SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

COURSE CODE: DJ19DSC501

COURSE NAME: Machine Learning – II **CLASS: AY 2022-23**

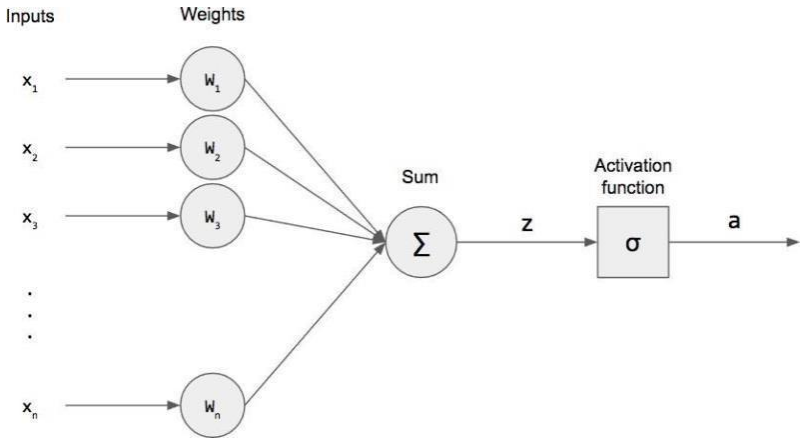
LAB EXPERIMENT NO.1

NAME: Bhuvi Ghosh **SAP ID: 60009210191** **BATCH: D22**

AIM : Implement Boolean gates using perceptron – Neural representation of Logic Gates.

THEORY:

Perceptron is a Supervised Learning Algorithm for binary classifiers.



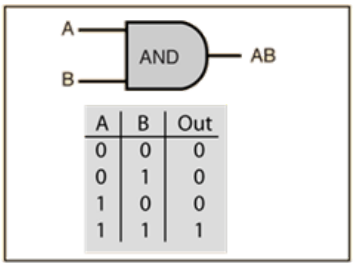
For a particular choice of the weight vector w and bias parameter b, the model predicts output ŷ for the corresponding input vector x.

$$\hat{y} = \Theta\left(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b\right)$$
$$= \Theta(\mathbf{w} \cdot \mathbf{x} + b)$$
$$\text{where } \Theta(v) = \begin{cases} 1 & \text{if } v \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

The input values, i.e., x1, x2, and bias is multiplied with their respective weight matrix that is W1, W2, and W0. The corresponding value is then fed to the summation neuron where the summed value is calculated. This is fed to a neuron which has a non-linear function(sigmoid in our case) for scaling the output to a desirable range. The scaled output of sigmoid is 0 if the output is less than 0.5 and 1 if the output is greater than 0.5. The main aim is to find the value of weights or the weight vector which will enable the system to act as a particular gate.

Boolean gates – Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic.

1) AND



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

IMPLEMENTATION CODE:

```
def andFun(df):
    w=[-2,1,1]
    return perceptron(df,w)
```

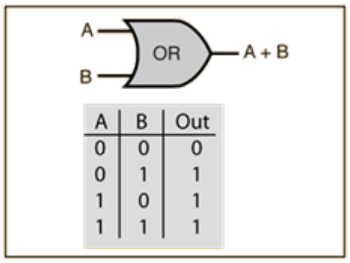
```
def perceptron(df,w):
    pred=[]
    for i in range(len(df)):
        yin=np.dot(df.drop('soln',axis=1).iloc[i],w)
        pred.append(stepfunc(yin))
    df['pred']=pred
    return df
```

```
dict={'inp0':[1,1,1,1],'inp1':[0,1,0,1],'inp2':[0,0,1,1],'soln':[0,0,0,1]}
df=pd.DataFrame(dict)
print(andFun(df))
```

```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

inp1	inp2	soln	pred
0	0	0	0
1	0	0	0
0	1	0	0
1	1	1	1

2) OR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

IMPLEMENTATION CODE:

```
def orFun(df):
    w=[-0.5,1,1]
    return perceptron(df,w)
```

INPUTS:

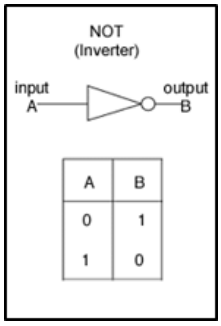
```
ordict={'inp0':[1,1,1,1],'inp1':[0,1,0,1],'inp2':[0,0,1,1],'soln':[0,1,1,1]}
ordf=pd.DataFrame(ordict)
print(orFun(ordf))
```

```
def perceptron(df,w):
    pred=[]
    for i in range(len(df)):
        yin=np.dot(df.drop('soln',axis=1).iloc[i],w)
        pred.append(stepfunc(yin))
    df['pred']=pred
    return df
```

```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

inp1	inp2	soln	pred
0	0	0	0
1	0	1	1
0	1	1	1
1	1	1	1

3) NOT



IMPLEMENTATION CODE:

```
def notFun(df):
    w=[0.5,-1]
    return perceptron(df,w)
```

```
def perceptron(df,w):
    pred=[]
    for i in range(len(df)):
        yin=np.dot(df.drop('soln',axis=1).iloc[i],w)
        pred.append(stepfunc(yin))
    df['pred']=pred
    return df
```

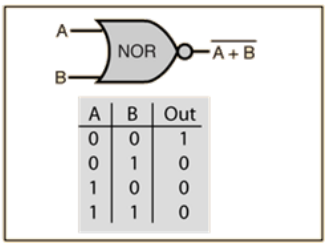
```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

