**CS6301 MACHINE LEARNING LAB WEEK – 4 MLP**

**SRIHARI. S – 2018103601**

**Date**: 15-03-2021 Monday

**Aim**: To experiment on Multilayer Perceptron by varying different hyperparameters and observing which results in better accuracy using two labelled datasets from UCI repository.

**Dataset 1**: **Acute Inflammations Dataset**

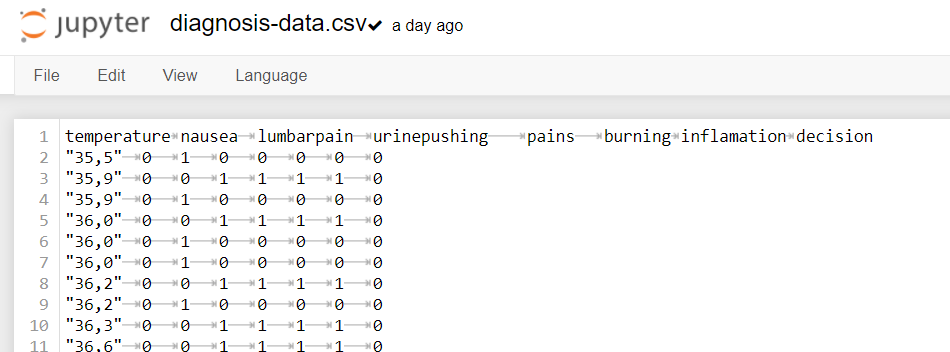
**Url**: <https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations>

**Description**:

The main idea of this data set is to prepare the algorithm of the expert system, which will perform the presumptive diagnosis of two diseases of urinary system. It will be the example of diagnosing of the acute inflammations of urinary bladder and acute nephritises. Acute inflammation of urinary bladder is characterised by sudden occurrence of pains in the abdomen region and the urination in form of constant urine pushing, micturition pains and sometimes lack of urine keeping. At proper treatment, symptoms decay usually within several days. However, there is inclination to returns. At persons with acute inflammation of urinary bladder, we should expect that the illness will turn into protracted form. It begins fever and is accompanied by shivers and one- or both-side lumbar pains, which are sometimes very strong. Symptoms of acute inflammation of urinary bladder appear very often. Quite not infrequently there are nausea and vomiting and spread pains of whole abdomen. The data was created by a medical expert as a data set to test the expert system, which will perform the presumptive diagnosis of two diseases of urinary system. The basis for rules detection was Rough Sets Theory. Each instance represents a potential patient. The data is in an ASCII file. Attributes are separated by TAB.

**Input**: The following 6 attributes

* Occurrence of nausea { 0, 1 },
* Lumbar pain { 0, 1},
* Urine pushing { 0, 1},
* Micturition pains { 0, 1},
* Burning of urethra { 0, 1},
* Inflammation of urinary bladder { 0, 1},



**Output**: decision: Nephritis of renal origin { 0, 1}

**TABULATION**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dataset | No of neurons in the hidden layers | | Activation Function | Learning Rate (eta) | | Epochs | | Accuracy (%) | | Inference | |
| **Acute Inflammations** | 6 | | Sigmoid | 0.1 | | 100 | | 52.09 | | Initially accuracy is 52.09% | |
| 4 | | Sigmoid | 0.1 | | 100 | | 43.05 | | When the number of hidden layer nodes is decreased the accuracy drops | |
| 8 | | Sigmoid | 0.0025 | | 50 | | 53.8 | | When no of hidden layer neurons is increased and learning rate and epochs are reduced **Accuracy is increased** | |
| 8 | Softmax | | | 0.1 | | 50 | | 57.08 | | When Softmax activation is used accuracy is maximised. |

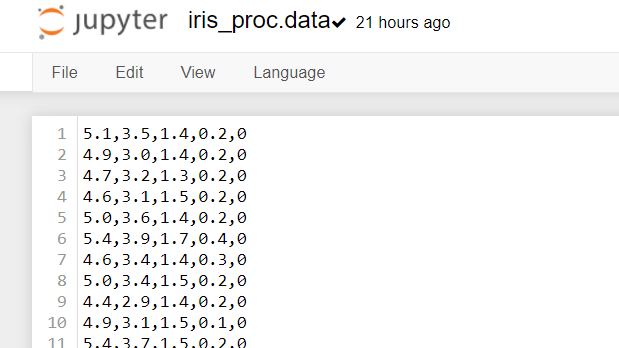
**Dataset 2**: **Iris Dataset**

**Url**: <https://archive.ics.uci.edu/ml/datasets/iris>

**Description**: The **Iris Dataset** contains four features (length and width of sepals and petals) of 50 samples of three species of **Iris** (**Iris** setosa, **Iris** virginica and **Iris** versicolor).

**Input**: The following 4 attributes

* sepal length in cm,
* sepal width in cm,
* petal length in cm,
* petal width in cm,



**Output**: decision: Multiclass classification among 3 classes of flowers: Iris Setosa, Iris Versicolour, Iris Virginica.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dataset | No of neurons in the hidden layers | | Activation Function | Learning Rate (eta) | | Epochs | | Accuracy (%) | | Inference | |
| **Iris** | 6 | | Sigmoid | 0.1 | | 100 | | 97.29 | | Initially accuracy is 97.29% | |
| 4 | | Sigmoid | 0.4 | | 100 | | 40.95 | | When the number of hidden layer nodes is decreased and the learning rate is increased the accuracy drops | |
| 8 | | Softmax | 0.025 | | 5000 | | 91.8 | | When no of hidden layer neurons and epochs are increased along with softmax activation **Accuracy is improves** | |
| 8 | Softmax | | | 0.1 | | 5000 | | 94.54 | | When learning rate is increased along with softmax activation accuracy is maximised. |

**VIVA QUESTIONS:**

1. **What is the effect of the hyperparameter eta (learning rate) on the model?**

While experimenting the effect of learning rate, it was observed that neither a learning rate which is high nor a lower one produced an impressive result. The model was trained using three different learning rates 0.1, 0.4 and 0.025. It is observed that the accuracy was at the maximum when eta = 0.1. While it was considerably lower at 0.4 and 0.025. Hence setting a learning rate between 0.1 and 0.4 is appreciated.

1. **What is the effect of increasing the number of hidden layer nodes?**

With respect to the two datasets taken, it was duly observed that as we increase the number of hidden layer nodes the accuracy increases. The accuracy reached the maximum of 57% when no. of hidden layer nodes is 12. On further increase the accuracy is observed to be stagnant at 58%

Hence concluding that more the number of hidden layer neurons the better the accuracy (in this case). But after some point it tends to saturate at the maximum value.

1. **Which activation function worked better on the model?**

The model was trained on 3 different activation functions – Sigmoid, SoftMax and Linear. For the given datasets, its observed that the Sigmoid activation function produces more accurate results than SoftMax or linear activations.

1. **Which approach is better – Increasing the number of hidden layers or the number of nodes in a hidden layer?**

In a Multilayer perceptron, increasing the number of hidden layers seem to be the wiser choice. With backpropagation the error is minimized if the no of hidden layers is more. The minimization of error isn’t found to be of a similar extent if we just increase the number of nodes in the hidden layer.

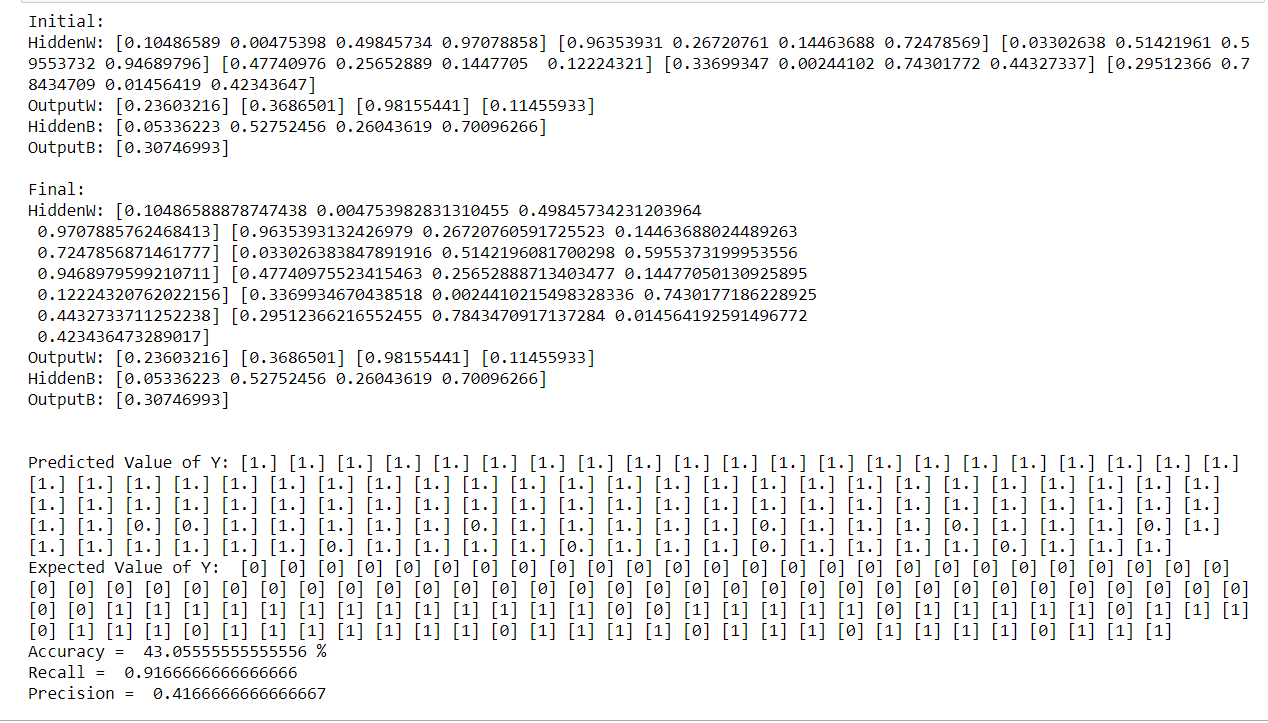
**EXPERIMENTATION:**

1. **Effect of no. of hidden layer nodes**

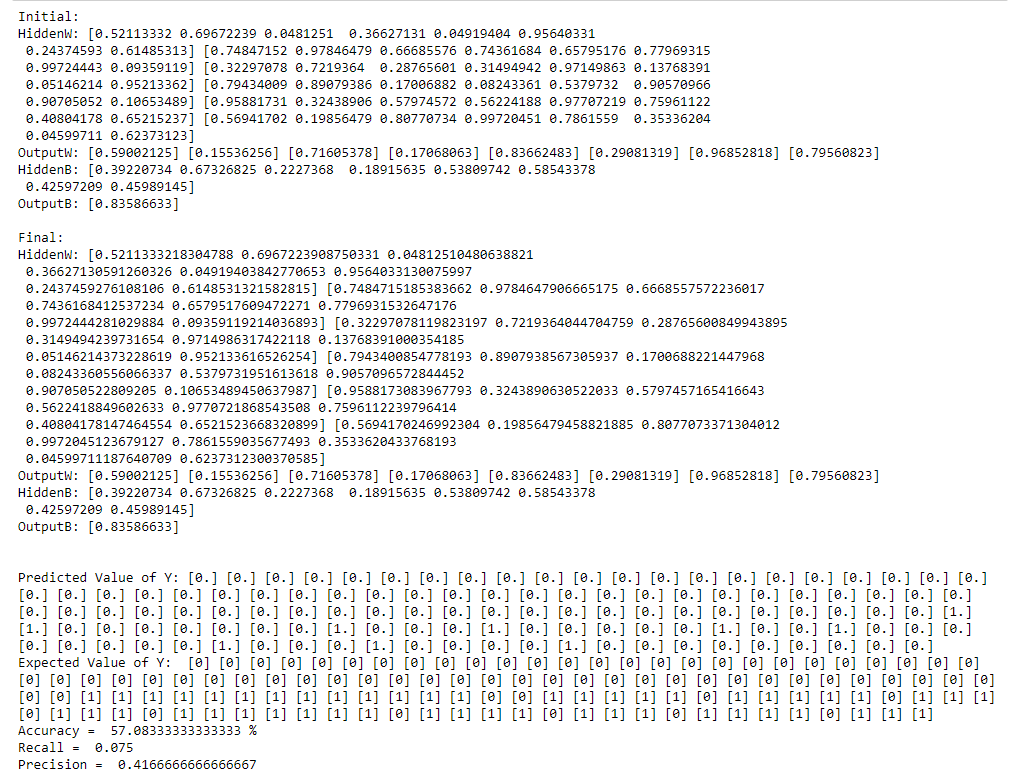
Illustrated from Dataset - 1

When no. of hidden layer nodes = 6A picture containing text

Description automatically generated

When no of hidden layer nodes = 4

When no. of hidden layer nodes = 8

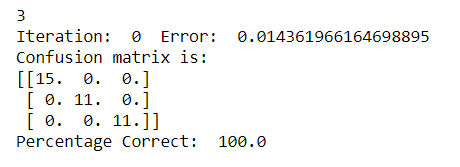


We see that as the number of hidden layer nodes increases the accuracy improves.

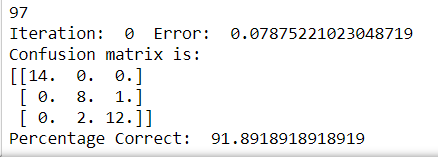
1. **Effect of Activation Function**

Illustrated using Dataset – 2

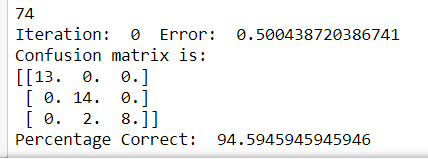
When **Sigmoid** Activation Function is applied



When **Softmax** Activation Function is applied



When **linear** Activation Function is applied

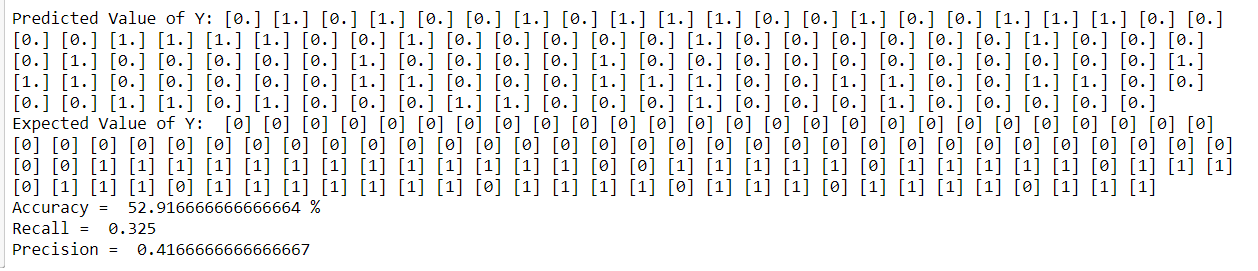


We can see that the sigmoid activation function converges quickly producing better accuracy, while the softmax and linear activations aren’t as accurate and take more time to converge.

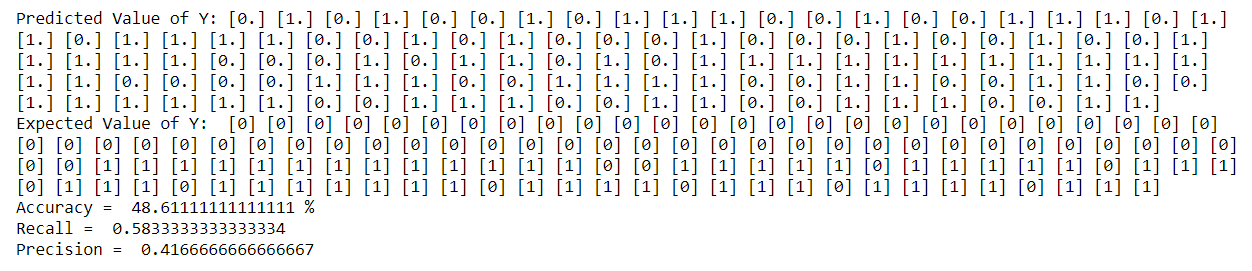
1. **Effect of the number of epochs**

Illustrated using Dataset – 1

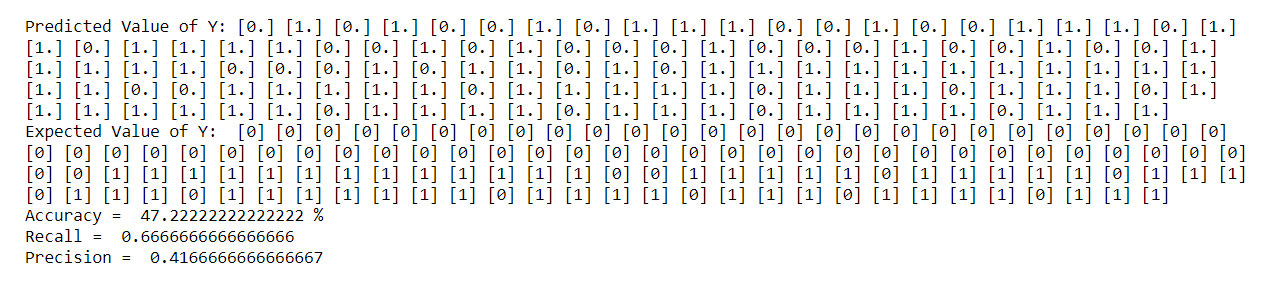
When no of epochs = 100



When no of epochs = 50



When no of epochs = 5000

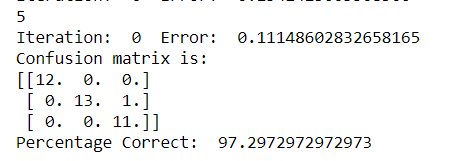


Though we can see that the accuracy improves as the number of epochs are increased from 50 to 100. But it is also necessary to see that increasing the no. of epochs doesn’t necessarily mean better accuracy as we see a dip in accuracy when we trained for 5000 epochs.

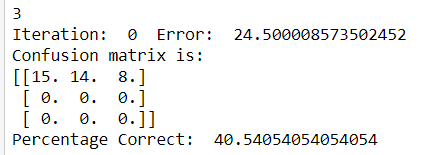
1. **Effect of Learning rate**

Illustrated using dataset-2

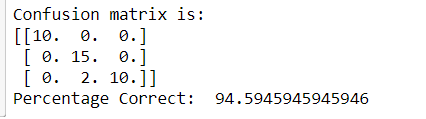
**When eta = 0.1**



**When eta = 0.4**



**When eta = 0.025**



Neither a learning rate which is high nor a lower one produced an impressive result. The model was trained using three different learning rates 0.1, 0.4 and 0.025. It is observed that the accuracy was at the maximum when eta = 0.1. While it was considerably lower at 0.4 and 0.025. Hence setting a learning rate between 0.1 and 0.4 is appreciated.

**CODE FOR DATASET – 1**

import numpy as np # for math functions

import pandas as pd # for importing and using datasets

import random # for generating random weights

def load\_data(filename, target):

dataset = pd.read\_csv("dataset/"+filename, sep="\t") # read .csv file

print(dataset,"\n")

x = np.array(dataset.drop([target],1))

y = np.array(dataset[target]) # y contains the target class

print("\nX = \n",x)

print("\nY = \n",y)

print("\n\n")

return (dataset,x,y,target)

inp = load\_data("diagnosis-data.csv","decision")

# x\_data = add\_initial\_column(inp[1],-1)

x\_data = np.array(inp[1])

y\_data = np.array([inp[2]])

y\_data = np.transpose(y\_data)

x\_data = x\_data[:,1:]

print(x\_data,"\n")

print(y\_data)

def activation\_func(x):

return sigmoid(x)

def sigmoid(x):

# print("Exp: ", np.exp(-x))

x = np.array(x, dtype=int)

return 1/(1 + np.exp(-x))

def sign(x):

res = 0 if x <= 0 else 1

# print(res)

return res

def sigmoid\_deriv(x):

x = np.array(x, dtype=int)

return x \* ( 1 - x)

class MLP\_single\_hidden():

def \_\_init\_\_(self, numI, numH, numO, x\_data, y\_data):

self.numI = numI

self.numH = numH

self.numO = numO

self.x = x\_data

self.y = y\_data

self.epochs = 5000

self.lr = 0.1

self.tp = 0

self.tn = 0

self.fp = 0

self.fn = 0

self.cost\_array = []

self.initialize\_weights()

self.initialize\_bias()

def initialize\_weights(self):

self.hiddenW = np.random.uniform(size=(self.numI, self.numH))

self.outputW = np.random.uniform(size=(self.numH, self.numO))

def initialize\_bias(self):

self.hiddenB = np.random.uniform(size=(1,self.numH))

self.outputB = np.random.uniform(size=(1,self.numO))

def forward\_propagation(self):

self.hidden\_layer\_activation = np.dot(self.x, self.hiddenW)

self.hidden\_layer\_activation += self.hiddenB

self.hidden\_layer\_output = sigmoid(self.hidden\_layer\_activation)

self.output\_layer\_activation = np.dot(self.hidden\_layer\_output, self.outputW)

self.output\_layer\_activation += self.outputB

self.y\_predict = sigmoid(self.output\_layer\_activation)

def backward\_propagation(self):

self.error = ((self.y - self.y\_predict)) #\*\*2).mean()

self.d\_y\_predict = self.error \* sigmoid\_deriv(self.y\_predict)

self.error\_hidden\_layer = self.d\_y\_predict.dot(self.outputW.T)

self.d\_hidden\_layer = self.error\_hidden\_layer\*sigmoid\_deriv(self.hidden\_layer\_output)

def update\_weights(self):

self.outputW = (self.outputW)+((self.hidden\_layer\_output.T.dot(self.d\_y\_predict))\* self.lr)

self.hiddenW = (self.hiddenW + (self.x.T.dot(self.d\_hidden\_layer)) \* self.lr )

def update\_bias(self):

self.outputB = self.outputB + (np.sum(self.d\_y\_predict, axis=0, keepdims=True) \* self.lr)

self.hiddenB = self.hiddenB + (np.sum(self.d\_hidden\_layer, axis=0, keepdims=True) \* self.lr)

def train(self):

for i in range(self.epochs):

self.forward\_propagation()

self.backward\_propagation()

self.update\_weights()

self.update\_bias()

self.cost\_array.append(self.error)

def print\_weights(self):

print("HiddenW: ",end="")

print(\*self.hiddenW)

print("OutputW: ",end="")

print(\*self.outputW)

def print\_bias(self):

print("HiddenB: ",end="")

print(\*self.hiddenB)

print("OutputB: ",end="")

print(\*self.outputB)

def print\_y\_predict(self):

print("\n\nPredicted Value of Y: ", end="")

print(\*self.y\_predict)

print("Expected Value of Y: ", end="")

print(\*self.y)

for i in self.y\_predict:

for j in self.y:

if i==1 and j==1:

self.tp += 1

elif i == 0 and j == 0:

self.tn += 1

elif i == 1 and j == 0:

self.fp += 1

else:

self.fn += 1

print("Accuracy = ",((self.tp+self.tn)/(self.tp+self.tn+self.fp+self.fn))\*100,"%")

print("Recall = ",(self.tp)/(self.tp+self.fn))

print("Precision = ",(self.tp)/(self.tp+self.fp))

def apply\_threshold(self):

for i in range(self.y\_predict.size):

if self.y\_predict[i][0] < 0.5:

self.y\_predict[i][0] = 0

else:

self.y\_predict[i][0] = 1

XOR = MLP\_single\_hidden(6,8,1,x\_data,y\_data)

print("Initial: ")

XOR.print\_weights()

XOR.print\_bias()

XOR.train()

print("\nFinal: ")

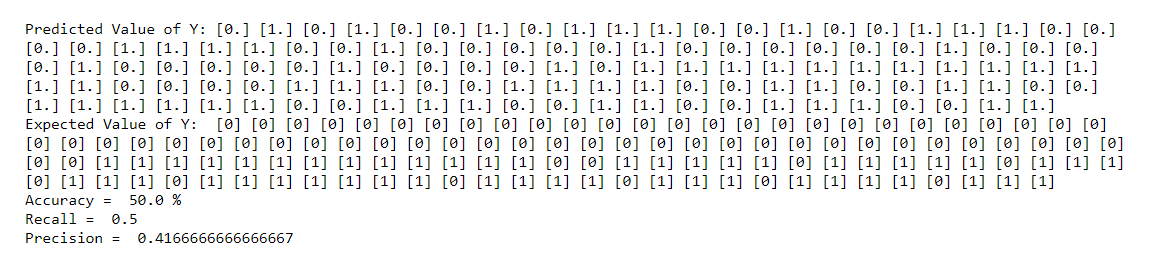
XOR.print\_weights()

XOR.print\_bias()

XOR.apply\_threshold()

XOR.print\_y\_predict()

**OUTPUT**



**CODE FOR DATASET – 2**

import numpy as np

class mlp: def\_\_init\_\_(self,inputs,targets,nhidden,beta=1,momentum=0.9,outtype='logistic'):

""" Constructor """

# Set up network size

self.nin = np.shape(inputs)[1]

self.nout = np.shape(targets)[1]

self.ndata = np.shape(inputs)[0]

self.nhidden = nhidden

self.beta = beta

self.momentum = momentum

self.outtype = outtype

# Initialise network

self.weights1 = (np.random.rand(self.nin+1,self.nhidden)-0.5)\*2/np.sqrt(self.nin)

self.weights2 = (np.random.rand(self.nhidden+1,self.nout)-0.5)\*2/np.sqrt(self.nhidden)

def earlystopping(self,inputs,targets,valid,validtargets,eta,niterations=100):

valid = np.concatenate((valid,-np.ones((np.shape(valid)[0],1))),axis=1)

old\_val\_error1 = 100002

old\_val\_error2 = 100001

new\_val\_error = 100000

count = 0

while (((old\_val\_error1 - new\_val\_error) > 0.001) or ((old\_val\_error2 - old\_val\_error1)>0.001)):

count+=1

print(count)

self.mlptrain(inputs,targets,eta,niterations)

old\_val\_error2 = old\_val\_error1

old\_val\_error1 = new\_val\_error

validout = self.mlpfwd(valid)

new\_val\_error = 0.5\*np.sum((validtargets-validout)\*\*2)

#print("Stopped", new\_val\_error,old\_val\_error1, old\_val\_error2)

return new\_val\_error

def mlptrain(self,inputs,targets,eta,niterations):

""" Train the thing """

# Add the inputs that match the bias node

inputs = np.concatenate((inputs,-np.ones((self.ndata,1))),axis=1)

change = range(self.ndata)

updatew1 = np.zeros((np.shape(self.weights1)))

updatew2 = np.zeros((np.shape(self.weights2)))

for n in range(niterations):

self.outputs = self.mlpfwd(inputs)

error = 0.5\*np.sum((self.outputs-targets)\*\*2)

if (np.mod(n,100)==0):

print("Iteration: ",n, " Error: ",error)

# Different types of output neurons

if self.outtype == 'linear':

deltao = (self.outputs-targets)/self.ndata

elif self.outtype == 'logistic':

deltao = self.beta\*(self.outputs-targets)\*self.outputs\*(1.0-self.outputs)

elif self.outtype == 'softmax':

deltao = (self.outputs-targets)\*(self.outputs\*(-self.outputs)+self.outputs)/self.ndata

else:

print("error")

deltah = self.hidden\*self.beta\*(1.0-self.hidden)\*(np.dot(deltao,np.transpose(self.weights2)))

updatew1 = eta\*(np.dot(np.transpose(inputs),deltah[:,:-1])) + self.momentum\*updatew1

updatew2 = eta\*(np.dot(np.transpose(self.hidden),deltao)) + self.momentum\*updatew2

self.weights1 -= updatew1

self.weights2 -= updatew2

def mlpfwd(self,inputs):

""" Run the network forward """

self.hidden = np.dot(inputs,self.weights1);

self.hidden = 1.0/(1.0+np.exp(-self.beta\*self.hidden))

self.hidden = np.concatenate((self.hidden,-np.ones((np.shape(inputs)[0],1))),axis=1)

outputs = np.dot(self.hidden,self.weights2);

# Different types of output neurons

if self.outtype == 'linear':

return outputs

elif self.outtype == 'logistic':

return 1.0/(1.0+np.exp(-self.beta\*outputs))

elif self.outtype == 'softmax':

normalisers = np.sum(np.exp(outputs),axis=1)\*np.ones((1,np.shape(outputs)[0]))

return np.transpose(np.transpose(np.exp(outputs))/normalisers)

else:

print("error")

def confmat(self,inputs,targets):

"""Confusion matrix"""

# Add the inputs that match the bias node

inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),axis=1)

outputs = self.mlpfwd(inputs)

nclasses = np.shape(targets)[1]

if nclasses==1:

nclasses = 2

outputs = np.where(outputs>0.5,1,0)

else:

# 1-of-N encoding

outputs = np.argmax(outputs,1)

targets = np.argmax(targets,1)

cm = np.zeros((nclasses,nclasses))

for i in range(nclasses):

for j in range(nclasses):

cm[i,j] = np.sum(np.where(outputs==i,1,0)\*np.where(targets==j,1,0))

print("Confusion matrix is:")

print(cm)

print("Percentage Correct: ",np.trace(cm)/np.sum(cm)\*100)

def preprocessIris(infile,outfile):

stext1 = 'Iris-setosa'

stext2 = 'Iris-versicolor'

stext3 = 'Iris-virginica'

rtext1 = '0'

rtext2 = '1'

rtext3 = '2'

fid = open(infile,"r")

oid = open(outfile,"w")

for s in fid:

if s.find(stext1)>-1:

oid.write(s.replace(stext1, rtext1))

elif s.find(stext2)>-1:

oid.write(s.replace(stext2, rtext2))

elif s.find(stext3)>-1:

oid.write(s.replace(stext3, rtext3))

fid.close()

oid.close()

import numpy as np

iris = np.loadtxt('dataset/iris\_proc.data',delimiter=',')

iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)

imax = np.concatenate((iris.max(axis=0)\*np.ones((1,5)),np.abs(iris.min(axis=0)\*np.ones((1,5)))),axis=0).max(axis=0)

iris[:,:4] = iris[:,:4]/imax[:4]

print(iris[0:5,:])

# Split into training, validation, and test sets

target = np.zeros((np.shape(iris)[0],3));

indices = np.where(iris[:,4]==0)

target[indices,0] = 1

indices = np.where(iris[:,4]==1)

target[indices,1] = 1

indices = np.where(iris[:,4]==2)

target[indices,2] = 1

order = list(range(np.shape(iris)[0]))

np.random.shuffle(order)

iris = iris[order,:]

target = target[order,:]

train = iris[::2,0:4]

traint = target[::2]

valid = iris[1::4,0:4]

validt = target[1::4]

test = iris[3::4,0:4]

testt = target[3::4]

net = mlp(train,traint,4,outtype='logistic')

net.earlystopping(train,traint,valid,validt,0.025)

net.confmat(test,testt)

**OUTPUT**

