Neural Representation of AND, NAND, OR, XOR, NOT and XNOR Logic Gates under

> ## (Perceptron Algorithm)

the neural network updates the weights, but the logic behind how the values are being changed in simple terms.

First, we need to know that the Perceptron algorithm states that: Prediction (y) = 1

> if Wx+b > 0 and 0 if Wx+b ≤ 0

the steps in this method are very similar to how Neural Networks learn, which is as follows;

1. Initialize weight values and bias
2. Forward Propagate
3. Check the error
4. Backpropagate and Adjust weights and bias
5. Repeat for all training examples

Now that we know the steps, let's get up and running:

AND Gate

- From our knowledge of logic gates, the output of an AND gate is 1 only if both inputs (in this case, x1 and x2) are 1.

> From w1*x1+w2*x2+b, initializing w1, w2, as 1 and b as –1, we get;

In [27]:
```python
# LogicGate's Implementation through class definitions
import numpy as np

class LogicGate:
    def __init__(self):
        pass

# AND Gate
# The AND gate gives an output of 1 if both the two inputs are 1, it gives 0 otherwise.
    def do_and(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
        _nodes = np.array([x1,x2])
        _weights = np.array([1,1])
        _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
        _eval = np.sum([_nodes*_weights]) + _bias
        return 1 if _eval>0 else 0
```

From w1*x1+w2*x2+b, initializing w1, w2, as 1 and b as –1, we get; x1(1)+x2(1)–1 Passing the

1. step-1 the AND logic table (x1=0, x2=0), we get; 0+0–1 = –1 From the Perceptron rule, if Wx+b≤0, then y =0. Therefore, this row is correct, and no need for Backpropagation.
2. step-2 Passing (x1=0 and x2=1), we get; 0+1–1 = 0 From the Perceptron rule, if Wx+b≤0, then y=0. This row is correct, as the output is 0 for the AND gate.From the Perceptron rule, this works (for both step 1, step 2 and step 3).
3. Passing (x1=1 and x2=1), we get; 1+1–1 = 1 Again, from the perceptron rule, this is still valid. Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is

In [28]:
```python
Values  = [(0,0),(0,1),(1,0),(1,1)]
print("+---------------+----------------+")
print(" | AND Truth Table | Result |")
for i  in Values:
    _result = LogicGate.do_and(i[0],i[1])
    print(" A = {}, B = {} | A AND B = {}".format(i[0],i[1],_result)," | ")
```

```
+---------------+----------------+
 | AND Truth Table | Result |
 A = 0, B = 0 | A AND B = 0  |
 A = 0, B = 1 | A AND B = 0  |
 A = 1, B = 0 | A AND B = 0  |
 A = 1, B = 1 | A AND B = 1  |
```

NAND Gate

From our knowledge of logic gates, the output of an NAND gate is negation of AND Gate.

1. From `w1x1+w2x2+b`, initializing w1 and w2 as 1, and b as -1, we get; x1(1)+x2(1)-1 Passing the first row of the NAND logic table (x1=0, x2=0), we get; 0+0-1 = -1 From the Perceptron rule, if Wx+b≤0, then y=0. This row is incorrect, as the output is 1 for the NAND gate. So we want values that will make input x1=0 and x2 = 0 to give y` a value of 1. If we change b to 1, we have; 0+0+1 = 1 From the Perceptron rule, this works.

2. Passing (x1=0, x2=1), we get; 0+1+1 = 2 From the Perceptron rule, if Wx+b > 0, then y=1. This row is also correct (for both step 2 and step 3).

3. Passing (x1=1, x2=1), we get; 1+1+1 = 3 This is not the expected output, as the output is 0 for a NAND combination of x1=1 and x2=1. Changing values of w1 and w2 to -1, and value of b to 2, we get; -1-1+2 = 0 It works for all rows. Therefore, we can conclude that the model to achieve a NAND gate, using the Perceptron algorithm is;

In [45]:
```python
# AND Gate
# The AND gate gives an output of 1 if both the two inputs are 1, it gives 0 otherwise.
def do_nand(x1,x2):
    return 0 if do_and(x1, x2) else 1
def do_and(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
    _nodes = np.array([x1,x2])
    _weights = np.array([1,1])
    _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
    _eval = np.sum([_nodes*_weights]) + _bias
    return 1 if _eval>0 else 0
```

In [46]:
```python
print("+---------------+----------------+")
print(" | NAND Truth Table | Result |")
for i  in Values:
    _result = do_nand(i[0],i[1])
    print(" A = {}, B = {} | A NAND B = {}".format(i[0],i[1],_result)," | ")
```

```
+---------------+----------------+
 | NAND Truth Table | Result |
 A = 0, B = 0 | A NAND B = 1  |
 A = 0, B = 1 | A NAND B = 1  |
 A = 1, B = 0 | A NAND B = 1  |
 A = 1, B = 1 | A NAND B = 0  |
```

OR Gate

From our knowledge of logic gates, the output of an OR gate is 0 only if both inputs (in this case, x1 and x2) are 0

1. From `w1x1+w2x2+b`, initializing w1, w2, as 1 and b as –1, we get; x1(1)+x2(1)–1 Passing the first row of the OR logic table (x1=0, x2=0), we get; 0+0–1 = –1 From the Perceptron rule, if Wx+b≤0, then y`=0. Therefore, this row is correct.
2. Passing (x1=0 and x2=1), we get; 0+1–1 = 0 From the Perceptron rule, if Wx+b <= 0, then y=0. Therefore, this row is incorrect. So we want values that will make inputs x1=0 and x2=1 give y` a value of 1. If we change w2 to 2, we have; 0+2–1 = 1 From the Perceptron rule, this is correct for both the row 1 and 2.
3. Passing (x1=1 and x2=0), we get; 1+0–1 = 0 From the Perceptron rule, if Wx+b <= 0, then y=0. Therefore, this row is incorrect. Since it is similar to that of row 2, we can just change w1 to 2, we have; 2+0–1 = 1 From the Perceptron rule, this is correct for both the row 1, 2 and 3.
4. Passing (x1=1 and x2=1), we get; 2+2–1 = 3 Again, from the perceptron rule, this is still valid. Quite Easy! Therefore, we can conclude that the model to achieve an OR gate, using the Perceptron algorithm

In [51]:
```python
def do_or(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
    _nodes = np.array([x1,x2])
    _weights = np.array([1,1])
    _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
    _eval = np.sum([_nodes*_weights]) + _bias
    return 0 if _eval>0 else 1
```

In [52]:
```python
print("+---------------+----------------+")
print(" | OR Truth Table | Result |")
for i  in Values:
    _result = do_or(i[0],i[1])
    print(" A = {}, B = {} | A OR B = {}".format(i[0],i[1],_result)," | ")
```

```
+---------------+----------------+
 | OR Truth Table | Result |
 A = 0, B = 0 | A OR B = 1  |
 A = 0, B = 1 | A OR B = 1  |
 A = 1, B = 0 | A OR B = 1  |
 A = 1, B = 1 | A OR B = 0  |
```

Do Not

1. From w1x1+b, initializing w1 as 1 (since single input), and b as –1, we get; x1(1)–1 Passing the first row of the NOT logic table (x1=0), we get; 0–1 = –1 From the Perceptron rule, if Wx+b≤0, then y=0. This row is incorrect, as the output is 1 for the NOT gate. So we want values that will make input x1=0 to give y a value of 1. If we change b to 1, we have; 0+1 = 1 From the Perceptron rule, this works.

2. Passing (x1=1), we get; 1+1 = 2 From the Perceptron rule, if Wx+b > 0, then y=1. This row is so incorrect, as the output is 0 for the NOT gate. So we want values that will make input x1=1 to give y a value of 0. If we change w1 to –1, we have; –1+1 = 0 From the Perceptron rule, if Wx+b ≤ 0, then y=0. Therefore, this works (for both row 1 and row 2). Therefore, we can conclude that the model to achieve a NOT gate, using the Perceptron algorithm is

In [55]:
```python
def do_not(x):
        # w are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w, b  = 1 , 0.8
        # if y = 0, x.w < b
        # if y = 1, x.w > b
        #_eval = x*w
    _nodes = np.array(x)
    _weights = np.array(1)
    _bias = -1
    _eval = np.array(_nodes* _weights) + _bias
    return 1 if _eval else 0
```

In [56]:
```python
sigValues= [0,1]
print("+---------------+----------------+")
print(" | NOT Truth Table | Result |")
for i  in range(len(sigValues)):
    _result = do_not(sigValues[i])
    print(" A = {} |        |       A NOT = {}".format(sigValues[i],_result))
```

```
+---------------+----------------+
 | NOT Truth Table | Result |
 A = 0 |        |       A NOT = 1
 A = 1 |        |       A NOT = 0
```

In [76]:
```python
def do_nor(x1,x2):
    return 0 if do_or(x1, x2) else 1
#OR Gate
#The OR gate gives an output of 1 if either of the two inputs is 1, it gives 0 otherwise
def do_or(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
    _nodes = np.array([x1,x2])
    _weights = np.array([1,1])
    _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
    _eval = np.sum([_nodes*_weights]) + _bias
    return 0 if _eval>0 else 1
```

In [77]:
```python
print("+---------------+----------------+")
print(" | XNOR Truth Table | Result |")
for i  in Values:
    _result = do_nor(i[0],i[1])
    print(" A = {}, B = {} | A NOR B = {}".format(i[0],i[1],_result)," | ")
```

```
+---------------+----------------+
 | XNOR Truth Table | Result |
 A = 0, B = 0 | A NOR B = 0  |
 A = 0, B = 1 | A NOR B = 0  |
 A = 1, B = 0 | A NOR B = 0  |
 A = 1, B = 1 | A NOR B = 1  |
```

XOR

In [80]:
```python
def do_xor(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.2
        # if y = ~(0), x1.w1 + x2.w2 < b
        # if y = ~(1), x1.w1 + x2.w2 > b
    y1 = do_or(x1, x2)
    y2 = do_nand(x1,x2)
    y =  do_and(y1, y2)
    return y
def do_or(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
    _nodes = np.array([x1,x2])
    _weights = np.array([1,1])
    _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
    _eval = np.sum([_nodes*_weights]) + _bias
    return 0 if _eval>0 else 1
def do_nand(x1,x2):
    return 0 if do_and(x1, x2) else 1
def do_and(x1,x2):
        # w1, w2 are weights of the paths to reach the destination to y
        # th ---> b threshold value
        #w1, w2 , b  = 0.5, 0.5 , 0.8
    _nodes = np.array([x1,x2])
    _weights = np.array([1,1])
    _bias = -1
        # if y = 0, x1.w1 + x2.w2 < b
        # if y = 1, x1.w1 + x2.w2 > b
        #_eval = x1*w1 + x2*w2
    _eval = np.sum([_nodes*_weights]) + _bias
    return 1 if _eval>0 else 0
```

In [85]:
```python
print("+---------------+----------------+")
print(" | XOR Truth Table | Result |")
for i  in Values:
    _result = do_xor(i[0],i[1])
    print(" A = {}, B = {} | A XOR B = {}".format(i[0],i[1],_result)," | ")
```

```
+---------------+----------------+
 | XOR Truth Table | Result |
 A = 0, B = 0 | A XOR B = 1  |
 A = 0, B = 1 | A XOR B = 1  |
 A = 1, B = 0 | A XOR B = 1  |
 A = 1, B = 1 | A XOR B = 0  |
```

In [ ]: