Experiment No. 2
Implement Multilayer Perceptron algorithm to simulate XOR gate
Date of Performance:
Date of Submission:

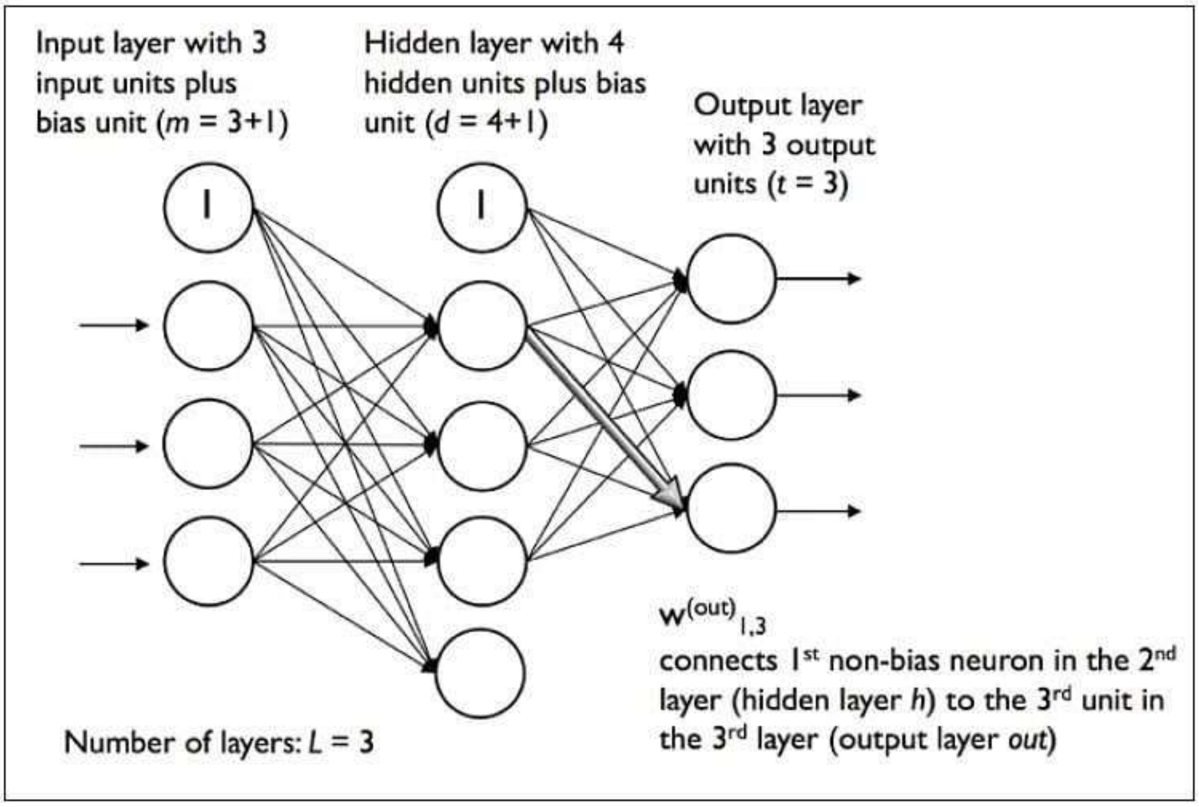
23_DL_PARTH MAHENDRA PURI_AI&DS

Aim: Implement Multilayer Perceptron algorithm to simulate XOR gate.

Objective: Ability to perform experiments on different architectures of multilayer perceptorn.

Theory:

multilayer artificial neuron network is an integral part of deep learning. And this lesson will help you with an overview of multilayer ANN along with overfitting and underfitting.


Input layer with 3 input units plus bias unit (m = 3+1)
Hidden layer with 4 hidden units plus bias unit (d = 4+1)
Output layer with 3 output units (t = 3)
1
1
$w^{(out)}_{1,3}$
connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer *out*)
Number of layers: L = 3

A fully connected multi-layer neural network is called a Multilayer Perceptron (MLP).

At has 3 layers including one hidden layer. If it has more than 1 hidden layer, it is called a deep ANN. An MLP is a typical example of a feedforward artificial neural network. In this figure, the ith activation unit in the lth layer is denoted as ai(l).

The number of layers and the number of neurons are referred to as hyperparameters of a neural network, and these need tuning. Cross-validation techniques must be used to find ideal values for these.

The weight adjustment training is done via backpropagation. Deeper neural networks are better at processing data. However, deeper layers can lead to vanishing gradient problems. Special algorithms are required to solve this issue.

A multilayer perceptron (MLP) is a feed forward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input nodes connected as a directed graph between the input and output layers. MLP uses backpropagation for training the network. MLP is a deep learning method.

Code-

```
import numpy as np
def converter(a):
    if (a >= 0):
        return 1
    else:
        return 0
def PerceptronModel(x, w, b):
    a = np.dot(w, x)+b
    y = converter(a)
    return y

# And Logic
def AND_logic(x):
    w = np.array([1, 1])
    bAND = -1.5
    return PerceptronModel(x, w, bAND)
# OR Logic
def OR_logic(x):
    w = np.array([2, 2])
    bOR = -1
    return PerceptronModel(x, w, bOR)
```

```
# NOT logic
def NOT_logic(x):
    w = -1
    bNOT = 0.5
    return PerceptronModel(x, w, bNOT)

def XOR_logic(x):
    y1 = AND_logic(x)
    y2 = OR_logic(x)
    y3 = NOT_logic(y1)
    Fin_x = np.array([y2, y3])
    F_Output = AND_logic(Fin_x)
    return F_Output



# Model Testing
test1 = np.array([0, 0])
test2 = np.array([0, 1])
test3 = np.array([1, 0])
test4 = np.array([1, 1])

print("XOR({}, {}) = {}".format(0, 0, XOR_logic(test1)))
print("XOR({}, {}) = {}".format(0, 1, XOR_logic(test2)))
print("XOR({}, {}) = {}".format(0, 1, XOR_logic(test3)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logic(test4)))
```

Conclusion:

Certainly! Neural networks are powerful models inspired by the brain's structure, with layers of interconnected nodes. The backpropagation algorithm is a technique to train these networks. It calculates the difference between predicted and actual outputs, adjusts weights accordingly to minimize the difference, and repeats until the network learns patterns in data. This approach has been transformative in machine learning but requires careful parameter tuning and can face challenges like vanishing/exploding gradients.