Feed-forward neural network with back-propagation

Neural networks are a supervised learning approach that enables us to model a complex and non-linear dataset. The model uses an artificial neuron called a perceptron. A perceptron takes several inputs(x1, x2,…) and produces an output. These are assigned some weights (w1,w2,…) to express the importance of the input. Output is computed as $h(x) = b + W_1x_1 + W_2x_2$. Here b is the bias unit.

Feed forward neural network has perceptron's arranged in layers. First layer taking the inputs, last layer being the output and hidden layers in between. The activation of nodes are computed using the below formula:

$a1=f(W_{11}x_1+W_{12}x_2+w_{13}x_3+b_1)$, similarly for other nodes.

Here f is the activation function. Commonly used activation function is sigmoid function which is given by the below formula:

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

Algorithm to implement feedforward neural network with backprogation:

- **Divide** input dataset to training set (80%) and testing set (20%)
- **Train** the model with 80% of data
  - **Initialise** weights vectors (w1 and w2), to some small random values.
  - Set bias vectors (b1 and b2) to zeros initially.
  - **Repeat** for each training case(epoch)
    - **Repeat** for maximum iteration(1500)
      - **Perform a Feed Forward pass**:
        $Z_i = Sum(W_{ij} x_j + b)$
        activation node = a=sigmoid(Z)
      - **Perform a Backpropagation pass:**

        Alpha(Learning rate) = 0.001
        Partial derivative: f '(z)=f(z)(1 - f(z))
        Error = y(training case label)-a3(Output node activation)

Output layer weight and bias adjustment

delta3 = f ’(z)*Error

$W_{ij}=W_{ij}$ + alpha * a_j * delta3

bias=bias+ alpha* delta3

Hidden layer weight adjustment

delta2 = f ’(z)*Sum(Weights*delta3)

$W_{ij}=W_{ij}$ + alpha * a_i * delta2

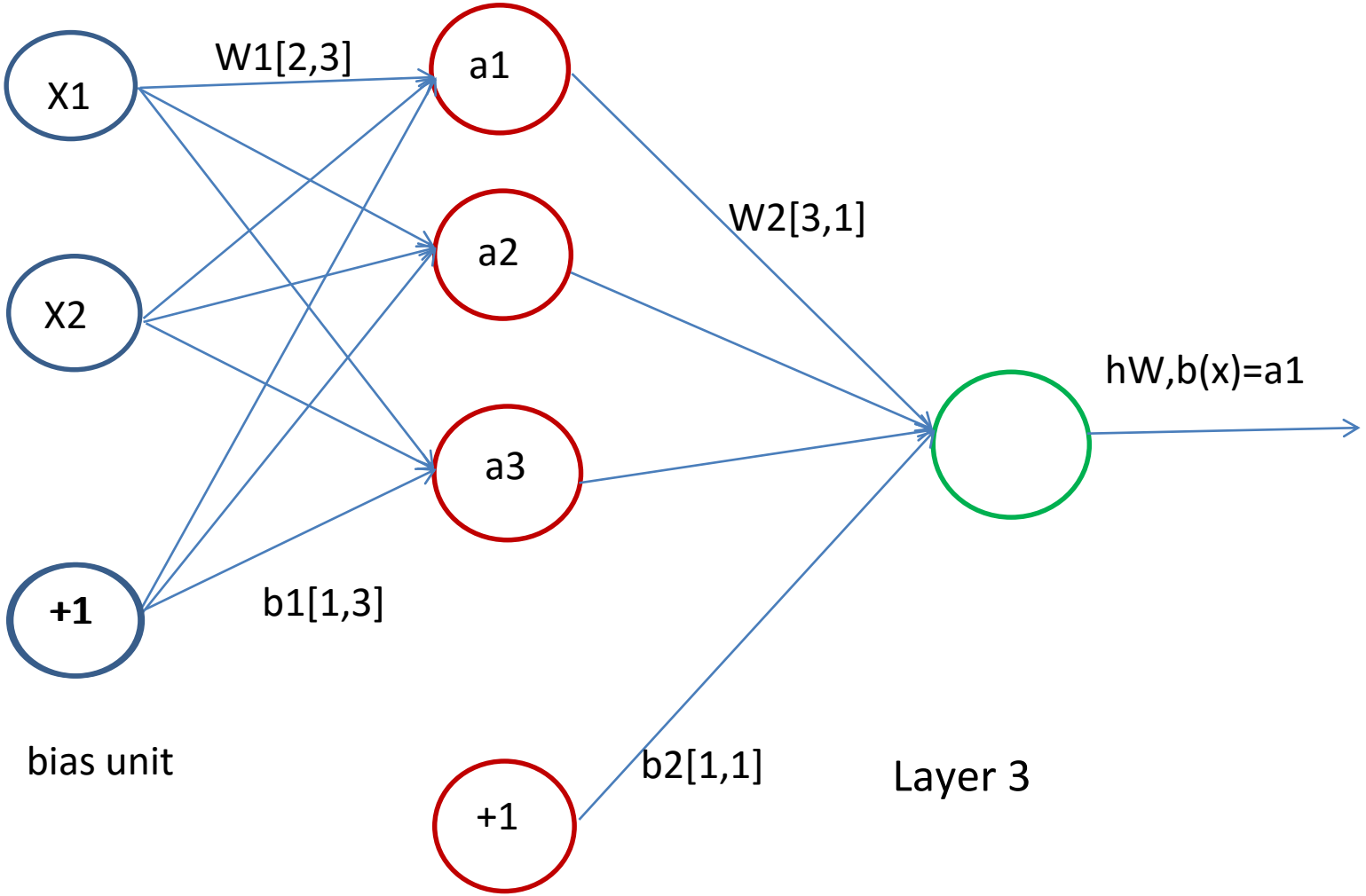
bias=bias+ alpha* delta2

- **Analyse Error:** Compare the Output from feedforward using the optimised weights and bias with class labels for each training case.
- Plot the difference in errors after each training case

- **Test** the model with 20% of data
- **Print** the results

Implementation

We have a dataset of {X0, X1} with class labels as y(0/1) for 400 training cases. I have implemented the model with one hidden layer and 3 activation nodes in the hidden layer. The model is shown as below.



I have defined the below functions to implement feedforward neural network with backpropagation.

- **feedforward()**

  This function accepts weights vectors, bias vectors and the input vector (XVec, contains x1 and x2). Weight vector W1 is a 2X3 matrix and W2 is a 3X1 matrix. Z value is calculated as the sum of weights, multiplied with the node value + the bias unit. This value is then passed to the sigmoid function to compute the activation node value.

  Output from the model is generated using the hard threshold on the output layer activation node value.

  Hard threshold rule:

  $$f(z)=o \text{ if } z<0.5$$
  $$=1 \text{ otherwise}$$

- **train_test_model()**
  This function is used to train and test the dataset. Training is done on 80% of the data and testing on the remaining 20%. Weights and bias are initially assigned some small random values. These are later optimised using the backpropagation pass. Forward and back propagation pass is iterated for 15000 times (maximum iteration) for each training case. In back propagation pass, I have calculated the

partial derivative of Z as f ’(z)=f(z)(1 - f(z)). Error is calculated as training case class label-Output node activation value. This value has to back propagate, we first calculate delta value as follows.
Delta Value for output layer= f ’(z)*Error.
Delta value for hidden layer= f ’(z)*Weights *delta from output layer

Weights and bias are now updated as below:
Updated Weight = Weight + (alpha * a *delta)
Updated bias = bias + (alpha * delta)
Updated weights and bias are used in subsequent iteration and for each training case (epochs). Upon convergence we get the most optimised weights and bias for the model. The optimised weights and biased are then used for testing.

Observations on your testing

Output:

Setting Intial values for Weights to small random value

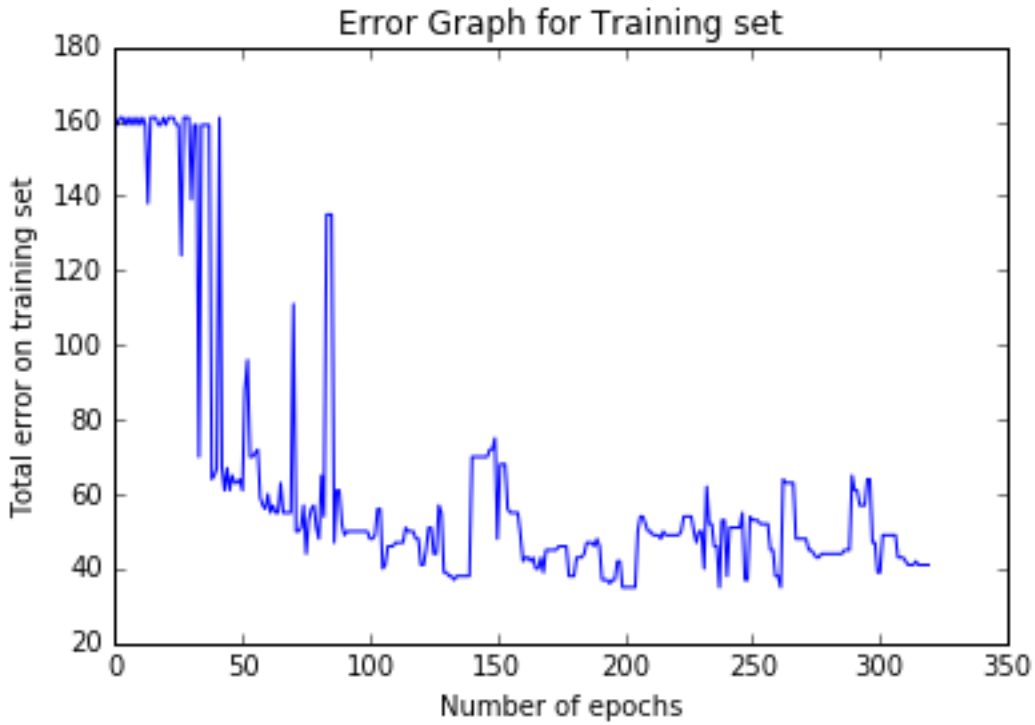
```
W1 =  [ [  5.43404942e-04   2.78369385e-04   4.24517591e-04]
        [  8.44776132e-04   4.71885619e-06   1.21569121e-04]]
W2 =  [ [ 0.00067075]
        [ 0.00082585]
        [ 0.00013671]]
```

Starting to train the model with the Training set (80% data)
Training completed

Wieghts and bias after training

```
W1 =  [[ 1.06120632  1.06174273  1.0601623 ]
       [-3.9671555  -3.97891416 -3.94678534]]
W2 =  [ [ 3.91948214]
        [ 3.9334059 ]
        [ 3.89496121]]
b1 =  [[ 0.06194001  0.06399768  0.05849626]]
b2 =  [[-4.61459467]]
```

Testing the model with the Test set (20% data)
Total number of errors for Testing set = 11


Error Graph for Training set
Total error on training set
Number of epochs
180
160
140
120
100
80
60
40
20
0
50
100
150
200
250
300
350

Inferences from Observations:

- Total number of errors from a test set of size 80 using the converged values of weights and bias w as observed as 11. This shows an error rate of 13.75% (Accuracy of 86.25%)
- Converged value of Weights and Bias are as below:

```
W1 =  [ [1.06120632  1.06174273  1.0601623 ]
        [-3.9671555  -3.97891416 -3.94678534]]
W2 =  [[ 3.91948214]
       [ 3.9334059 ]
       [ 3.89496121]]
b1 =  [[ 0.06194001  0.06399768  0.05849626]]
b2 =  [[-4.61459467]]
```

- From the 'Error Graph of Training set', we can observe that total number of errors was initially very high using random weights. 160 out of 320 training cases was predicted wrong which means model has a 50% accuracy.
- Error rate follows a flat line (50%) until training size is 50-75.
- It starts converging, but with high fluctuations until training size is 150-175.
- Convergence seems to be stabilised from 175-320.

Code

```
# Package imports
import matplotlib
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import pandas as pd
import numpy as np

# Display plots inline and change default figure size
%matplotlib inline


IMPLEMENTATION OF FEED FORWARD NEURAL NETWORK WITH BACK PROPOGATION

#Implementation of Feedforward path
#Implementation with one hidden layer and 3 nodes in hidden layer

import math

#Helper function for sigmoid
#Activiation function chosen is sigmoid (1/1+e^-z)
def sigmoid(x):
 s = np.zeros((1, len(x[0])))
 for i in range (0, len(x[0])):
   s[0,i]= 1 / (1 + math.exp(-x[0,i]))
 return s
```

```
#Helper Function to calculate the hard threshold for output layer
def hard_threshold(x):
  ot =0
  if(x<0.5):
     ot=0
  else:
     ot=1
  return ot


# Helper function for Feed forward.
#Input - W1 - Wieghts vector to the hidden layer
#Input - W2 - Wieghts vector  to the output layer
#Input - b1 - bias vector to the hidden layer
#Input - b2 - bias vector to the output layer
#Input - x - Input vector
#Output - a3 - 0 or 1 from output layer
def feedforward(W1, W2, b1, b2 , Xvec):
   out = np.zeros((len(Xvec), 1))
   # Feed Forward propagation
   for i in range (0, len(Xvec)):
      z1 = Xvec[i].dot(W1) + b1
      a2 = sigmoid(z1)
      z2 = a2.dot(W2) + b2
      a3 = sigmoid(z2)
      #Applying hard threshold
      out[i] = hard_threshold(a3)
```

```
    return out

#Algorithm Implementation of feed forward with backpropogation to adjust the weights and bias units

import matplotlib.pyplot as plt

#Model to train and test
def train_test_model(trainSet,testSet,trainSetout,testSetout):
    # Initialize the Weights and bias vectors .
    np.random.seed(100)
    #Taking small Random values for weights
    #Number of nodes in hidden layer is chosen as 3.
    print ("Setting Intial values for Weights to small random value")
    W1 = np.random.random((2, 3))/1000
    W2 = np.random.random((3, 1))/1000
    print ("W1 = ",W1)
    print ("W2 = ",W2)
    print ("\n")
    #Initializing bias as zeros
    b1 = np.zeros((1, 3))
    b2 = np.zeros((1, 1))
    ErrorData = np.zeros((320, 1))

    #Training the trainset data (80% of data)
    print ("Starting to train the model with the Training set (80% data)")
    #Iterating over the trainsSet length
    for k in range (0, len(trainSet)):
```

```
#Number of iterations per epoch is set as 1500
for i in range (0, 1500):

  ##Perform a Feed Forward pass
  z1 = trainSet[k].dot(W1) + b1
  a2 = sigmoid(z1)
  z2 = a2.dot(W2) + b2
  a3 = sigmoid(z2)

  #Taking alpha value as 0.01
  alpha = 0.01

  ##Back Propagation pass
  #Output layer weight adjustment
  f_z3 = a3*(1-a3)
  Err3 = y[k]-a3
  delta3 = f_z3*Err3
  for i in range (0, len(W2)):
    W2[i,0]= W2[i,0] +alpha*a2[0,i]*delta3

  #Output layer Bias adjustment
  b2 = b2 + alpha*delta3

  #hidden layer weight adjustment
  f_z2 = a2*(1-a2)
  delta1_2 = f_z2[0,0] * W2[0,0] * delta3
  delta2_2 = f_z2[0,1] * W2[1,0] * delta3
  delta3_2 = f_z2[0,2] * W2[2,0] * delta3
```

```
        W1[0,0] = W1[0,0] +alpha*trainSet[k,0]*delta1_2
        W1[0,1] = W1[0,1] +alpha*trainSet[k,0]*delta2_2
        W1[0,2] = W1[0,2] +alpha*trainSet[k,0]*delta3_2
        W1[1,0] = W1[1,0] +alpha*trainSet[k,1]*delta1_2
        W1[1,1] = W1[1,1] +alpha*trainSet[k,1]*delta2_2
        W1[1,2] = W1[1,2] +alpha*trainSet[k,1]*delta3_2

        #hidden layer bias adjustment
        b1[0,0] = b1[0,0]+alpha*delta1_2
        b1[0,1] = b1[0,1]+alpha*delta2_2
        b1[0,2] = b1[0,2]+alpha*delta3_2
    #Analyzing errors
    ErrorVec = feedforward(W1, W2, b1, b2 , trainSet) - np.reshape(trainSetout, (320,1))
    ErrorData[k] = np.count_nonzero(ErrorVec)
print ("Training completed")
plt.plot(ErrorData)
plt.xlabel('Number of epochs')
plt.ylabel('Total error on training set')
plt.title('Error Graph for Training set')
print ("Weights and bias after training")
print ("W1 = ",W1)
print ("W2 = ",W2)
print ("b1 = ",b1)
print ("b2 = ",b2)

print ("\n")
print ("Testing the model with the Test set (20% data)")
```

```
#Testing the model with 20% of data.
numErrors = np.count_nonzero(feedforward(W1, W2, b1, b2 , testSet) - np.reshape(testSetout, (80,1)))
print ("Total number of errors for Testing set = ", numErrors)
```

LOADING INPUT DATA FROM MOONS DATASET

```
# Reusing Load data,separate the attribute values from the class labels,
# and plot the data code from LoadDataset.ipynb provided(Sample iPython Notebook Script)
#Assignment 1 - Part 2 - Loading the input moons dataset

#Use pandas to read the CSV file as a dataframe
df = pd.read_csv("C:\Assignments\moons400.csv")

# The y values are those labelled 'Class': extract their values
y = df['Class'].values

# The x values are all other columns
del df['Class']    # drop the 'Class' column from the dataframe
X = df.as_matrix() # convert the remaining columns to a numpy array
```

GENERATING TRAIN AND TEST SET DATA

```
trainSet = X[0:320,:]
testSet =  X[320:400,:]
trainSetout = y[0:320]
testSetout = y[320:400]
```

CONSTRUCTING THE MODEL (TRAINING AND TESTING)

```
#Constructing model
train_test_model(trainSet,testSet,trainSetout,testSetout)
```

REFERENCE

http://deeplearning.stanford.edu/wiki/index.php/Backpropagation_Algorithm

http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/

http://www.rueckstiess.net/snippets/show/17a86039

https://triangleinequality.wordpress.com/2014/03/31/neural-networks-part-2/