CSL 303 : Artificial Intelligence

TUTORIAL ASSIGNMENT 10

Machine Learning-I

Date Assigned: 12th November, 2021 **Date Submitted :** 21st November, 2021

Submitted by:

Name: S. Bhavyesh

Roll Number: 191210045

Branch: Computer Science and Engineering

Semester: 5th

Submitted to: Dr. Chandra Prakash

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI
A-7, Institutional Area, near Satyawadi Raja Harish Chandra Hospital, New Delhi, Delhi 110040

PART A: Exposition Problems

1. Consider the below dataset as discussed in class with 4 attributes/features and two class (Play and not Play Tennis). Write a program to classify a new sample X using Bayesian Classifier when :

PlayTennis: training examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

• outlook = sunny, temperature = cool, humidity = high and windy = false
• X = rain, hot, high, false Compare the your results with the in-build library of Bayesian Classifier in python .

Compare the your results with the in-build library of Bayesian Classifier in python .

CODES and OUTPUTS

In-built Naive Bayes classifier

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# load csv data into dataframe
df = pd.read_csv('playTennis.csv', sep = ',')
encoder = LabelEncoder()

# encode the labels in the dataframe
for i in df.columns[1:]:
 df[i] = encoder.fit_transform(df[i])

# remove unwated columns and create the input and the output sets
del df['day']
Y = df['play'].to_numpy()
del df['play']
X = df.to_numpy()

# train the model and do prediction
model = GaussianNB()
model.fit(X,Y)
result = model.predict([[2,0,0,1],[1,1,0,1]])

label = {1:"Yes", 0:"No"} # dictionary to map integer to output label

for i in range(len(result)):
  print("Output label for test case ",i+1," is", label[result[i]] )
```

```
Output label for test case  1  is No
Output label for test case  2  is Yes
```

__Custom Naive Bayes classifier__

```python
[32] import pandas as pd
     import numpy as np

     df = pd.read_csv('playTennis.csv', sep = ',') # load csv data into dataframe
```

```python
[33] col = df.columns[1:] # extract column heads
```

```python
[34] vals = []
     for i in col:
       vals.append(df[i].unique()) # retrieve the unique values of each column
                                   # these are the possible target values for each parameter
```

```python
[35] prob_tables = [] # stores probability tables (list of lists)

     for i in range(len(col[-1])):
       priors = []
       for j in vals[i]:
         row = []
         for k in vals[-1]:
           row.append(sum((df[col[i]] == j) & (df[col[-1]] == k)) / sum(df[col[-1]] == k))
           # example: P(outlook = Sunny|Output = Yes)
           # = P(outlook = Sunny & Output = Yes) / P(Output = Yes)

         priors.append(row)
       prob_tables.append(priors)
```

✓ 0s completed at 15:58

```python
test1 = ["Sunny","Cool","High","Weak"] # test vector 1
test2 = ["Rain","Hot","High","Weak"] # test vector 2

prob_yes = sum(df['play'] == "Yes")/len(df) # P(Yes)
prob_no = sum(df['play'] == "No")/len(df)   # P(No)

play_no1, play_yes1 = 1, 1 # stores the chance of each output case (vector 1)
play_no2, play_yes2 = 1, 1 # stores the chance of each output case (vector 2)

for i in range(len(col[-1])): # repeatedly multipy respective conditional priors by picking out
                              # probabilities from prob_tables (3D list)
  # Vector 1
  play_no1 *= prob_tables[i][np.where(vals[i] == test1[i])[0][0]][0]
  play_yes1 *= prob_tables[i][np.where(vals[i] == test1[i])[0][0]][1]

  # Vector 2
  play_no2 *= prob_tables[i][np.where(vals[i] == test2[i])[0][0]][0]
  play_yes2 *= prob_tables[i][np.where(vals[i] == test2[i])[0][0]][1]

# finally multiply the probability of output values
play_yes1 *= prob_yes
play_no1 *= prob_no
play_yes2 *= prob_yes
play_no2 *= prob_no
```

```python
print("Chance of play = Yes, given test1: ", play_yes1)
print("Chance of play = No, given test1: ", play_no1)

if play_yes1 > play_no1: print("Output label for ", test1, "is Yes",)
else: print("Output label for ", test1, "is No\n\n")

print("Chance of play = Yes, given test2: ", play_yes2)
print("Chance of play = No, given test2: ", play_no2)

if play_yes2 > play_no2: print("Output label for ", test2, "is Yes",)
else: print("Output label for ", test2, "is No")
```

```
Chance of play = Yes, given test1:  0.010582010582010581
Chance of play = No, given test1:  0.013714285714285715
Output label for  ['Sunny', 'Cool', 'High', 'Weak'] is No


Chance of play = Yes, given test2:  0.010582010582010581
Chance of play = No, given test2:  0.018285714285714287
Output label for  ['Rain', 'Hot', 'High', 'Weak'] is No
```

OBSERVATION/COMMENTS:

There is a difference between the results for test case 2 in the custom Naive Bayes classifier and In-built Naive Bayes classifier. This could be because in the in-built Naive Bayes classifier, the likelihood is assumed to be Gaussian, while in the custom case, we are simply enumerating the occurences on a conditional basis.

2. Write a program for the decision tree for the below scenario with

a) Gini as impurity index
b) Information Gain as impurity index
Write a program to classify a new sample X using decision tree with above 2 cases, when :
• outlook = sunny, temperature = cool, humidity = high and windy = false
• X = rain, hot, high, false

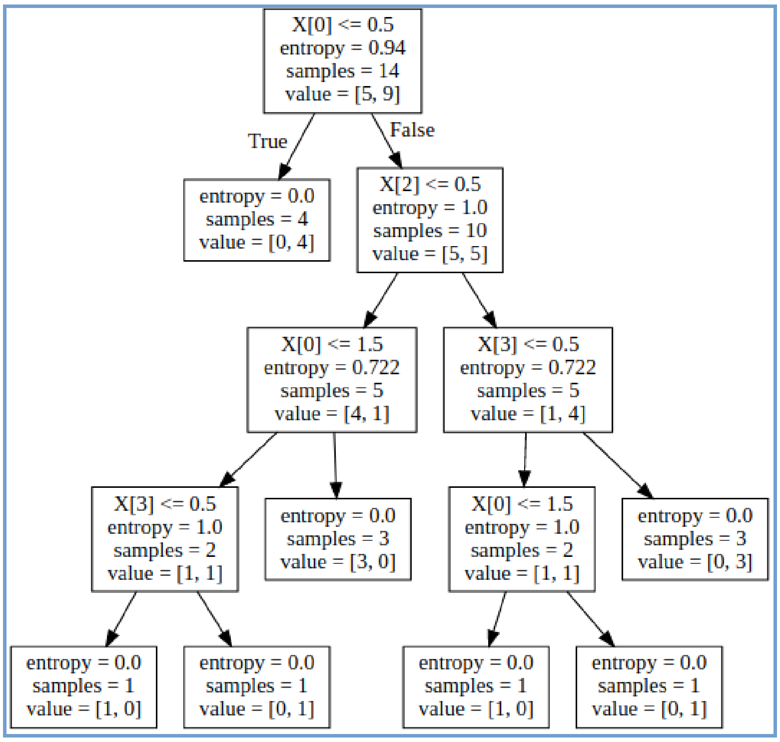
a) Compare the your results with the in-build library of decision tree in python .
b) Compare the results with the Bayesian Classifier.

CODES and OUTPUTS

In-built Decision Trees

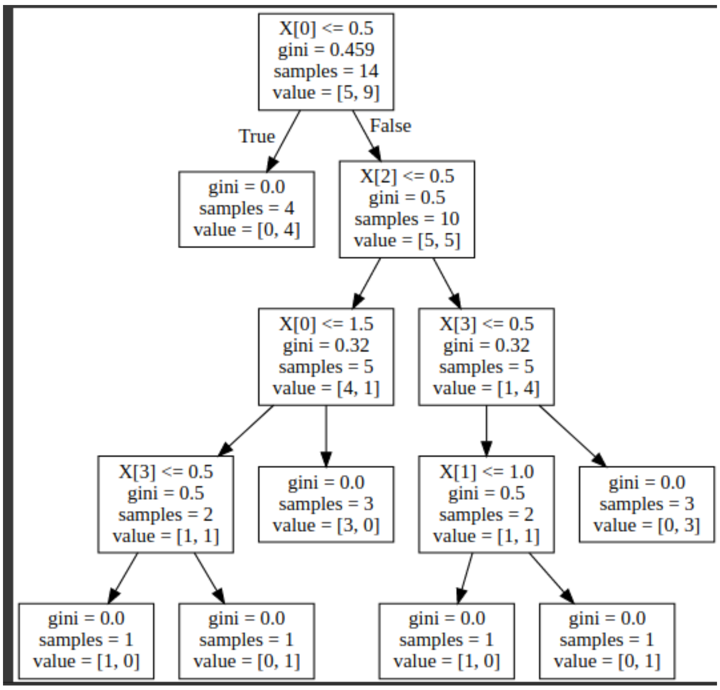
```
####################################################
#   PART 1: USE IN-BUILT LIBRARIES AND FUNCTIONS   #
####################################################

import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier, export_graphviz
import graphviz

# load csv data into dataframe
df = pd.read_csv('playTennis.csv', sep = ',')

# define encoder
encoder = LabelEncoder()

# convert string data to integer for ease of handling
for i in df.columns[1:]:
 df[i] = encoder.fit_transform(df[i])

# remove unwanted columns and split
# dataframe into input features and output
# features (labels)
del df['day']
Y = df['play'].to_numpy()
del df['play']
X = df.to_numpy()
```

```
# first tree, use entropy as the impurity parameter
tree1 = DecisionTreeClassifier(criterion = 'entropy')
tree1.fit(X,Y)

# diagrammatically show the tree
dot_data1 = export_graphviz(tree1, out_file=None)
graph1 = graphviz.Source(dot_data1)
graph1
```




```
[4]  # predict on the given test set
     tree1.predict([[2,0,0,1],[1,1,0,1]])

     # output 0 -> No, output 1 -> Yes

     array([0, 1])
```

```
# second tree, use GINI index as the impurity parameter
tree2 = DecisionTreeClassifier(criterion = 'gini')
tree2.fit(X,Y)

# diagrammatically show the tree
dot_data2 = export_graphviz(tree2, out_file=None)
graph2 = graphviz.Source(dot_data2)
graph2
```



```
# predict on the given test set
tree2.predict([[2,0,0,1],[1,1,0,1]])

# output 0 -> No, output 1 -> Yes
```

```
array([0, 1])
```

Custom Decision Trees

```
#####################################################
#   PART 2: MAKE DECISION TREES FROM SCRATCH       #
#####################################################
```

```
[14] import pandas as pd
     import math

     X = pd.read_csv("playTennis.csv")  # read file, convert to a dataframe
     del X['day']  # drop unnecessary column
```

```
[15] def entropy(train_set):
         # below segment checks if entropy calculation is possible
         # checking purity of the set
         num_yes = sum(train_set['play'] == "Yes")
         num_no = len(train_set) - num_yes

         if num_yes == 0 or num_no == 0: return 0.0
         else: # apply formula of entropy
             p = num_yes / (num_yes + num_no)
             return -p*math.log2(p) - (1 - p)*math.log2(1-p)

     def info_gain(train_set, feature): # information gain criterion

         values = train_set[feature].unique()
         gain = entropy(train_set) # entropy of parent node
         # summation of weighted entropy of each value of the feature
         for value in values:
             temp = train_set[train_set[feature] == value]
             gain -= entropy(temp)*len(temp)/len(train_set)
         return gain
```

✓ 0s comple

```
def gini(train_set):
    # below segment checks if entropy calculation is possible
    # checking purity of the set
    num_yes = sum(train_set['play'] == "Yes")
    num_no = len(train_set) - num_yes

    if num_yes == 0 or num_no == 0: return 0.0
    else: # apply formula of GINI
        p = num_yes / (num_yes + num_no)
        return 1 - p**2 - (1-p)**2

def gini_split(train_set, feature): # GINI split criterion

    values = train_set[feature].unique()
    gain = gini(train_set) # GINI index of parent node
    # summation of weighted GINI of each value of the feature
    for value in values:
        temp = train_set[train_set[feature] == value]
        gain += gini(temp)*len(temp)/len(train_set)
    return gain
```

```
[17] # class to generate the decision tree
     # relevant variables are declared to store critical data
     class Node:
         def __init__(self,children,value,terminal,prediction):
             self.children = children
             self.value = value
             self.terminal = terminal
             self.prediction = prediction
```

```python
# recursive function to generate tree from the training set
def build_tree(train_set, feature_set, criterion):

    root = Node([],None,False,None)
    temp_gain = 0
    temp_feat = None
    # in info-gain criterion, we choose the feature
    # with highest purity value among all
    if criterion == "info_gain":
        temp_gain = -10000
        for feature in feature_set:
            gain = eval(criterion+"(train_set, feature)") # call info-gain method
            if gain > temp_gain:
                temp_gain = gain
                temp_feat = feature

    # in GINI index criterion, we choose the feature
    # with lowest weighted-GINI value among all
    elif criterion == "gini_split":
        temp_gain = 10000
        for feature in feature_set:
            gain = eval(criterion+"(train_set, feature)") # call weighted-GINI method
            if gain < temp_gain:
                temp_gain = gain
                temp_feat = feature

    # update root node and retrieve values
    # corresponding to the chosen feature
    root.value = temp_feat
    values = train_set[temp_feat].unique()
```

```python
    for value in values:

        temp = train_set[train_set[temp_feat] == value]
        # if we have reached the case of a terminal node,
        # declare the node as terminal
        if (criterion == "info_gain" and entropy(temp) == 0.0) \
        or (criterion == "gini_split" and gini(temp) == 0.0):
            child = Node([],value,True,temp['play'].unique()[0])
            root.children.append(child)

        else:
            # child node will actually have the feature value
            # the grandchild will actually propagate the tree forward
            child = Node([],value,False,None)

            # modify feature to prevent repeated use of same feature
            new_feature_set = feature_set.copy()
            new_feature_set.remove(temp_feat)

            grandchild = build_tree(temp, new_feature_set, criterion)
            child.children.append(grandchild)
            root.children.append(child)

    return root
```

```python
# recursive method to draw prediction from given test data
def predict(root, test):

    # lowest depth of tree, decision is to be made
    if root.terminal:
        return root.prediction
    # intermediate node that only stores a feature value(child node)
    elif root.value not in features:
        return predict(root.children[0],test)
    # node that actually connects to further nodes(grandchild node)
    else:
        index = features.index(root.value)
        for child in root.children:
            if child.value == test[index]:
                return predict(child,test)
```

```python
# execution point
features=[feat for feat in X]
features.remove('play')
root1 = build_tree(X, features, "info_gain")
root2 = build_tree(X, features, "gini_split")

# test cases
test1 = ["Sunny","Cool","High","Weak"]
test2 = ["Rain","Hot","High","Weak"]

# prediction in the case of information gain criterion
print("When info gain is criterion...")
print("Output Label for ",test1," is ", predict(root1,test1))
print("Output Label for ",test2," is ", predict(root1,test2))

print("\n")

# prediction in the case of GINI index criterion
print("When GINI split is criterion...")
print("Output Label for ",test1," is ", predict(root2,test1))
print("Output Label for ",test2," is ", predict(root2,test2))
```

```
When info gain is criterion...
Output Label for  ['Sunny', 'Cool', 'High', 'Weak']  is  No
Output Label for  ['Rain', 'Hot', 'High', 'Weak']  is  Yes


When GINI split is criterion...
Output Label for  ['Sunny', 'Cool', 'High', 'Weak']  is  No
Output Label for  ['Rain', 'Hot', 'High', 'Weak']  is  Yes
```

OBSERVATION/COMMENTS:

1. The outputs generated by the inbuilt Bayesian classifier, and the two decision trees(info-gain and GINI) yield the same output. However, the custom Naive Bayes classifier slightly deviates from the rest in the second test case.

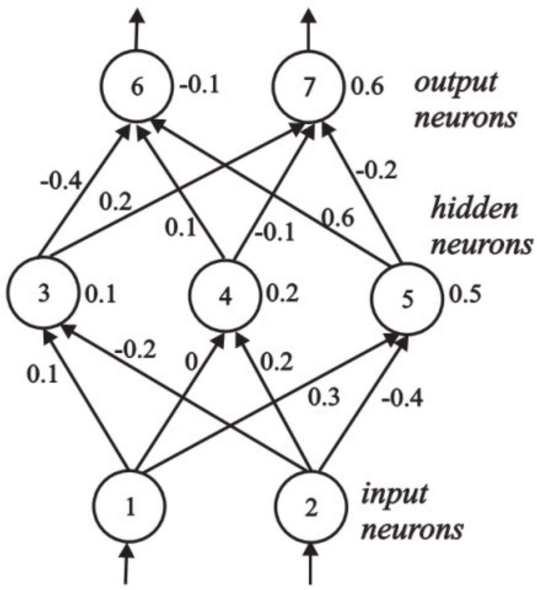
2. Although all 4 decision trees give the same output, the algorithms that both use are different. The inbuilt decision trees use a kind of binary spiltting algorithm and utilize all the features, while the custom decision trees use ID3 algorithm and ignore the *temperature* feature.

3. Consider the same example discussed in the class (2 classes, 2 dim. input data) :
The training set is :
ex.1: 0.6 0.1 — class 1 (banana)
ex.2: 0.2 0.3 — class 2 (orange)
Mention the following in lab file :
a) Network architecture **2-3-2 architecture**
b) How many inputs? **2 inputs**
c) How many hidden neurons? **3 hidden neurons**
d) How many output neurons? **2 output neurons**
e) What encoding of the outputs? **10 for class 1, 01 for class 2**
f) Initial weights and learning rate

**Hidden layer weights: [[0.1, 0.0, 0.3],
[-0.2, 0.2, -0.4]]**

**Output layer weights: [[-0.4, 0.2],
[0.1, -0.1],
[0.6, -0.2]]**
Learning rate = 0.1

Initial weights are given in the neural network below:



Write a code for Back Propagation Neural Network for the following Network architecture . Let's learning rate (η) = 0.1 and the weights are set as in the figure below. Show what will be the final weight after 1st,2nd and 50 iteration.

classmate
Date
Page

$*$ → element wise multiply

eg $(0.1 \ \ 0.0) * (0.2 \ \ 0.3) = (0.02 \ \ 0)$

Let $W_h = \begin{bmatrix} 0.1 & 0.0 & 0.3 \\ -0.2 & 0.2 & -0.4 \end{bmatrix}$, $W_o = \begin{bmatrix} -0.4 & 0.2 \\ 0.1 & -0.1 \\ 0.6 & -0.2 \end{bmatrix}$

$B_h = [0.1 \ \ 0.2 \ \ 0.5]$, $B_o = [-0.1 \ \ 0.6]$

Iteration 1 example → $(0.6 \ \ 0.1)$

$net_h = X \cdot W_h + B_h$

$= (0.6 \ \ 0.1) \begin{pmatrix} 0.1 & 0.0 & 0.3 \\ -0.2 & 0.2 & -0.4 \end{pmatrix} + (0.1 \ \ 0.2 \ \ 0.5)$

$= (\underset{net_3}{0.14} \ \ \underset{net_4}{0.22} \ \ \underset{net_5}{0.64})$

$output_h = sigmoid\,(net_h)$

$= (sig\,(0.14) \ \ sig\,(0.22) \ \ sig\,(0.64))$

$= (\underset{out_3}{0.53} \ \ \underset{out_4}{0.55} \ \ \underset{out_5}{0.65})$

$net_o = output_h \cdot W_o + B_o$

$= (0.53 \ \ 0.55 \ \ 0.65) \begin{pmatrix} -0.4 & 0.2 \\ 0.1 & -0.1 \\ 0.6 & -0.2 \end{pmatrix} + (-0.1 \ \ 0.6)$

$= (\underset{net_6}{0.13} \ \ \underset{net_7}{0.53})$

$output_o = sigmoid\,(net_o) = (\underset{out_6}{0.53} \ \ \underset{out_7}{0.63})$

$\delta_o = (Y - output) * output * (1 - output)$

$= (1-0.53 \ \ 0-0.63) * (0.53 \ \ 0.13) * (1-0.53 \ \ 1-0.63)$

$= (\underset{\delta_6}{0.12} \ \ \underset{\delta_7}{-0.15})$

$$\Delta W_o = \eta \cdot output_h^T \cdot \delta_o$$

$$= 0.1 \times \begin{pmatrix} 0.53 \\ 0.55 \\ 0.65 \end{pmatrix} \times \begin{pmatrix} 0.12 & -0.15 \end{pmatrix}$$

$$= \begin{pmatrix} 0.006 & -0.008 \\ 0.007 & -0.008 \\ 0.008 & -0.010 \end{pmatrix}$$

$$\Delta B_o = \eta \cdot \delta_o$$
$$= 0.1 \times \begin{pmatrix} 0.12 & -0.15 \end{pmatrix}$$
$$= \begin{pmatrix} 0.012 & -0.015 \end{pmatrix}$$

$$\delta_h = output_h * (1 - output_h) * \left(W_o \cdot \delta_o^T \right)$$
$$= \begin{pmatrix} 0.53 & 0.55 & 0.65 \end{pmatrix} * \begin{pmatrix} 0.47 & 0.45 & 0.35 \end{pmatrix} * \left(\begin{pmatrix} -0.4 & 0.2 \\ 0.1 & -0.1 \\ 0.6 & -0.2 \end{pmatrix} \cdot \begin{pmatrix} 0.12 \\ 0.15 \end{pmatrix} \right)$$
$$= \begin{pmatrix} -0.019 & 0.007 & 0.023 \end{pmatrix}$$

$$\Delta W_h = \eta \cdot X^T \cdot \delta_h$$
$$= 0.1 \times \begin{pmatrix} 0.6 \\ 0.1 \end{pmatrix} \times \begin{pmatrix} -0.019 & 0.007 & 0.023 \end{pmatrix}$$
$$= \begin{pmatrix} -0.001 & 0.0004 & 0.001 \\ -0.0002 & 0.00007 & 0.0002 \end{pmatrix}$$

$$\Delta B_h = \eta \cdot \delta^h = 0.1 \begin{pmatrix} -0.019 & 0.007 & 0.023 \end{pmatrix}$$
$$= \begin{pmatrix} -0.0019 & 0.0007 & 0.0023 \end{pmatrix}$$

$$W_o^{new} = W_o + \Delta W_o = \begin{pmatrix} -0.394 & 0.192 \\ 0.107 & -0.108 \\ 0.608 & -0.21 \end{pmatrix}$$

$$B_o^{new} = B_o + \Delta B_o = \begin{pmatrix} -0.088 & 0.565 \end{pmatrix}$$

$$W_h^{new} = W_h + \Delta W_h = \begin{pmatrix} 0.099 & 0.0004 & 0.301 \\ -0.2002 & 0.2007 & -0.3998 \end{pmatrix}$$

$$B_h^{new} = B_h + \Delta B_h = \begin{pmatrix} 0.0981 & 0.2007 & 0.5023 \end{pmatrix}$$

Iteration 2 example 2 (0.2 0.3)

$$net_h = X \cdot W_h + B_h$$

$$= (0.2 \quad 0.3) \begin{pmatrix} 0.099 & 0.10004 & 0.8301 \\ -0.2002 & 0.2007 & -0.3998 \end{pmatrix}$$

$$+ (0.0981 \quad 0.2007 \quad 0.5023)$$

$$= (0.057 \quad 0.26 \quad 0.44)$$

$$output_h = sigmoid((0.057 \quad 0.26 \quad 0.44))$$

$$= (0.51 \quad 0.57 \quad 0.60)$$

$$net_o = output_h \cdot W_o + B_o$$

$$= (0.51 \quad 0.57 \quad 0.60) \begin{pmatrix} -0.394 & 0.192 \\ 0.107 & -0.108 \\ 0.608 & -0.021 \end{pmatrix} + (-0.088 \quad 0.585)$$

$$= (0.14 \quad 0.49)$$

$$output_o = sigmoid((0.14 \quad 0.49)) = (0.53 \quad 0.62)$$

$$\delta_o = (Y - output_o) * output * (1 - output)$$

$$= (0 - 0.53 \quad 1 - 0.62) * (0.53 \quad 0.62) * (0.47 \quad 0.38)$$

$$= (-0.133 \quad 0.089)$$

$$\Delta W_o = \eta \cdot output_h^T \cdot \delta_o$$

$$\cancel{= 0.1 \begin{pmatrix} -0.133 \\ 0.089 \end{pmatrix}}$$

$$= 0.1 \cdot \begin{pmatrix} 0.51 \\ 0.57 \\ 0.60 \end{pmatrix} (-0.133 \quad 0.089)$$

$$= \begin{pmatrix} -0.006 & 0.004 \\ -0.007 & 0.005 \\ -0.008 & 0.005 \end{pmatrix}$$

$$\Delta B_o = \eta \cdot \delta_o$$

$$= 0.1 \times (-0.133 \quad 0.089)$$

$$= (-0.0133 \quad 0.0089)$$

$$\delta_h = \text{output}_h * (1 - \text{output}_h) * \left(W_o \cdot \delta_o^T \right)$$

$$= \begin{pmatrix} 0.51 & 0.57 & 0.60 \end{pmatrix} * \begin{pmatrix} 0.49 & 0.43 & 0.40 \end{pmatrix} * \left(\begin{pmatrix} -0.394 & 0.192 \\ 0.107 & -0.108 \\ 0.608 & -0.021 \end{pmatrix} \cdot \begin{pmatrix} -0.133 \\ 0.084 \end{pmatrix} \right)$$

$$= \begin{pmatrix} 0.017 & -0.005 & -0.024 \end{pmatrix}$$

$$\Delta W_h = \eta \cdot x^T \cdot \delta_h$$

$$= 0.1 \cdot \begin{pmatrix} 0.2 \\ 0.3 \end{pmatrix} \cancel{\begin{pmatrix} -0.013 & 0.005 \end{pmatrix}} \begin{pmatrix} 0.017 & -0.005 & -0.024 \end{pmatrix}$$

$$= \begin{pmatrix} 0.0003 & -0.0001 & -0.0005 \\ 0.0005 & -0.0002 & -0.0007 \end{pmatrix}$$

$$\Delta B_h = \eta \cdot \delta_h = 0.1 \begin{pmatrix} 0.017 & -0.005 & -0.024 \end{pmatrix}$$

$$= \begin{pmatrix} 0.0017 & -0.0005 & -0.0024 \end{pmatrix}$$

$$W_o = W_o + \Delta W_o = \begin{pmatrix} -0.4007 & 0.196 \\ 0.098 & -0.103 \\ 0.599 & -0.204 \end{pmatrix}$$

$$B_o = B_o + \Delta B_o = \begin{pmatrix} -0.101 & 0.594 \end{pmatrix}$$

$$W_h = W_h + \Delta W_h = \begin{pmatrix} 0.099 & 0.00027 & 0.301 \\ -0.199 & 0.199 & -0.4005 \end{pmatrix}$$

$$B_h = B_h + \Delta B_h = \begin{pmatrix} 0.0998 & 0.2006 & 0.4998 \end{pmatrix}$$

CODE and OUTPUT

```python
import numpy as np

def sigmoid (x): # activation function
    return 1/(1 + np.exp(-x))

X = np.array([[0.6, 0.1], [0.2, 0.3]]) # test data
Y = np.array([[1.0, 0.0], [0.0, 1.0]]) # output labels

epoch = 50 # number of epochs for which backprop is to be run
lr = 0.1   # learning rate

W_h = np.array([[0.1, 0.0, 0.3], [-0.2, 0.2, -0.4]])    # weights in hidden layer
B_h = np.array([0.1, 0.2, 0.5])                          # bias in hidden layer
W_o = np.array([[-0.4, 0.2], [0.1, -0.1], [0.6, -0.2]]) # weights in output layer
B_o = np.array([-0.1, 0.6])                              # bias in output layer
```

```python
for e in range(epoch):

    for i in range(X.shape[0]):

        h_temp = np.dot(X[i],W_h) + B_h  # forward pass from input -> hidden
        h_act = sigmoid(h_temp)          # activate hidden-layer neurons

        o_temp = np.dot(h_act,W_o) + B_o # forward pass from hidden -> output
        output = sigmoid(o_temp)         # activate output-layer neurons

        del_out = (Y[i] - output) * output * (1 - output)         # error in output
        dW_o = lr * np.array([h_act]).T.dot(np.array([del_out]))  # find delta(Weight_output)
        dB_o = lr * del_out * 1                                   # find delta(Bias_output)

        del_hidden = h_act * (1 - h_act) * W_o.dot(del_out.T)     # error in internal activation
        dW_h = lr * np.array([X[i]]).T.dot(np.array([del_hidden])) # find delta(Weight_hidden)
        dB_h = lr * del_hidden * 1                                # find delta(Bias_hidden)

        # update the weights and biases accordingly
        W_o += dW_o
        B_o += dB_o
        W_h += dW_h
        B_h += dB_h

    if e in [0,1,49]: # print parameters for 1st, 2nd and 50th iterations
        print("Iteration: ", e+1,"\n")
        print("Weights in hidden layer: \n" , str(W_h),"\n")
        print("Biases in hidden layer: \n" , str(B_h),"\n")
        print("Weights in output layer: \n" ,str(W_o),"\n")
        print("Biases in output layer: \n" , str(B_o),"\n\n\n")
```

```
Iteration:  1

Weights in hidden layer:
 [[ 9.92162570e-02  2.72431456e-04  3.00868637e-01]
 [-1.99667657e-01  1.99889457e-01 -4.00487120e-01]]

Biases in hidden layer:
 [0.09985118 0.20006418 0.499868  ]

Weights in output layer:
 [[-0.40063383  0.19673691]
 [ 0.09892641 -0.10310539]
 [ 0.59950097 -0.20417922]]

Biases in output layer:
 [-0.1016944   0.59424127]
```

```
Iteration:  2

Weights in hidden layer:
 [[ 0.09843748  0.00054864  0.30174373]
 [-0.19933512  0.19977811 -0.40097576]]

Biases in hidden layer:
 [0.09970904 0.20013109 0.49974026]

Weights in output layer:
 [[-0.40125028  0.19353339]
 [ 0.09787272 -0.10614852]
 [ 0.59902536 -0.20829079]]

Biases in output layer:
 [-0.10335376  0.58859417]
```

```
Iteration:  50

Weights in hidden layer:
 [[ 0.06519173  0.01597844  0.34831635]
 [-0.18368757  0.19330999 -0.42661493]]

Biases in hidden layer:
 [0.09727073 0.20324803 0.49385811]

Weights in output layer:
 [[-0.41708125  0.09743366]
 [ 0.06374033 -0.19230297]
 [ 0.59581789 -0.3401919 ]]

Biases in output layer:
 [-0.15446236  0.42535047]
```

OBSERVATION/COMMENTS:

1. The updation of weights and biases is not done until all the layers have been backpropagated.

2. Note that derivative(sigmoid(x)) = sigmoid(x) * (1 – sigmoid(x)). The simple nature of the derivative could be a reason why the sigmoid function is the most popular actiavtion function in deep learning.