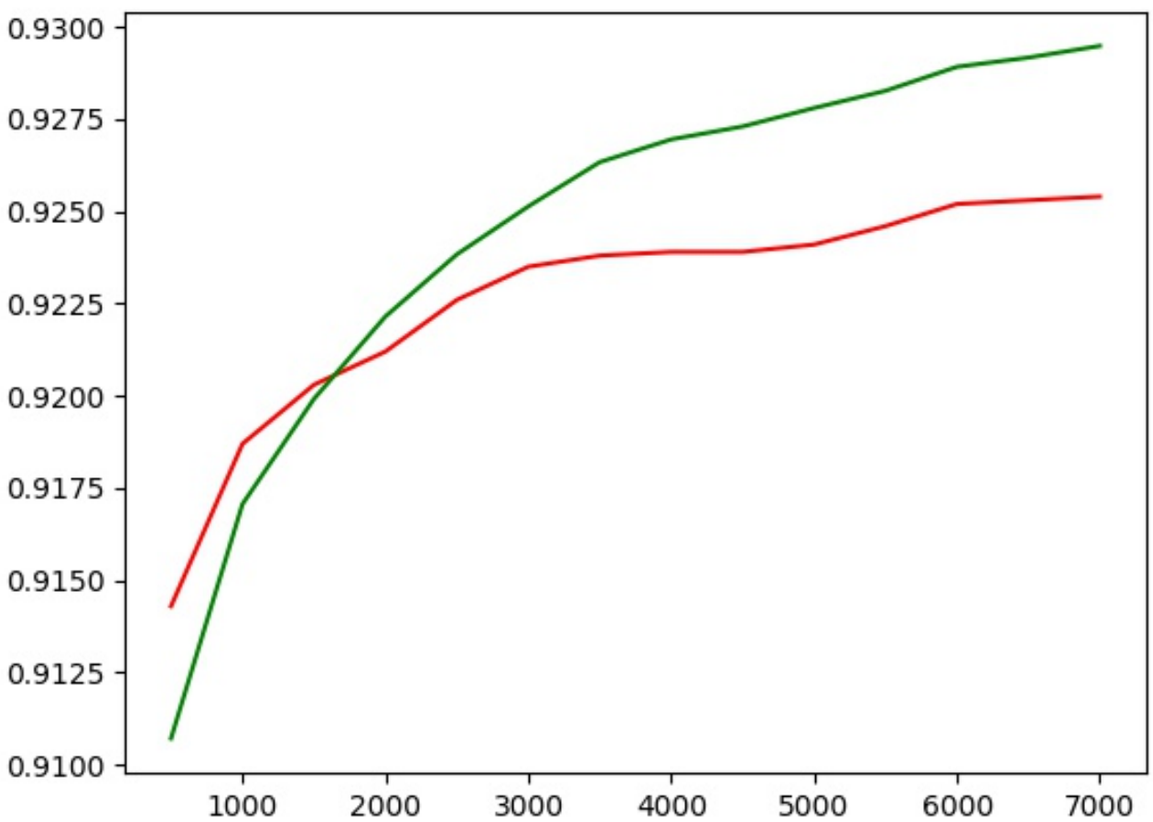
(b)

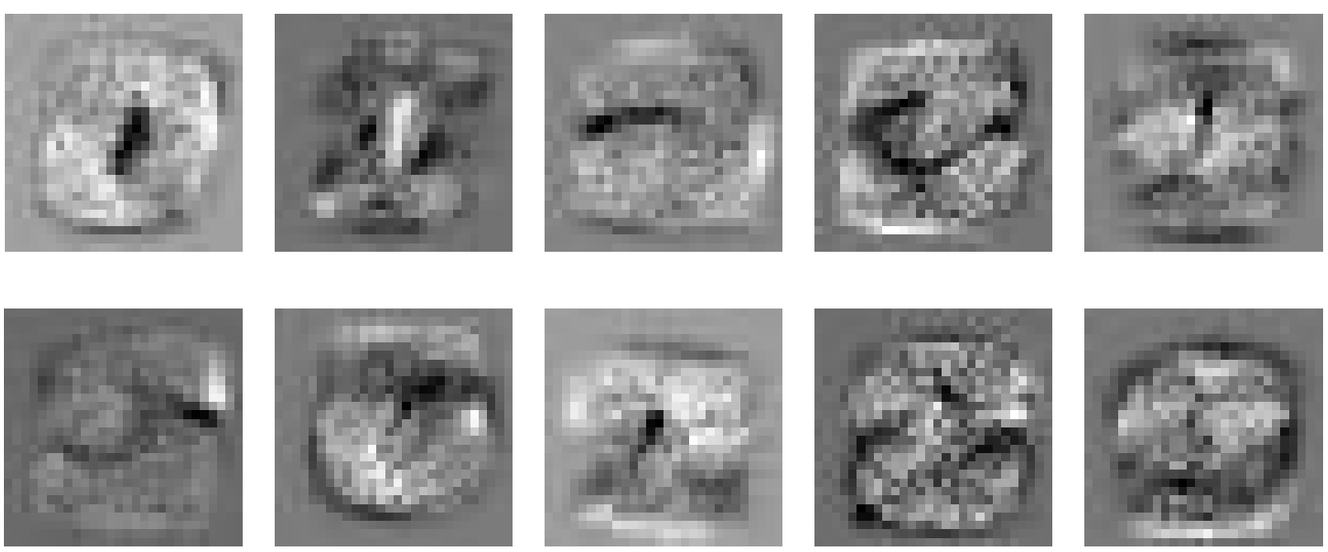
```
def finite_differences(x, y, w, b, i, j):
    h = 10e-5
    c = cost_function(x,y, w, b)
    w1 = w
    w1[i, j] += h
    c1 = cost_function(x,y, w1, b)
    return (c1 - c)/h

def part3(x, y, b, w):
    index1 = [377, 488, 599]
    index2 = [5, 6, 7]
    for k in range(3):
        i = index1[k]
        j = index2[k]
        fd = finite_differences(x, y, w, b, i, j)
        g = gradient_W(x,y, w)[i, j]
        diff = abs(g - fd)
        print("Test"+str(k)+":")
        print("finitedifferences="+str(fd))
        print("gradient="+str(g))
        print("difference="+str(diff))

x1 = M["train8"][0]
x2 = M["train8"][1]
x = vstack((x1, x2)).T
y = array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
           [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]).T
w = zeros((x.shape[0], 10))
b = zeros((10, 2))
part3(x, y, b, w)

####################################
Test 0:
finite differences = 49.4422055337
gradient = 49.9879565015
difference = 0.545750967803


Test 1:
finite differences = 37.92927574
gradient = 38.2535285738
difference = 0.324252833793


Test 2:
finite differences = 21.8181347273
gradient = 21.9305348307
difference = 0.11240010331
```

