ROLL NO: 225229128

NAME: Princy A

Lab2. Design of Logic Gates using Perceptron and Keras

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

Step1: Define helper functions

In [1]:
```
import pandas as pd
import numpy as np
import math
```

In [2]:
```
def sigmoid(x):
    s=1/(1+math.exp(-x))
    return s
```

In [3]:
```
def logic_gate(w1,w2,b):
    return lambda x1,x2: sigmoid(w1*x1+w2*x2+b)
```

In [4]:
```
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 [5]:
```
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

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 [6]:
```
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 [7]:
```
nor_gate = logic_gate(-20,-20,10)
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 [8]:
```
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 [9]:
```
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 [10]:
```
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 [11]:
```
!pip install keras
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: keras in c:\users\2mscdsa45\appdata\roaming\py
thon\python39\site-packages (2.13.1)
```

!pip install tensorflow

In [12]:
```
import tensorflow as tf
from tensorflow import keras
```

```
C:\ProgramData\Anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarnin
g: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy
(detected version 1.24.3
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

In [17]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In [18]:
```
#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())
```

```
Epoch 1/100
1/1 - 1s - loss: 0.2568 - binary_accuracy: 0.5000 - 706ms/epoch - 706ms/st
ep
Epoch 2/100
1/1 - 0s - loss: 0.2560 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 3/100
1/1 - 0s - loss: 0.2552 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 4/100
1/1 - 0s - loss: 0.2544 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 5/100
1/1 - 0s - loss: 0.2536 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 6/100
1/1 - 0s - loss: 0.2528 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 7/100
1/1 - 0s - loss: 0.2520 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 8/100
1/1 - 0s - loss: 0.2512 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 9/100
1/1 - 0s - loss: 0.2504 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
```

In [19]:
```
#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())
```

```
Epoch 1/100
1/1 - 0s - loss: 0.3276 - binary_accuracy: 0.2500 - 433ms/epoch - 433ms/st
ep
Epoch 2/100
1/1 - 0s - loss: 0.3252 - binary_accuracy: 0.0000e+00 - 4ms/epoch - 4ms/st
ep
Epoch 3/100
1/1 - 0s - loss: 0.3229 - binary_accuracy: 0.0000e+00 - 4ms/epoch - 4ms/st
ep
Epoch 4/100
1/1 - 0s - loss: 0.3206 - binary_accuracy: 0.0000e+00 - 6ms/epoch - 6ms/st
ep
Epoch 5/100
1/1 - 0s - loss: 0.3184 - binary_accuracy: 0.0000e+00 - 4ms/epoch - 4ms/st
ep
Epoch 6/100
1/1 - 0s - loss: 0.3161 - binary_accuracy: 0.0000e+00 - 5ms/epoch - 5ms/st
ep
Epoch 7/100
```

In [20]:
```
#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())
```

```
Epoch 1/100
1/1 - 0s - loss: 0.2312 - binary_accuracy: 0.5000 - 480ms/epoch - 480ms/st
ep
Epoch 2/100
1/1 - 0s - loss: 0.2307 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 3/100
1/1 - 0s - loss: 0.2303 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 4/100
1/1 - 0s - loss: 0.2298 - binary_accuracy: 1.0000 - 5ms/epoch - 5ms/step
Epoch 5/100
1/1 - 0s - loss: 0.2294 - binary_accuracy: 1.0000 - 5ms/epoch - 5ms/step
Epoch 6/100
1/1 - 0s - loss: 0.2289 - binary_accuracy: 1.0000 - 4ms/epoch - 4ms/step
Epoch 7/100
1/1 - 0s - loss: 0.2284 - binary_accuracy: 1.0000 - 4ms/epoch - 4ms/step
Epoch 8/100
1/1 - 0s - loss: 0.2280 - binary_accuracy: 1.0000 - 4ms/epoch - 4ms/step
Epoch 9/100
1/1 - 0s - loss: 0.2275 - binary_accuracy: 1.0000 - 4ms/epoch - 4ms/step
```

In [21]:
```
#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())
```

```
Epoch 1/100
1/1 - 0s - loss: 0.2500 - binary_accuracy: 0.5000 - 430ms/epoch - 430ms/st
ep
Epoch 2/100
1/1 - 0s - loss: 0.2494 - binary_accuracy: 0.7500 - 5ms/epoch - 5ms/step
Epoch 3/100
1/1 - 0s - loss: 0.2489 - binary_accuracy: 0.7500 - 7ms/epoch - 7ms/step
Epoch 4/100
1/1 - 0s - loss: 0.2483 - binary_accuracy: 0.7500 - 4ms/epoch - 4ms/step
Epoch 5/100
1/1 - 0s - loss: 0.2478 - binary_accuracy: 0.7500 - 7ms/epoch - 7ms/step
Epoch 6/100
1/1 - 0s - loss: 0.2472 - binary_accuracy: 0.7500 - 5ms/epoch - 5ms/step
Epoch 7/100
1/1 - 0s - loss: 0.2467 - binary_accuracy: 0.7500 - 5ms/epoch - 5ms/step
Epoch 8/100
1/1 - 0s - loss: 0.2462 - binary_accuracy: 0.7500 - 5ms/epoch - 5ms/step
Epoch 9/100
1/1 - 0s - loss: 0.2456 - binary_accuracy: 0.7500 - 3ms/epoch - 3ms/step
```

In [22]:
```
#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())
```

```
Epoch 1/100
1/1 - 0s - loss: 0.3072 - binary_accuracy: 0.0000e+00 - 422ms/epoch - 422m
s/step
Epoch 2/100
1/1 - 0s - loss: 0.3050 - binary_accuracy: 0.0000e+00 - 4ms/epoch - 4ms/st
ep
Epoch 3/100
1/1 - 0s - loss: 0.3028 - binary_accuracy: 0.0000e+00 - 5ms/epoch - 5ms/st
ep
Epoch 4/100
1/1 - 0s - loss: 0.3007 - binary_accuracy: 0.0000e+00 - 5ms/epoch - 5ms/st
ep
Epoch 5/100
1/1 - 0s - loss: 0.2985 - binary_accuracy: 0.0000e+00 - 4ms/epoch - 4ms/st
ep
Epoch 6/100
1/1 - 0s - loss: 0.2964 - binary_accuracy: 0.0000e+00 - 5ms/epoch - 5ms/st
ep
Epoch 7/100
```

In [23]:
```
#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 - 0s - loss: 0.2410 - binary_accuracy: 0.5000 - 426ms/epoch - 426ms/st
ep
Epoch 2/100
1/1 - 0s - loss: 0.2406 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 3/100
1/1 - 0s - loss: 0.2400 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 4/100
1/1 - 0s - loss: 0.2395 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
Epoch 5/100
1/1 - 0s - loss: 0.2390 - binary_accuracy: 0.5000 - 3ms/epoch - 3ms/step
Epoch 6/100
1/1 - 0s - loss: 0.2385 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/step
Epoch 7/100
1/1 - 0s - loss: 0.2380 - binary_accuracy: 0.5000 - 4ms/epoch - 4ms/step
Epoch 8/100
1/1 - 0s - loss: 0.2376 - binary_accuracy: 0.5000 - 6ms/epoch - 6ms/step
Epoch 9/100
1/1 - 0s - loss: 0.2371 - binary_accuracy: 0.5000 - 5ms/epoch - 5ms/step
```

In [ ]: