Fengnan Wang fwang40

HW5

1. Use the back propagate and gradient descent algorithm for fitting curve

a) First draw x,v,d for inputs and desired outputs

b) Build a network with(input=1, hidden layer=24, output=1) for one neuron

c) I need to consider bias for input and hidden so I create an array with(input=2*[1], hidden=25*[1]), you can find these inputs and hidden neurons as ai and ah in my code

d) The activate function for hidden layer is tanh(), and y=x for output layer that means derivative would be dtanh(1-y**2) and 1

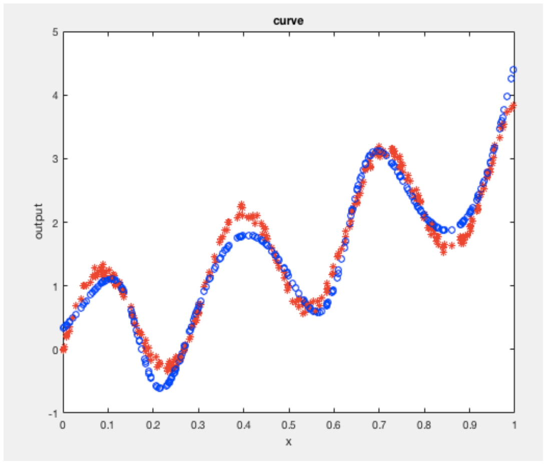
e) Consider the gradient vector(g), I use a 4*4 matrix to present it, the first column is derivative for wn0, the second column is derivative for wni, the third column is derivative for wni' and the first item in last column is derivative for wn0'

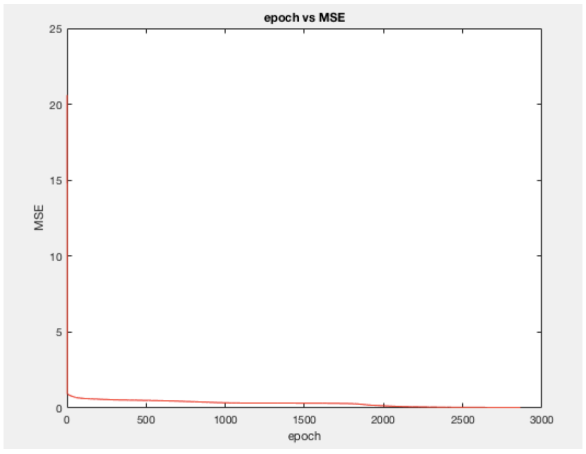
f) Start training, the first error gets from the first neuron by 1/n*(for i=1:n (di –f(xi,w))**2) and n =1, then we update weights by w=w-u*g, and calculate the next error (1/2*(d1-f(x1,w)**2+(d0-f(x0,w))**2), we should train all of these 300 neurons to find the optimal weights

g) Then for j=1:100000(the number can be pretty large because the loop will break whatever), train these 300 neurons to find the best weights for making error cost function get to its global lowest point, which means when g ==0.

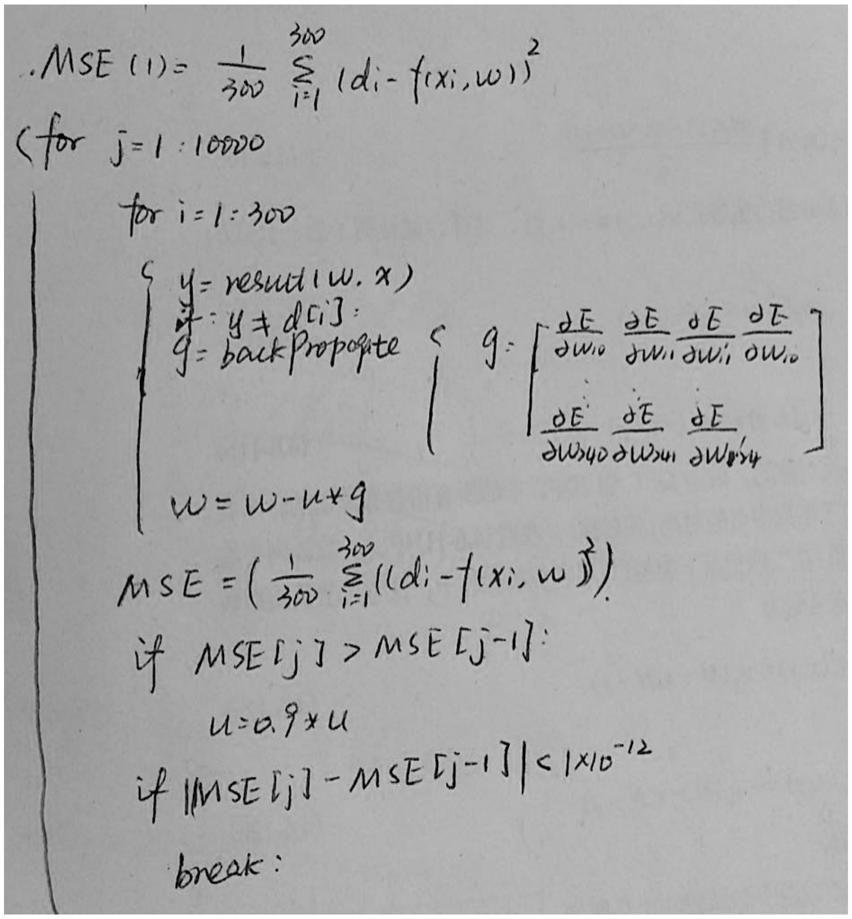
h) Show the figure and we can find it fits the original curve perfectly.

Results：





Pseudocode:



```
.MSE (1) = $\frac{1}{300}\sum_{i=1}^{300}(d_i - f(x_i, w))^2$
{for j = 1 : 10000
    for i = 1 : 300
    { y = result(w. x)
      if : y ≠ d[i] :
      g = backPropogate  { g : $\begin{bmatrix} \frac{\partial E}{\partial w_{10}} & \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{i1}} & \frac{\partial E}{\partial w_{i0}} \\ \vdots & \vdots & \vdots & \\ \frac{\partial E}{\partial w_{340}} & \frac{\partial E}{\partial w_{341}} & \frac{\partial E}{\partial w_{354}} & \end{bmatrix}$
      w = w - u * g
    MSE = ($\frac{1}{300}\sum_{i=1}^{300}((d_i - f(x_i, w))^2$)
    if MSE[j] > MSE[j-1] :
        u = 0.9 * u
    if |MSE[j] - MSE[j-1]| < $1\times10^{-12}$
        break :
```

HW6

2. design a neural network for digit classification using the backpropagation algorithm

a) network topology, i.e. how many layers, how many neurons in each layer

   input neurons=784; hidden neurons=300; output neurons=10

   Reason: there is no problem in choosing input and output layer, and 300 for hidden layer, I just take the complexity into my consideration, if the number is too big, I will make training period longer; if the number is too small, I' m afraid it will be hard to get optimal result. I have tried 500 in hidden layer, it presents a better result, but takes longer time.

b) The way represented digits 0, . . . , 9 in the output layer.

   I use the same setup as in Homework 2, 10 output neurons, with [1 0 ···0] representing a 0, [0 1 0…0] representing a 1

c) Neuron activation functions, learning rates for each neuron, and any dynamic update of learning rates

   The activation function for hidden neurons is tanh()

   The activation function for output neurons is sigmoid()

   I use these two functions because they are the most common functions and these functions are centered at 0 and they make the calculation easier.

   for a given fixed u, the algorithm may not always result in a monotonically decreasing MSE (the descent may overshoot the locally optimal point). I modify the gradient descent algorithm in such a way that decreasing u=0.9*u when the MSE has increased.

d) The energy/distance functions of choice.

   The energy function is the most common one MSE=1/n*(for i=1:n(di-f(xi,w))**2), because I need a quadratic function to find the global lowest point.

e) other tricks such as regularization, dropout, momentum method, etc.

   Other things I do:

   1) I normalize the inputs: x=x/255 to make x ranging from 0-1, because if the input neurons are close to each other (I mean there is no big diversity among them like 56, or 113), it will be easier for my network to train them
   2) Improve weight initialization, I choose initial weights of a neural network from the range (−1 / sqrt(d),1/sqrt(d)), where d is the number of inputs. It is assumed, that the sets are normalized - mean 0, variance 1. This is really important, because if I set the initial weights too large, it will saturation, otherwise, it will be slow learning. It is proved to be highly effective for the fisrt mse is nearly to 0.05 without training.
   3) I use momentum method to train my network, the velocity vector gets from the delta, which records the velocity

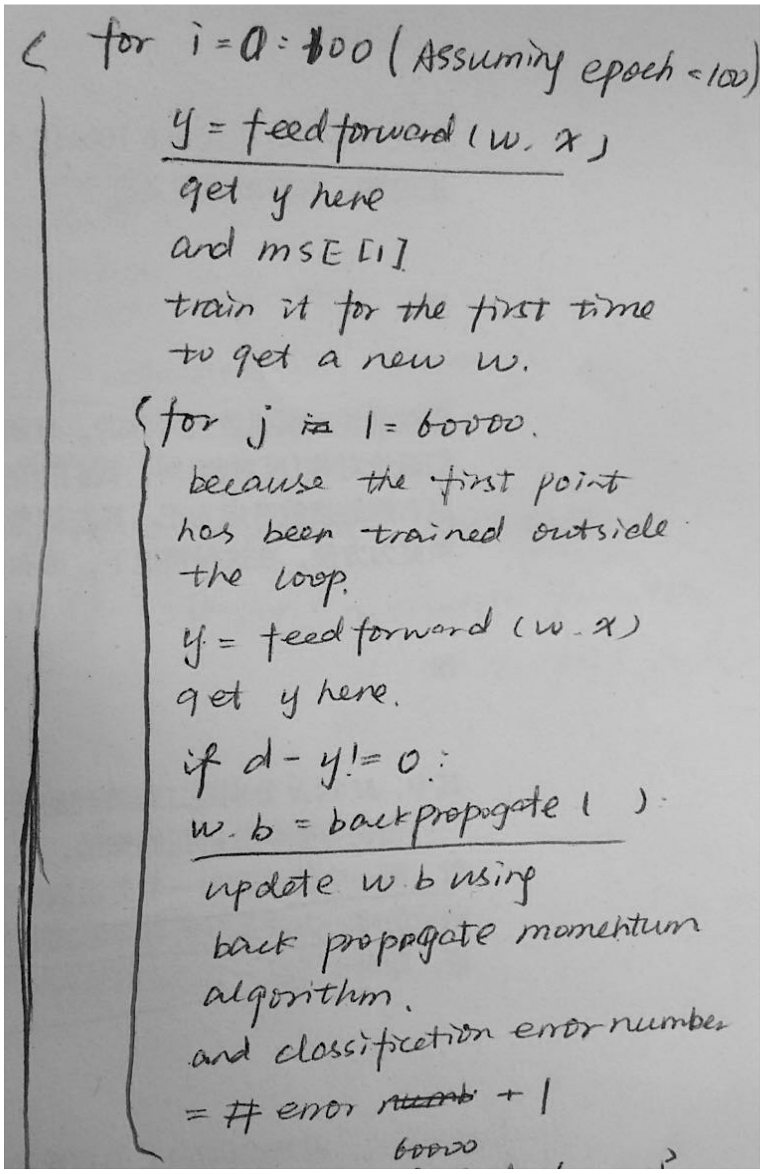
f) My design process

   1) At first, I inputed the wrong variables to dtanh and dsigmoid, because I am misguided by function taught, I thought it should be dtanh(v) and dsigmoid(v).