Part 4

(a)

The learning curve can be presented as the image below. The green line stands for the performance on training set, while the red line is the performance curve of images from test set. Y axis is the performance at each stage. X axis is the iteration time. As you can see, the performance of green line exceed the red line when the iteration times is over 2000 and reach a plateau as the iteration time increases.



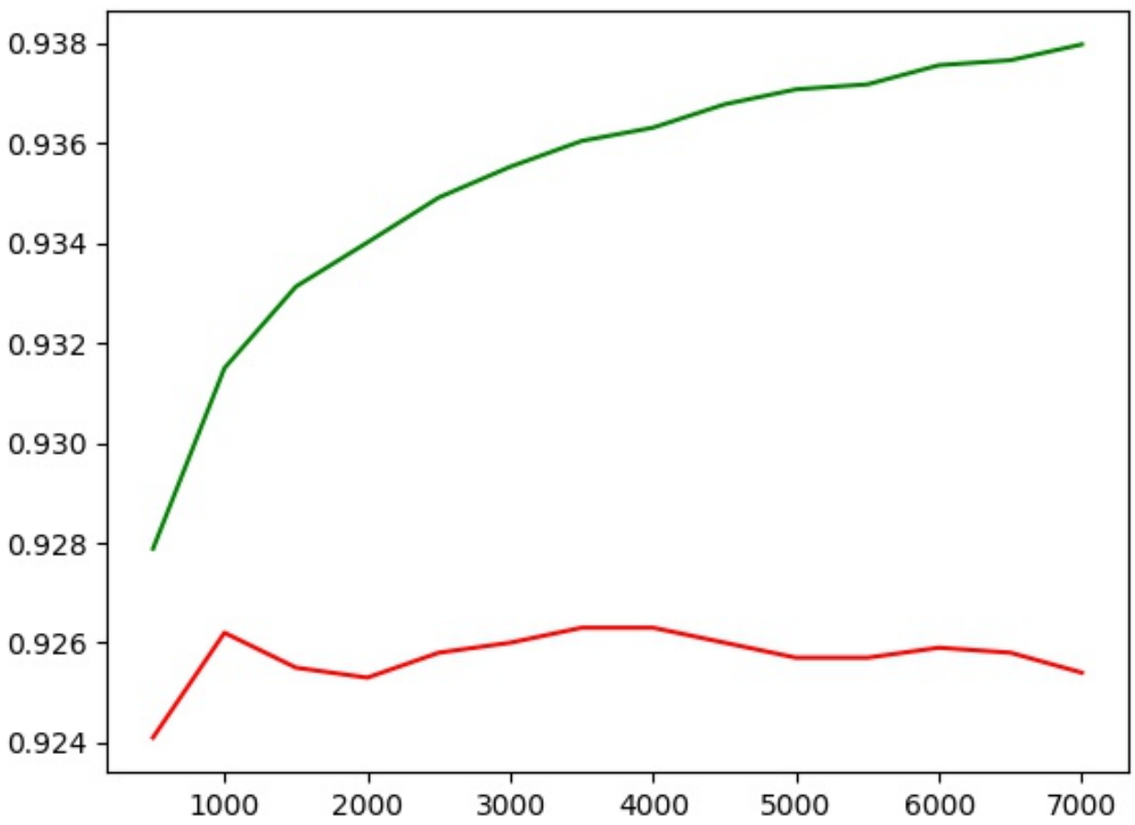


Above are the visualization of the weights of neural network for all different digits going to all output units. To optimize the weights of neural network, first I initialize the weights to be 784 by 10 zero matrix and the learning rate is set to be $1e-10$ in each 500 iterations, record the weights of current neural network and save as files. And after the gradient descent is complete, reproduce the weights and compute performance by reading from file. Below I provide some source code.

```
def grad_descent_p4(df, x, y, init_w, alpha):
    EPS = 1e-15   #EPS = 10**(-5)
    max_iter = 7000
    iter  = 1
    t = init_w
    #t = vstack((b, init_w))
    prev_t = t-10*EPS;
    while norm(t - prev_t) >  EPS and iter <= max_iter:
        if(iter % 500 == 0):
            filename ='w_'+str(iter)+'.txt'
            np.savetxt(filename, t)
            print(t);
        prev_t = t.copy()
        t -= alpha*df(x, y, t)
        iter += 1
    return t

one_hot_encode = [array([[1] + [0] * 9]),
    array([[0] * 1 + [1] + [0] * 8]),
    array([[0] * 2 + [1] + [0] * 7]),
    array([[0] * 3 + [1] + [0] * 6]),
    array([[0] * 4 + [1] + [0] * 5]),
    array([[0] * 5 + [1] + [0] * 4]),
    array([[0] * 6 + [1] + [0] * 3]),
    array([[0] * 7 + [1] + [0] * 2]),
    array([[0] * 8 + [1] + [0] * 1]),
    array([[0] * 9 + [1]])]

#construct p4_x, p4_y
p4_x = M[train[0]]
p4_y = one_hot_encode[0]

for i in range(1, M[train[0]].shape[0]):
    p4_y = vstack((p4_y, one_hot_encode[0]))
for n in range(1, 10):
    p4_x =vstack((p4_x, M[train[n]]))
    for j in range(M[train[n]].shape[0]):
        p4_y = vstack((p4_y, one_hot_encode[n]))
p4_x= vstack((ones((1, p4_x.shape[0])), p4_x.T))
p4_y = p4_y.T

#construct init_w, b, alpha
init_w = np.zeros((784,10))
b = np.zeros((1,10))
alpha = 1e-10;
init_w = vstack((b, init_w))

if not os.path.exists("w_7000.txt"):
    p4_weight = grad_descent_p4(gradient_W, p4_x, p4_y, init_w, alpha)
else:
    p4_weight = np.loadtxt("w_7000.txt")
```

Part 5

Below is the new learning curve produced by the gradient descent with momentum included. As the figure shows, both the green line and red line reached the plateau at a faster speed than the gradient descent without momentum and the performances for both line have improvement to different extents. The only difference between part4 and part5 is the gradient descent function.




```
def grad_descent_p4(df, x, y, init_w, alpha):
    EPS = 1e-15   #EPS = 10**(-5)
    max_iter = 7000
    iter  = 1
    t = init_w
    #t = vstack((b, init_w))
    prev_t = t-10*EPS;
    while norm(t - prev_t) >  EPS and iter <= max_iter:
        if(iter % 500 == 0):
            filename ='w_'+str(iter)+'.txt'
            np.savetxt(filename, t)
            print(t);
        prev_t = t.copy()
        t -= alpha*df(x, y, t)
        iter += 1
    return t
```

Part 6

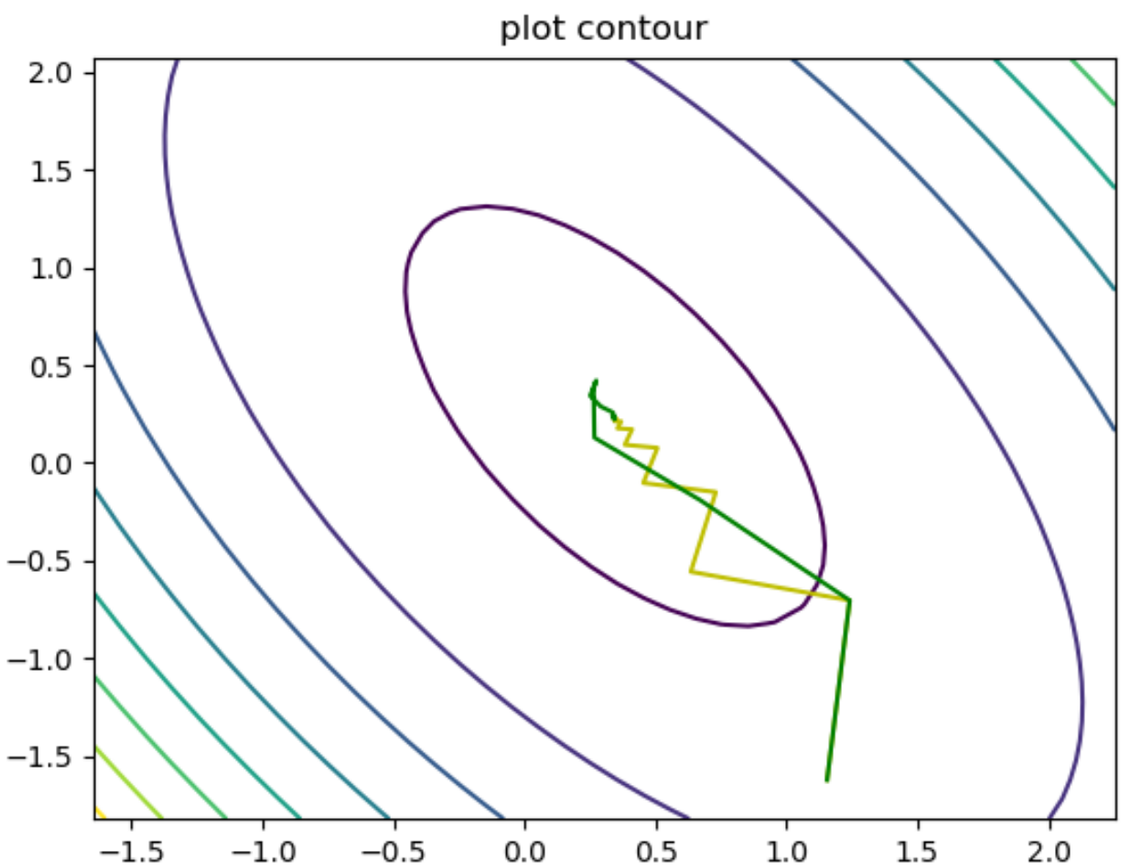
```
def grad_descent_p6_v2(x, y, init_t, w1,w2,w1r,w1c,w2r,w2c, f, df, loss_functi
    EPS = 1e-5
    max_iter=20
    enable_momentum = gamma
    row, column = init_t.shape[0],init_t.shape[1]
    v = np.zeros((row, column))
    trajectory = []
    t = init_t.copy()
    prev_t = t-10*EPS
    t[w1r,w1c] = w1
    t[w2r,w2c] = w2
    current_alpha = 0.0035
    current_loss = loss_function(y, f(x,t))
```

```
    iter  = 1
    while norm(t - prev_t) >  EPS and iter <= max_iter:
        trajectory.append((t[w1r,w1c],t[w2r,w2c]))
        prev_t = t.copy()
        v = enable_momentum*v + current_alpha*df(y, f(x,t), x)
        t[w1r,w1c] -= v[w1r,w1c]
        t[w2r,w2c] -= v[w2r,w2c]
        if iter % 5 == 0:
            current_loss = loss_function(y, f(x,t))
            print "iteration", iter
            print current_loss
        iter += 1

    return trajectory

def loss_function_p6(Wb, x, y, w1, w2, w1r, w1c, w2r, w2c):
    Wb_cp = Wb.copy()
    Wb_cp[w1r,w1c] = w1
    Wb_cp[w2r,w2c] = w2
    b1 = Wb_cp[:,-1:]
    W1 = Wb_cp[:,:-1]
    L1 = np.matmul(W1,x.T)+b1
    output = softmax(L1)

    return NLL(y.T, output)
```

```
def part6(w1r,w1c,w2r,w2c):
    np.random.seed(0)
    train_x_p6, train_y_p6,validation_x,validation_y, test_x,
    test_y = load_data()
    np.random.seed(1)
    Wb = np.random.normal(0.,0.001,[10,785])

    if not os.path.exists("Wb.txt"):
        Wb = grad_descent_p6(forward, backward, NLL, train_x_p6,
        train_y_p6.T, Wb,alpha=0.000001)
        np.savetxt("Wb.txt",Wb)
    else:
        Wb = np.loadtxt("Wb.txt")
    temp1 = Wb[w1r,w1c]
    temp2 = Wb[w2r,w2c]
    w1s = np.arange(temp1-2, temp1+2, 0.1)
    w2s = np.arange(temp2-2, temp2+2, 0.1)

    Cost = np.zeros([w1s.size, w2s.size])
    for ind1, w1 in enumerate(w1s):
        for ind2, w2 in enumerate(w2s):
            Cost[ind2,ind1] =  loss_function_p6(Wb,
                train_x_p6,train_y_p6,w1,w2,w1r,w1c,w2r,w2c)
    w1_, w2_ = np.meshgrid(w1s, w2s)
    without_mo = grad_descent_p6_v2(train_x_p6, train_y_p6.T, Wb,
        temp1+0.8,temp2-1.8,w1r,w1c,w2r,w2c,forward, backward, NLL,
        gamma = 0)
    with_mo = grad_descent_p6_v2(train_x_p6, train_y_p6.T, Wb,
        temp1+0.8,temp2-1.8,w1r,w1c,w2r,w2c,forward, backward, NLL,
        gamma = 0.4)

    contour_plot = plt.contour(w1_, w2_, Cost, camp=cm.coolwarm)
    plt.plot([a for a, b in without_mo], [b for a,b in without_mo], 'y',
    label="without Momentum")
    plt.plot([a for a, b in with_mo], [b for a,b in with_mo], 'g',
    label="with Momentum")
    plt.title('plot contour')
    plt.savefig('part6.jpg')

np.random.seed(1)
```

```
w1c = 28*(14  + int(round(6*np.random.rand())) - 3) + 14 +
int(round(6*np.random.rand())) - 3
w2c = 28*(14  + int(round(6*np.random.rand())) - 3) + 14 +
int(round(6*np.random.rand())) - 3
w1r = int(round((10-1)*np.random.rand()))
w2r = int(round((10-1)*np.random.rand()))

part6(w1r,w1c,w2r,w2c)
```

As the figure above shows, green line uses the method of momentum, so the green line approach the cave at a faster speed and a more direct way. The yellow contour didn't use momentum, so it hovers around the minimum and eventually arrive the point in a slower way.



Part 7

First of all, we will need to define the number of layers to be N, excluding the input layer. A number of K Neurons are in each layer, so therefore, there are $K \times N$ neurons in total. Because the neurons are fully connected, there are also $K \times K \times N$ total weights. The Cost for this fully vectorized network can be computed as:

$$\frac{\partial C}{\partial W(N, 1...K, 1...K)}$$

Notice that we also have

$$\frac{\partial C}{\partial N} = [\frac{\partial C}{\partial N1} ... \frac{\partial C}{\partial Nk}]$$

Similar to the situation of other layers, we will need $KN+K^2N$ computations to compute the cost regarding all weights, including the cost of matrix multiplication. This is because we will require $K \times N$ computations to compute the following equation at each layer.

$$\frac{\partial C}{\partial N} = [\frac{\partial C}{\partial N1} ... \frac{\partial C}{\partial Nk}]$$

$$\frac{\partial C}{\partial W(N, 1...K, 1...K)}$$

requires us add k^2 more computations, to get the gradient computation cost with respect to each weight separately.

$$\frac{\partial C}{\partial W(N, 1...1)} ... \frac{\partial C}{\partial W(N, K...K)}$$

This requires k^2 plus each one requires the corresponding derivative of the previous layer, So the total cost: $2k^2$ total for every weight: $k^2 \sum_{i=0}^{N-1}(N-i)$

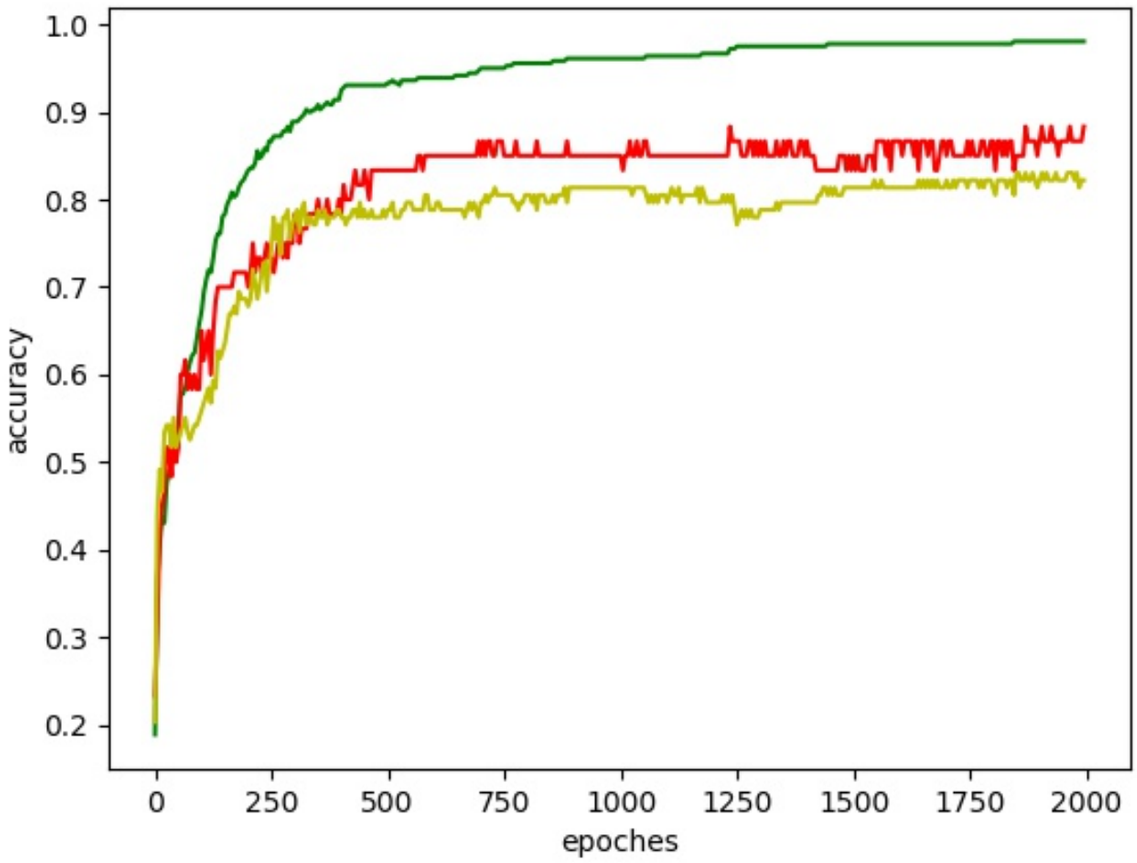
Part 8

```
def construct_set(target):
    y = np.empty((6,0))
    data = np.empty((1024,0))
    for a in one_hot_encoding.keys():
        for image in target[a]:
            y = np.hstack((y,np.array(one_hot_encoding[a])))
            im = imread("cropped/"+image)
            flat_im = (im/255.).reshape(1024,1)
            data = np.hstack((data,flat_im))
    return data.T,y.T

def produce():
    train_data,train_label = construct_set(train_set)
    test_data,test_label = construct_set(test_set)
    valid_data,valid_label = construct_set(valid_set)
    return train_data,train_label,test_data,
        test_label,valid_data,valid_label
```

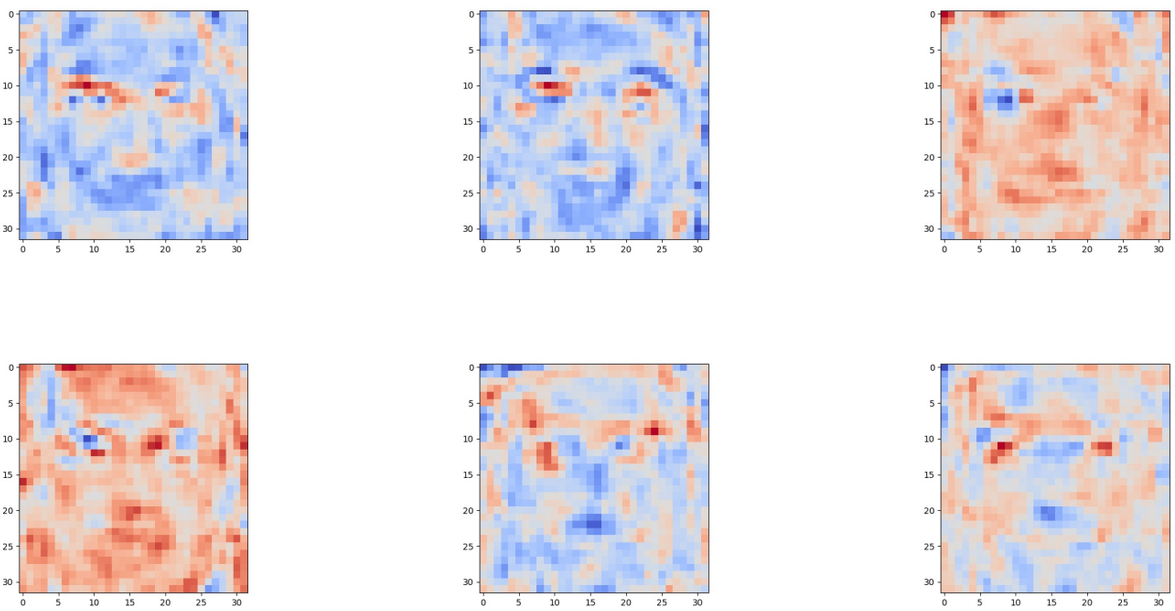
```
def part8(alpha, epoch_times):
    torch.manual_seed(0)
    train_data,train_label,test_data,
        test_label,valid_data,valid_label = produce()
    TRAIN = np.hstack((train_data,
    np.argmax(train_label, 1).reshape((train_data.shape[0],1))))
    VALID = np.hstack((valid_data,
    np.argmax(valid_label, 1).reshape((valid_data.shape[0],1))))
    TEST = np.hstack((test_data,
    np.argmax(test_label, 1).reshape((test_data.shape[0],1))))

    dim_x = 32*32
    dim_h = 12
    dim_out = 6
    dtype_float = torch.FloatTensor
    dtype_long = torch.LongTensor
    model = torch.nn.Sequential(
            torch.nn.Linear(dim_x, dim_h),
            torch.nn.Tanh(),
            torch.nn.Linear(dim_h, dim_out),
            torch.nn.Softmax()
            )
    model[0].weight.data.normal_(0.0,0.01)
    model[2].weight.data.normal_(0.0,0.01)
    model[0].bias.data.normal_(0.0,0.01)
    model[2].bias.data.normal_(0.0,0.01)
    dataloader = DataLoader(TRAIN, batch_size=32,shuffle=True)
    loss_fn = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=alpha)
    train_x = Variable(torch.from_numpy(TRAIN[:,:-1]),
        requires_grad=False).type(dtype_float)
    validation_x = Variable(torch.from_numpy(VALID[:,:-1]),
        requires_grad=False).type(dtype_float)
```

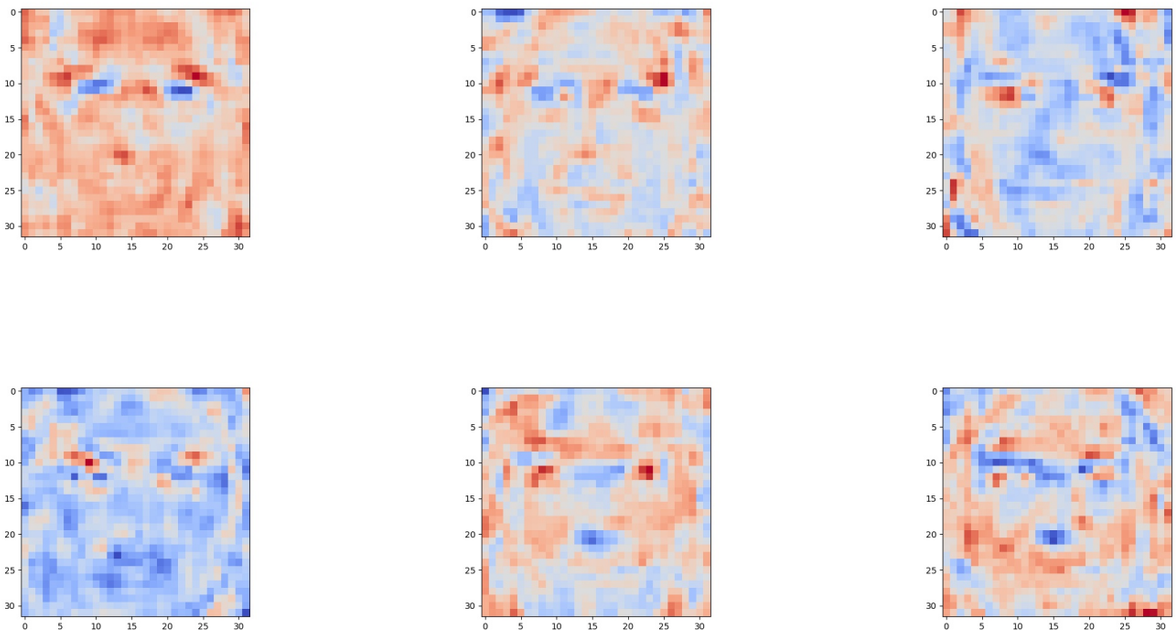
```
    test_x = Variable(torch.from_numpy(TEST[:,:-1]),
        requires_grad=False).type(dtype_float)

    tr,va, te, cx= [],[],[],[]
    for epoch in range(epoch_times):
        for i, data in enumerate(dataloader):
            x = Variable(data[:,:-1], requires_grad=False).type(dtype_float)
            y_classes =
                Variable(data[:,-1], requires_grad=False).type(dtype_long)
            y_pred = model(x)
            loss = loss_fn(y_pred, y_classes)
            model.zero_grad()
            loss.backward()
            optimizer.step()

        if epoch % 5 == 0:
            cx.append(epoch)

            train_set_result = model(train_x).data.numpy()
            tr.append((np.mean(np.argmax(train_set_result, 1)
                == TRAIN[:,-1])))

            valid_set_result = model(validation_x).data.numpy()
            va.append((np.mean(np.argmax(valid_set_result, 1)
                == VALID[:,-1])))

            test_set_result = model(test_x).data.numpy()
            te.append((np.mean(np.argmax(test_set_result, 1)
                == TEST[:,-1])))

    plt.plot(cx, tr,'g',cx, va,'r',cx, te, 'y')
    plt.xlabel('epoches')
    plt.ylabel('accuracy')
    plt.savefig('part8.jpg')
    plt.show()

    y_pred = model(test_x).data.numpy()
    print ("accuracy:")
    print (np.mean(np.argmax(y_pred, 1) == TEST[:,-1]))
    return model
model = part8(2e-4, 2000)
```

Before continuing with part8, what needs to be ready is training, validation, test data with their corresponding labels. Then use the build in function np.hstack() to concatenate the data and label together and form a matrix. respectively name them TRAIN, VALID, TEST. create model using torch.nn.Sequential. Load data using the torch dataloader and define loss function as torch.nn.CrossEntropyLoss(). Finally optimize the model with the optimizer in pytorch by a learning rate of 2e-4. Use the model after optimizing and calculate the performance in each of 5 epoched with epoch numbers increasing from 0 to 2000. the plot below is the product. green line is train set performance, red line is the accuracy of model on validation set. yellow line is the performance on test set. As you can see, train set has the best accuracy with value more than 0.95 and validation set is at the second place with performance about 0.85. Test set result is below valid set but more than 0.75.


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
accuracy
0
250
500
750
1000
1250
1500
1750
2000
epoches

Part 9

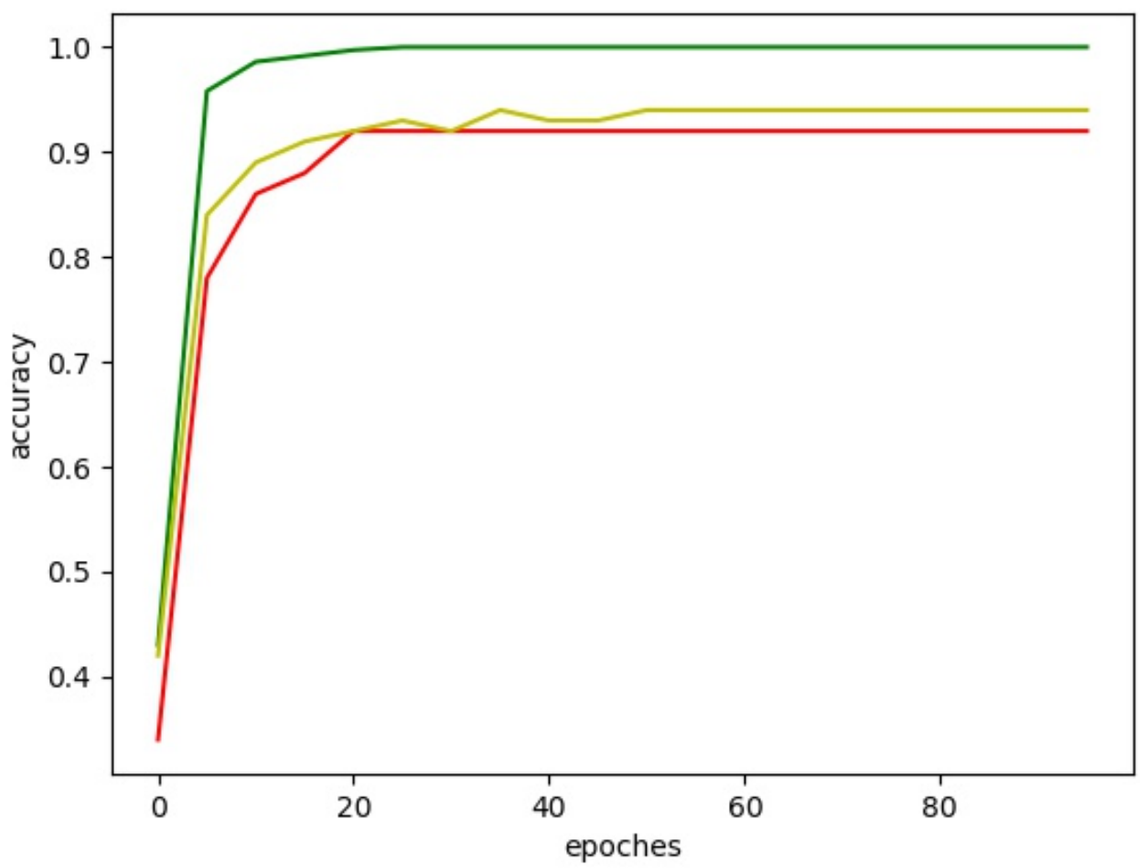




Part 10

```
def part10(alpha, epoch_times):
    torch.manual_seed(0)
    model = MyAlexNet()
    train_data,train_label,test_data,test_label,valid_data,valid_label
        = produce(model)
    TRAIN = np.hstack((train_data,
        np.argmax(train_label, 1).reshape((train_data.shape[0],1))))
    VALID = np.hstack((valid_data,
        np.argmax(valid_label, 1).reshape((valid_data.shape[0],1))))
    TEST = np.hstack((test_data,
        np.argmax(test_label, 1).reshape((test_data.shape[0],1))))

    dtype_float = torch.FloatTensor
    dtype_long = torch.LongTensor
    dataloader = DataLoader(TRAIN, batch_size=32,shuffle=True)

    loss_fn = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.classifier.parameters(), lr=alpha)

    train_x = Variable(torch.from_numpy(TRAIN[:,:-1]),
        requires_grad=False).type(dtype_float)
    validation_x = Variable(torch.from_numpy(VALID[:,:-1]),
        requires_grad=False).type(dtype_float)
    test_x = Variable(torch.from_numpy(TEST[:,:-1]),
        requires_grad=False).type(dtype_float)
    tr, va, te, cx = [],[],[],[]

    for epoch in range(epoch_times):
        for i, data in enumerate(dataloader):
            x = Variable(data[:,:-1], requires_grad=False).type(dtype_float)
            y_classes = Variable(data[:,-1],
                requires_grad=False).type(dtype_long)
            y_predit_result = model.classifier(x)
            loss = loss_fn(y_predit_result, y_classes)
            model.classifier.zero_grad()
            loss.backward()
            optimizer.step()

        if epoch % 5 == 0:
            cx.append(epoch)
            train_set_result = model.classifier(train_x).data.numpy()
            tr.append((np.mean(np.argmax(train_set_result, 1)
```

```
                == TRAIN[:,-1])))
            valid_set_result = model.classifier(validation_x).data.numpy()
            va.append((np.mean(np.argmax(valid_set_result, 1)
                == VALID[:,-1])))

            test_set_result = model.classifier(test_x).data.numpy()
            te.append((np.mean(np.argmax(test_set_result, 1)
                == TEST[:,-1])))
    plt.plot(cx, tr,'g',cx, va,'r', cx, te, 'y')
    plt.xlabel('epoches')
    plt.ylabel('accuracy')
    plt.savefig('part10.jpg')
    plt.show()
    test_x = Variable(torch.from_numpy(TEST[:,:-1]),
        requires_grad=False).type(dtype_float)
    y_predit_result = model.classifier(test_x).data.numpy()
    print ("accuracy:" + str(np.mean(np.argmax(y_predit_result, 1)
        == TEST[:,-1])))
    return model

model = part10(1e-4, 400)
```



look at the plot above, you can see that with an iteration time of only 100, the accuracy had already reach the plateau and become stable. Moreover the accuracy of both test set and validation set reached 0.9 and the performance of training set approach 1.0. this result largely improve the model in part 8. the way to improve the model are in the code bow above. when constructing the data matrix, we use the function below instead of the previous construct_set function in part8.

```
def construct_set(target,model):
    data = np.empty((0,9216))
    y = np.empty((6,0))
    for a in one_hot_encoding.keys():
        images_set = target[a]
```

```
        for image in images_set:
            y = np.hstack((y,np.array(one_hot_encoding[a])))
            img = imread("cropped_227/" + image)[:,:,:3]
            img = imresize(img,(227,227))
            img = img - np.mean(img.flatten())
            img = img/np.max(np.abs(img.flatten()))
            img = np.rollaxis(img, -1).astype(np.float32)
            img = Variable(torch.from_numpy(img).unsqueeze_(0),
                requires_grad=False)
            img = model.process(img)
            data = np.vstack((data,img))
    return data,y.T
```

the general idea of how to extract the values of the activations of AlexNet on the face images in a particular layer. use the below function process data when creating the training data matrix.

```
class MyAlexNet(nn.Module):
    def __init__(self, num_classes = 6):
        ...
    def forward(self,x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x

    def process(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        feature = x.data.numpy()
        return feature

def construct_set(target,model):
    data = np.empty((0,9216))
    y = np.empty((6,0))
    for a in one_hot_encoding.keys():
        images_set = target[a]
        for image in images_set:
            y = np.hstack((y,np.array(one_hot_encoding[a])))
            img = imread("cropped_227/" + image)[:,:,:3]
            img = imresize(img,(227,227))
            img = img - np.mean(img.flatten())
            img = img/np.max(np.abs(img.flatten()))
            img = np.rollaxis(img, -1).astype(np.float32)
            img = Variable(torch.from_numpy(img).unsqueeze_(0),
                requires_grad=False)
            img = model.process(img)
            data = np.vstack((data,img))
    return data,y.T
```

also we can use the trained model one indivial image and recognize the actor in that image by parsing the image in to torch variable and pass in the model.forward() function as argument.

```
test_img_list = ['hader0.jpg', 'harmon0.jpg']

for image in test_img_list:
    img = imread("cropped_227/"+image)[:,:,:3]
    img = imresize(img,(227,227))
    img = img - np.mean(img.flatten())
    img = img/np.max(np.abs(img.flatten()))
    img = np.rollaxis(img, -1).astype(np.float32)
    img = Variable(torch.from_numpy(img).unsqueeze_(0), requires_grad=False)

    prob = model.forward(img).data.numpy()[0]
    answer = np.argsort(prob)
```

```
    for i in range(6):
        print("------------------------")
        print("answer:", part10_act[answer[i]])
        print("probability:", prob[answer[i]])

    print("========================")
    ind = np.argmax(model.forward(img).data.numpy())
    prob = model.forward(img).data.numpy()[0][ind]
    print("best␣candidate␣is:␣" + part10_act[ind] + "␣with␣prob:␣"+ str(prob))
```