Python Tutorial: Learning

Setup:

- network of [3,4,4,2] :
  - 3 inputs, 2 outputs
  - 2 hidden layers with 4 neurons each
- One-hot encoding
- Various Initialization Schemes

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

class1 = np.random.randn(20,3) + np.array([2,2,2])       #20x3
class2 = np.random.randn(20,3) + np.array([4,4,4])

X = np.vstack ((class1,class2))                          #40x2 combined samples
Y = np.vstack(( np.zeros((20,1))+([1,0]),
                np.zeros((20,1))+([0,1])))

width0 = 3                # input layer neurons
width1 = 4                # hidden layer 1 neurons
width2 = 4                # hidden layer 2 neurons
width3 = 2                # output layer neurons

#Normal distr
b0nn = np.random.randn(1, width1)
b1nn = np.random.randn(1, width2)
b2nn = np.random.randn(1, width3)
W0nn = np.random.randn(width0, width1)
W1nn = np.random.randn(width1, width2)
W2nn = np.random.randn(width2, width3)

#uniform distr
W0Xu = 2*(np.random.rand(width0, width1)-0.5)
W1Xu = 2*(np.random.rand(width1, width2)-0.5)
W2Xu = 2*(np.random.rand(width2, width3)-0.5)

#Xavier-like initialization sigmoid
W0 = W0nn/ np.sqrt(width0+width1)
W1 = W1nn/ np.sqrt(width1+width2)
W2 = W2nn/ np.sqrt(width2+width3)

#Xavier initialization sigmoid
W0 = W0Xu * np.sqrt(6/(width0+width1))
W1 = W1Xu * np.sqrt(6/(width1+width2))
W2 = W2Xu * np.sqrt(6/(width2+width3))
b0 = b0nn * np.sqrt(2)
b1 = b1nn * np.sqrt(2)
b2 = b2nn * np.sqrt(2)
```

Example 1: BGD vs mini-BGD vs SGD

- Previous setup network of [3,4,4,2], Sigmoid activations

```
alpha = 1
b = 5                               #batch size. b=1 for SGD, b<m for miniBGD, b=m for BGD
m = X.shape[0]                  #number of samples
E = []                              #initialize error vector (not needed)
acc= [0]                            #accuracy vector after each iteration

while acc[-1]<100:
    ix = list(range(m))
    np.random.shuffle (ix)      #randomly shuffle the data to improve performance
    X = X[ix,:].reshape(X.shape[0],X.shape[1])
    Y = Y[ix,:]
    for i in range(0,m,b):                  #get a batch
        Z1 = np.dot(X[i:i+b,:].reshape(b,X.shape[1]),W0)+b0     #1st layer (output)
        a1 = 1/(1+np.exp(-Z1))      #sigmoid activation of hidden layer 1
        Z2 = np.dot(a1,W1)+b1
        a2 = 1/(1+np.exp(-Z2))      #sigmoid activation of hidden layer 2
        Z3 = np.dot(a2,W2)+b2
        a3 = 1/(1+np.exp(-Z3))      #sigmoid activation of output, layer 3

        Yhat = a3                       #nnet output
        d = Yhat - Y[i:i+b,:]           #delta

        g1 = a1*(1-a1)                  #sigm derivative of layer 1
        g2 = a2*(1-a2)
        g3 = a3*(1-a3)

        dEda3 = d * g3
        dEda2 = np.dot(dEda3 , W2.T) * g2
        dEda1 = np.dot(dEda2 , W1.T) * g1

        dEdW2 = np.dot(a2.T, dEda3)
        dEdb2 = np.sum(dEda3, axis=0, keepdims=True)

        dEdW1 = np.dot( a1.T,dEda2)
        dEdb1 = np.sum(dEda2, axis=0)

        dEdW0 = np.dot( X[i:i+b,:].reshape(b,X.shape[1]).T, dEda1)
        dEdb0 = np.sum(dEda1, axis=0)

        W0 -= alpha/m * dEdW0
        b0 -= alpha/m * dEdb0
        W1 -= alpha/m * dEdW1
        b1 -= alpha/m * dEdb1
        W2 -= alpha/m * dEdW2
        b2 -= alpha/m * dEdb2

        #forward propagate to capture error and accuracy:
        Z1 = np.dot(X,W0)+b0            #1st layer (output)
        a1 = 1/(1+np.exp(-Z1))      #sigmoid activation of hidden layer 1
        Z2 = np.dot(a1,W1)+b1
        a2 = 1/(1+np.exp(-Z2))      #sigmoid activation of hidden layer 2
        Z3 = np.dot(a2,W2)+b2
        Yhat = 1/(1+np.exp(-Z3))        #sigmoid activation of output, layer 3

        E = np.append(E,np.sum(0.5*((Yhat-Y)**2)))
        acc = np.append(acc, np.sum(np.argmax(Yhat,axis=1)==np.argmax(Y,axis=1))/.4)
        print("accuracy: %d, batch number: %d" %(acc[-1],len(acc)), "\r", end="", flush=True)
```

Example 2: ReLu Activations

- Previous setup network of [3,4,4,2]

```
W0 = W0Xu*np.sqrt(6/width0)
W1 = W1Xu*np.sqrt(6/width1)
W2 = W2Xu*np.sqrt(6/width2)

alpha = .5
b = 10                              #batch size. b=1 for SGD, b<m for miniBGD, b=m for BGD
m = X.shape[0]                  #number of samples
E = []                              #initialize error vector (not needed)
acc= [0]                            #accuracy vector after each iteration

while acc[-1]<100:
    ix = list(range(m))
    np.random.shuffle (ix)      #randomly shuffle the data to improve performance
    X = X[ix,:].reshape(X.shape[0],X.shape[1])
    Y = Y[ix,:]
    for i in range(0,m,b):              #get a batch
        Xn = X[i:i+b,:].reshape(b,X.shape[1])
        #Xn = (Xn - np.mean(Xn))/np.std(Xn)
        Z1 = np.dot(Xn,W0)+b0           #1st layer (output)
        a1 = np.maximum(0,Z1)           #relu activation of hidden layer 1
        Z2 = np.dot(a1,W1)+b1
        a2 = np.maximum(0,Z2)           #relu activation of hidden layer 2
        Z3 = np.dot(a2,W2)+b2
        expo = np.exp(Z3-np.max(Z3))    #softmax numerator. subtract stabilize.
        a3 = expo/np.sum(expo,axis=1, keepdims=True)        #softmax

        Yhat = a3                       #nnet output
        d = Yhat - Y[i:i+b,:]           #delta

        g1 = (a1>0)*1                    #relu derivative of layer 1
        g2 = (a2>0)*1
        g3 = 1                          #softmax

        dEda3 = d * g3
        dEda2 = np.dot(dEda3 , W2.T) * g2
        dEda1 = np.dot(dEda2 , W1.T) * g1

        dEdW2 = np.dot(a2.T, dEda3)
        dEdb2 = np.sum(dEda3, axis=0, keepdims=True)

        dEdW1 = np.dot(a1.T,dEda2)
        dEdb1 = np.sum(dEda2, axis=0, keepdims=True)

        dEdW0 = np.dot(X[i:i+b,:].reshape(b,X.shape[1]).T, dEda1)
        dEdb0 = np.sum(dEda1, axis=0, keepdims=True)

        W0 -= alpha/m * dEdW0
        b0 -= alpha/m * dEdb0
        W1 -= alpha/m * dEdW1
        b1 -= alpha/m * dEdb1
        W2 -= alpha/m * dEdW2
        b2 -= alpha/m * dEdb2

        #forward propagate to capture error and accuracy:
        Z1 = np.dot(X,W0)+b0            #1st layer (output)
        a1 = np.maximum(0,Z1)
```

```
        Z2 = np.dot(a1,W1)+b1
        a2 = np.maximum(0,Z2)
        Z3 = np.dot(a2,W2)+b2
        expo = np.exp(Z3-np.max(Z3))                #softmax to numerator. subtract stabilize.
        Yhat = expo/np.sum(expo,axis=1, keepdims=True)

        E = np.append(E,np.sum(0.5*((Yhat-Y)**2)))
        acc = np.append(acc, np.sum(np.argmax(Yhat,axis=1)==np.argmax(Y,axis=1))/.4)
        print("accuracy: %d, batch number: %d" %(acc[-1],len(acc)), "\r", end="", flush=True)
```

Example 3: AdaGrad

- Previous setup network of [3,4,4,2]

```
...
r0=0
r1=0
r2=0
stab = 1e-7                        #stabilizing factor
...
while acc[-1]<100:
    ...
    for i in range(0,m,b):           #get a batch
        ...
        dEdW2 = np.dot(a2.T, dEda3)
        dEdb2 = np.sum(dEda3, axis=0, keepdims=True)

        dEdW1 = np.dot( a1.T,dEda2)
        dEdb1 = np.sum(dEda2, axis=0)

        dEdW0 = np.dot( X[i:i+b,:].reshape(b,X.shape[1]).T, dEda1)
        dEdb0 = np.sum(dEda1, axis=0)

        r0 += np.square(dEdW0)
        r1 += np.square(dEdW1)
        r2 += np.square(dEdW2)

        W0 += -alpha/(stab+np.sqrt(r0)) * dEdW0/m
        b0 += -alpha * dEdb0/m
        W1 += -alpha/(stab+np.sqrt(r1)) * dEdW1/m
        b1 += -alpha * dEdb1/m
        W2 += -alpha/(stab+np.sqrt(r2)) * dEdW2/m
        b2 += -alpha * dEdb2/m
        ...
```

Example 4: Modularize

- Use functions to perform: network description, activation, initialization, forward propagation, backward propagation, loss calculation, learning function
- Use dictionaries to store values

Define Initialize scheme:

```
def initialize_normal(layer_width):
    Ws = {}                 #Weights & Biases as a dictionary
    bs = {}
    layers = len(layer_width)    #Layers

    for l in range(0, layers-1):
        Ws[l] = np.random.randn(?,?)
        bs[l] = np.random.randn(1, ?)

    return Ws,bs
```

Define activation & derivatives of activation

```
def sigmoid (Input, Ws, bs):
    Z = np.dot(Input, Ws)+bs
    return 1/(1+np.exp(-Z))

def dsigmoid(A):              #derivative of sigmoid
    return ?

def relu (Input, Ws, bs):
    Z = np.dot(Input, Ws)+bs
    return ?

def drelu (A):                #deriv of relu = 0 (negative Z), or 1
    return ?

def softmax (Input, Ws, bs):
    Z = np.dot(Input, Ws)+bs
    expo = ?
    return ?
```

Define forward propagation:

```
def forwardprop(X, Ws, bs):
    #X=input,
    #layer_width = array containing width of each layer input to output
    #sigmoid activations + softmax output

    layers = len(Ws)
    layerout = {}
    layerout[0] = X

    for l in range(0, layers-1):
        layerout[l+1] = sigmoid(?,Ws[l],bs[l])      #sigmoid

    pred = softmax(?,Ws[l+1],bs[l+1])

    layerout.pop(0)     #remove X from layers

    return layerout, pred
```

Define Error & Loss:

```
def Loss(y_pred, y_true):
    return -np.sum(np.log(np.sum(?,axis=-1)))/y_true.shape[0]

def error (y_pred, y_true):
    m = y_true.shape[0]
    return ?
```

Define Back propagation:

```
def backprop(x_true, y_true, y_pred, Ws, bs, activations, alpha=0.01):
    layers = len(Ws)                               #total layers
    A    = activations
    A[0] = x_true                                  #add input to activations
    dLda = {}

    m = y_true.shape[0]


    #softmax layer:
    delta = y_pred - y_true
    g = 1
    dLda[layers] = delta*g

    #Other layers
    for l in range(layers-1,-1,-1):
        dLdW = np.dot(?, ?])
        dLdb = np.sum(?, axis=0, keepdims=True)

        Ws[l] -= alpha/m*?
        bs[l] -= alpha/m*?
        g    = dsigmoid(?)
        dLda[l] = ?

    A = A.pop(0)                                   #remove input from activations

    return Ws, bs
```

Creating a network & training (this can also be incorporated into a function!)

```
network = [3,4,5,2]      #input,hidden width1,hidden width2,output

W, b = initialize_normal(?)

itera =1000
L = np.zeros((itera))

for epoc in range(itera):

    act, pred = forwardprop(X, ?, ?)
    L[epoc] = Loss(?, ?)
    W, b = backprop(X,Y, ?, ?, ?, ?,0.2)
    err = error(?,?)*100
    print("error: %3d , epoc: %d" %(err,epoc), "\r", end="", flush=True)
```