```
notdict={'inp0':[1,1],'inp1': [0,1],'soln':[1,0]}
notdf=pd.DataFrame(notdict)
print(notFun(notdf))
```

OUTPUTS:

inp0	inp1	soln	pred
1	0	1	1
1	1	0	0

4)NOR



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

IMPLEMENTATION FUNCTION:

```
def norFun(df):
    w=[0.5,-1,-1]
    return perceptron(df,w)
```

```
def perceptron(df,w):
    pred=[]
    for i in range(len(df)):
        yin=np.dot(df.drop('soln',axis=1).iloc[i],w)
        pred.append(stepfunc(yin))
    df['pred']=pred
    return df
```

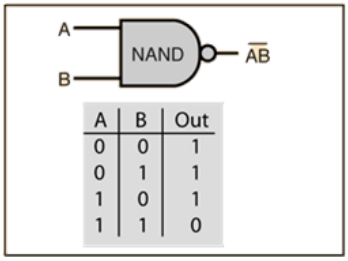
```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

```
nordict={'inp0':[1,1,1,1],'inp1':[0,1,0,1],'inp2':[0,0,1,1],'soln':[1,0,0,0]}
nordf=pd.DataFrame(nordict)
print(norFun(nordf))
```

OUTPUTS:

inp1	inp2	soln	pred
0	0	1	1
1	0	0	0
0	1	0	0
1	1	0	0

4) NAND



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

IMPLEMENTATION FUNCTION:

INPUTS:

```
def NandFun(df):
    w=[1.5,-1,-1]
    return(perceptron(df,w))
```

```
nanddict={'inp0':[1,1,1,1],'inp1':[0,1,0,1],'inp2':[0,0,1,1],'soln':[1,1,1,0]}
nanddf=pd.DataFrame(nanddict)
print(NandFun(nanddf))
```

```
def perceptron(df,w):
    pred=[]
    for i in range(len(df)):
        yin=np.dot(df.drop('soln',axis=1).iloc[i],w)
        pred.append(stepfunc(yin))
    df['pred']=pred
    return df
```

```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

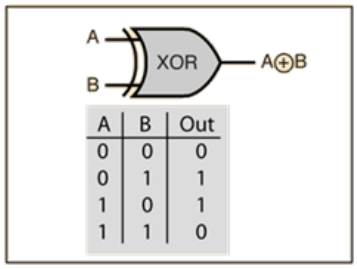
inp1	inp2	soln	pred
0	0	1	1
1	0	0	0
0	1	0	0
1	1	0	0

```
def xorFun(xordf):

    tempdf['inp1']=NandFun(xordf)['pred']
    tempdf['inp2']=orFun(xordf.drop('pred',axis=1))['pred']
    xordf['pred']=andFun(tempdf.drop(['pred'],axis=1))['pred']
    return(xordf)
```

```
def perceptron(df,w):
    pred=[]
    for i in range(len(df))
        yin=np.dot(df.drop(
        pred.append(stepfun
    df['pred']=pred
    return df
```

5) XOR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

IMPLEMENTATION FUNCITON:

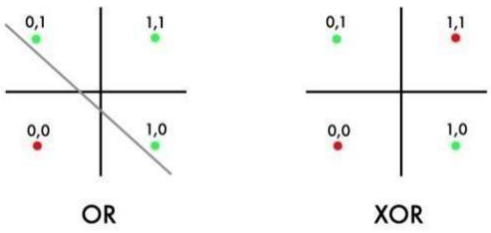
```
def stepfunc(ans):
    if(ans>=0):
        return 1
    return 0
```

INPUTS:

```
xordict={'inp0':[1,1,1,1],'inp1':[0,1,0,1],'inp2':[0,0,1,1],'soln':[0,1,1,0]}
xordf=pd.DataFrame(xordict)
print(xorFun(xordf))
```

inp1	inp2	soln	pred
0	0	0	0
1	0	1	1
0	1	1	1
1	1	0	0

For the XOR gate, the output can't be linearly separated. Uni layered perceptrons can only work with linearly separable data. But in the following diagram drawn in accordance with the truth table of the XOR logical operator, we can see that the data is NOT linearly separable.



To solve this problem, we add an extra layer to our vanilla perceptron, i.e., we create a Multi Layered Perceptron (or MLP). We call this extra layer as the Hidden layer. To build a perceptron, we first need to understand that the XOR gate can be written as a combination of AND gates, NOT gates and OR gates in the following way:

$$XOR(x_1, x_2) = AND(NOT(AND(x_1, x_2)), OR(x_1, x_2))$$

Hidden layers are those layers with nodes other than the input and output nodes. An MLP is generally restricted to having a single hidden layer. The hidden layer allows for non-linearity. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with their own weights and biases, and an output is computed, generally with an activation function.

CONCLUSION:

- This experiment illustrates the concept that complex logical operations can be broken down into perceptron-based building blocks, making it accessible for beginners in neural network programming.
- The outcomes of this experiment emphasize the importance of fine-tuning perceptron parameters, such as learning rates, to optimize their performance.
- By successfully coding logic gates with perceptrons, I gained valuable experience in troubleshooting and debugging neural network implementations.

- The versatility of perceptrons in reproducing logic gates showcases their potential as essential components in artificial intelligence systems, ranging from chatbots to selfdriving cars.

- This study provides practical insights into the role of perceptrons in creating adaptive and intelligent systems capable of learning from data.

- The precision achieved in this experiment underscores the significance of welldesigned training datasets for perceptron-based models.