Lab2. Design of Logic Gates using Perceptron and Keras

```
Roll no:225229124
```

Part-I: Design OR gate using the concept of Perceptron

Step1:

Define helper functions You can implement gate operations by identifying the appropriate weights for w1 and w2 and bias b for the single neuron.

In [3]:

```
import numpy as np
import math
```

In [5]:

```
def sigmoid(x):
    s=1/(1+math.exp(-x))
    return s
```

In [6]:

```
def logic_gate(w1,w2,b):
    return lambda x1,x2: sigmoid(w1*x1+w2*x2+b)
```

In [7]:

```
def test(gate):
    for a,b in (0,0),(0,1),(1,0),(1,1):
        print("{},{}: {}".format(a,b,np.round(gate(a,b))))
```

Step2:

Identify values for weights, w1 and w2 and bias, b, for OR gate. Then, call logic_gate() function first with the values of weights and bias and test the outputs. For example, do the following steps and verify OR gate operations.

In [8]:

```
or_gate = logic_gate(20, 20, -10)
test(or_gate)
```

```
0,0: 0.0
0,1: 1.0
1,0: 1.0
1,1: 1.0
```

Part-II: Implement the operations of AND, NOR and NAND gates

Step1:

Identify values for weights, w1 and w2 and bias, b, for AND gate. Then, call logic_gate() function first with the values of weights and bias and test the outputs. Draw manually using pen the diagram of OR gate.

In [9]:

```
and_gate = logic_gate(10, 10, -10)
test(and_gate)
```

```
0,0: 0.0
0,1: 0.0
1,0: 0.0
1,1: 1.0
```

Step2:

Identify values for weights, w1 and w2 and bias, b, for NOR gate. Then, call logic_gate() function first with the values of weights and bias and test the outputs. Draw manually using pen the diagram of NOR gate.

In [11]:

```
nor_gate=logic_gate(-20,-20,20)
test(nor_gate)
```

```
0,0: 1.0
0,1: 0.0
1,0: 0.0
1,1: 0.0
```

Step3:

Identify values for weights, w1 and w2 and bias, b, for NAND gate. Then, call logic_gate() function first with the values of weights and bias and test the outputs. Draw manually using pen the diagram of NAND gate.

In [18]:

```
nand_gate = logic_gate(-1,-1,2)
test(nand_gate)
```

```
0,0: 1.0
0,1: 1.0
1,0: 1.0
1,1: 0.0
```

Part-III: Limitations of single neuron for XOR operation

```
Can you identify a set of weights such that a single neuron can output the values for
XOR gate?. Single
neurons can't correlate inputs, so it's just confused. So individual neurons are out.
Can we still use neurons to
somehow form an XOR gate?.
```

In [21]:

```
def xor_gate(a,b):
    c=or_gate(a,b)
    d=nand_gate(a,b)
    return and_gate(c,d)
test(xor_gate)
```

```
0,0: 0.0
0,1: 1.0
1,0: 1.0
1,1: 1.0
```

In [28]:

```
def logic_gate(w1, W2, b):
    return lambda x1, x2: sigmoid(w1 * x1 + W2 * x2 + b)
def final(gate):
    for a, b in zip(result1, result2):
        print("{}, {}: {}".format(a, b, np.round(gate(a, b))))
result1 = []
result2 = []

or_gate = logic_gate(20,20,-10)
for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
    result1.append(np.round(or_gate(a,b)))
nand_gate = logic_gate(-23,-25,35)
for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
    result2.append(np.round(nand_gate(a,b)))
xor_gate = logic_gate(20,20,-30)
print("XOR Gate truth table \n")
print("X, Y X+Y")
final(xor_gate)
```

```
XOR Gate truth table

X, Y X+Y
0.0, 1.0: 0.0
1.0, 1.0: 1.0
1.0, 1.0: 1.0
1.0, 0.0: 0.0
```

Part-IV: Logic Gates using Keras library

```
In this part of the lab, you will create and implement the operations of logic gates
such as AND, OR, NOT,
NAND, NOR and XOR in Keras.
```

In [29]:

```
from keras.models import Sequential
from keras.layers.core import Dense
```

In [30]:

```python
#AND gate
# the four different states of the AND gate
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# the four expected results in the same order
target_data = np.array([[0],[0],[0],[1]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
ep
Epoch 17/100
1/1 - 0s - loss: 0.2548 - binary_accuracy: 0.5000 - 9ms/epoch - 9ms/st
ep
Epoch 18/100
1/1 - 0s - loss: 0.2538 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/st
ep
Epoch 19/100
1/1 - 0s - loss: 0.2527 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/st
ep
Epoch 20/100
1/1 - 0s - loss: 0.2517 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/st
ep
Epoch 21/100
1/1 - 0s - loss: 0.2508 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/st
ep
Epoch 22/100
1/1 - 0s - loss: 0.2498 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/st
ep
Epoch 23/100
```

In [31]:

```
#OR gate
# the four different states of the OR gate
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# the four expected results in the same order
target_data = np.array([[0],[1],[1],[1]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
ep
Epoch 22/100
1/1 - 0s - loss: 0.2116 - binary_accuracy: 0.5000 - 7ms/epoch - 7ms/st
ep
Epoch 23/100
1/1 - 0s - loss: 0.2104 - binary_accuracy: 0.5000 - 9ms/epoch - 9ms/st
ep
Epoch 24/100
1/1 - 0s - loss: 0.2092 - binary_accuracy: 0.5000 - 8ms/epoch - 8ms/st
ep
Epoch 25/100
1/1 - 0s - loss: 0.2080 - binary_accuracy: 0.5000 - 7ms/epoch - 7ms/st
ep
Epoch 26/100
1/1 - 0s - loss: 0.2069 - binary_accuracy: 0.5000 - 7ms/epoch - 7ms/st
ep
Epoch 27/100
1/1 - 0s - loss: 0.2057 - binary_accuracy: 0.5000 - 7ms/epoch - 7ms/st
ep
Epoch 28/100
```

In [32]:

```python
#NOT gate
# the four different states of the NOT gate
training_data = np.array([[0],[1]], "float32")
# the four expected results in the same order
target_data = np.array([[1],[0]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=1, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
1/1 - 0s - loss: 0.1819 - binary_accuracy: 1.0000 - 6ms/epoch - 6ms/st
ep
Epoch 24/100
1/1 - 0s - loss: 0.1815 - binary_accuracy: 1.0000 - 6ms/epoch - 6ms/st
ep
Epoch 25/100
1/1 - 0s - loss: 0.1811 - binary_accuracy: 1.0000 - 8ms/epoch - 8ms/st
ep
Epoch 26/100
1/1 - 0s - loss: 0.1807 - binary_accuracy: 1.0000 - 7ms/epoch - 7ms/st
ep
Epoch 27/100
1/1 - 0s - loss: 0.1802 - binary_accuracy: 1.0000 - 7ms/epoch - 7ms/st
ep
Epoch 28/100
1/1 - 0s - loss: 0.1798 - binary_accuracy: 1.0000 - 8ms/epoch - 8ms/st
ep
Epoch 29/100
1/1 - 0s - loss: 0.1794 - binary_accuracy: 1.0000 - 6ms/epoch - 6ms/st
ep
```

In [33]:

```
#NAND gate
# the four different states of the NAND gate
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# the four expected results in the same order
target_data = np.array([[1],[1],[1],[0]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
ep
Epoch 22/100
1/1 - 0s - loss: 0.2245 - binary_accuracy: 0.7500 - 6ms/epoch - 6ms/st
ep
Epoch 23/100
1/1 - 0s - loss: 0.2238 - binary_accuracy: 1.0000 - 6ms/epoch - 6ms/st
ep
Epoch 24/100
1/1 - 0s - loss: 0.2232 - binary_accuracy: 1.0000 - 7ms/epoch - 7ms/st
ep
Epoch 25/100
1/1 - 0s - loss: 0.2225 - binary_accuracy: 1.0000 - 4ms/epoch - 4ms/st
ep
Epoch 26/100
1/1 - 0s - loss: 0.2219 - binary_accuracy: 1.0000 - 7ms/epoch - 7ms/st
ep
Epoch 27/100
1/1 - 0s - loss: 0.2212 - binary_accuracy: 1.0000 - 6ms/epoch - 6ms/st
ep
```

```
In [34]:
```

```python
#NOR gate
# the four different states of the NOR gate
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# the four expected results in the same order
target_data = np.array([[1],[0],[0],[0]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
1/1 - 0s - loss: 0.2441 - binary_accuracy: 0.2500 - 20ms/epoch - 20ms/
step
Epoch 5/100
1/1 - 0s - loss: 0.2432 - binary_accuracy: 0.2500 - 15ms/epoch - 15ms/
step
Epoch 6/100
1/1 - 0s - loss: 0.2424 - binary_accuracy: 0.2500 - 8ms/epoch - 8ms/st
ep
Epoch 7/100
1/1 - 0s - loss: 0.2415 - binary_accuracy: 0.2500 - 8ms/epoch - 8ms/st
ep
Epoch 8/100
1/1 - 0s - loss: 0.2406 - binary_accuracy: 0.2500 - 6ms/epoch - 6ms/st
ep
Epoch 9/100
1/1 - 0s - loss: 0.2398 - binary_accuracy: 0.2500 - 8ms/epoch - 8ms/st
ep
Epoch 10/100
1/1 - 0s - loss: 0.2389 - binary_accuracy: 0.2500 - 4ms/epoch - 4ms/st
ep
```

In [35]:

```
#XOR gate
# the four different states of the XOR gate
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# the four expected results in the same order
target_data = np.array([[0],[1],[1],[0]], "float32")
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=100, verbose=2)
print(model.predict(training_data).round())
```

```
Epoch 1/100
1/1 - 1s - loss: 0.2435 - binary_accuracy: 0.7500 - 1s/epoch - 1s/step
Epoch 2/100
1/1 - 0s - loss: 0.2430 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/st
ep
Epoch 3/100
1/1 - 0s - loss: 0.2425 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/st
ep
Epoch 4/100
1/1 - 0s - loss: 0.2419 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/st
ep
Epoch 5/100
1/1 - 0s - loss: 0.2414 - binary_accuracy: 0.7500 - 7ms/epoch - 7ms/st
ep
Epoch 6/100
1/1 - 0s - loss: 0.2409 - binary_accuracy: 0.7500 - 4ms/epoch - 4ms/st
ep
Epoch 7/100
1/1 - 0s - loss: 0.2403 - binary_accuracy: 0.7500 - 6ms/epoch - 6ms/st
```

In [ ]: