```
# sources: Neural Network Design by Hagan, Demuth,n ..., chapters 4 and 11

import pdb
from test import dSigmoid
import numpy as np
#import random
import matplotlib.pyplot as plt

# debug config
NBUG = True

dataOR = [( 1,  1,  1,  1,  1),
          ( 1,  1, -1,  1,  1),
          ( 1, -1,  1,  1,  1),
          ( 1, -1, -1,  1,  1),
          (-1,  1,  1,  1,  1),
          (-1,  1, -1,  1,  1),
          (-1, -1,  1,  1,  1),
          (-1, -1, -1,  1, -1)]

dataXOR = [( 1,  1,  1,  1, -1),
           ( 1,  1, -1,  1,  1),
           ( 1, -1,  1,  1,  1),
           ( 1, -1, -1,  1,  1),
           (-1,  1,  1,  1,  1),
           (-1,  1, -1,  1,  1),
           (-1, -1,  1,  1,  1),
           (-1, -1, -1,  1, -1)]

class nn_2layer:
  def __init__(self, X, Y, lr, iters, prt=True, loggerEnable=True):
    self.X = np.concatenate((X, np.zeros((8,1))), axis = 1)
    self.nSamples, self.mFeatures = X.shape
    self.Y = Y
    self.Yest = np.zeros((1,8)) # network's current estimate
    self.nnDims = [5, 5, 1]
    self.lr = lr
    self.iters = iters
    self.printRate = iters / 10
    self.loss = None # training error
    # Print and data logging
    self.prt = prt
    self.batch_size = self.nSamples
    self.p = {}
    self.ch = {}
    self.grd = {}


    ''' Initialize network weights
    '''
    # first layer
    self.p['W1'] = np.random.uniform(low=-.1, high=.1, \
      size=(self.nnDims[1], self.nnDims[0])) / np.sqrt(self.nnDims[0]) # we norm
alize W vectors by dividing by the sqrt of size of the previous layer
    # second layer
    self.W2 = np.random.uniform(low=-.1, high=.1, \
      size=(self.nnDims[2], self.nnDims[1])) / np.sqrt(self.nnDims[1]) # we norm
alize W vectors by dividing by the sqrt of size of the previous layer

    if self.prt:
```

```
      print('Initialize model parameters:')
      print('W1:', self.W1)
      print('W2:', self.W2)
      print('\n\n')
    #pdb.set_trace()
    #if self.logOn:
    # init datalogger
    self.log = datalogger(loggerEnable)
    return

  def cEntropy(self,Yest): # batch normalized cross entropy loss
    loss = (1./5) * (-np.dot(self.Y, np.log(Yest).T) - np.dot(1-self.Y, np.log(1
-Yest).T))
    return loss

  def sigmoid(self, vec):
    res = 1/(1 + np.exp(-vec))
    #pdb.set_trace()
    return res

  def hardlim(self, vec):
    res = np.zeros(vec.shape)
    for i in range(len(vec)):
      if vec[i,:] > .5: res[i,:] = 1
    else: res[i,:] = 0
    return res

  def dSigmoid(self, var):
    res = self.sigmoid(var) * (1.0 - self.sigmoid(var))
    return res

  def MLE(self, Yact, Yest):
    loss = (1/2) * (Yest - Yact)**2
    return loss

  def getacc(self, groundtruth, prediction):
    correct = 0
    for i in range(len(groundtruth)):
      if groundtruth[i] == prediction[i]:
        correct += 1
    return correct / float(len(groundtruth)) * 100.0

  # forward pass is basically an estimator
  def forwardpass(self):
    #if NBUG:
    #  print('In forward pass...')
    #pdb.set_trace()
    # 1st layer
    self.Z1 = self.X.dot(self.W1.T)
    self.Y1 = self.sigmoid(self.Z1)
    # 2nd layer
    self.Z2 = self.Y1.dot(self.W2.T)
    self.Y2  = self.sigmoid(self.Z2)
    self.Yest = self.hardlim(self.Z2)
    self.loss = self.cEntropy(self.Yest)
    #pdb.set_trace()
    # calc loss

    #pdb.set_trace()
    # log data
```

```
    if self.log.on:
      self.log.Yest.append(self.Yest)
      self.log.loss.append(self.loss)
    return

  # backprop is similar to feedback in with much higher dimensions and in state
space
  def backprop(self):
    '''
      Backpropagation is done by performing partial derivative of the entire
      network (the output loss - end of forwardpass) with respect to the network
      parameters, W1, B1, W2, B2. Once output sensitivity (rate of change) is
      determined with respect to each parameter, we update them accordingly.
    '''
    #if NBUG:
    #  print('In backprop...')
    pdb.set_trace()
    # start with the output - s^2 (sensitivity of the 2nd layer)
    # get gradient loss of Yest
    dLYest = - (np.divide(self.Y, self.Yest) - np.divide(1-self.Y, 1-self.Yest))

    dLZ2 = dLYest * self.dSigmoid(self.Z2)
    dLY1 = np.dot(dLZ2, self.W2) #                 Grad error for hidden weights
    # dW2 1x5
    dLW2 = 1./self.Y1.shape[1] * np.dot(dLZ2.T, self.Y1)

    dLZ1 = dLY1 * self.dSigmoid(self.Z1)
    #dLA0 = np.dot(self.W1, dLZ1.T)
    dLW1 = 1./self.X.shape[1] * np.dot(self.X.T, dLZ1)

    # perform update
    self.W1 = self.W1 - self.lr * dLW1
    self.W2 = self.W2 - self.lr * dLW2

    # log updates and updated parameters
    #if self.log.on:
      #self.log.cost.append(cost)
    return

  '''getAcc()
    By calculating accuracy per batch, it is essentially calculating the moving
    average for accuracy which is a much better representation than overall
    average.
  '''
  def getAcc(self, Yest, Yact):
    if (len(Yact) != len(Yest)):
      print(">>>Err in (self.getAcc()): Batch output length mismatch!")
    correct = 0
    for i in range(len(Yact)):
      if Yact[i] == Yest[i]:
        correct += 1
    return correct / float(len(Yact)) * 100.0

  def BGD(self):
    for i in range(0, self.iters):
      self.forwardpass()
      self.backprop()
      if (self.prt & (i%self.printRate==0)):
        print ("Cost after iteration {}".format(i),": {}".format(self.loss))
        print('\n\n')
```

```
    return


# data logger with easy data structure handler
class datalogger:
  def __init__(self, enable):
    self.on = enable # enable flag
    self.W1 = list()
    self.B1 = list()
    self.Z1 = list()
    self.Y1 = list()
    self.W2 = list()
    self.B2 = list()
    self.Z2 = list()
    self.Yest = list()
    self.loss = list()
    self.iters = list()
    self.acc = list()
    self.batchAcc = list()
    self.cost = list()
    return

def get_data(data, Ymax,  Ymin):
  X = list()
  Y = list()
  for i in range(len(data)):
    X.append(np.asarray(data[i][:-1]))
    Y.append(np.asarray(data[i][-1]))
  X = np.asarray(X)
  Y = [Ymax if i > 0 else Ymin for i in Y]
  Y = np.asarray(Y)
  Y = np.expand_dims(Y,axis=1)
  XY = np.concatenate((X, Y), axis=1)
  print('dataset: ')
  print(XY)
  print('\n\n')
  return X, Y




if __name__ == '__main__':
  print('-->> Training and testing with OR')
  X, Y = get_data(dataOR, Ymax=1, Ymin=0) # rescale output to [0,1] since we're
using Sigmoid activation function

  nnOR = nn_2layer(X, Y, lr=.001, iters=1000)
  nnOR.BGD()

  pdb.set_trace()

  plt.plot(range(len(nnOR.log.loss)), nnOR.log.loss, label='Network Batch Loss')
  #plt.show()
  #figOR = plt.figure()
  #ax = figOR.add_subplot(111, projection='3d')


  '''
  print("\n\n")
  print('-->> Training and testing with XOR')
```

```
xorX, xorY = get_data(dataXOR, Ymax=1, Ymin=0) # rescale output to [0,1] since
we're using Sigmoid activation function
pXOR = nn_2layer(.01, 1000, True, L2, True)
pXOR.train(xorX,xorY)
plt.style.use('ggplot')
#plt.style.use('fivethirtyeight')

plt.plot(range(len(pXOR.accHist)), pXOR.accHist, label='XOR Perceptron Taining
Acc')
'''

plt.xlabel('Training iterations')
plt.ylabel('Training accuracy')
plt.title('OR vs. XOR training accuracy')
plt.legend()
#plt.grid(True)
#plt.tight_layout()
plt.show()
```