

# Hwk5&6

Xiaohan Liu 659692941 CS 559 November 6, 2017

## Q.1

(a) Draw  $n = 300$  real numbers uniformly at random on  $[0, 1]$ , call them  $x_1, \dots, x_n$ .

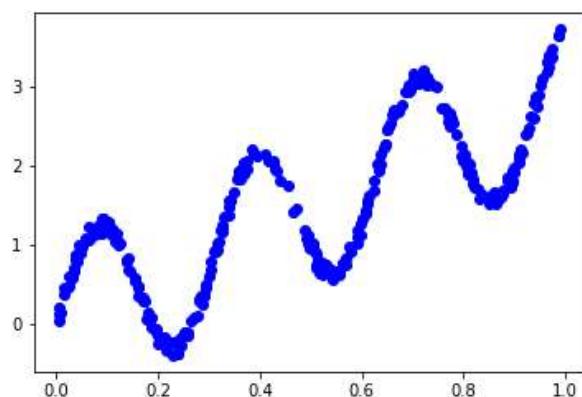
(b) Draw  $n$  real numbers uniformly at random on  $[1, 1]$ , call them  $1, \dots, n$ .

```
In [1]: import os
import struct
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl

np.random.seed(42)
n=300
X = np.random.uniform(0,1,size = n).reshape((n,1))
MU = np.random.uniform(-.1,.1,size = n).reshape((n,1))
Y = np.zeros(n).reshape((n,1))
```

(c) Let  $d_i = \sin(20x_i) + 3x_i + i$ ,  $i = 1, \dots, n$ . Plot the points  $(x_i, d_i)$ ,  $i = 1, \dots, n$ .

```
In [2]: D = np.sin(20*X) + 3*X + MU
```



We will consider a 1by N by 1 neural network with one input,  $N = 24$  hidden neurons, and 1 output neuron. The network will thus have  $3N + 1$  weights including biases. Let  $w$  denote the vector of all these  $3N + 1$  weights. The output neuron will use the activation function  $(v) = v$ ; all other neurons will use the activation function  $(v) = \tanh v$ . Given input  $x$ , we use the notation  $f(x,w)$  to represent the network output.

In [3]: *### Step 1 to 4*

```
# Number of hidden layer's neurons
N = 24
W = np.random.uniform(-.1,.1,size = (3*N+1)).reshape((3*N+1,1))
eta = 4
epsilon = 1e-8
mse = list()
mse.append(1e2)
mse.append(1e1)
l=0
m = 0

epoch = 2
U = np.zeros(N)
Z = U
A = W[0:N]
B = W[N:2*N]
C = W[2*N:3*N]
e = W[-1]
```

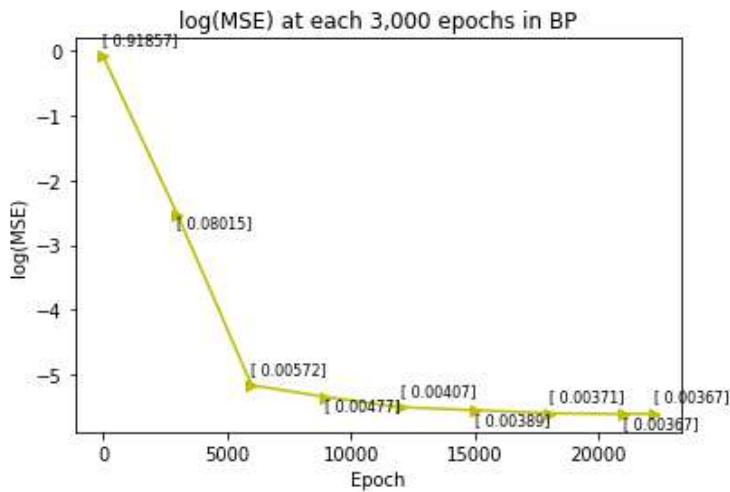
In [4]: *### Step 5*

```
while (np.abs(mse[-1]-mse[-2])>=epsilon):
    m += 1
    index = np.random.choice(np.arange(n), size= n, replace=False)
    # Step 5.A and C
    for i in index:
        U = X[i]*A + B
        Z = np.tanh(U)
        Y[i] = np.transpose(Z).dot(C) + e
        # update A:
        A += 2 / n * eta * X[i] * (D[i]-Y[i]) * (1-Z) * (1+Z) * C
        # update B:
        B += 2 / n * eta * (D[i]-Y[i]) * (1-Z) * (1+Z) * C
        # update C:
        C += 2 / n * eta * (D[i]-Y[i]) * Z
        # update e:
        e += 2 / n * eta * (D[i]-Y[i])
    # Step 5.B
    mse.append(np.sum((D-Y)**2)/n)
    epoch +=1

    if (epoch-2) % 2000 == 0:
        eta *= 0.5

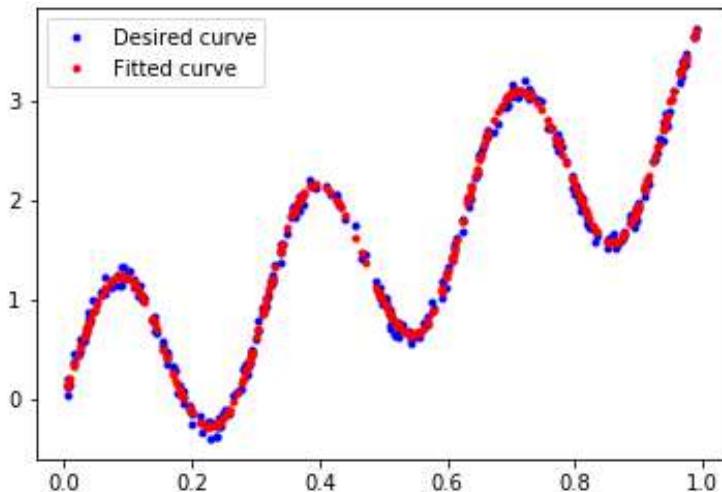
# delete the 2 dummy points      2
epoch -= 2
del(mse[0:2])
```

(d) Use the backpropagation algorithm with online learning to find the optimal weights/network that minimize the mean-squared error (MSE)  $\frac{1}{n} \sum_i f(x_i, w)^2$ . Use some of your choice. Plot the number of  $n_i=1$  epochs vs the MSE in the backpropagation algorithm.



(e) Plot the curve  $f(x, w_0)$  as  $x$  ranges from 0 to 1 on top of the plot of points in (c). The fit should be a “good” fit.

In [72] :



**(f) Pseudocode of implementation.**

1. Given  $n, N$ , and  $\epsilon$ .
2. Initialize  $\eta \in \mathbb{R}, W \in \mathbb{R}^{(3N+1) \times 1}$  randomly. Note:  $W = A, B, C, d$ , where  $A \in \mathbb{R}^N, B \in \mathbb{R}^N, C \in \mathbb{R}^N, d \in \mathbb{R}$ .
3. Initialize epoch = 0.
4. Initialize  $MSE_{epoch} = 0$  for epoch = 0, 1, ... .
5. Do SGD, so sample  $n$  numbers out range( $n$ ) without replacement as the index:
  1. for i in index, do (this loop is where we compute the mse of each epoch):
    1. Compute the induced local fields as a vector with the current training sample and weights by:

$$U = x_i \cdot A + B \in \mathbb{R}^N$$

2. Then, get the  $N$  outputs from the  $N$  hidden neurons as a vector by:

$$Z = \tanh(U) \in \mathbb{R}^N$$

3. So, the output of the last neuron will be:

$$y_i = Z^T \cdot C + e$$

2. Compute the MSE for current epoch:

$$MSE_{epoch} = \frac{1}{n} \sum_{i=1}^n (d_i - y_i)^2$$

Then, update the epoch:

$$epoch \leftarrow epoch + 1.$$

3. for  $i = 1$  to  $n$ , do (this loop is where we update the weights):

Note:

$$\frac{\partial \tanh(U)}{\partial U} = \begin{bmatrix} \frac{\partial \tanh(u_1)}{\partial u_1} & \frac{\partial \tanh(u_1)}{\partial u_2} & \dots & \frac{\partial \tanh(u_1)}{\partial u_{24}} \\ \frac{\partial \tanh(u_2)}{\partial u_1} & \frac{\partial \tanh(u_2)}{\partial u_2} & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial \tanh(u_{24})}{\partial u_1} & \dots & \dots & \frac{\partial \tanh(u_{24})}{\partial u_{24}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \tanh(u_1)}{\partial u_1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{\partial \tanh(u_{24})}{\partial u_{24}} \end{bmatrix} = \begin{bmatrix} 1 - \tanh^2(u_1) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 - \tanh^2(u_{24}) \end{bmatrix}$$

3. update the  $A$  here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial MSE}{\partial A} = -x_i \cdot \frac{2}{n} (d_i - y_i) \cdot \frac{\partial \tanh(U)}{\partial U} \cdot C = -x_i \cdot \frac{2}{n} (d_i - y_i) \cdot (1 - \tanh(U)) \odot (1 + \tanh(U)) \odot C$$

2. The update would be:

$$A \leftarrow A - \eta \frac{\partial MSE}{\partial A} = A + \eta x_i \cdot \frac{2}{n} (d_i - y_i) \cdot (1 - \tanh(U)) \odot (1 + \tanh(U)) \odot C$$

4. update the B here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial MSE}{\partial B} = -1 \cdot \frac{2}{n} (d_i - y_i) \cdot \frac{\partial \tanh(U)}{\partial U} \cdot C = -1 \cdot \frac{2}{n} (d_i - y_i) \cdot (1 - \tanh(U)) \odot (1 + \tanh(U)) \odot C$$

2. The update would be:

$$B \leftarrow B - \eta \frac{\partial MSE}{\partial B} = B + \eta \frac{2}{n} (d_i - y_i) \cdot (1 - \tanh(U)) \odot (1 + \tanh(U)) \odot C$$

5. update the C here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial MSE}{\partial C} = -\tanh(U) \cdot \frac{2}{n} (d_i - y_i)$$

2. The update would be:

$$C \leftarrow C - \eta \frac{\partial MSE}{\partial C} = C + \eta \frac{2}{n} (d_i - y_i) \cdot \tanh(U)$$

6. update the e here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial MSE}{\partial e} = -\frac{2}{n} (d_i - y_i)$$

2. The update would be:

$$e \leftarrow e - \eta \frac{\partial MSE}{\partial e} = e + \eta \frac{2}{n} (d_i - y_i)$$

7. for every kth ( $k = 6000$ ) epoch:

$$\eta = 0.5 * \eta$$

4. Loop to A, if  $abs(MSE_{epoch} - MSE_{epoch-1}) > \epsilon$ .

Reamrk: Could consider to reduce the  $\eta$  after a certain epochs systematically.

## Q.2.

(150pts) In this computer project, we will design a neural network for digit classification using the backpropagation algorithm (see the notes above). You should use the MNIST data set (see Homework 2 for details) that consists of 60000 training images and 10000 test images. The training set should only be used for training, and the test set should only be used for testing. Your final report should include the following:

### 0. Input the data files

In [2]: # save the original binary MNIST data files in 0-255

## 1. First try

1. Architecture of NN: 784 input neurons, 1 hidden layer with  $N_1 = 100$  neurons, 10 output neurons with 0 or 1 as their outputs to be compatible with the use of 0-1 vector outputs.

2. Output: With  $[1 \ 0 \dots \ 0]$  representing a 0,  $[0 \ 1 \ 0 \dots \ 0]$  representing a 1 and so on, it is denoted as  $f(x_i, w)$ .

3.1 Activation functions:,  $\tanh()$  from input to 1st hidden layer and softmax from 1st hidden layer to output.

3.2 Learning rate: 0.1.

3.2 Weight initialization: all weights are initialized randomly from uniform(-0.5,0.5).

4. Energy function: MSE in the form of:  $\frac{1}{n} \sum_{i=1}^n \|d_i - f(x_i, w)\|^2$ .

5. Other tricks: reduce the eta by 50% when at plateau.

```
In [6]: ## initialization
np.random.seed(2)
# use first n samples from training data to train the NN
n = 60000
# number of neurons in 1st hidden layer, N1
N1 = 100
# learning rate
eta = 1e-2
# convergence threshold
epsilon = 5e-8
# epoch number
epoch = 2
# batch size
bs = 1
# initialize errors
mse = list()
mse.append(1e3)
mse.append(5e2)
# initialize real outputs given the current w
out = np.zeros((n,10))
# initialize weights from input to 1st hidden layer
u = np.random.uniform(-0.05,0.05,size = (784*N1)).reshape(784,N1)
# initialize weights from 1st hidden layer to output
v = np.random.uniform(-0.2,0.2,size = (10*N1)).reshape(N1,10)
# initialize the bias
b = np.random.normal(-0.1,0.1,size = (bs*N1)).reshape(bs,N1)

# define softmax activation function
def softmax(w):
```

```

e = np.exp(np.array(w) - np.max(w))
dist = e / np.sum(e)
return dist

# define a continuous vector to 0-1 vector function
def vec_to_01(x):
    s1 = np.zeros(x.shape[0])
    q = np.argmax(x)
    s1[q] = 1
    return s1

# define softmax's Jacobian matrix function
def softmax_jacobian(s):
    return np.diagflat(s) - np.outer(s, s)

# define tanh's Jacobian matrix function
def tanh_jacobian(t):
    return np.diagflat(1-t**2)

def cross_S(out, Label):
    return - np.sum(np.multiply(out, np.log(Label)) + np.multiply((1-out), np.log(1-Label)))

In [10]: while (np.abs(mse[-1]-mse[-2])>=epsilon):
            index = np.random.choice(np.arange(n), size=n, replace=False)
            X_train_rndm = X_train[index]
            # this loop is where we update the weights
            for i in range(0,n, bs):
                lf1 = np.dot(X_train_rndm[i:(i+bs)], u) + b
                z0 = np.tanh(lf1)           #compute the input layer's out
                lf2 = np.dot(z0,v)          #compute the 1st H layers' induced local field
                out[i:(i+bs)] = softmax(lf2)      #compute the 1st H layer's out

                # update u:
                temp1 = 2 / n * eta * (Y_train[i:(i+bs)] - out[i:(i+bs)]).dot(softmax_jacobian(lf2))
                temp2 = np.dot(temp1, v.T)
                temp2 = temp2.dot(tanh_jacobian(lf1))
                u -= np.outer(X_train_rndm[i:(i+bs)], temp2)
                # update b:
                b -= temp2
                # update v:
                v -= np.outer(z0, temp1)

                # calculate mse for the current epoch
                mse.append(np.sum((out-Y_train)**2)/n)
                epoch +=1

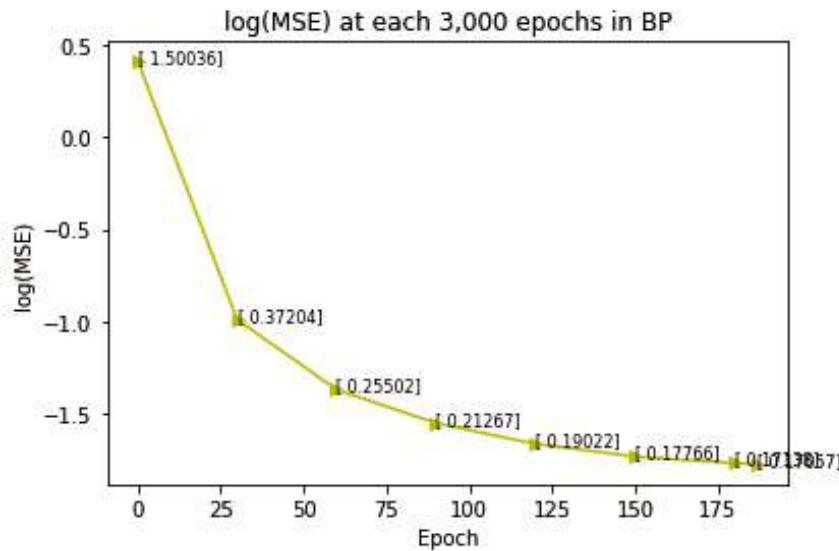
                if mse[epoch-2] <= mse[epoch-1]:
                    eta *= 0.5
                #if (epoch-2) % 2 == 0:

                # delete the 2 dummy points
                epoch -= 2
                del(mse[0:2])

```

```
In [12]: u0 = u
b0 = b
v0 = v
# Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    lf1 = np.dot(X_train_rndm[i:(i+bs)],u) + b #compute the input layers' induced local field
    z0 = np.tanh(lf1) #compute the input layer's output
    lf2 = np.dot(z0,v) #compute the 1st H layers' induced local field
    out[i:(i+bs)] = softmax(lf2) #compute the 1st H layer's output
    # Find the largest component of v0 = [v0', v1', ...v9']^T
    predic_out = np.argmax(out[i,:])
    # If the predicted output is different to the testing label, error +=1
    diff = predic_out - np.argmax(Y_test[i,:])
    if diff != 0:
        error_test += 1
error_test/10000
```

Out[12]: 0.122



Clearly, we are not satisfied with a 12.2% error rate on the test data. And I concluded that it was due to the undernumber of the neurons in the hidden layer, careless initialization of parameters and/or hyperparameters, and lack of other tricks. Thus, I had my 2nd try:

## 2. Second try

1. Architecture of NN: 784 input neurons, 1 hidden layer with  $N_1 = 150$  neurons, 10 output neurons with 0 or 1 as their outputs to be compatible with the use of 0-1 vector outputs.

2. Output: With  $[1 \ 0 \dots \ 0]$  representing a 0,  $[0 \ 1 \ 0 \dots \ 0]$  representing a 1 and so on, it is denoted as  $f(x_i, w)$ .

3.1 Activation functions:  $\tanh()$  from input to 1st hidden layer and  $\text{softmax}$  from 1st hidden layer to output.

**3.2 Learning rate:** Set to 0.03 due to the empirical principle  $\eta \sim O(1/\sqrt{m})$ , m is 784 here.

**3.2 Weight initialization:** For hyperbolic tangent units: sample a Uniform(-r, r) with  $r = \sqrt{\frac{6}{fan-in+fan-out}}$ ; For softmax tangent units: sample a Uniform(-r, r) with  $r = 4\sqrt{\frac{6}{fan-in+fan-out}}$  (fan-in is the number of inputs of the unit, fan-out is the number of outputs of the unit); For bias: sample the same way as the associated weight.

**4. Energy function:** MSE in the form of:  $\frac{1}{n} \sum_{i=1}^n \|d_i - f(x_i, w)\|^2$ .

**5. Other tricks:** SGD optimizer; reduce the eta by 50% when at plateau; carry 90% of the last epoch's gradient over to the current epoch when updating; Early termination (done manually)

```
In [278]: ## initialization
np.random.seed(2)
# use first n samples from training data to train the NN
n = 60000
# number of neurons in 1st hidden layer, N1
N1 = 150
# learning rate
eta = 0.03
# convergence threshold
epsilon = 5e-7
# epoch number
epoch = 2
# batch size
bs = 1
# initialize errors
mse = list()
mse.append(1e3)
mse.append(5e2)
# initialize real outputs given the current w
out = np.zeros((n,10))
# initialize weights from input to 1st hidden layer
u = np.random.uniform(-0.05,0.05,size = (784*N1)).reshape(784,N1)
# initialize weights from 1st hidden layer to output
v = np.random.uniform(-0.2,0.2,size = (10*N1)).reshape(N1,10)
# initialize the bias
b = np.random.uniform(-0.05,0.05,size = (bs*N1)).reshape(bs,N1)
# initialize condi
m = 0
l = 0
# intialize momentum
momentum = 0.9
```

```
In [279]: while (np.abs(mse[-1]-mse[-2])>=epsilon):
```

```

m += 1
index = np.random.choice(np.arange(n), size= n, replace=False)
X_train_rndm = X_train[index]
# re-initialzie last epoch's gradients
grad_uold = 0
grad_bold = 0
grad_vold = 0
# this loop is where we update the weights
for i in range(0,n, bs):
    l +=1
    lf1 = np.dot(X_train_rndm[i:(i+bs)],u) + b #compute the input layers' induced
    z0 = np.tanh(lf1) #compute the input layer's out
    lf2 = np.dot(z0,v) #compute the 1st H layers' induced local field
    out[i:(i+bs)] = softmax(lf2) #compute the 1st H layer's out
    if (True):
        # update u:
        temp1 = 2 / n * eta * (Y_train[i:(i+bs)]-out[i:(i+bs)]).dot(softmax_jacobian(lf1))
        temp2 = np.dot(temp1,v.T)
        temp2 = temp2.dot(tanh_jacobian(lf1))
        grad_u= np.outer(X_train_rndm[i:(i+bs)],temp2)
        grad_u = grad_uold*momentum+grad_u
        u -= grad_u
        grad_uold = grad_u
        # update b:
        grad_b = (temp2)
        grad_b = grad_bold*momentum+grad_b
        b -=grad_b
        grad_bold = grad_b
        # update v:
        grad_v = np.outer(z0, temp1)
        grad_v = grad_vold*momentum+grad_v

        v -=grad_v
        grad_vold = grad_v
    # calculate mse for the current epoch
    mse.append(np.sum((out-Y_train)**2)/n)
    epoch +=1
    if mse[epoch-2] <= mse[epoch-1]:
        eta *= 0.5
# delete the 2 dummpy points

```

```

In [14]: u0 = u
b0 = b
v0 = v
# Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    lf1 = np.dot(X_train_rndm[i:(i+bs)],u) + b #compute the input layers' induced local field
    z0 = np.tanh(lf1) #compute the input layer's out
    lf2 = np.dot(z0,v) #compute the 1st H layers' induced local field
    out[i:(i+bs)] = softmax(lf2) #compute the 1st H layer's out
    # Find the largest component of v0 = [v0', v1', ...v9']^T

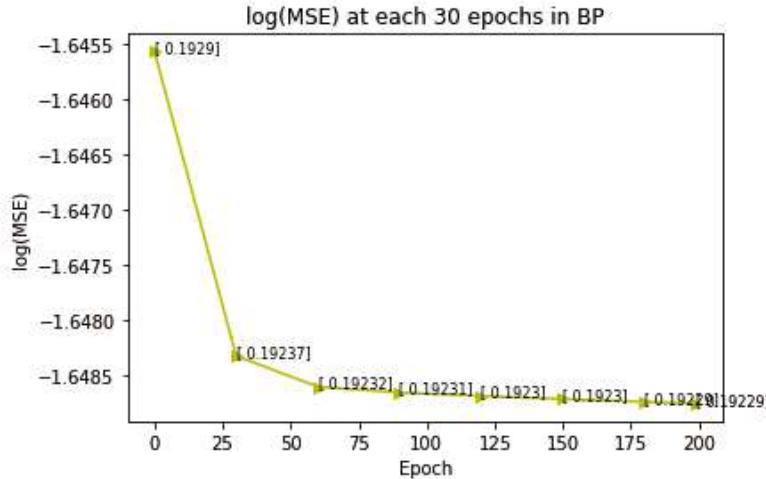
```

```

predic_out = np.argmax(out[i,:])
# If the predicted output is different to the testing label, error +=1
diff = predic_out - np.argmax(Y_test[i,:])
if diff != 0:
    error_test += 1
error_test/10000

```

Out [14]: 0.0672



6.72% to 12.2% error rate on the test data was a big boost. But I concluded that cross-entropy will be a better choice of the energy function of the than good old MSE. Meanwhile, adding more neurons will definitely help me to get into the 5% error rate club. Thus, I had my 3rd try:

### 3. Third try

1. Architecture of NN: 784 input neurons, 1 hidden layer with  $N_1 = 250$  neurons, 10 output neurons with 0 or 1 as their outputs to be compatible with the use of 0-1 vector outputs.

2. Output: With  $[1 \ 0 \dots \ 0]$  representing a 0,  $[0 \ 1 \ 0 \dots \ 0]$  representing a 1 and so on, it is denoted as  $f(x_i, w)$ .

3.1 Activation functions:  $\tanh()$  from input to 1st hidden layer and softmax from 1st hidden layer to output.

3.2 Learning rate: Set to 0.03 due to the empirical principle  $\eta \sim O(1/\sqrt{m})$ , m is 784 here.

3.2 Weight initialization: For hyperbolic tangent units: sample a Uniform(-r, r) with  $r = \sqrt{\frac{6}{fan-in+fan-out}}$ ; For softmax units: sample a Uniform(-r, r) with  $r = 4\sqrt{\frac{6}{fan-in+fan-out}}$  (fan-in is the number of inputs of the unit, fan-out is the number of outputs of the unit); For bias: sample the same way as the associated weight.

4. Energy function: Cross-entropy in the form of:  $-\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{10} d_{ij} \log(f(x_i, w)_j)$ , where j represents the 10 output classes.

5. Other tricks: SGD optimizer; reduce the eta by 50% when at plateau; carry 82% of the last epoch's gradient over to the current epoch when updating; Early termination (done manually)

```

np.random.seed(2)
# use first n samples from training data to train the NN
n = 60000
# number of neurons in 1st hidden layer, N1
N1 = 250
# learning rate
eta = 0.03
# convergence threshold
epsilon = 5e-8
# epoch number
epoch = 2
# batch size
bs = 1
# initialize errors
entropy = list()
entropy.append(1e1)
entropy.append(5e0)
# initialize real outputs given the current w
out = np.zeros((n,10))
# initialize weights from input to 1st hidden layer
u = np.random.uniform(-0.05,0.05,size = (784*N1)).reshape(784,N1)
# initialize weights from 1st hidden layer to output
v = np.random.uniform(-0.2,0.2,size = (10*N1)).reshape(N1,10)
# initialize the bias
b = np.random.normal(-0.1,0.1,size = (bs*N1)).reshape(bs,N1)
# initialize condi
m = 0
l = 0
# initialize momentum
momentum = 0.82

```

```

In [5]: while (np.abs(entropy[-1]-entropy[-2])>=epsilon):
    m += 1
    index = np.random.choice(np.arange(n), size= n, replace=False)
    X_train_rndm = X_train[index]
    # re-initialzie last epoch's gradients
    grad_uold = 0
    grad_bold = 0
    grad_vold = 0
    # this loop is where we update the weights
    for i in range(0,n, bs):
        lf1 = np.dot(X_train_rndm[i:(i+bs)],u) + b #compute the input layers' induced local
        z0 = np.tanh(lf1) #compute the input layer's out
        lf2 = np.dot(z0,v) #compute the 1st H layers' induced local field
        out[i:(i+bs)] = softmax(lf2) #compute the 1st H layer's out
        if (True):

```

```

# update u:
temp1 = 1 / n * eta * np.sum(Y_train[i:(i+bs)] / out[i:(i+bs)], axis = 0).dot(softmax_jacobi
temp2 = np.dot(temp1,v.T)
temp2 = temp2.dot(tanh_jacobian(lf1))
grad_u= np.outer(X_train_rndm[i:(i+bs)],temp2)
grad_u = grad_uold*momentum+grad_u
u -= grad_u
grad_uold = grad_u

# update b:
grad_b = (temp2)
grad_b = grad_bold*momentum+grad_b
b -=grad_b
grad_bold = grad_b

# update v:
grad_v = np.outer(z0, temp1)
grad_v = grad_vold*momentum+grad_v
v -=grad_v
grad_vold = grad_v

# calculate mse for the current epoch
entropy.append(-np.sum(np.log(out) * Y_train)/n)
epoch +=1

if entropy[epoch-2] <= entropy[epoch-1]:
    eta *= 0.5

# delete the 2 dummy points
epoch -= 2
del(entropy[0:2])

```

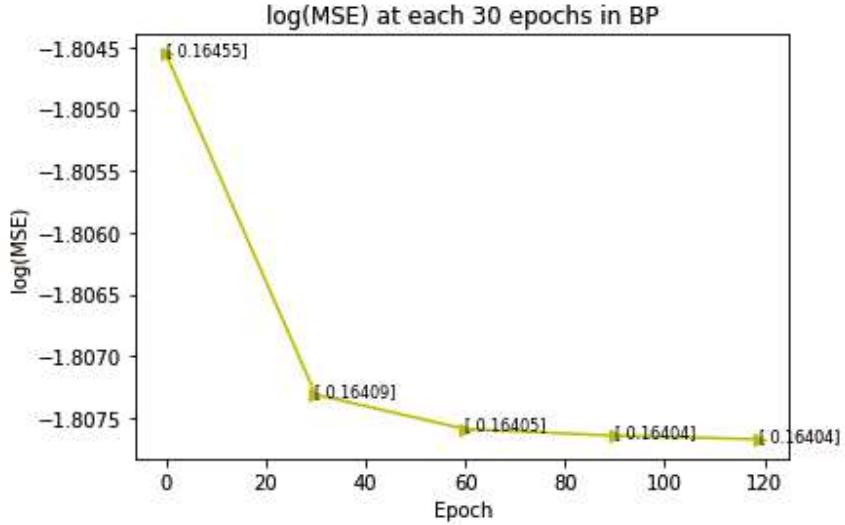
In [157]:

```

u0 = u
b0 = b
v0 = v
# Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    lf1 = np.dot(X_train_rndm[i:(i+bs)],u) + b #compute the input layers' induced local field
    z0 = np.tanh(lf1) #compute the input layer's out
    lf2 = np.dot(z0,v) #compute the 1st H layers' induced local field
    out[i:(i+bs)] = softmax(lf2) #compute the 1st H layer's out
    # Find the largest component of v0 = [v0', v1', ...v9']^T
    predic_out = np.argmax(out[i,:])
    # If the predicted output is different to the testing label, error +=1
    diff = predic_out - np.argmax(Y_test[i,:])
    if diff != 0:
        error_test += 1
error_test/10000

```

Out[157]: 0.0463



Now, with an errorrate of 4.63%. We have accomplished our goal with the 3rd try, where the pseudocode was listed below:

1. Given  $n, N_1$ , momentum and  $\epsilon$ .
2. Initialize  $\eta \sim O(1/\sqrt{m}) \in \mathbb{R}, u \in \mathbb{R}^{(784) \times N_1}$  and  $b \in \mathbb{R}^{1 \times N_1}$  by Uniform(-r, r) with  $r = \sqrt{\frac{6}{fan-in+fan-out}}$ , then  $v \in \mathbb{R}^{N_1 \times 10}$  by Uniform(-r, r) with  $r = 4\sqrt{\frac{6}{fan-in+fan-out}}$ . (fan-in is the number of inputs of the unit, fan-out is the number of outputs of the unit)
3. Initialize epoch = 0.
4. Initialize  $loss_{epoch} = 0$  for epoch = 0, 1, ... .
5. Do SGD, so sample n numbers out range(n) without replacement as the index:
  1. for i in index, do (this loop is where we compute the mse of each epoch):
    1. Compute the 1st induced local fields as a vector with the current training sample and weights by:  

$$K = x_i \cdot u + b \in \mathbb{R}^{1 \times N_1}$$
  2. Then, get the outputs from the  $N_1$  hidden neurons as a vector by:  

$$Z = \tanh(K) \in \mathbb{R}^{1 \times N_1}$$
  3. Do the same for the rest, the outputs of the 10 output neurons will be:  

$$L = Z^T \cdot v$$
  

$$y_i = softmax(L)$$

2. Compute the Cross-entropy for current epoch:

$$loss_{epoch} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{10} d_{ij} \log(f(x_i, w)_j)$$

Then, update the epoch:

$$epoch \leftarrow epoch + 1.$$

for i = 1 to n, do (this loop is where we update the weights):

Note:

$$\frac{\partial \tanh(U)}{\partial U} = \begin{bmatrix} \frac{\partial \tanh(u_1)}{\partial u_1} & \frac{\partial \tanh(u_1)}{\partial u_2} & \dots & \frac{\partial \tanh(u_1)}{\partial u_q} \\ \frac{\partial \tanh(u_2)}{\partial u_1} & \frac{\partial \tanh(u_2)}{\partial u_2} & & \vdots \\ \vdots & \ddots & & \vdots \\ \frac{\partial \tanh(u_q)}{\partial u_1} & \dots & \dots & \frac{\partial \tanh(u_q)}{\partial u_q} \end{bmatrix} = \begin{bmatrix} 1 - \tanh^2(u_1) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 - \tanh^2(u_q) \end{bmatrix} = (1 - \tanh(U)) \odot (1 + \tanh(U))$$

$$\frac{\partial \text{softmax}(U)}{\partial U} = \begin{bmatrix} \frac{\partial \text{softmax}(u_1)}{\partial u_1} & \frac{\partial \text{softmax}(u_1)}{\partial u_2} & \dots & \frac{\partial \text{softmax}(u_1)}{\partial u_q} \\ \frac{\partial \text{softmax}(u_2)}{\partial u_1} & \frac{\partial \text{softmax}(u_2)}{\partial u_2} & & \vdots \\ \vdots & \ddots & & \vdots \\ \frac{\partial \text{softmax}(u_{24})}{\partial u_1} & \dots & \dots & \frac{\partial \text{softmax}(u_{24})}{\partial u_q} \end{bmatrix} = \begin{bmatrix} u_1(\delta_{11} - u_1) & \dots & u_1(\delta_{1q} - u_q) \\ \vdots & \ddots & \vdots \\ u_q(\delta_{q1} - u_1) & \dots & u_q(\delta_{qq} - u_q) \end{bmatrix} = U \cdot I - U \otimes U^T$$

3. update the u here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial \text{loss}}{\partial u} = -x_i \cdot \frac{1}{n} (d_i - y_i) \cdot \frac{\partial \text{softmax}(L)}{\partial L} \cdot v \cdot \frac{\partial \tanh(K)}{\partial K} = -x_i \cdot \frac{1}{n} (d_i - y_i) \cdot (L \cdot I - L \otimes L^T) \cdot v \cdot (1 - \tanh(K)) \odot (1 + \tanh(K))$$

2. The update would be:

$$u_{(t)} \leftarrow u_{(t-1)} \cdot \text{momentum} - \eta \frac{\partial \text{loss}}{\partial u} = u_{(t-1)} \cdot \text{momentum} - x_i \cdot \frac{1}{n} (d_i - y_i) \cdot (L \cdot I - L \otimes L^T) \cdot v \cdot (1 - \tanh(K)) \odot (1 + \tanh(K))$$

4. update the b here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial \text{loss}}{\partial b} = -\frac{1}{n} (d_i - y_i) \cdot \frac{\partial \text{softmax}(L)}{\partial L} \cdot v \cdot \frac{\partial \tanh(K)}{\partial K} = -\frac{1}{n} (d_i - y_i) \cdot (L \cdot I - L \otimes L^T) \cdot v \cdot (1 - \tanh(K)) \odot (1 + \tanh(K))$$

2. The update would be:

$$b_{(t)} \leftarrow b_{(t-1)} \cdot \text{momentum} - \eta \frac{\partial \text{loss}}{\partial b} = b_{(t-1)} \cdot \text{momentum} - \frac{1}{n} (d_i - y_i) \cdot (L \cdot I - L \otimes L^T) \cdot v \cdot (1 - \tanh(K)) \odot (1 + \tanh(K))$$

5. update the v here:

1. According to the backpropagation algorithm, we have:

$$\frac{\partial \text{loss}}{\partial v} = -\tanh(K) \cdot \frac{1}{n} (d_i - y_i) \cdot \frac{\partial \text{softmax}(L)}{\partial L}$$

2. The update would be:

$$v_{(t)} \leftarrow v_{(t-1)} \cdot \text{momentum} - \eta \frac{\partial \text{loss}}{\partial v} = v_{(t-1)} \cdot \text{momentum} - \frac{1}{n} (d_i - y_i) \cdot Z \cdot (L \cdot I - L \otimes L^T)$$

6. for every kth (k = 6000) epoch:

$$\eta = 0.5 * \eta$$

if loss\_{epoch} - loss\_{epoch-1} > 0

Loop to A, if abs(loss\_{epoch} - loss\_{epoch-1}) > ε.