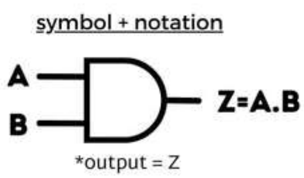
Name: Vishwa R

Reg. No.: 21BAI1772

Perceptron implementation for AND, OR, NAND and XOR Gate

AND Gate



truth table

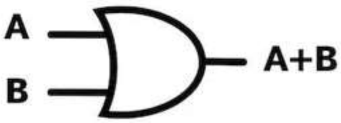
A	B	Z=A.B
0	0	0
0	1	0
1	0	0
1	1	1

```
import numpy as np

def g(Z,b):
  return np.where((np.sum(Z)+b)>0,1,0)

def dense(a_in,W,b):
 Z=np.matmul(a_in,W)
 A_out=g(Z,b)
 return A_out


X1=np.array([[0,0]])
X2=np.array([[0,1]])
X3=np.array([[1,0]])
X4=np.array([[1,1]])
W=np.array([[1],[0.5]])
B=np.array([[-1]])


print("The AND output for (0,0): ",dense(X1,W,B))
print("The AND output for (0,1): ",dense(X2,W,B))
print("The AND output for (1,0): ",dense(X3,W,B))
print("The AND output for (1,1): ",dense(X4,W,B))
```

```
The AND output for (0,0):  [[0]]
The AND output for (0,1):  [[0]]
The AND output for (1,0):  [[0]]
The AND output for (1,1):  [[1]]
```

We get correct outputs for all of the possible inputs when the weights are 1, 0.5 and bias is -1.

OR Gate



2 input OR gate

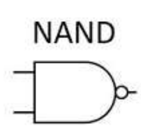
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

```
import numpy as np

def g(Z,b):
  return np.where((np.sum(Z)+b)>0,1,0)

def dense(a_in,W,b):
 Z=np.matmul(a_in,W)
 A_out=g(Z,b)
 return A_out


X1=np.array([[0,0]])
X2=np.array([[0,1]])
X3=np.array([[1,0]])
X4=np.array([[1,1]])
W=np.array([[1.5],[1.5]])
B=np.array([[-1]])


print("The OR output for (0,0): ",dense(X1,W,B))
print("The OR output for (0,1): ",dense(X2,W,B))
print("The OR output for (1,0): ",dense(X3,W,B))
print("The OR output for (1,1): ",dense(X4,W,B))
```

```
The OR output for (0,0):  [[0]]
The OR output for (0,1):  [[1]]
The OR output for (1,0):  [[1]]
The OR output for (1,1):  [[1]]
```

We get correct outputs for all of the possible inputs when the weights are 1.5, 1.5 and bias is -1.

NAND Gate



A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

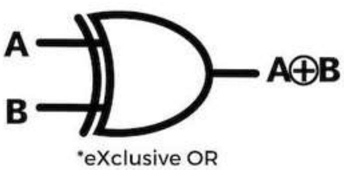
```
import numpy as np

def g(Z,b):
  return np.where((np.sum(Z)+b)>0,1,0)

def dense(a_in,W,b):
 Z=np.matmul(a_in,W)
 A_out=g(Z,b)
 return A_out
```

```
X1=np.array([[0,0]])
X2=np.array([[0,1]])
X3=np.array([[1,0]])
X4=np.array([[1,1]])
W=np.array([[-0.6],[-0.5]])
B=np.array([[1]])


print("The NAND output for (0,0): ",dense(X1,W,B))
print("The NAND output for (0,1): ",dense(X2,W,B))
print("The NAND output for (1,0): ",dense(X3,W,B))
print("The NAND output for (1,1): ",dense(X4,W,B))
```

```
The NAND output for (0,0):  [[1]]
The NAND output for (0,1):  [[1]]
The NAND output for (1,0):  [[1]]
The NAND output for (1,1):  [[0]]
```

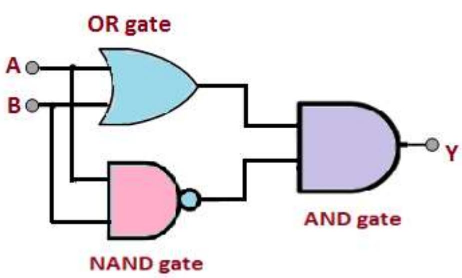
We get correct outputs for all of the possible inputs when the weights are -0.5, -0.6 and bias is 1.

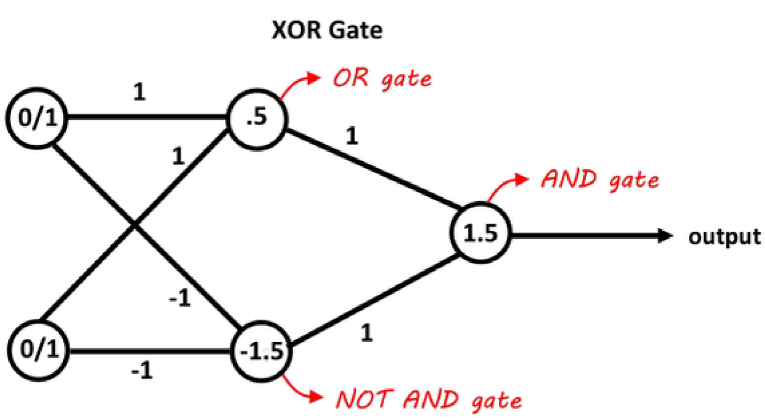
XOR Gate



2 input XOR gate

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0





```
import numpy as np

def g(Z,b):
  return np.where((np.sum(Z)+b)>0 ,1,0)

def dense(a_in,W,b):
 Z=np.matmul(a_in,W)
 A_out=g(Z,b)
 return A_out


X1=np.array([[0,0]])
X2=np.array([[0,1]])
X3=np.array([[1,0]])
X4=np.array([[1,1]])
W_AND=np.array([[1],[0.5]])
B_AND=np.array([[-1]])
W_OR=np.array([[1.5],[1.5]])
B_OR=np.array([[-1]])
W_NAND=np.array([[-0.6],[-0.5]])
B_NAND=np.array([[1]])


X1=np.append(dense(X1,W_OR,B_OR),dense(X1,W_NAND,B_NAND))
X2=np.append(dense(X2,W_OR,B_OR),dense(X2,W_NAND,B_NAND))
X3=np.append(dense(X3,W_OR,B_OR),dense(X3,W_NAND,B_NAND))
X4=np.append(dense(X4,W_OR,B_OR),dense(X4,W_NAND,B_NAND))
print(X1,X2,X3,X4)
```

```
[0 1] [1 1] [1 1] [1 0]
```

```
print("The XOR output for (0,0): ",dense(X1,W_AND,B_AND))
print("The XOR output for (0,1): ",dense(X2,W_AND,B_AND))
print("The XOR output for (1,0): ",dense(X3,W_AND,B_AND))
print("The XOR output for (1,1): ",dense(X4,W_AND,B_AND))
```

```
The XOR output for (0,0):  [[0]]
The XOR output for (0,1):  [[1]]
The XOR output for (1,0):  [[1]]
The XOR output for (1,1):  [[0]]
```

We get correct outputs for all of the possible inputs when we combine AND, OR and NAND gates.

Perceptron with Back Propagation AND gate

```
lr=0.5
X=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([[0],[0],[0],[1]])
W=np.array([[1.2],[0.6]])
B=np.array([[-1]])
def g(Z):
  return np.where((Z)>0,1,0)

def dense(a_in,W,b):
 Z=np.matmul(a_in,W)+b
 A_out=g(Z)
 return A_out


y_pred=dense(X,W,B)
print(y_pred)
```

```
[[0]
 [0]
 [1]
 [1]]
```

Since the 3rd row doesn't satisfy the condition, We update the weights using that row.

```
del_W_0=np.array(lr*(y[2]-y_pred[2])*X[2][0])
W[0]=W[0]+del_W_0
del_W_1=np.array(lr*(y[2]-y_pred[2])*X[2][1])
W[1]=W[1]+del_W_1
```

```
print(W)
```

```
[[0.7]
 [0.6]]
```

Now we can test each row and see if they satisfy the condition.

```
X1=np.array([[0,0]])
X2=np.array([[0,1]])
X3=np.array([[1,0]])
X4=np.array([[1,1]])
```

```
print("The AND output for (0,0): ",dense(X1,W,B))
print("The AND output for (0,1): ",dense(X2,W,B))
print("The AND output for (1,0): ",dense(X3,W,B))
print("The AND output for (1,1): ",dense(X4,W,B))
```

```
The AND output for (0,0):  [[0]]
The AND output for (0,1):  [[0]]
The AND output for (1,0):  [[0]]
The AND output for (1,1):  [[1]]
```

We get correct outputs for all of the possible inputs when the weights are 0l7, 0.6 and bias is -1.

ANN for MNIST classification

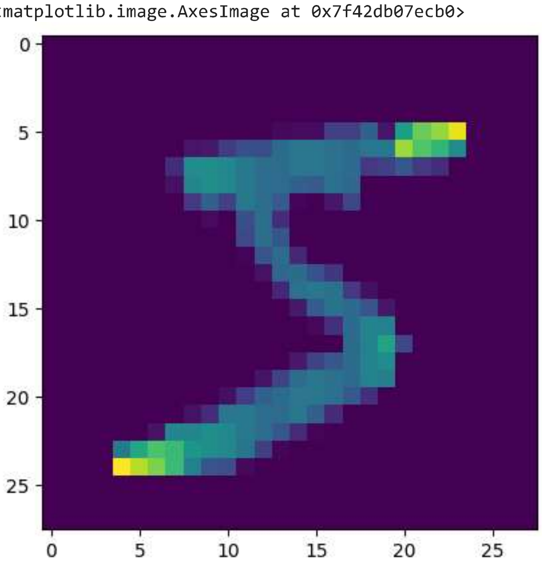
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
```

```
(train_img,train_label),(test_img,test_lab)=mnist.load_data()
```

```
train_img=tf.keras.utils.normalize(train_img,axis=1)
test_img=tf.keras.utils.normalize(test_img,axis=1)
```

```
plt.imshow(train_img[0])
```

```
<matplotlib.image.AxesImage at 0x7f42db07ecb0>
```



```
model=Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(256,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_15 (Dense)            (None, 256)               200960

 dense_16 (Dense)            (None, 256)               65792

 dense_17 (Dense)            (None, 10)                2570

=================================================================
Total params: 269322 (1.03 MB)
Trainable params: 269322 (1.03 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(train_img,train_label,epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2243 - accuracy: 0.9331
Epoch 2/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0876 - accuracy: 0.9729
Epoch 3/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0596 - accuracy: 0.9809
Epoch 4/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0413 - accuracy: 0.9866
Epoch 5/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0329 - accuracy: 0.9890
Epoch 6/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0241 - accuracy: 0.9921
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0216 - accuracy: 0.9928
Epoch 8/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0155 - accuracy: 0.9949
Epoch 9/10
1875/1875 [==============================] - 20s 10ms/step - loss: 0.0164 - accuracy: 0.9945
Epoch 10/10
1875/1875 [==============================] - 18s 9ms/step - loss: 0.0144 - accuracy: 0.9950
<keras.src.callbacks.History at 0x7f42daf86b00>
```

```
print(model.evaluate(test_img,test_lab))
```

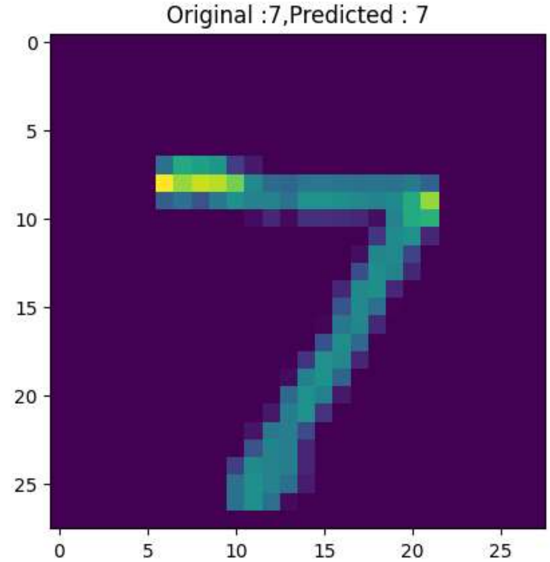
```
313/313 [==============================] - 1s 2ms/step - loss: 0.1023 - accuracy: 0.9783
[0.10231593996286392, 0.9782999753952026]
```

```
from sklearn.metrics import accuracy_score
accuracy_score(p[:10],test_lab[:10])
```

```
1.0
```

```
plt.imshow(test_img[0])
plt.title("Original :{},Predicted : {}".format(test_lab[0],p[0]))
plt.figure
```

```
<function matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, *, facecolor=None, edgecolor=None, frameon=True, FigureClass=
<class 'matplotlib.figure.Figure'>, clear=False, **kwargs)>
```



ANN for Iris species Classification

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
df=pd.read_csv("IRIS.csv")
df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
df['species'] = df['species'].map({'Iris-versicolor': 0, 'Iris-virginica': 1, 'Iris-setosa': 2})
```

```
X = df.drop(['species'], axis=1)
y = df['species']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
model=tf.keras.models.Sequential([
    tf.keras.layers.Dense(64,activation='relu',input_shape=(4,)),
    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dense(3,activation='softmax')])
```

```
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_18 (Dense)            (None, 64)                320

 dense_19 (Dense)            (None, 64)                4160
```

```
 dense_20 (Dense)            (None, 3)                 195

=================================================================
Total params: 4675 (18.26 KB)
Trainable params: 4675 (18.26 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

```
model.fit(X_train_scaled,y_train,epochs=30)
```

```
4/4 [==============================] - 0s 4ms/step - loss: 1.0425 - accuracy: 0.6190
Epoch 3/30
4/4 [==============================] - 0s 4ms/step - loss: 1.0056 - accuracy: 0.6667
Epoch 4/30
4/4 [==============================] - 0s 4ms/step - loss: 0.9732 - accuracy: 0.6571
Epoch 5/30
4/4 [==============================] - 0s 4ms/step - loss: 0.9425 - accuracy: 0.6476
Epoch 6/30
4/4 [==============================] - 0s 4ms/step - loss: 0.9128 - accuracy: 0.6476
Epoch 7/30
4/4 [==============================] - 0s 4ms/step - loss: 0.8828 - accuracy: 0.6476
Epoch 8/30
4/4 [==============================] - 0s 4ms/step - loss: 0.8529 - accuracy: 0.6476
Epoch 9/30
4/4 [==============================] - 0s 4ms/step - loss: 0.8224 - accuracy: 0.6476
Epoch 10/30
4/4 [==============================] - 0s 5ms/step - loss: 0.7912 - accuracy: 0.6476
Epoch 11/30
4/4 [==============================] - 0s 4ms/step - loss: 0.7593 - accuracy: 0.6476
Epoch 12/30
4/4 [==============================] - 0s 4ms/step - loss: 0.7279 - accuracy: 0.6476
Epoch 13/30
4/4 [==============================] - 0s 4ms/step - loss: 0.6963 - accuracy: 0.6476
Epoch 14/30
4/4 [==============================] - 0s 3ms/step - loss: 0.6670 - accuracy: 0.6476
Epoch 15/30
4/4 [==============================] - 0s 3ms/step - loss: 0.6396 - accuracy: 0.6476
Epoch 16/30
4/4 [==============================] - 0s 4ms/step - loss: 0.6126 - accuracy: 0.6476
Epoch 17/30
4/4 [==============================] - 0s 4ms/step - loss: 0.5888 - accuracy: 0.6476
Epoch 18/30
4/4 [==============================] - 0s 4ms/step - loss: 0.5675 - accuracy: 0.6476
Epoch 19/30
4/4 [==============================] - 0s 5ms/step - loss: 0.5494 - accuracy: 0.6476
Epoch 20/30
4/4 [==============================] - 0s 4ms/step - loss: 0.5349 - accuracy: 0.6476
Epoch 21/30
4/4 [==============================] - 0s 5ms/step - loss: 0.5177 - accuracy: 0.6476
Epoch 22/30
4/4 [==============================] - 0s 4ms/step - loss: 0.5048 - accuracy: 0.6667
Epoch 23/30
4/4 [==============================] - 0s 5ms/step - loss: 0.4910 - accuracy: 0.6952
Epoch 24/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4811 - accuracy: 0.7810
Epoch 25/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4707 - accuracy: 0.8762
Epoch 26/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4603 - accuracy: 0.8762
Epoch 27/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4506 - accuracy: 0.8381
Epoch 28/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4408 - accuracy: 0.8571
Epoch 29/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4312 - accuracy: 0.8762
Epoch 30/30
4/4 [==============================] - 0s 4ms/step - loss: 0.4215 - accuracy: 0.8571
<keras.src.callbacks.History at 0x7f42dad40ee0>
```

```
model.evaluate(X_test_scaled,y_test)
```

```
2/2 [==============================] - 0s 10ms/step - loss: 0.3304 - accuracy: 0.9333
[0.3304286599159241, 0.9333333373069763]
```

```
y_prob=model.predict(X_test_scaled)
y_pred=[]
for i in y_prob:
  y_pred.append(np.argmax(i))
```

```
2/2 [==============================] - 0s 6ms/step
```

```
nn_confusion_matrix = confusion_matrix(y_test, y_pred)
nn_confusion_matrix
```

```
array([[10,  3,  0],
       [ 0, 13,  0],
       [ 0,  0, 19]])
```

Our model is doing very well. It only predicts 3 records wrongly