**Assignment 1**

**Code:-**

import numpy as np

import matplotlib.pyplot as plt

# Binary Step Activation Funciton

# Returns 0 if less than 0, other wise 1

def binaryStep(x):

    return np.heaviside(x,1)

x = np.linspace(-10, 10)

plt.plot(x, binaryStep(x))

plt.title('binaryStep Activation Function')

plt.show()

# Linear Activation function

def linearFun(x):

    return x

x = np.linspace(-10,10)

plt.plot(x, linearFun(x))

plt.title("Linear Activation Function")

plt.show()

# Sigmoid Activation Funciton

# 1/(1+exp(-x))

def sigmoidFun(x):

    return 1 / (1+np.exp(-x))

x = np.linspace(-10,10)

plt.plot(x, sigmoidFun(x))

plt.title("Sigmoid Activation Function")

plt.show()

# Tanh Activation Function

# (1-exp(-2x))/(1+exp(-2x))

def tanhFun(x):

    return np.tanh(x)

x = np.linspace(-10,10)

plt.plot(x, tanhFun(x))

plt.axis("tight")

plt.title("Tanh Activation Function")

plt.show()

# Relu Activation function

# It returns zero if the input is less than zero otherwise it returns the given input

def ReluFun(x):

    x1=[]

    for i in x:

        if i<0:

            x1.append(0)

        else:

            x1.append(i)

    return x1

x = np.linspace(-10,10)

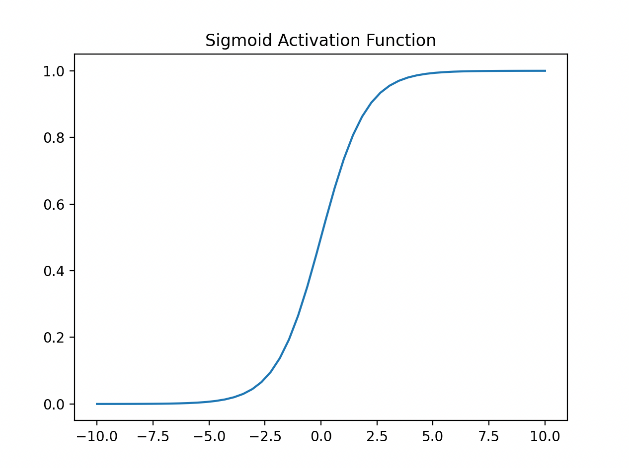
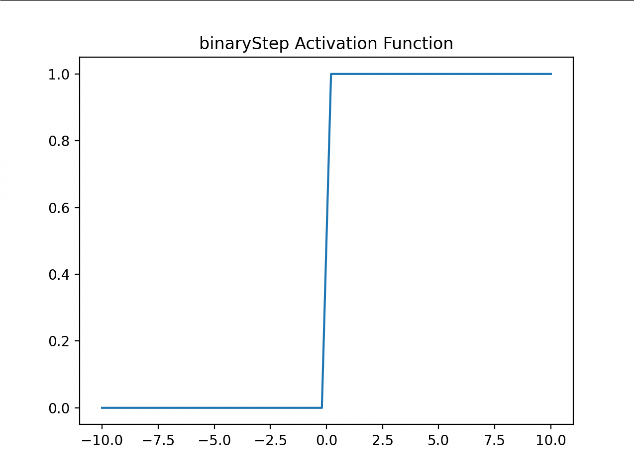
plt.plot(x, ReluFun(x))

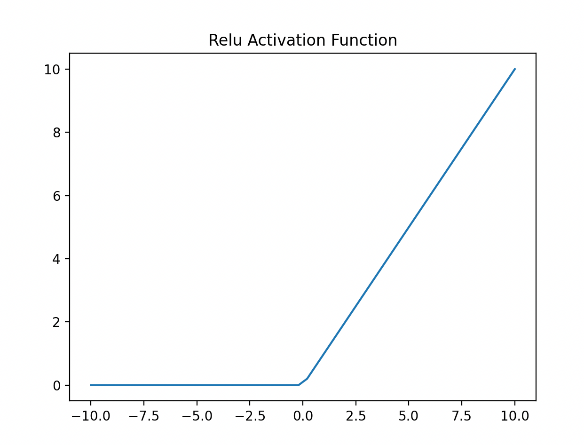
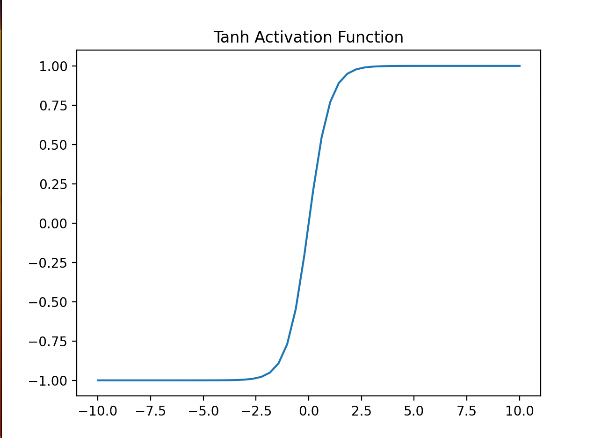
plt.axis("tight")

plt.title("Relu Activation Function")

plt.show()

**Output :-**





**Assigment 2**

**Code:**

def mp\_neuron(x, w, theta):

# Calculate the net input

net\_input = sum([x[i]\*w[i] for i in range(len(x))])

# Apply the activation function (Heaviside step function)

if net\_input >= theta:

return 1

else:

return 0

# Define the NAND function

def nand(x1, x2):

# Set the weights and threshold

w1, w2, theta = -2, -2, -3

# Input the values into the McCulloch-Pitts neuron

x = [x1, x2]

output = mp\_neuron(x, [w1, w2], theta)

return output

print(nand(1, 1))

**Output:**

0

**Assignment 4**

import numpy as np

import matplotlib.pyplot as plt

# Define the training data

training\_data = np.array([

[1.2, 3.7, 1],

[2.3, 4.2, 1],

[3.1, 5.0, 1],

[4.4, 1.8, -1],

[5.5, 2.4, -1],

[5.7, 1.9, -1],

])

# Separate the inputs (features) from the targets (class labels)

inputs = training\_data[:, :2]

targets = training\_data[:, 2:]

# Define the perceptron function

def perceptron(inputs, targets, learning\_rate, num\_epochs):

num\_inputs = inputs.shape[1]

num\_outputs = targets.shape[1]

# Initialize the weights and bias to small random values

weights = np.random.randn(num\_inputs, num\_outputs)

bias = np.random.randn(num\_outputs)

# Iterate over the training data for the specified number of epochs

for epoch in range(num\_epochs):

# Iterate over each training example

for input\_, target in zip(inputs, targets):

# Compute the output of the perceptron for the current input

output = np.dot(input\_, weights) + bias

# Compute the error (difference between target and output)

error = target - output

# Update the weights and bias according to the perceptron learning law

weights += learning\_rate \* np.outer(input\_, error)

bias += learning\_rate \* error

return weights, bias

# Train the perceptron on the training data

weights, bias = perceptron(inputs, targets, learning\_rate=0.1, num\_epochs=100)

# Plot the training data and the decision boundary

x\_min, x\_max = inputs[:, 0].min() - 1, inputs[:, 0].max() + 1

y\_min, y\_max = inputs[:, 1].min() - 1, inputs[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.01), np.arange(y\_min, y\_max, 0.01))

Z = np.sign(np.dot(np.c\_[xx.ravel(), yy.ravel()], weights) + bias)

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.4)

plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0], cmap=plt.cm.bwr)

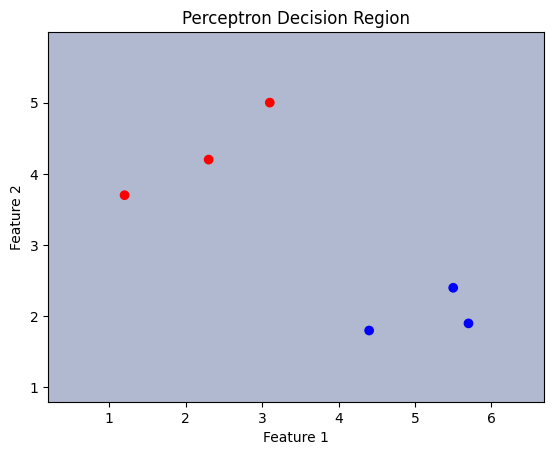
plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Perceptron Decision Region')

plt.show()

**Output:**



**Assignment 7**

**Code:**

import numpy as np

def sigmoid(x):

    """sigmoid activation function"""

    return 1/(1+np.exp(-x))

def sigmoid\_derivative(x):

    """derivative of sigmoid activation function"""

    return x\*(1-x)

# input dataset

X = np.array([[0,0,1],

              [0,1,1],

              [1,0,1],

              [1,1,1]])

# output dataset

y = np.array([[0],

              [1],

              [1],

              [0]])

# set random seed

np.random.seed(1)

# initialize weights randomly with mean 0

syn0 = 2\*np.random.random((3,4)) - 1

syn1 = 2\*np.random.random((4,1)) - 1

def forward\_propagation(X, syn0, syn1):

    """forward propagation function"""

    layer0 = X

    layer1 = sigmoid(np.dot(layer0, syn0))

    layer2 = sigmoid(np.dot(layer1, syn1))

    return layer0, layer1, layer2

def back\_propagation(layer0, layer1, layer2, y, syn1, syn0, learning\_rate):

    """back propagation function"""

    layer2\_error = y - layer2

    layer2\_delta = layer2\_error \* sigmoid\_derivative(layer2)

    layer1\_error = layer2\_delta.dot(syn1.T)

    layer1\_delta = layer1\_error \* sigmoid\_derivative(layer1)

    syn1 += learning\_rate \* layer1.T.dot(layer2\_delta)

    syn0 += learning\_rate \* layer0.T.dot(layer1\_delta)

    return syn0, syn1, layer2\_error

# set learning rate and number of iterations

learning\_rate = 0.1

iterations = 60000

for i in range(iterations):

    # forward propagation

    layer0, layer1, layer2 = forward\_propagation(X, syn0, syn1)

    # back propagation

    syn0, syn1, layer2\_error = back\_propagation(layer0, layer1, layer2, y, syn1, syn0, learning\_rate)

    # print error every 10000 iterations

    if i % 10000 == 0:

        print("Error: " + str(np.mean(np.abs(layer2\_error))))

**Output:**

Error: 0.4964100319027255

Error: 0.042888017000115755

Error: 0.02409899422852161

Error: 0.018110652146797853

Error: 0.014987616272210909

Error: 0.013014490538142583

**Assignment 8**

**Code:**

import pandas as pd

import tensorflow as tf

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

iris = load\_iris()

X = iris.data

y = iris.target

scaler = StandardScaler()

X = scaler.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_train, y\_train, test\_size=0.2, random\_state=42)

model = Sequential([

Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),

Dense(32, activation='relu'),

Dense(3, activation='softmax')

])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

history = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val))

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print('Test loss:', test\_loss)

print('Test accuracy:', test\_acc)

**Output:**

Test loss: 0.14251665771007538

Test accuracy: 0.9333333373069763

**Assignment B1**

import numpy as np

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

def binary\_cross\_entropy(y\_true, y\_pred):

    return -np.mean(y\_true \* np.log(y\_pred) + (1 - y\_true) \* np.log(1 - y\_pred))

# XOR input and output data

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

outputs = np.array([[0], [1], [1], [0]])

# Initialize weights and biases

np.random.seed(42)

input\_layer\_neurons = 2

hidden\_layer\_neurons = 2

output\_layer\_neurons = 1

hidden\_weights = np.random.uniform(size=(input\_layer\_neurons, hidden\_layer\_neurons))

hidden\_bias = np.random.uniform(size=(1, hidden\_layer\_neurons))

output\_weights = np.random.uniform(size=(hidden\_layer\_neurons, output\_layer\_neurons))

output\_bias = np.random.uniform(size=(1, output\_layer\_neurons))

# Hyperparameters

learning\_rate = 0.5

epochs = 50000

# Training the network

for epoch in range(epochs):

    # Forward pass

    hidden\_layer\_input = np.dot(inputs, hidden\_weights) + hidden\_bias

    hidden\_layer\_output = sigmoid(hidden\_layer\_input)

    output\_layer\_input = np.dot(hidden\_layer\_output, output\_weights) + output\_bias

    predictions = sigmoid(output\_layer\_input)

    # Compute the error

    error = outputs - predictions

    # Backpropagation

    output\_delta = error \* sigmoid\_derivative(predictions)

    hidden\_delta = output\_delta.dot(output\_weights.T) \* sigmoid\_derivative(hidden\_layer\_output)

    # Update weights and biases

    output\_weights += hidden\_layer\_output.T.dot(output\_delta) \* learning\_rate

    output\_bias += np.sum(output\_delta, axis=0, keepdims=True) \* learning\_rate

    hidden\_weights += inputs.T.dot(hidden\_delta) \* learning\_rate

    hidden\_bias += np.sum(hidden\_delta, axis=0, keepdims=True) \* learning\_rate

    # Print loss at every 5000 epochs

    if epoch % 5000 == 0:

        loss = binary\_cross\_entropy(outputs, predictions)

        print(f"Epoch {epoch}: Loss {loss}")

# Test the trained network

print("\nTrained XOR neural network outputs:")

for i in range(len(inputs)):

    print(f"Input: {inputs[i]} -> Output: {predictions[i]}")

**Output:**

Epoch 0: Loss 0.8725511895307583

Epoch 5000: Loss 0.026277935522086827

Epoch 10000: Loss 0.01723706961771534

Epoch 15000: Loss 0.013684583220941508

Epoch 20000: Loss 0.011669435992336757

Epoch 25000: Loss 0.01033299864828558

Epoch 30000: Loss 0.009365096893987802

Epoch 35000: Loss 0.008623155665348624

Epoch 40000: Loss 0.00803138014781044

Epoch 45000: Loss 0.0075453093964910935

Trained XOR neural network outputs:

Input: [0 0] -> Output: [0.00799187]

Input: [0 1] -> Output: [0.99323485]

Input: [1 0] -> Output: [0.99322976]

Input: [1 1] -> Output: [0.00691861]

**Assignment B3**

import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Dense

from tensorflow.keras.models import Sequential

model = Sequential([

Dense(10, activation='relu', input\_shape=(4,)),

Dense(8, activation='relu'),

Dense(3, activation='softmax')

])

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

X\_train = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

y\_train = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

history = model.fit(X\_train, y\_train, epochs=100)

X\_new = np.array([[2, 4, 6, 8], [10, 12, 14, 16]])

y\_pred = model.predict(X\_new)

print(y\_pred)

**Output:**

[[0.06911165 0.25591102 0.6749773 ]

[0.02979936 0.27925494 0.6909457 ]]

**Assignment B4**

import numpy as np

def to\_bipolar(vec):

    print(np.array([1 if elem > 0 else -1 for elem in vec]))

    return np.array([1 if elem > 0 else -1 for elem in vec])

def hebbian\_learning\_rule(vectors):

    n = vectors.shape[1]

    weight\_matrix = np.zeros((n, n))

    for vec in vectors:

        weight\_matrix += np.outer(vec, vec)

    np.fill\_diagonal(weight\_matrix, 0)

    print("weight\_matrix",weight\_matrix)

    return weight\_matrix

def synchronous\_update(weight\_matrix, input\_vector):

    return np.sign(np.dot(weight\_matrix, input\_vector))

def is\_recall\_success(weight\_matrix, input\_vector):

    output\_vector = synchronous\_update(weight\_matrix, input\_vector)

    return np.array\_equal(input\_vector, output\_vector)

if \_\_name\_\_ == "\_\_main\_\_":

    input\_vectors = np.array([

        to\_bipolar([0, 0, 0, 1, 1]),

        to\_bipolar([0, 1, 0, 1, 0]),

        to\_bipolar([0, 1, 0, 0, 1]),

        to\_bipolar([0, 0, 0, 1, 1]),

    ])

    weight\_matrix = hebbian\_learning\_rule(input\_vectors)

    for i, vec in enumerate(input\_vectors):

        if is\_recall\_success(weight\_matrix, vec):

            print(f"Input vector {i+1} is successfully recalled.")

        else:

            print(f"Input vector {i+1} is not successfully recalled.")

**Output:**

[-1 -1 -1 1 1]

[-1 1 -1 1 -1]

[-1 1 -1 -1 1]

[-1 -1 -1 1 1]

weight\_matrix [[ 0. 0. 4. -2. -2.]

[ 0. 0. 0. -2. -2.]

[ 4. 0. 0. -2. -2.]

[-2. -2. -2. 0. 0.]

[-2. -2. -2. 0. 0.]]

Input vector 1 is successfully recalled.

Input vector 2 is not successfully recalled.

Input vector 3 is not successfully recalled.

Input vector 4 is successfully recalled.

**Assignment B5**

import time

import numpy as np

import cv2

model\_config = "yolov3.cfg"

model\_weights = "yolov3.weights"

model = cv2.dnn.readNetFromDarknet(model\_config, model\_weights)

min\_probability = 0.5

image = cv2.imread('image.jpeg')

image = cv2.resize(image, (416, 416))

height, width, channels = image.shape

blob = cv2.dnn.blobFromImage(image, 1/255.0, (416, 416), swapRB=True, crop=False)

model.setInput(blob)

output\_layer\_names = model.getUnconnectedOutLayersNames()

start\_time = time.time()

outputs = model.forward(output\_layer\_names)

end\_time = time.time()

print('Time taken: {} seconds'.format(end\_time - start\_time))

boxes = []

confidences = []

class\_ids = []

for output in outputs:

for detection in output:

scores = detection[5:]

class\_id = np.argmax(scores)

confidence = scores[class\_id]

if confidence > min\_probability:

center\_x = int(detection[0] \* width)

center\_y = int(detection[1] \* height)

w = int(detection[2] \* width)

h = int(detection[3] \* height)

x = int(center\_x - w/2)

y = int(center\_y - h/2)

boxes.append([x, y, w, h])

confidences.append(float(confidence))

class\_ids.append(class\_id)

indices = cv2.dnn.NMSBoxes(boxes, confidences, min\_probability, 0.4)

print(indices)

if len(indices) > 0:

for i in indices:

box = boxes[i]

x = box[0]

y = box[1]

w = box[2]

h = box[3]

label = str(class\_ids[i])

score = confidences[i]

print('Detected object with label {} and score {}'.format(label, score))

color = (0, 255, 0)

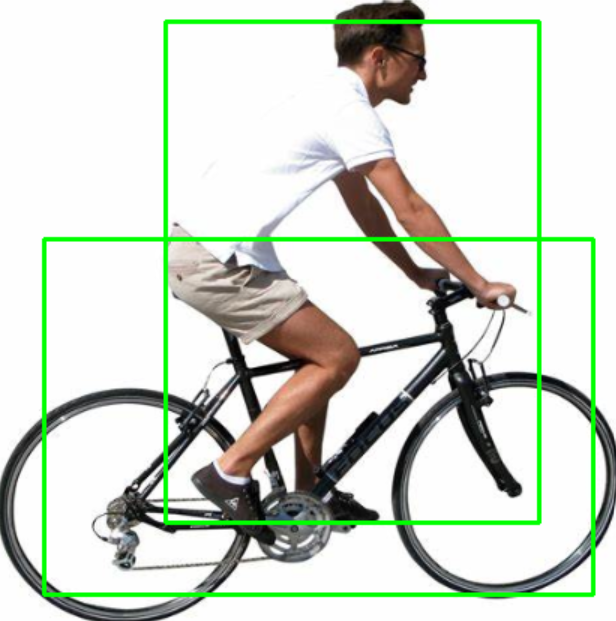
cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)

cv2.imshow('Object Detection', image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output:**



**Assignment C1**

import tensorflow as tf

from tensorflow import keras

import numpy as np

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

model = keras.Sequential([

keras.layers.Flatten(input\_shape=(28, 28)),

keras.layers.Dense(1, activation='relu'), #logistic regression

keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

y\_pred = np.argmax(model.predict(x\_test), axis=1)

from sklearn.metrics import accuracy\_score

accuracy\_score(y\_test, y\_pred)

**Output:**

0.3979

**Assignment C2**

import tensorflow as tf

from tensorflow import keras

import numpy as np

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

model = keras.Sequential([

keras.layers.Reshape((28, 28, 1), input\_shape=(28, 28)),

keras.layers.Conv2D(2, (3,3), padding='same', input\_shape=(28, 28)),

keras.layers.Flatten(),

keras.layers.Dense(128, activation='relu'),

keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

y\_pred = np.argmax(model.predict(x\_test), axis=1)

from sklearn.metrics import accuracy\_score

accuracy\_score(y\_test, y\_pred)

**Output:**

0.9759

**Assignment C4**

import tensorflow as tf

from tensorflow import keras

import numpy as np

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

model = keras.Sequential([

keras.layers.Flatten(input\_shape=(28, 28)),

keras.layers.Dense(128, activation='relu'),

keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

y\_pred = np.argmax(model.predict(x\_test), axis=1)

from sklearn.metrics import accuracy\_score

accuracy\_score(y\_test, y\_pred)

Output :-

0.9762

**Assignment 3**

import tensorflow as tf

import numpy as np

# Define the training data

training\_inputs = np.array([

[1, -1, 1, -1, 1, 1, 1, -1, 1, 1], # 0

[1, 1, -1, -1, 1, 1, 1, -1, 1, 1], # 1

[1, -1, 1, 1, -1, 1, 1, -1, 1, 1], # 2

[1, -1, 1, -1, -1, -1, 1, -1, 1, -1], # 3

[1, 1, -1, -1, -1, -1, 1, -1, 1, -1], # 4

[1, -1, -1, 1, -1, -1, 1, -1, 1, 1], # 5

[1, -1, -1, 1, -1, 1, 1, -1, 1, 1], # 6

[1, -1, 1, -1, -1, -1, 1, -1, 1, -1], # 7

[1, -1, 1, -1, -1, 1, 1, -1, 1, 1], # 8

[1, -1, 1, -1, -1, -1, 1, -1, 1, 1], # 9

])

# Define the target outputs (1 for even, 0 for odd)

target\_outputs = np.array([

[1],

[0],

[1],

[0],

[0],

[1],

[1],

[0],

[1],

[0],

])

# Define the model

model = tf.keras.Sequential([

tf.keras.layers.Dense(10, input\_dim=10, activation='relu'),

tf.keras.layers.Dense(1, activation='sigmoid'),

])

# Compile the model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(training\_inputs, target\_outputs, epochs=100)

# Test the model

test\_inputs = np.array([

[1, -1, 1, -1, -1, -1, 1, -1, 1, 1], # odd number

[1, -1, 1, -1, -1, -1, 1, -1, 1, -1], # even number

])

predictions = model.predict(test\_inputs)

for i in range(len(predictions)):

if predictions[i] >= 0.5:

print(f"{i+1} is even")

else:

print(f"{i+1} is odd")

**Output:**

1 is odd

2 is odd