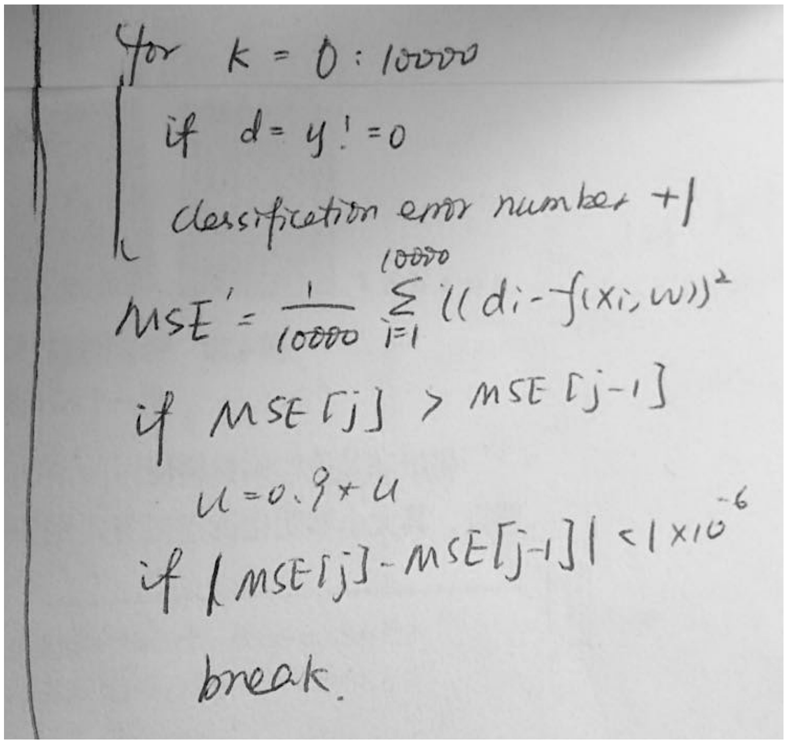
However, y=tanh(x), and tanh’ (x)=1-y*2, that means I should input y

2) I gave a big value to u such that u=1, and the mse turn out to be uncountable, then I changed the value to be 0.01, it worked.
3) I assumed I would get better result if I added momentum in my algorithm, however, it presented not as good as I had expected. I show my result below, I think my problem is a wrong value given for the momentum.

Pseucode:



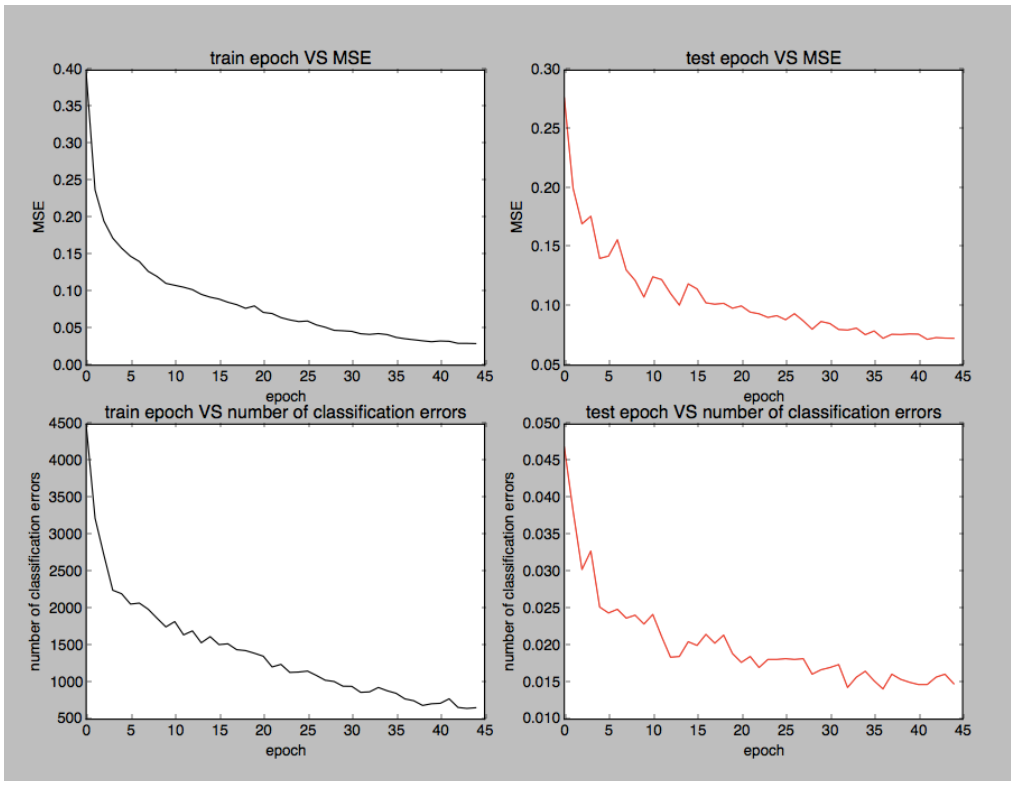
```
{ for i = 0 : 100 ( Assuming epoch = 100)
    y = feedforward ( w. x )
    get y here
    and msE [i]
    train it for the first time
    to get a new w.
    { for j ~~in~~ 1 = 60000.
        because the first point
        has been trained outside
        the loop.
        y = feedforward ( w. x )
        get y here.
        if d - y != 0 :
        w. b = backpropogate (  ) :
        update w. b using
        back propogate momentum
        algorithm.
        and classification error number
        = # error ~~numb~~ + 1
                  60000
    }
```



```
for k = 0 : 10000
    if d = y != 0
    classification error number +1
$MSE' = \frac{1}{10000} \sum_{i=1}^{10000} ((d_i - f(x_i, w))^2$
if MSE[j] > MSE[j-1]
    $u = 0.9 * u$
if |MSE[j] - MSE[j-1]| < $1 \times 10^{-6}$
    break.
```

Result:

momentum=0,hidden_layer=500,break rate=0.0000001



```
training time 0.000328063964844
losscost 0.0295613583333
MSE [0.3960937083333318, 0.23757013500000013, 0.1953543800000011,
0.1722801500000011, 0.15869280166666816, 0.14775070833333559,
0.1405525216666686, 0.12725901833333531, 0.12026622000000152,
0.11095817666666799, 0.10832080666666784, 0.105733065000001,
0.10241161333333462, 0.096024938333334128, 0.092219240000000674,
0.089670196666667173, 0.085343303333333689, 0.082088113333333407,
0.077084500000000056, 0.080345585000000247, 0.071581069999999983,
0.070035353333332995, 0.064518898333333047, 0.061378321666666277,
0.059130193333333306, 0.059855521666666446, 0.054606918333333053,
0.051441346666666492, 0.047279279999999896, 0.046662124999999798,
0.045831536666666443, 0.042630314999999884, 0.041719054999999935,
0.042831276666666612, 0.041459151666666645, 0.037721696666666638,
0.035836644999999973, 0.034464103333333329, 0.033052260000000007,
0.031799344999999993, 0.032810974999999971, 0.032365628333333285,
0.029583599999999953, 0.029561358333333298, 0.029271413333333343]
training time 1380.63118196
the number of classification errors [4483, 3225, 2725, 2248, 2201, 2061, 2074, 1991,
1869, 1751, 1823, 1643, 1697, 1536, 1619, 1511, 1522, 1442, 1430, 1394, 1354,
1208, 1243, 1134, 1139, 1152, 1093, 1028, 1011, 947, 945, 864, 870, 930, 885, 851,
775, 750, 686, 709, 715, 776, 659, 646, 655]
the number of classification errors in training [0.0469, 0.0382, 0.0303, 0.0328,
0.0252, 0.0244, 0.0249, 0.0237, 0.0241, 0.0229, 0.0242, 0.0212, 0.0184, 0.0185,
0.0205, 0.02, 0.0215, 0.0203, 0.0214, 0.0189, 0.0177, 0.0185, 0.017, 0.0181, 0.0181,
0.0182, 0.0181, 0.0182, 0.0161, 0.0167, 0.017, 0.0174, 0.0143, 0.0157, 0.0165,
0.0152, 0.0141, 0.0161, 0.0154, 0.015, 0.0147, 0.0147, 0.0157, 0.0161, 0.0148]
mse1 [0.27654487999999949, 0.19973274999999988, 0.16974736999999993,
0.17618446000000018, 0.14046906999999997, 0.14246983999999988,
0.15628424999999999, 0.13063650999999984, 0.12185151999999994,
0.10771982999999993, 0.12488766999999983, 0.12252384999999995,
0.11084050999999986, 0.10087490999999987, 0.11890652999999989,
0.11460281999999988, 0.10284411999999978, 0.10169329999999985,
0.1024223199999998, 0.098274399999999887, 0.10019756999999992,
0.094970869999999791, 0.093556129999999751, 0.090477749999999829,
0.091954099999999761, 0.088520269999999887, 0.093637999999999888,
0.087540469999999829, 0.080468739999999969, 0.087014349999999852,
0.085322039999999877, 0.080240879999999945, 0.079757699999999876,
0.081401149999999894, 0.075860539999999976, 0.078980539999999946,
0.072782119999999978, 0.076178229999999916, 0.075957289999999858,
0.076477579999999892, 0.076309109999999916, 0.071899879999999944,
0.073371749999999958, 0.072936059999999983, 0.072798339999999934]
```

the number of classification errors in testing 0.0148 = 98.52%

I achieve 98.52% success rate on the test set

My code:

hw5

```
#!/usr/local/bin/python
import numpy as np
import random
import matplotlib.pyplot as plt
import sympy as sp
import math
#calculate a random number where: a<= rand < b
def rand(a,b):
   return a+(b-a)*random.random()
#make a matrix
def initalize(n):
   x=[]
   v=[]
   d=[float(0.0)]*n
   for i in range(n):
      x.append(rand(0.0,1.0))
      v.append(rand(-0.1,0.1))
      d[i]=math.sin(20*x[i])+3*x[i]+v[i]
   return x,d
def makeMatrix(I,J,fill=0.0):
   m=[]
   for i in range(J):
      m.append([fill]*I)
   m=np.array(m)
   return m
def sigmoid(x):
   return math.tanh(x)
```

```
# derivative of our sigmoid function, in terms of the output
(i.e. y)
def dsigmoid(y):
    return 1-y**2
# create weights
w_hi= makeMatrix(2, 24)
w_oh= np.array([0.0]*(24+1))#no=1
#set them to random value
for i in range(2):
    for j in range(24):
        w_hi[j][i] = rand(-0.1,0.1)
for j in range(24+1):
    w_oh[j] = rand(-0.1,0.1)
def result(x,w_hi,w_oh):
    ai=np.array([1,x])
    ah=np.array([1]*(24+1))
    # ai=[1,x]
    # hidden activations
    v=np.dot(w_hi,ai)
    #print 'v',v
    for j in range(1,len(v)+1):
        ah[j] = sigmoid(v[j-1])
    vprime=np.dot(w_oh,ah)
    #print 'w_oh',w_oh
    ao = vprime
    #print 'ao',ao
    return ai,ah,ao # #ao=1
def backPropagate(ai,ah,ao,w_oh,target):
    g=np.array([[0.0 for x in range(24)] for y in range(4)])
    delta=[]
    g[3][0]=-1*2*(target-ao)#e/w10'
    for j in range(24):
        g[0][j]=-1*2*(target-ao)*dsigmoid(ah[j+1])*w_oh[j+1]
        g[1][j]=-1*ai[1]*2*(target-ao)*dsigmoid(ah[j+1])*w_oh[j+1]
        g[2][j]=-1*ah[j+1]*2*(target-ao)
    #print'g',g
    return g
x,d=initalize(300)
u=0.01
errorresult=[]
e=[]
y=[0]*300
epoch=[0]
for i in range(300):
    ai,ah,ao=result(x[i],w_hi,w_oh)
    e.append((d[i]-ao)**2)
errorresult.append(sum(e)/300.0)
#i=0
for j in range(1,100000):
    error=[0]*300
    for i in range(300):
        ai,ah,ao=result(x[i],w_hi,w_oh)
        error[i]=((d[i]-ao)**2)
        if d[i]-ao!=0:
            #print 1,'j',j,'i',i
            g=backPropagate(ai,ah,ao,w_oh,d[i])
            w_hi[:,0]=w_hi[:,0]-u*g[0]
            w_hi[:,1]=w_hi[:,1]-u*g[1]
            w_oh[1:25]=w_oh[1:25]-u*g[2]
            w_oh[0]=w_oh[0]-u*g[3][0]
            #print 'g',g[2]
            #print 'w_hi_n',w_hi
            #print 'w_kh_n',w_oh
        y[i]=ao
    #print 'error=ao',ao,'desired',d[i]
    errorresult.append((sum(error))/(300.0))
    epoch.append(j)
    if errorresult[j]>errorresult[j-1]:
        u=0.9*u
        #print 'u',u
    if (abs(errorresult[j]-errorresult[j-1]))<=1e-12 and
errorresult[j]<1:
        break
    #print 'error',error[1:3],error[298:300],len(error),'j',j
    #if j % 500==0:
    print 'errorresult',errorresult[j]
    print'j',j
plt.plot(x,d,'ro')
plt.plot(x,y,'bo')
plt.xlabel('x')
plt.ylabel('output')
plt.title(' curve')
plt.show()
plt.plot(epoch,errorresult,'r-')
plt.xlabel('epoch')
plt.ylabel('MSE')
plt.title('epoch vs MSE')
plt.show()
#plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
```

My code:

hw6

```
#!/usr/local/bin/python
from keras.datasets import mnist
# I used this package to import MINST datasets, I didn't use any
other functions in this package to train my network
import matplotlib.pyplot as plt
import numpy as np
import time
# load the MNIST dataset
(train_image, train_label), (test_image, test_label) =
mnist.load_data()
pixels=train_image.shape[1]*train_image.shape[2]
#print train_image.shape (60000, 28, 28)
x=train_image.reshape(train_image.shape[0],pixels).astype('float3
2')
xprime=test_image.reshape(test_image.shape[0],pixels).astype('flo
at32')
#normalize the input form 0-255 to 0-1
x=x/255
xprime=xprime/255
#print train_ima.shape (60000, 784)
#print train_label.shape (60000,)
#print train_label[0]# 5
#change train_label to 10 elements matrix
d=np.zeros((train_label.shape[0],10))
for i in range(train_label.shape[0]):
   d[i][train_label[i]]=1
dprime=np.zeros((test_label.shape[0],10))
for i in range(test_label.shape[0]):
    dprime[i][test_label[i]]=1
#print d[3] [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
#print train_label[3] 1
#print dprime[5] [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
#print test_label[5] 1
def sigmoid(x):
    return 1/(1+np.exp(-x))
def dsigmoid(y):
    return y*(1-y)
def tanh(x):
    return np.tanh(x)
def dtanh(y):
    return 1-y*y
#normalize weight
def initialize(input_num,hidden_num,output_num):
    input_d=1.0/(input_num)**(1/2)
    hidden_d=1.0/(hidden_num)**(1/2)

wi=np.random.normal(loc=0,scale=input_d,size=(input_num,hidden_nu
m))

wo=np.random.normal(loc=0,scale=hidden_d,size=(hidden_num,output_
num))
    bi=np.ones(hidden_num)
    bo=np.ones(output_num)
    vi=np.zeros((input_num,hidden_num))
    vo=np.zeros((hidden_num,output_num))
    vbi=np.zeros(hidden_num)
    vbo=np.zeros(output_num)
    return wi,wo,bi,bo,vi,vo,vbi,vbo
```

```
#wi,wo=initialize(784,300,10)
#print wi.shape (784, 300)
#print wo.shape (300, 10)
def feedforward(wi,wo,x,bi,bo):
    v=np.dot(wi.T,x)+bi
    #v (20,)
    y=tanh(v)
    vprime=np.dot(wo.T,y)+bo
    #print 'vprime',vprime
    #vprime (3,)
    yprime=sigmoid(vprime)
    yprime=np.round(yprime,2)
    return v,y,vprime,yprime
def
backpropagate(d,x,v,vprime,y,yprime,momentum,u,vo,vi,vbo,vbi,bo,b
i,wo,wi):
    err=2*(d-yprime)
    #print 'err', err.shape,err
    delta_prime=dsigmoid(yprime)*err
    #print 'delta_prime' ,delta_prime.shape
    #print 'v',v
    delta=dtanh(y)*np.dot(wo,delta_prime)
    #print 'delta',delta.shape,delta
    delta_wo=-1*delta_prime*np.reshape(y,(y.shape[0],1))
    #print 'delta_wo',delta_wo.shape,delta_wo
    delta_wi=-1*delta*np.reshape(x,(x.shape[0],1))
    #print 'delta_wi',delta_wi.shape,delta_wi
    delta_bo=-1*delta_prime
    #print 'delta_bo',delta_bo,delta_bo
    delta_bi=-1*delta
    #print 'delta_bi', delta_bi.shape,delta_bi
    wo=wo-u*delta_wo-momentum*vo
    vo=delta_wo
    wi=wi-u*delta_wi-momentum*vi
    vi=delta_wi
    bo=bo-u*delta_bo-momentum*vbo
    vbo=delta_bo
    bi=bi-u*delta_bi-momentum*vbi
    vbi=delta_bi
    return wo,wi,bo,bi
##################################
# main
##################################
######################
# initialize variables
#######################
wi,wo,bi,bo,vi,vo,vbi,vbo=initialize(x.shape[1],500,10)
#print 'wi',wi
#print 'wo',wo
u=0.01
MSE=[1]
miss=[]
#print x.shape (60000, 784)
######################
#start training
######################
# take 60000 samples for tarining
start=time.time()
for i in range(100):
    print i
    #calculate the first error
    mse=[0]*60000
    v,y,vprime,yprime=feedforward(wi,wo,x[0],bi,bo)

wo,wi,bo,bi=backpropagate(d[0],x[0],v,vprime,y,yprime,0,u,vo,vi,v
bo,vbi,bo,bi,wo,wi)
    print 'yprime',yprime
    print 'd',d[0]
    error_begin=sum((d[0]-yprime)**2)
    mse[0]=error_begin
    mis=0
    mis1=0
    errorcost=[0]*10000
    for j in range(1,60000):
        start_loop=time.time()
        v,y,vprime,yprime=feedforward(wi,wo,x[j],bi,bo)
        if sum(d[j]-yprime)!=0:

wo,wi,bo,bi=backpropagate(d[j],x[j],v,vprime,y,yprime,0,u,vo,vi,v
bo,vbi,bo,bi,wo,wi)
            mis=mis+1
        error=sum((d[j]-yprime)**2)
        mse[j]=(error)
        end_loop=time.time()
    miss.append(mis)
    print 'training time',end_loop-start_loop
    MSE.append(sum(mse)/60000.0)
    print 'losscost',MSE[i]
    for k in range(10000):
        v,y,vprime,yprime=feedforward(wi,wo,xprime[k],bi,bo)
        if sum(d[k]-yprime)!=0:
            mis1=mis1+1
        errorcost[k]=(sum((dprime[k]-yprime)**2))
    mse1.append(sum(errorcost)/10000.0)
    miss1.append(mis1/10000.0)
    if MSE[i]>MSE[i-1]:
        u=0.9*u
    if abs(MSE[i]-MSE[i-1])<1e-6:
        break
end=time.time()
print 'MSE',MSE[1:]
print 'training time', end-start
print 'the number of classification errors',miss
print 'the number of classification errors in training',miss1
print 'mse1',mse1

fig, axes = plt.subplots(nrows=2, ncols=2)
axes[0, 0].plot( MSE[1:], 'k')
axes[0, 0].set_title(' train epoch VS MSE')
axes[0, 0].set_xlabel('epoch')
axes[0, 0].set_ylabel('MSE')
axes[0, 1].plot( mse1, 'r')
axes[0, 1].set_title('test epoch VS MSE')
axes[0, 1].set_xlabel('epoch')
axes[0, 1].set_ylabel('MSE')
axes[1, 0].plot( miss, 'k')
axes[1, 0].set_title('train epoch VS number of classification 
errors')
axes[1, 0].set_xlabel('epoch')
axes[1, 0].set_ylabel('number of classification errors')
```

```
axes[1, 1].plot( miss1, 'r')
axes[1, 1].set_title('test epoch VS number of classification
errors')
axes[1, 1].set_xlabel('epoch')
axes[1, 1].set_ylabel('number of classification errors')
plt.show()
```