SCT Experiment No: 3

Name: Abhishek S Waghchaure

PRN: 1032221714

Dept: FY M Tech DSA(2022-24)

Aim:

Implementing Artificial Neural Network for XOR & XNOR Logic Gate.

Introduction:

The basic steps involved in training of a neural network are given below-

1-Random initialization of weights.

2-Iterating over full data-

a) Feed Forward- Computing the the output of the Neural network with the randomly initialized weights and computing the loss function between the actual output and predicted output.

b) Back Propagation- Compute the gradient of Loss function with respect to weights and update the weights using gradient descent.

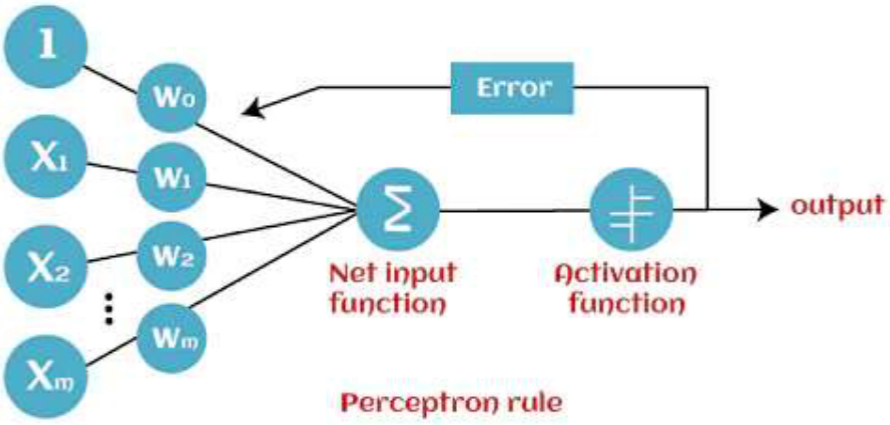
4-Repeat until the loss is minimum.

Dataset:

The neural networks are more useful when the dataset is non-linearly separable and it can not be separated with the help of basics techniques such as linear regession, logistic regression,etc. For this case we will be using a non-linearly separable Dataset (XOR Gate) and we will show that how neural networks can easily classify this data set using feature engineering with the help of different hidden layers.

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

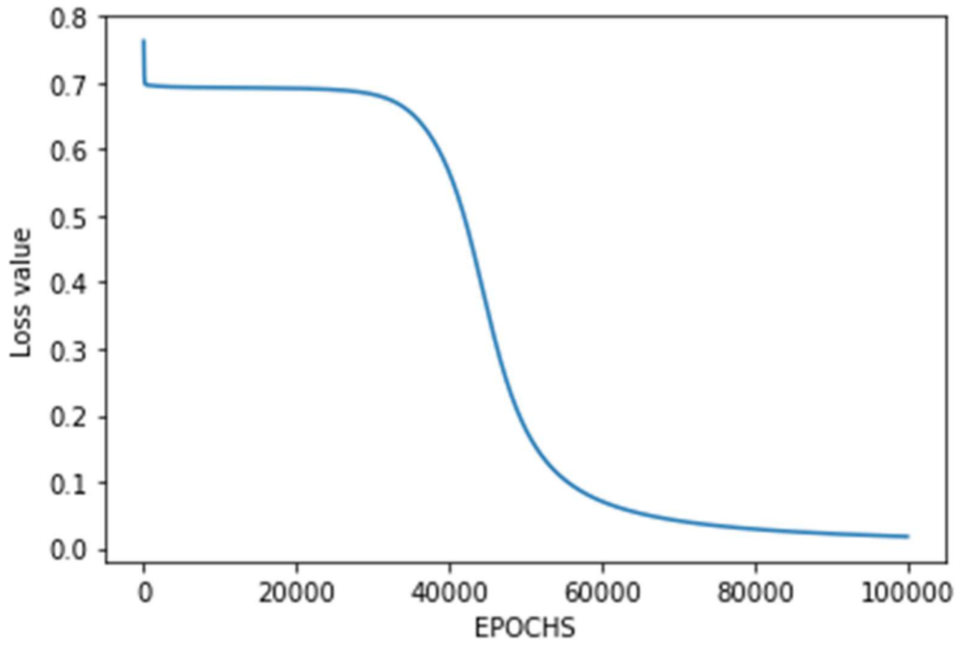
Truth Table for XOR Gate



Perceptron rule

```
# Forward Propagation
def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))

    cost = -np.sum(logprobs) / m
    return cost, cache, A2

# Backward Propagation
def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,"dZ1": dZ1, "dW1": dW1
,  "db1": db1} return
    gradients

#  Updating the weights based on the negative gradients def
updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"] return
    parameters

#  Model to learn the XOR truth table
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XOR input Y =
np.array([[0, 1, 1, 0]]) # XOR output

# Define model parameters
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, ou
```

```
tputFeatures)
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))

for i in range(epoch):
      losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
      gradients = backwardPropagation(X, Y, cache)
          parameters = updateParameters(parameters, gradients, learningRate)

#  Evaluating the performance
plt.figure() plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()


#  Testing
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input cost, _, A2 =
forwardPropagation(X, Y, parameters) prediction = (A2 > 0.5)
* 1.0
#  print(A2)
print(prediction)
```




```
[[1. 0. 0. 1.]]
```

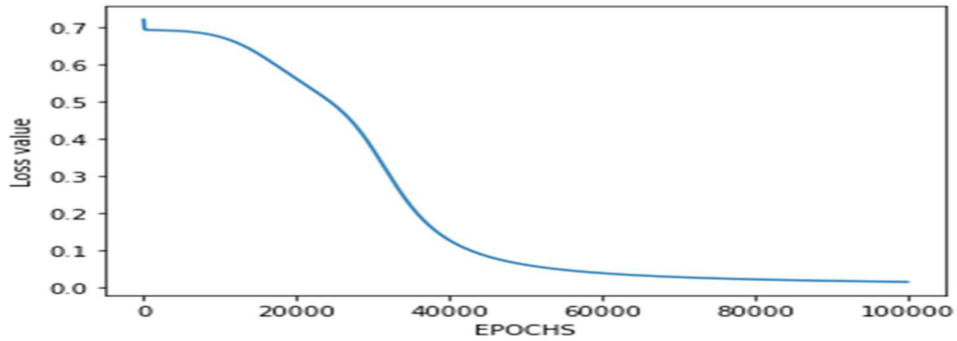
Implementation of Artificial Neural Network for XNOR Logic Gate¶

In [2]:

```
#  import Python Libraries
import numpy as np
from matplotlib import pyplot as plt
```

```
#  Sigmoid Function
def sigmoid(z):
      return 1 / (1 + np.exp(-z))

#  Initialization of the neural network parameters
#  Initialized all the weights in the range of between 0 and 1
#  Bias values are initialized to 0
def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatu res):

      W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
      W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
      b1 = np.zeros((neuronsInHiddenLayers, 1))
      b2 = np.zeros((outputFeatures, 1))

      parameters = {"W1" : W1, "b1": b1,
                    "W2" : W2, "b2": b2}
      return parameters

# Forward Propagation
def forwardPropagation(X, Y, parameters):
      m = X.shape[1]
      W1 = parameters["W1"]
      W2 = parameters["W2"]
      b1 = parameters["b1"]
      b2 = parameters["b2"]

      Z1 = np.dot(W1, X) + b1
      A1 = sigmoid(Z1)
      Z2 = np.dot(W2, A1) + b2
      A2 = sigmoid(Z2)

      cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
      logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))

      cost = -np.sum(logprobs) / m
      return cost, cache, A2

# Backward Propagation
def backwardPropagation(X, Y, cache):
      m = X.shape[1]
      (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

      dZ2 = A2 - Y
      dW2 = np.dot(dZ2, A1.T) / m
      db2 = np.sum(dZ2, axis = 1, keepdims = True)

      dA1 = np.dot(W2.T, dZ2)
      dZ1 = np.multiply(dA1, A1 * (1- A1))
      dW1 = np.dot(dZ1, X.T) / m
      db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

      gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,
                   "dZ1": dZ1, "dW1": dW1, "db1": db1}
```

```
    return gradients

# Updating the weights based on the negative gradients def
updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"] return
    parameters

# Model to learn the XNOR truth table
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XNOR input Y =
np.array([[1, 0, 0, 1]]) # XNOR output

# Define model parameters
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, ou
tputFeatures)
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))

for i in range(epoch):
    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
    gradients = backwardPropagation(X, Y, cache)
        parameters = updateParameters(parameters, gradients, learningRate)

# Evaluating the performance
plt.figure() plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()


# Testing
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XNOR input cost, _, A2 =
forwardPropagation(X, Y, parameters) prediction = (A2 > 0.5) *
1.0
# print(A2)
print(prediction)
```



[[0. 1. 1. 0.]]

Conclusion:

In conclusion, Neural network is a collection of connected units with input and output mechanism, each of the connections has an associated weight. Backpropagation is the "backward propagation of errors" and is useful to train neural networks. It is fast, easy to implement and simple. Backpropagation is very beneficial for deep neural networks working over error prone projects like speech or image recognition.

Reference:

https://www.mygreatlearning.com/blog/backpropagation-algorithm/
https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron
https://www.geeksforgeeks.org/implementation-of-perceptron-algorithm-for-nand-logic-gate-with-2-bit-binary-input/