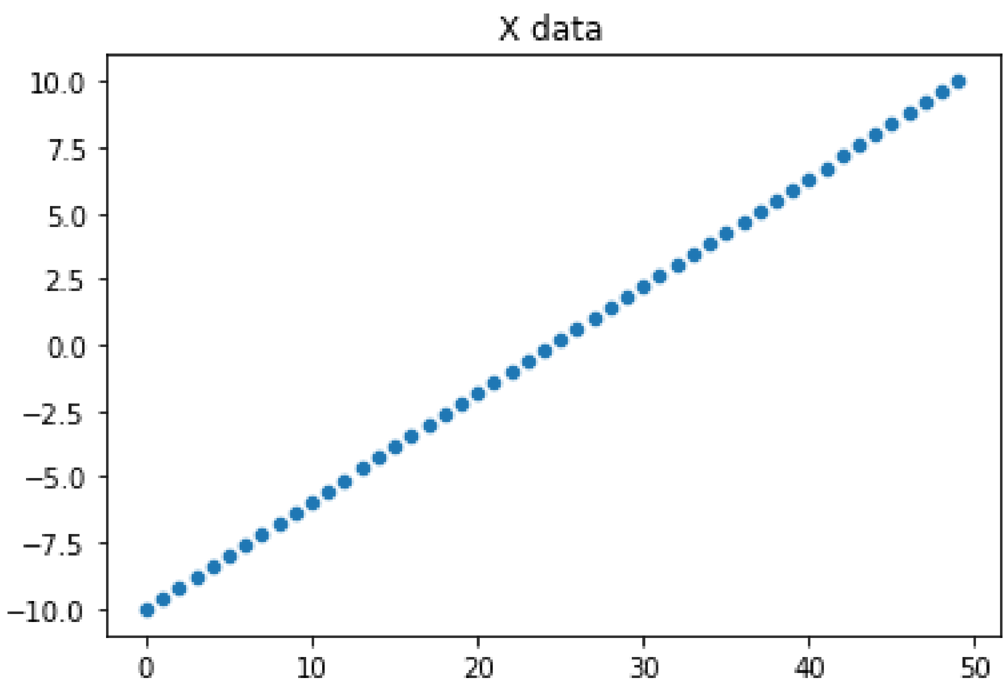
Lab Assignment No. 1

#Sigmoid Function

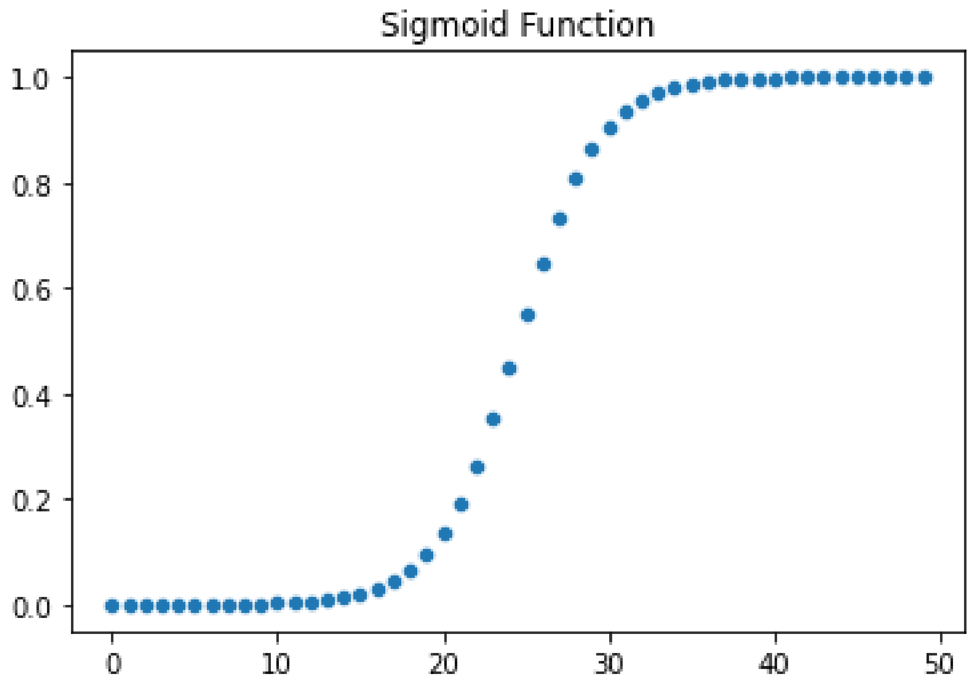
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

def sigmoid_function(x):
  return 1/(1+np.exp(-x))

x = np.linspace(-10, 10)

sns.scatterplot(x)
plt.title("X data")
```

```
Text(0.5, 1.0, 'X data')
```



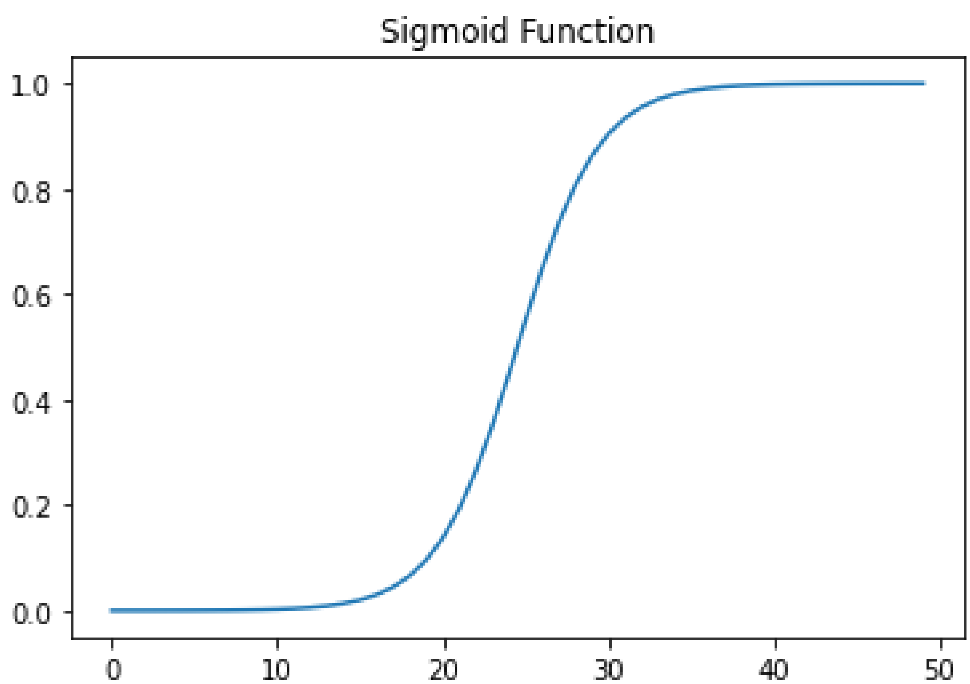

```
y = sigmoid_function(x)

sns.scatterplot(y)
plt.title("Sigmoid Function")
```

```
Text(0.5, 1.0, 'Sigmoid Function')
```



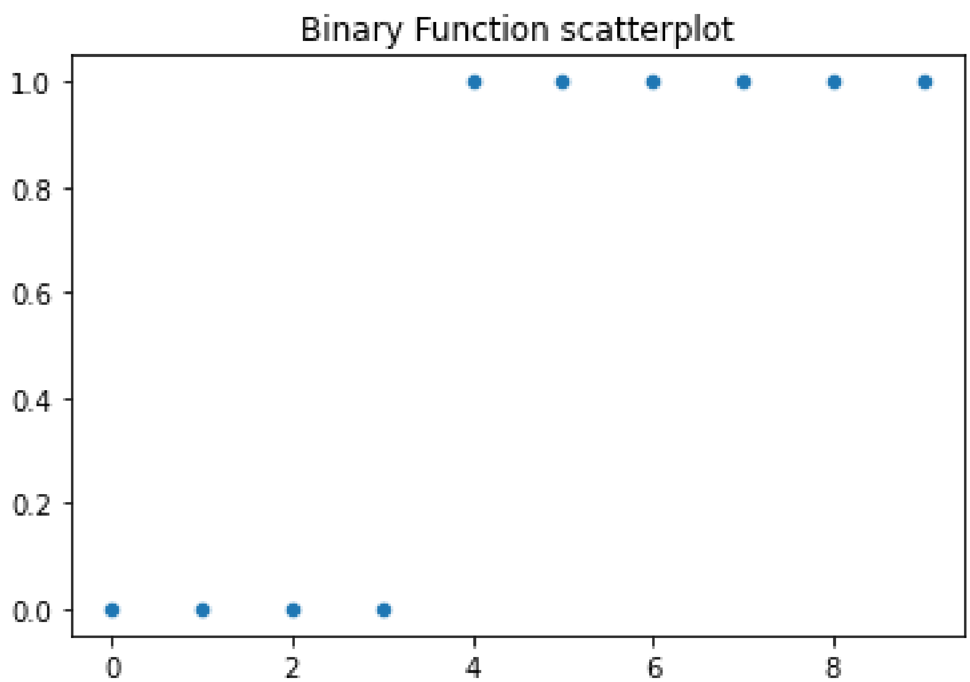

```
plt.plot(y)
plt.title("Sigmoid Function")
```

```
Text(0.5, 1.0, 'Sigmoid Function')
```

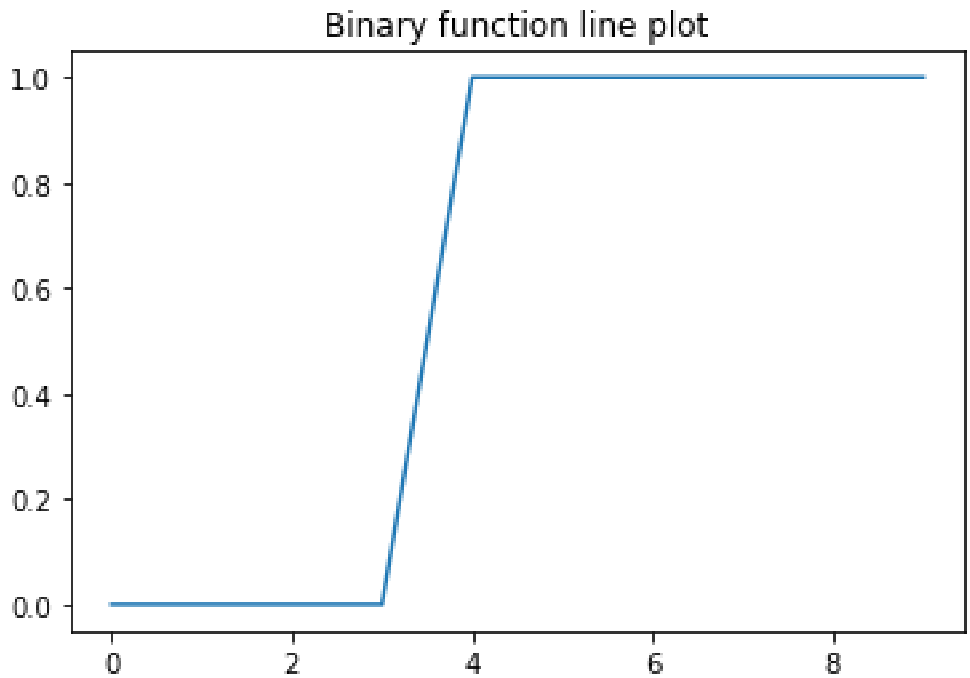


#Binary function

```
def binary_function(x):
  return 0 if x<0 else 1

binary_output = []
for i in [-10, -8, -5, -2, 0, 1, 2, 3, 8, 10]:
  binary_output.append(binary_function(i))

binary_output

[0, 0, 0, 0, 1, 1, 1, 1, 1, 1]

sns.scatterplot(binary_output)
plt.title("Binary Function scatterplot")

Text(0.5, 1.0, 'Binary Function scatterplot')
```

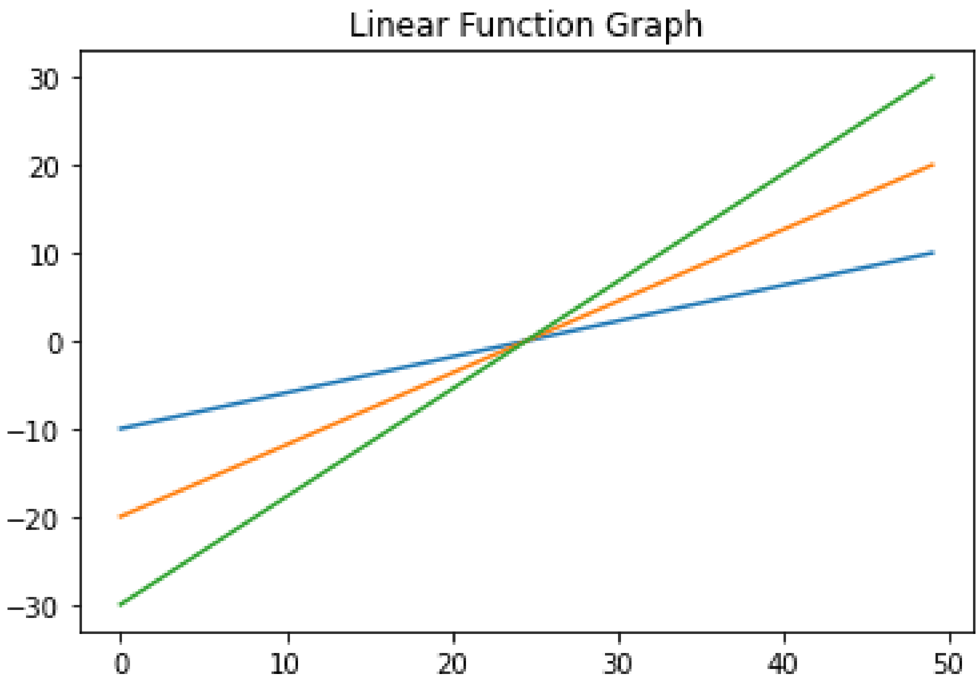



```
plt.plot(binary_output)
plt.title("Binary function line plot")

Text(0.5, 1.0, 'Binary function line plot')
```



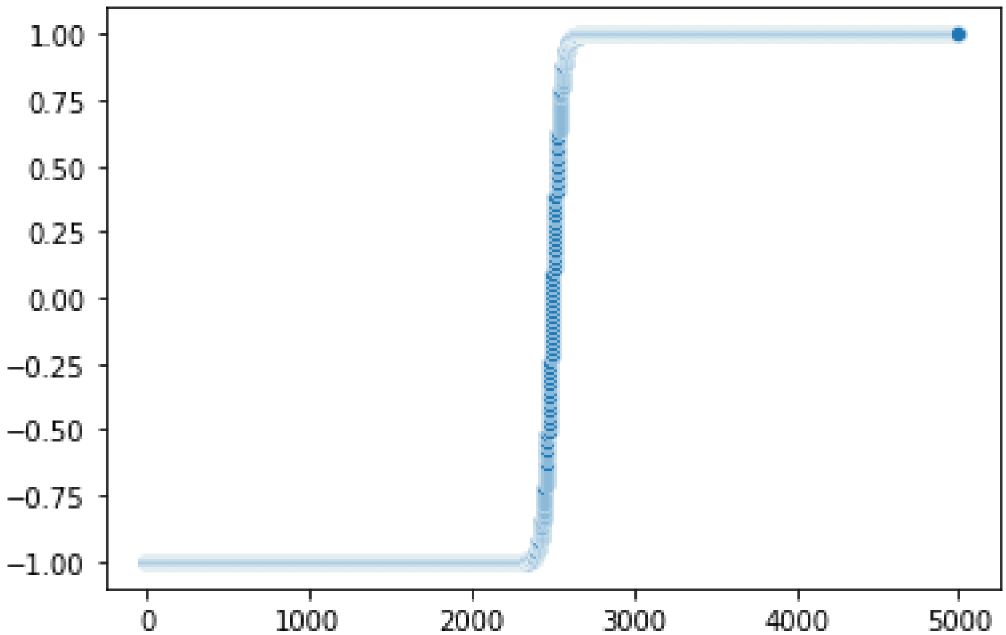
#Linear Function

```
def linear_function(a, x):
  return a*x;

x = np.linspace(-10, 10)

y1 = [linear_function(1, i) for i in x]

y2 = [linear_function(2, i) for i in x]

y3 = [linear_function(3, i) for i in x]

plt.plot(y1)
plt.plot(y2)
plt.plot(y3)
plt.title("Linear Function Graph")

Text(0.5, 1.0, 'Linear Function Graph')
```



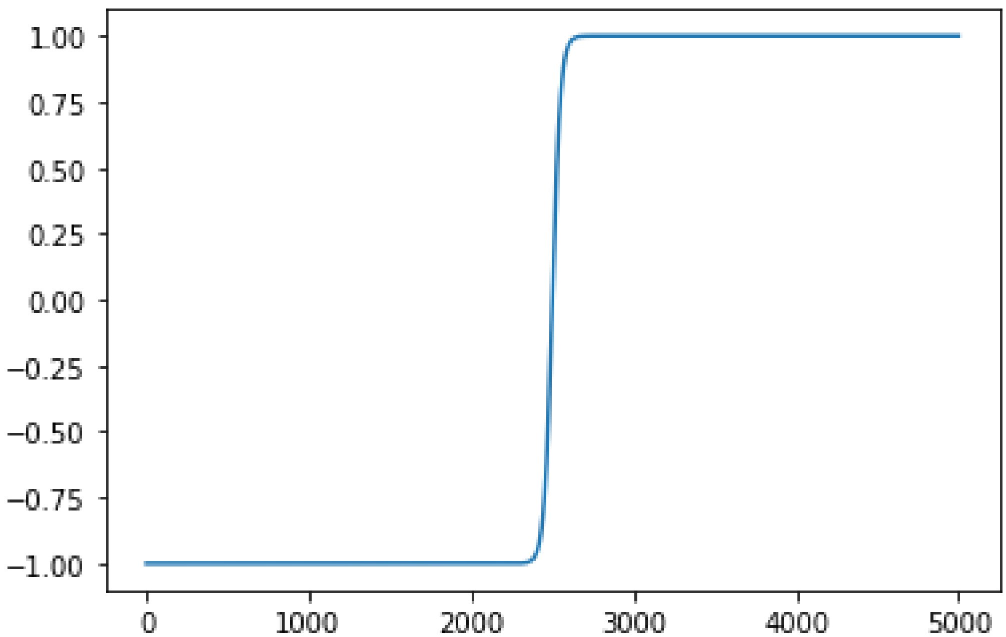
#Tanh

```
def tanh_function(x):
  return (np.exp(x) - np.exp(-x) / np.exp(x) + np.exp(-x))

def tanh_function2(x):
  return 2*sigmoid_function(2*x)-1

x = np.linspace(-50, 50, 5000)

y = [tanh_function(i) for i in x]

y_tanh = [tanh_function2(i) for i in x]

sns.scatterplot(y_tanh)

<Axes: >
```



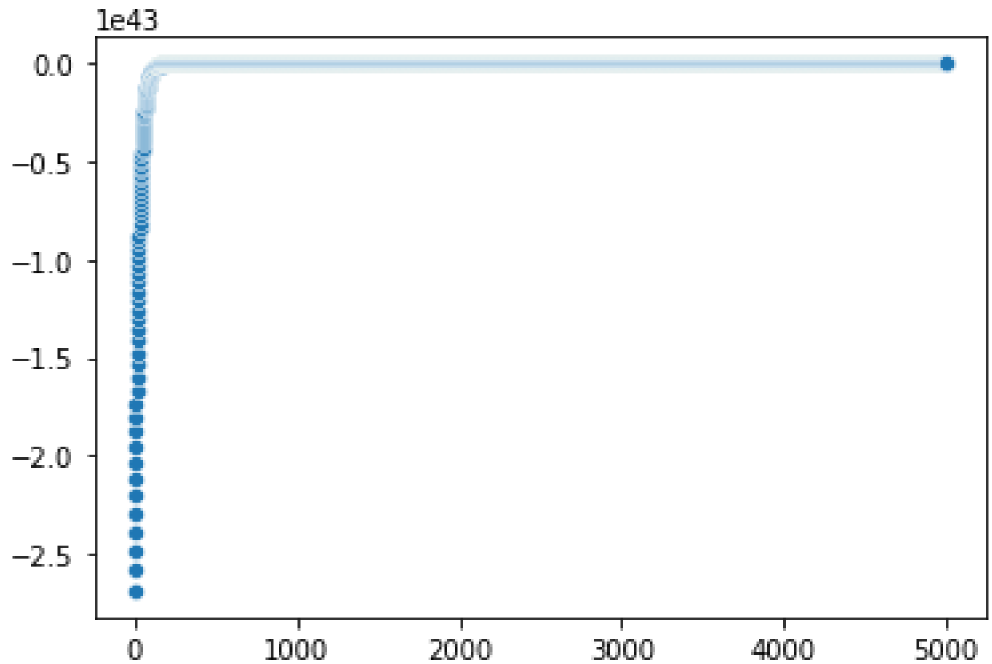

```
plt.plot(y_tanh)
```

[<matplotlib.lines.Line2D at 0x7f9a52d413a0>]



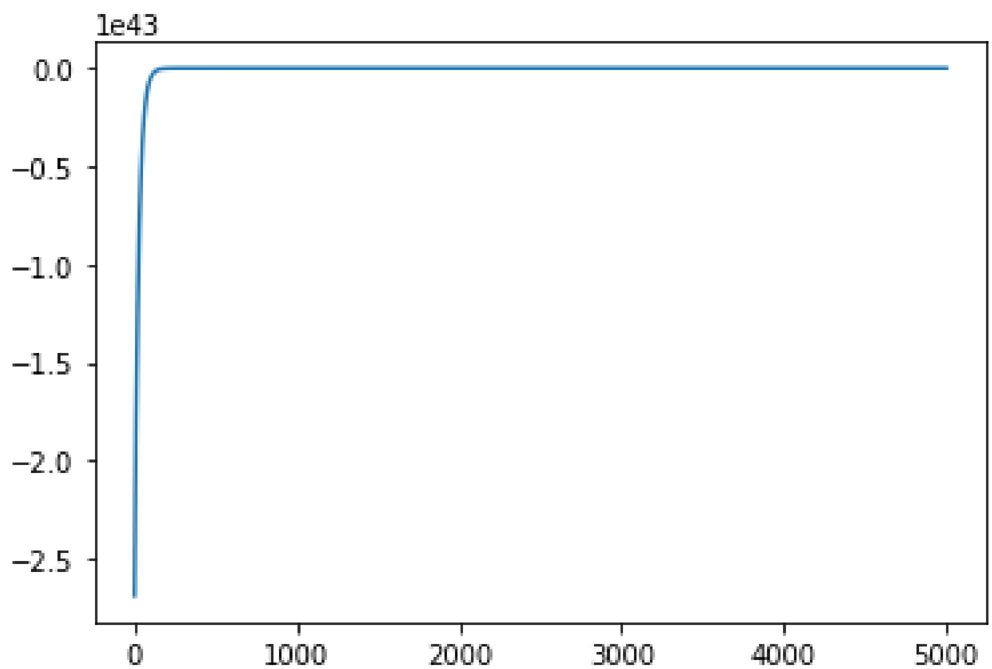

```
sns.scatterplot(y)
```

<Axes: >




```
plt.plot(y)
```

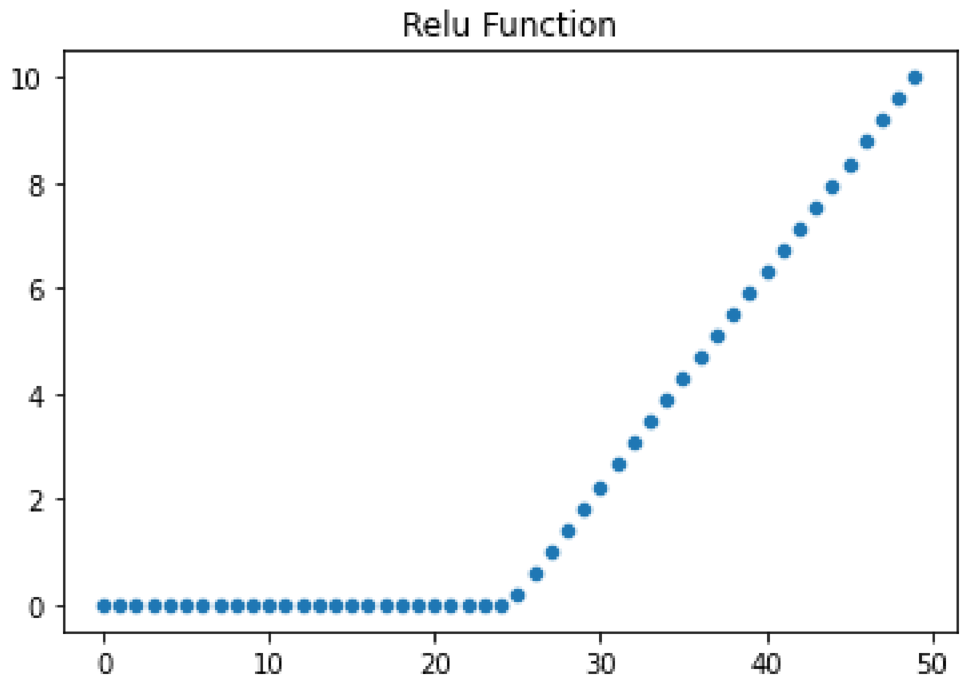
```
[<matplotlib.lines.Line2D at 0x7f9a52c26790>]
```



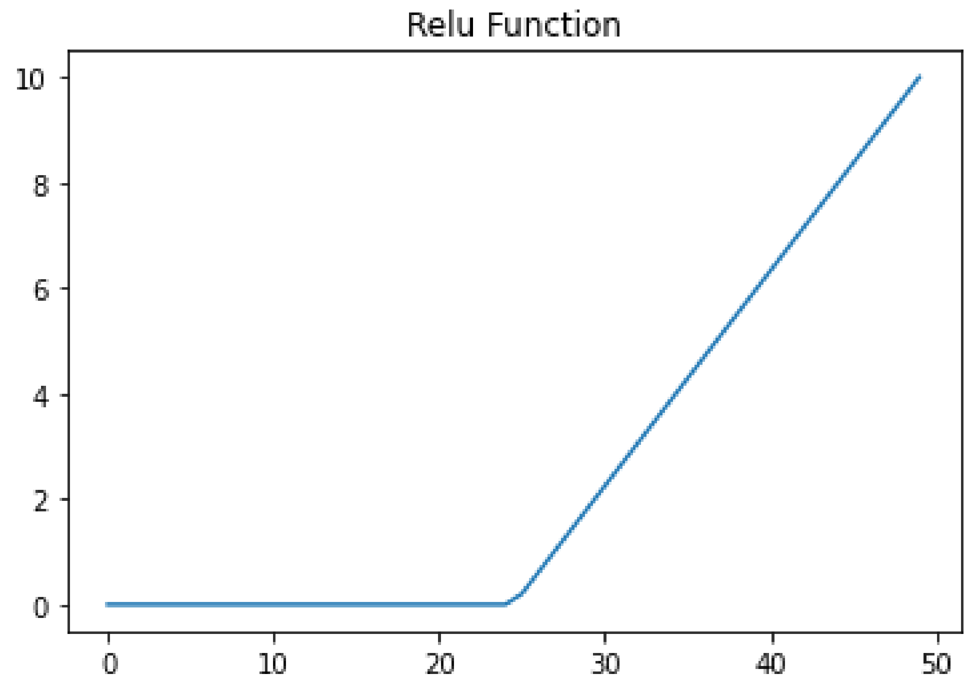
#Relu

```
def relu_function(x):
  return np.array([0, x]).max()
```

```
x = np.linspace(-10, 10)

y = [relu_function(i) for i in x]

sns.scatterplot(y)
plt.title("Relu Function")

Text(0.5, 1.0, 'Relu Function')
```

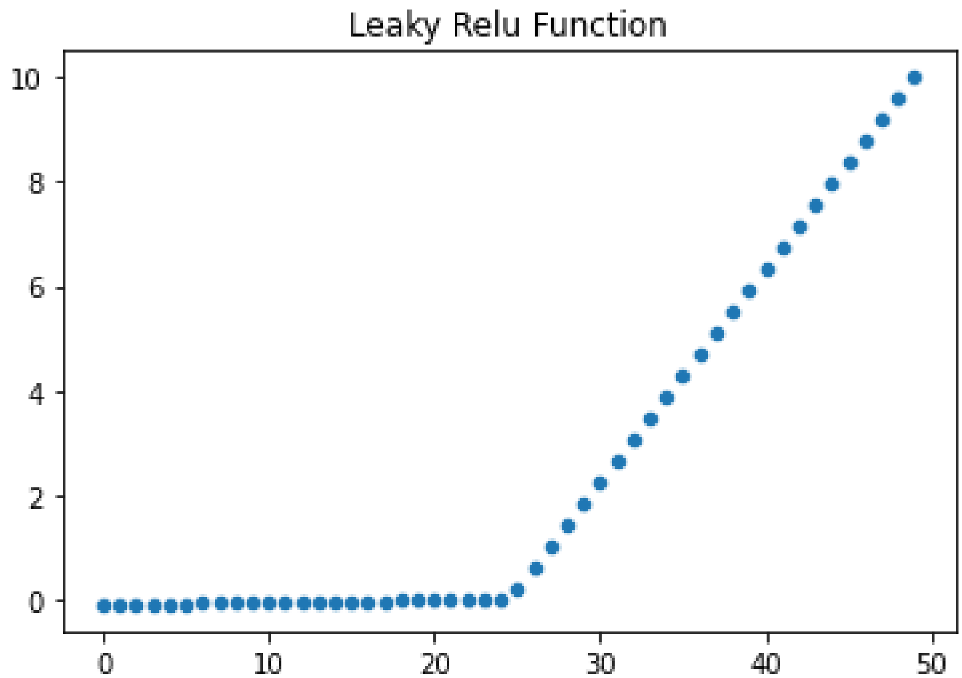



```
plt.plot(y)
plt.title("Relu Function")

Text(0.5, 1.0, 'Relu Function')
```



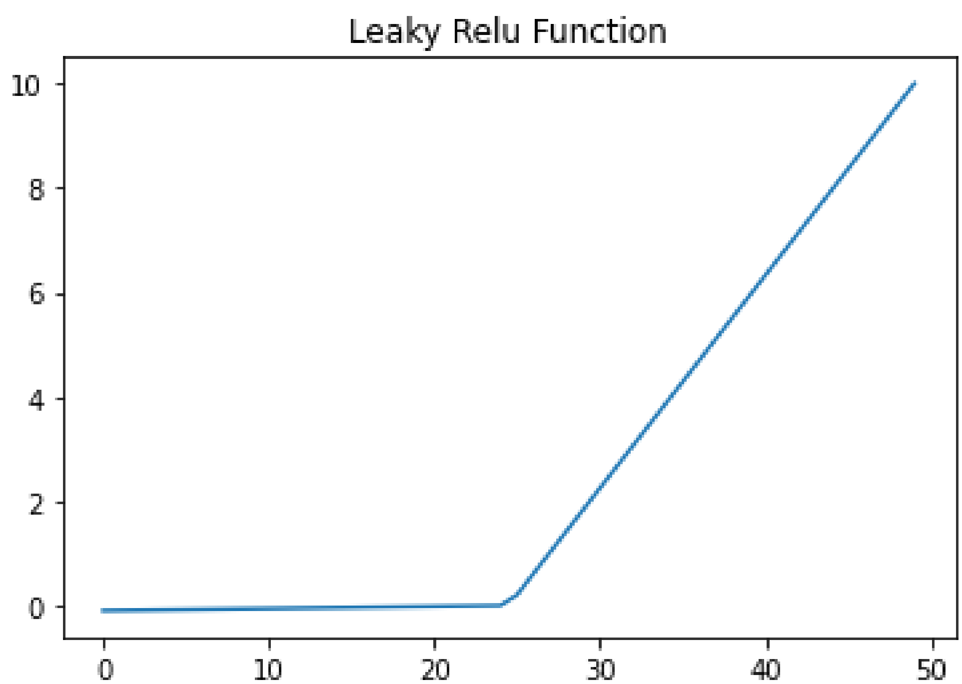
#leaky Relu

```
def leaky_relu_function(x):
  return  0.01*x if x < 0 else x

x = np.linspace(-10, 10)

y = [leaky_relu_function(i) for i in x]

sns.scatterplot(y)
plt.title('Leaky Relu Function')

Text(0.5, 1.0, 'Leaky Relu Function')
```




```
plt.plot(y)
plt.title("Leaky Relu Function")
```

```
Text(0.5, 1.0, 'Leaky Relu Function')
```



Lab Assignment No. 2

Code:

```
import numpy as np

class McCullochPittsNeuron():
  def __init__(self, threshold, weights):
    self.threshold = threshold
    self.weights = weights
    self.output = []

  def andNot(self, inputs):
    for inputXY in inputs:
      self.weightedSum = self.weights[0]*inputXY[0] +
self.weights[1]*inputXY[1]
      if self.weightedSum >= self.threshold:
        self.output.append(1)
      else:
        self.output.append(0)
    return self.output

mcpn = McCullochPittsNeuron(1, [1, -1])
output = mcpn.andNot([(0,0), (0,1), (1, 0), (1, 1)])
print("Output of McCulloch Pitts Neuron",output)
```

Output:

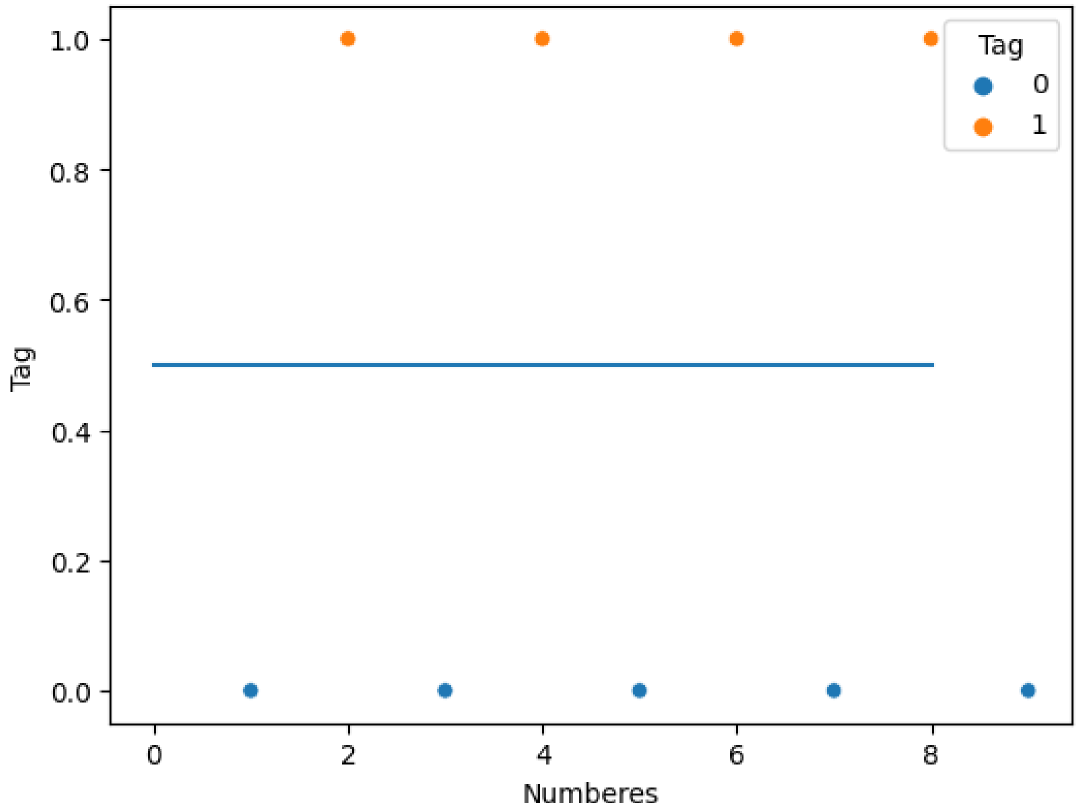
```
Output of McCulloch Pitts Neuron [0, 0, 1, 0]
```

Lab Assignment No. 3

Write a Python Program using Perceptron Neural Network to recognise even and odd numbers. Given numbers are in ASCII form 0 to 9

Code:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns

data = {"Numberes" : [1,3,5,4,2,9,7,6,8], "Tag" : [0,0,0, 1,1,0,0,1,1]}
df = pd.DataFrame(data)

df

   Numberes  Tag
0         1    0
1         3    0
2         5    0
3         4    1
4         2    1
5         9    0
6         7    0
7         6    1
8         8    1

x = df["Numberes"]
y = df["Tag"]

sns.scatterplot(x=df["Numberes"],y=y, hue=y)
plt.plot([0.5 for _ in df["Tag"] ])

[<matplotlib.lines.Line2D at 0x7f3457dc1ac0>]
```




```
2 * np.random.random((10, 1)) - 1
```

```
array([[-0.35812149],
       [ 0.47412758],
       [ 0.62511668],
       [-0.89430268],
       [ 0.32154228],
       [ 0.08212657],
       [ 0.4911432 ],
       [ 0.7819753 ],
       [ 0.92660091],
       [ 0.62252329]])
```

```
array([[ 0.2082653 ],
       [-0.74252417],
       [-0.77242322],
       [-0.86954873],
       [ 0.33953798],
       [ 0.74940269],
       [-0.29060187],
       [-0.72214394],
       [-0.49388729],
       [-0.09347683]])
```

```
int(bin(2)[2:])

10

a = [np.random.choice([0,1]) for _ in range(4)]
a

[0, 1, 0, 1]

1 if 8>0 else 0

1

np.ones(4)

array([1., 1., 1., 1.])

a = 2 + np.dot([1,2, 4], [2, 2,2])
a

16
```

#Class for binary input

```
class Perceptron():
  def __init__(self, epochs, lr, input_size):
    self.weight = np.ones(input_size)
    self.epochs = epochs
    self.lr = lr
    self.bias = 0.0

  def predict(self, x_test):
    a = self.bias
    for i in range(len(x_test)):
      a += self.weight[i] * x_test[i]
    return 1 if a>=0 else 0

  def train(self, train_data):
    for i in range(self.epochs):
      for x_train, y_train in train_data:
        predicted = self.predict(x_train)
        error = y_train - predicted
        self.bias += self.lr * error
        for j in range(len(self.weight)):
          self.weight[j] += self.lr * error * x_train[j]


perceptron = Perceptron(1000, 0.001, 8)

perceptron.train([([0,0,0,0,0,0,0,1], 0), ([0,0,0,0,0,0,1,0], 1),
([0,0,0,0,0,0,1,1], 0), ([0,0,0,0,0,1,0,0], 1), ([0,0,0,0,0,1,0,1], 0),
```

```
([0,0,0,0,0,1,1,0], 1), ([0,0,0,0,0,1,1,1], 0), ([0,0,0,0,1,0,0,0], 1),
([0,0,0,0,1,0,0,1], 0), ([0,0,0,0,1,0,1,0], 1), ([0,0,0,0,1,0,1,1], 0)])
```

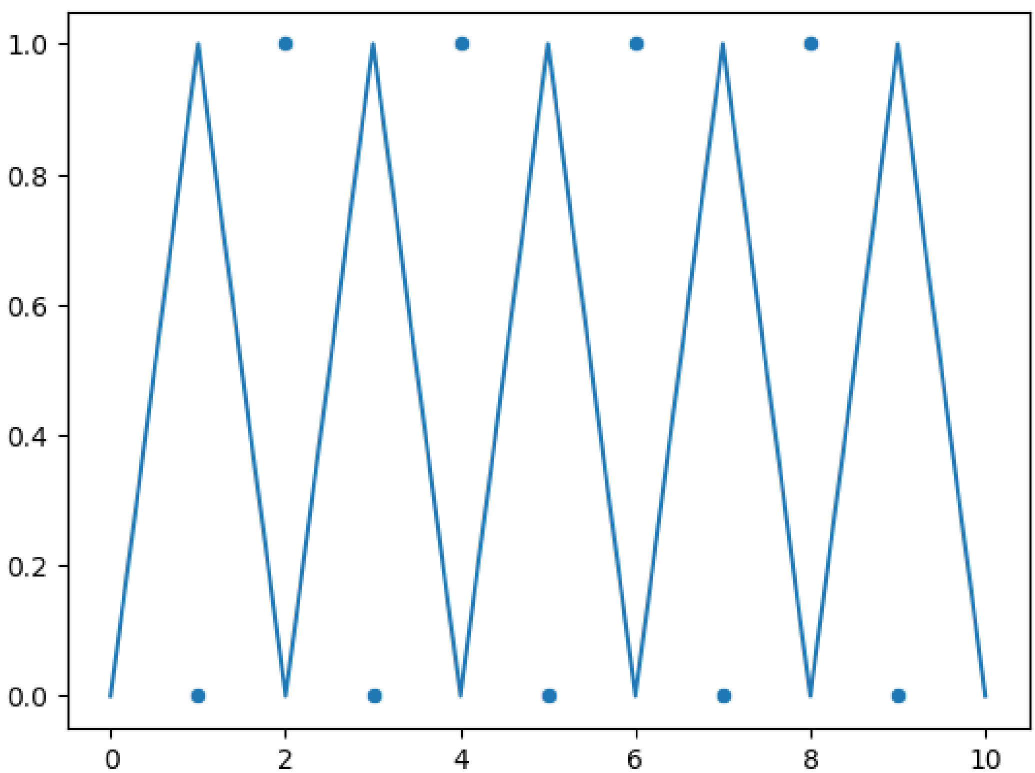
```
predictA = [ perceptron.predict(X_test) for X_test, i in [([0,0,0,0,0,0,0,1],
0), ([0,0,0,0,0,0,1,0], 1), ([0,0,0,0,0,0,1,1], 0), ([0,0,0,0,0,1,0,0], 1),
([0,0,0,0,0,1,0,1], 0), ([0,0,0,0,0,1,1,0], 1), ([0,0,0,0,0,1,1,1], 0),
([0,0,0,0,1,0,0,0], 1), ([0,0,0,0,1,0,0,1], 0), ([0,0,0,0,1,0,1,0], 1),
([0,0,0,0,1,0,1,1], 0)]]
```

```
perceptron.predict()
```

```
1
```

```
sns.scatterplot(x = [1, 2, 3, 4,5 ,6,7,8,9], y =[0, 1, 0, 1, 0, 1, 0, 1, 0])
plt.plot(predictA)
```

```
[<matplotlib.lines.Line2D at 0x7f34580cc820>]
```




```
a = [int(i) for i in bin(21)[2:]]
A = [0 for _ in range(8-len(a)) ] + a
A
```

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

#Class For ASCII Input

```
class Perceptron():
  def __init__(self, epochs, lr, input_size):
    self.weight = np.ones(input_size)
    self.epochs = epochs
    self.lr = lr
    self.bias = 0.0

  def predict(self, x_test):
    a = self.bias
    x_test = self.binary(x_test)
    for i in range(len(x_test)):
      a += self.weight[i] * x_test[i]
    return 1 if a>=0 else 0

  def binary(self, x):
    a = [int(i) for i in bin(x)[2:]]
    A = [0 for _ in range(8-len(a)) ] + a
    return A

  def train(self, train_data):
    for i in range(self.epochs):
      for x_train, y_train in train_data:
        # print(x_train)
        # print(x_train)
        predicted = self.predict(x_train)
        error = y_train - predicted
        x_train = self.binary(x_train)
        self.bias += self.lr * error
        for j in range(len(self.weight)):
          self.weight[j] += self.lr * error * x_train[j]


p = Perceptron(1000, 0.001, 8)

x_train = []
for i in range(1, 100):
  if i % 2 == 0:
    x_train.append((i, 1))
  else:
    x_train.append((i, 0))

p.train(x_train)

p.weight

array([ 0.782,  0.781,  0.288,  0.288,  0.205,  0.183,  0.181, -1.749])

p.bias

-0.18000000000000005
```

```
predictions = [(i, p.predict(i)) for i in range(1, 13)]

predictions
```

Output:

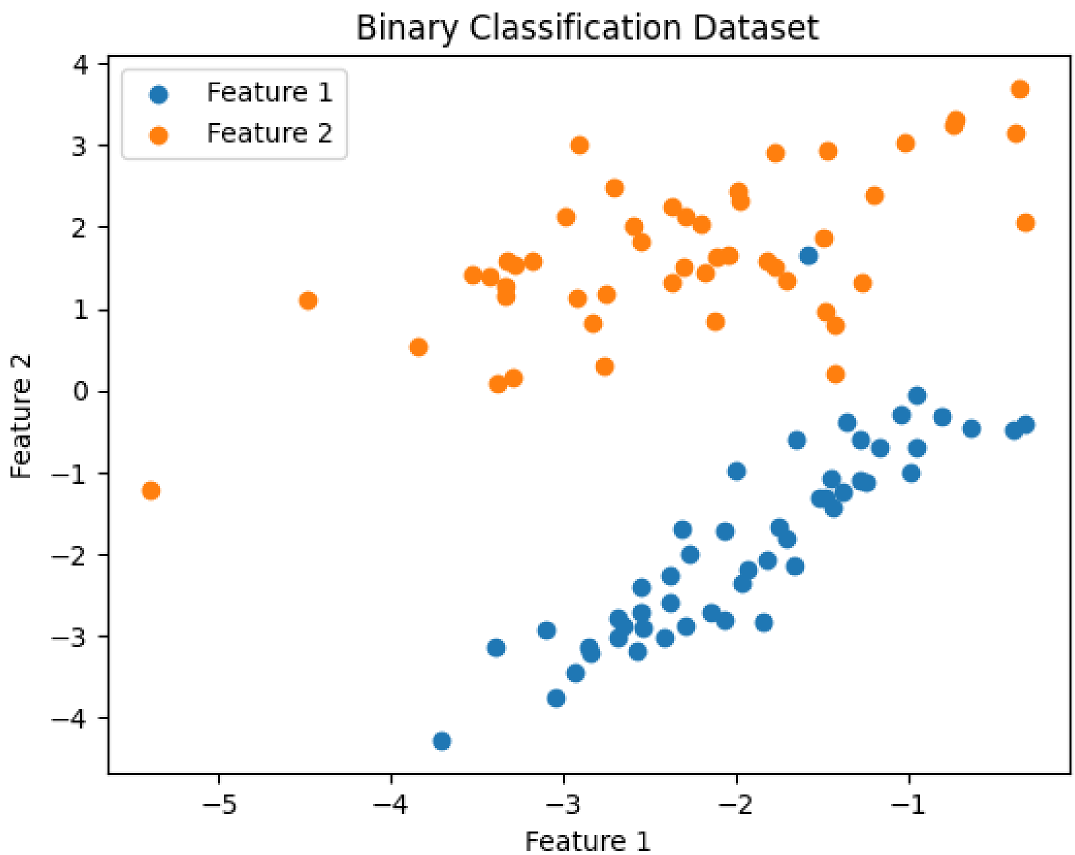
```
Even odd numbers are:

[(1, 0),
 (2, 1),
 (3, 0),
 (4, 1),
 (5, 0),
 (6, 1),
 (7, 0),
 (8, 1),
 (9, 0),
 (10, 1),
 (11, 0),
 (12, 1),
 (13, 0)]
```

Lab Assignment No. 4

Code:

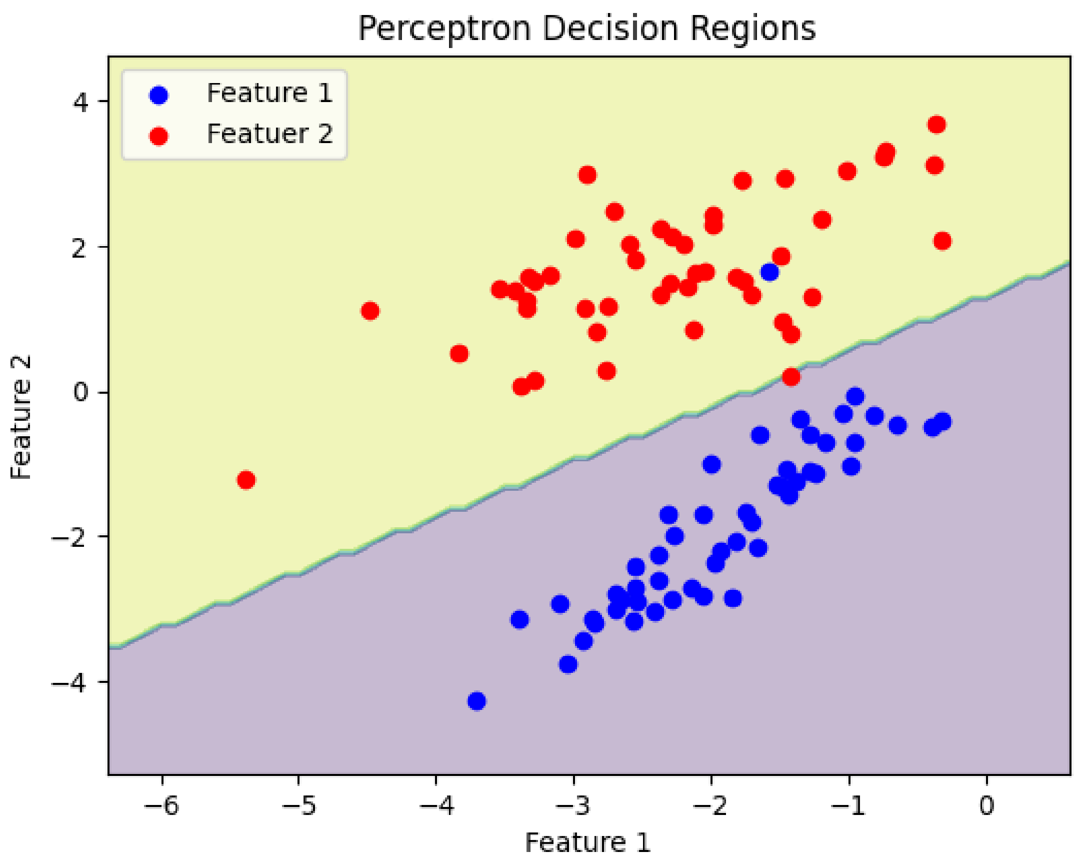
```
from sklearn.datasets import make_classification
import numpy as np
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, input_size, learning_rate=0.01):
        self.weights = np.zeros(input_size)
        self.bias = 0
        self.learning_rate = learning_rate

    def activation(self, weighted_sum):
      return np.where(weighted_sum <= 0, 0, 1)

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return self.activation(weighted_sum)

    def train(self, inputs, labels, num_epochs):
        for _ in range(num_epochs):
            for x, y in zip(inputs, labels):
                predicted = self.predict(x)
                error = y - predicted
                self.weights += self.learning_rate * error * x
                self.bias += self.learning_rate * error

X, y = make_classification(n_samples=100, n_features=2, n_informative=2,
                           n_redundant=0, n_clusters_per_class=1,
                           class_sep=2)
inputs = X
labels = y

labels

array([0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0,
       1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1,
       1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0])

plt.scatter(X[labels == 0][:,0], X[labels == 0][:, 1], label="Feature 1")
plt.scatter(X[labels == 1][:,0], X[labels == 1][:, 1], label="Feature 2")
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
```

```
plt.title('Binary Classification Dataset')
plt.legend()
plt.show()
```




```
perceptron = Perceptron(input_size=2)
perceptron.train(inputs, labels, num_epochs=10)
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, 0.1),
                     np.arange(y_min, y_max, 0.1))
Z = perceptron.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(inputs[labels == 0][:, 0], inputs[labels == 0][:, 1],
color='blue', label='Feature 1')
plt.scatter(inputs[labels == 1][:, 0], inputs[labels == 1][:, 1],
color='red', label='Featuer 2')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Perceptron Decision Regions')
plt.legend()
plt.show()
```

Output:


Perceptron Decision Regions
Feature 1
Featuer 2
Feature 2
Feature 1
4
2
0
−2
−4
−6
−5
−4
−3
−2
−1
0

Lab Assignment No. 5

Code:

```
import numpy as np

class BAM:
    def __init__(self):
        self.weights = None

    def train(self, X, Y):
        X = np.array(X)
        Y = np.array(Y)
        self.weights = np.dot(Y.T, X)

    def recall(self, X):
        X = np.array(X)
        Y = np.dot(X, self.weights.T)
        Y[Y >= 0] = 1
        Y[Y < 0] = -1
        return Y

if __name__ == '__main__':
    bam = BAM()

    X = [[1, 1, -1, -1],
         [-1, -1, 1, 1]]
    Y = [[1, -1],
         [-1, 1]]

    bam.train(X, Y)

    test_X = [[1, -1, 1, -1],
              [1, 1, -1, -1]]

    for x in test_X:
        recalled_Y = bam.recall(x)
        print(f"Input: {x}")
        print(f"Recalled Output: {recalled_Y}\n")
```

Output:

```
Input: [1, -1, 1, -1]
Recalled Output: [1 1]

Input: [1, 1, -1, -1]

Recalled Output: [ 1 -1]
```

Lab Assignment No. 6

Code:

```
import numpy as np

class NeuralNetwork:
    def __init__(self, layer_sizes):
        self.layer_sizes = layer_sizes
        self.weights = [np.random.randn(y, x) for x, y in zip(layer_sizes[:-
1], layer_sizes[1:])]
        self.biases = [np.random.randn(y, 1) for y in layer_sizes[1:]]

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def forward_propagation(self, X):
        activation = X
        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, activation) + b
            activation = self.sigmoid(z)
        return activation

    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))

    def backpropagation(self, X, y):
        m = X.shape[1]
        delta_weights = [np.zeros(w.shape) for w in self.weights]
        delta_biases = [np.zeros(b.shape) for b in self.biases]
        # Forward propagation
        activation = X
        activations = [activation]
        zs = []
        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = self.sigmoid(z)
            activations.append(activation)

        # Backpropagation
        delta = (activations[-1] - y) * self.sigmoid_derivative(zs[-1])
        delta_weights[-1] = np.dot(delta, activations[-2].T)
        delta_biases[-1] = np.sum(delta, axis=1, keepdims=True)
        for l in range(2, len(self.layer_sizes)):
            delta = np.dot(self.weights[-l+1].T, delta) *
self.sigmoid_derivative(zs[-l])
            delta_weights[-l] = np.dot(delta, activations[-l-1].T)
```

```
            delta_biases[-l] = np.sum(delta, axis=1, keepdims=True)

        return delta_weights, delta_biases

    def train(self, X, y, num_epochs, learning_rate):
        m = X.shape[1]
        for epoch in range(num_epochs):
            delta_weights, delta_biases = self.backpropagation(X, y)
            self.weights = [w - (learning_rate / m) * dw for w, dw in
zip(self.weights, delta_weights)]
            self.biases = [b - (learning_rate / m) * db for b, db in
zip(self.biases, delta_biases)]

    def predict(self, X):
        return self.forward_propagation(X)

if __name__ == '__main__':
    layer_sizes = [2, 4, 1]  # Input layer: 2 neurons, Hidden layer: 4
neurons, Output layer: 1 neuron
    nn = NeuralNetwork(layer_sizes)

    # Training data
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T
    y = np.array([[0, 1, 1, 0]])

    # Train the neural network
    num_epochs = 10000
    learning_rate = 0.1
    nn.train(X, y, num_epochs, learning_rate)

    predictions = nn.predict(X)
    print("Predictions:")
    print(predictions)
```

Output:

```
Predictions:
[[0.23317429 0.62300796 0.69296428 0.49537369]]

X

array([[0, 0, 1, 1],
       [0, 1, 0, 1]])

X[0]

array([0, 0, 1, 1])

y[0]
```

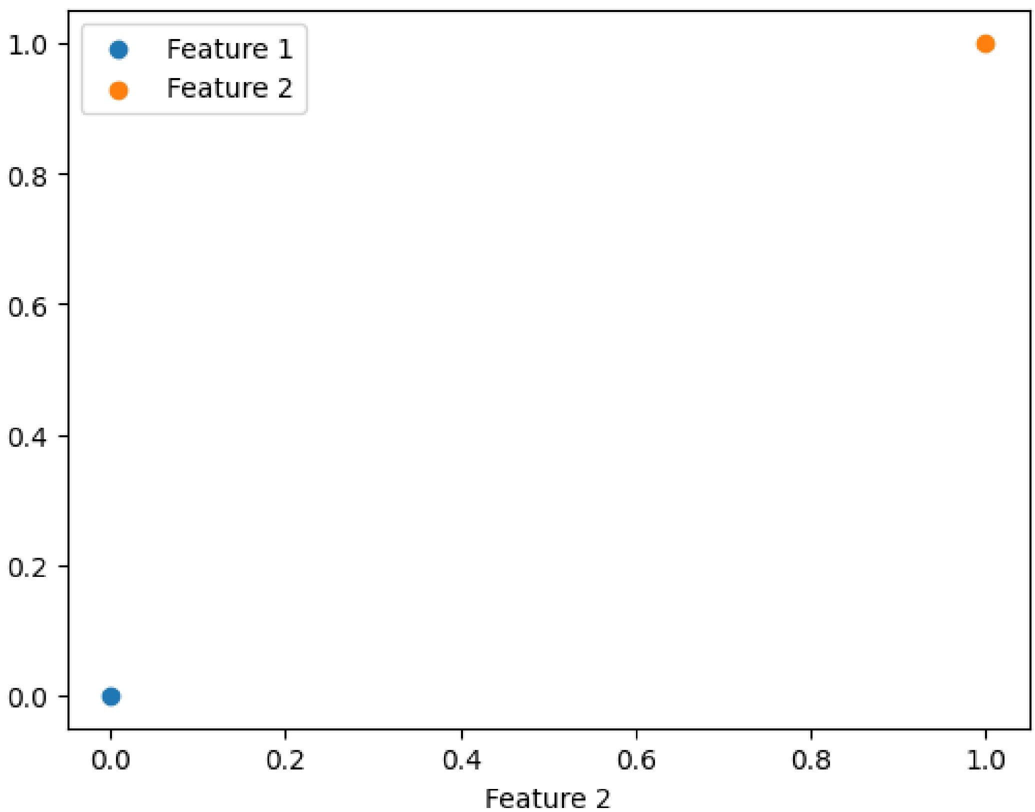
```
array([0, 1, 1, 0])
```

```
x = [(x, y) for x, y in zip(X[:,0], X[:,1])]
x
```

```
[(0, 0), (0, 1)]
```

```
x0 = [i for i in X[0] if i == 0]
x1 = [i for i in X[1] if i == 1]
```

```
import matplotlib.pyplot as plt
plt.scatter(x0[0], x0[1], label="Feature 1")
plt.scatter(x1[0], x1[1], label="Feature 2")
plt.xlabel("Feature 1")
plt.xlabel("Feature 2")
plt.legend()
plt.show()
```




```
final_preditions = [1 if predict >= 0.5 else 0 for predict in predictions[0]]
```

```
final_preditions
```

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

Lab Assignment No. 7

B.1. Write a python program to show Back Propagation Network for XOR function with Binary Input and Output

Code:

```
import numpy as np

#Activation sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

#Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

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

np.random.seed(42)

input_dim = 2
hidden_dim = 2
output_dim = 1

weights_input_hidden = 2 * np.random.random((input_dim, hidden_dim)) - 1
weights_hidden_output = 2 * np.random.random((hidden_dim, output_dim)) - 1

biases_hidden = np.zeros((1, hidden_dim))
biases_output = np.zeros((1, output_dim))

learning_rate = 0.1
num_epochs = 10000

for epoch in range(num_epochs):
    # Forward propagation
    hidden_layer_input = np.dot(X, weights_input_hidden) + biases_hidden
    hidden_layer_activation = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_activation,
weights_hidden_output) + biases_output
    output_layer_activation = sigmoid(output_layer_input)

    # Backpropagation
    error = y - output_layer_activation
    output_layer_delta = error * sigmoid_derivative(output_layer_activation)
```

```
    hidden_layer_error = output_layer_delta.dot(weights_hidden_output.T)
    hidden_layer_delta = hidden_layer_error *
sigmoid_derivative(hidden_layer_activation)

    weights_hidden_output +=
hidden_layer_activation.T.dot(output_layer_delta) * learning_rate
    biases_output += np.sum(output_layer_delta, axis=0, keepdims=True) *
learning_rate

    weights_input_hidden += X.T.dot(hidden_layer_delta) * learning_rate
    biases_hidden += np.sum(hidden_layer_delta, axis=0, keepdims=True) *
learning_rate

test_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
hidden_layer_output = sigmoid(np.dot(test_input, weights_input_hidden) +
biases_hidden)
predicted_output = sigmoid(np.dot(hidden_layer_output, weights_hidden_output)
+ biases_output)

print("Predicted Output:")
print(predicted_output)
```

```
Predicted Output:
[[0.0961913 ]
 [0.89393519]
 [0.89410922]
 [0.08557778]]
```

```
test_input
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
y
```

```
array([[0],
       [1],
       [1],
       [0]])
```

```
final_preditions = [1 if predict >= 0.5 else 0 for predict in
predicted_output]
```

Output:

```
final_preditions
```

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

Lab Assignment Group 8

Code:

```
import numpy as np

#Activation sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

#Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

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

np.random.seed(42)

input_dim = 2
hidden_dim = 2
output_dim = 1

weights_input_hidden = 2 * np.random.random((input_dim, hidden_dim)) - 1
weights_hidden_output = 2 * np.random.random((hidden_dim, output_dim)) - 1

biases_hidden = np.zeros((1, hidden_dim))
biases_output = np.zeros((1, output_dim))

learning_rate = 0.1
num_epochs = 10000

for epoch in range(num_epochs):
    # Forward propagation
    hidden_layer_input = np.dot(X, weights_input_hidden) + biases_hidden
    hidden_layer_activation = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_activation, weights_hidden_outpu
t) + biases_output
    output_layer_activation = sigmoid(output_layer_input)

    # Backpropagation
    error = y - output_layer_activation
    output_layer_delta = error * sigmoid_derivative(output_layer_activation)

    hidden_layer_error = output_layer_delta.dot(weights_hidden_output.T)
```

```
    hidden_layer_delta = hidden_layer_error * sigmoid_derivative(hidden_layer
_activation)

    weights_hidden_output += hidden_layer_activation.T.dot(output_layer_delta
) * learning_rate
    biases_output += np.sum(output_layer_delta, axis=0, keepdims=True) * lear
ning_rate

    weights_input_hidden += X.T.dot(hidden_layer_delta) * learning_rate
    biases_hidden += np.sum(hidden_layer_delta, axis=0, keepdims=True) * lear
ning_rate

test_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
hidden_layer_output = sigmoid(np.dot(test_input, weights_input_hidden) + bias
es_hidden)
predicted_output = sigmoid(np.dot(hidden_layer_output, weights_hidden_output)
+ biases_output)

print("Predicted Output:")
print(predicted_output)
```

```
Predicted Output:
[[0.0961913 ]
 [0.89393519]
 [0.89410922]
 [0.08557778]]
```

```
test_input
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
y
```

```
array([[0],
       [1],
       [1],
       [0]])
```

```
final_preditions = [1 if predict >= 0.5 else 0 for predict in predicted_outpu
t]
```

Output:

```
final_preditions
```

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

Lab Assignment No. 9

Q4. Write a python program to design a Hopfield Network which stores 4 vectors

Code:

```
import numpy as np

class HopfieldNetwork:
    def __init__(self, num_neurons):
        self.num_neurons = num_neurons
        self.weights = np.zeros((num_neurons, num_neurons))

    def train(self, vectors):
        num_vectors = len(vectors)
        for vector in vectors:
            vector = np.reshape(vector, (self.num_neurons, 1))
            self.weights += np.dot(vector, vector.T) / num_vectors
            np.fill_diagonal(self.weights, 0)

    def recall(self, input_vector, max_iter=100):
        output_vector = np.copy(input_vector)
        for _ in range(max_iter):
            prev_output = np.copy(output_vector)
            output_vector = np.sign(np.dot(self.weights, output_vector))
            if np.array_equal(output_vector, prev_output):
                break
        return output_vector

network = HopfieldNetwork(4)

vectors = np.array([[1, 1, 1, 1],
                    [1, -1, 1, -1],
                    [-1, 1, -1, 1],
                    [-1, -1, -1, -1]])


network.train(vectors)

for vector in vectors:
    output = network.recall(vector)
    print("Input:", vector)
    print("Output:", output)
    print()
```

Output:

```
Input: [1 1 1 1]
Output: [1. 1. 1. 1.]

Input: [ 1 -1  1 -1]
Output: [ 1. -1.  1. -1.]

Input: [-1  1 -1  1]
Output: [-1.  1. -1.  1.]

Input: [-1 -1 -1 -1]
Output: [-1. -1. -1. -1.]
```

Lab Assignment No. 10

Code:

```
import torch
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.transforms import functional as F
from PIL import Image, ImageDraw

model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights=True)

model.eval()

def transform_image(image):
    image = F.to_tensor(image)
    return image.unsqueeze(0)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior
is equivalent to passing `weights=FasterRCNN_ResNet50_FPN_Weights.COCO_V1`.
You can also use `weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT` to get the
most up-to-date weights.
  warnings.warn(msg)
```

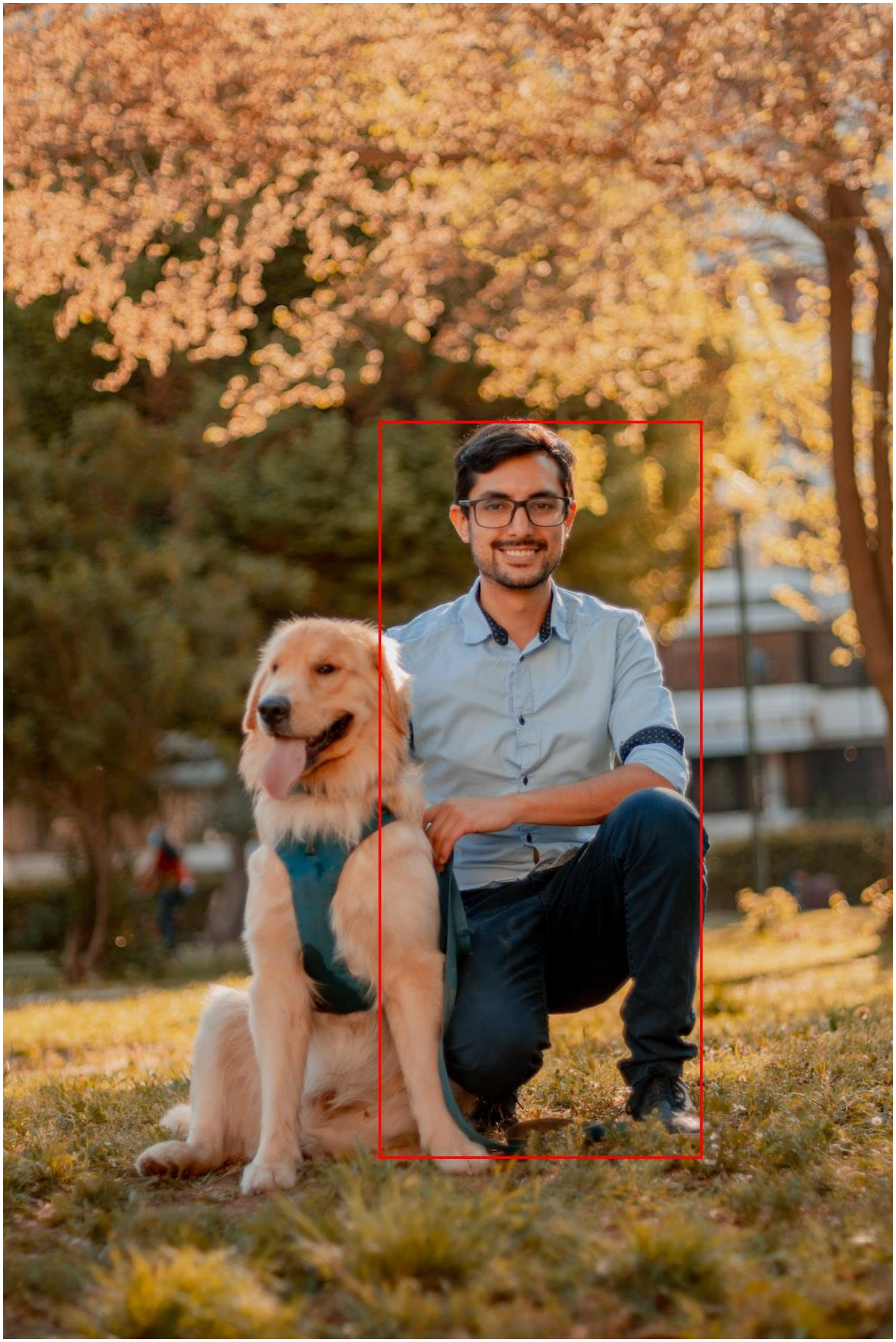
```
def calculate_area(box):
  maxX = max(box[0], box[2])
  maxY = max(box[1], box[3])
  minX = min(box[0], box[2])
  minY = min(box[1], box[3])
  width = (maxX-minX)
  height = (maxY-minY)
  return width*height

def calculate_iou(box1, box2):
    # Calculate the intersection area
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])
    intersection_area = max(0, x2 - x1 + 1) * max(0, y2 - y1 + 1)

    # Calculate the union area
    box1_area = (box1[2] - box1[0] + 1) * (box1[3] - box1[1] + 1)
    box2_area = (box2[2] - box2[0] + 1) * (box2[3] - box2[1] + 1)
    union_area = box1_area + box2_area - intersection_area

    # Calculate the IoU
```

```
    iou = intersection_area / union_area
    return iou

def max_area_box(boxes, threshold):
  maxArea = []
  maxAreaBox = []
  for box1 in boxes:
    for box2 in boxes:
      if box1 != box2:
        iou = calculate_iou(box1, box2)
        if iou < threshold:
          calAB1 = calculate_area(box1)
          calAB2 = calculate_area(box2)
          maxBox = max(calAB1, calAB2)
          # print(maxBox)
          if maxBox not in maxArea:
            maxArea.append(maxBox)
          if maxBox == calAB1:
            if box1 not in maxAreaBox:
              maxAreaBox.append(box1)
          if maxBox == calAB2:
            if box2 not in maxAreaBox:
              maxAreaBox.append(box2)

    return {"MaxArea":maxArea, "MaxAreaBox":maxAreaBox}

def detect_object(img, threshold):
  image_path = img
  image = Image.open(image_path).convert("RGB")
  transformed_image = transform_image(image)

  with torch.no_grad():
      predictions = model(transformed_image)

  boxes = predictions[0]['boxes'].tolist()
  scores = predictions[0]['scores'].tolist()
  labels = predictions[0]['labels'].tolist()

  draw = ImageDraw.Draw(image)
  # print(max_area_box(boxes)['MaxAreaBox'])
  for box in max_area_box(boxes, threshold)['MaxAreaBox']:
    draw.rectangle(box, outline='red', width=3)

  image.show()
  # print(score)

detect_object('/content/dog.jpg', 0.3)
```

Output:

```
detect_object('/content/manwithdog.jpeg', 0.36)
```

```
Accuracy 0.89 loss 0.076
```



Lab Assignment No. 11

Code:

```
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Dropout
from sklearn.datasets import make_gaussian_quantiles
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns


X1, y1 = make_gaussian_quantiles(cov=3.,
                                 n_samples=10000, n_features=2,
                                 n_classes=2, random_state=1)
X1 = pd.DataFrame(X1,columns=['x','y'])
y1 = pd.Series(y1)
```

```
X1
```

```
             x         y
0     0.759772  1.418316
1     2.429896 -2.974839
2    -1.312662 -3.837630
3     1.544247  0.904236
4     0.675905  3.471664
...        ...       ...
9995 -1.519436 -0.076489
9996 -2.862951  1.931277
9997 -0.977937  0.364132
9998 -3.888984 -2.809069
9999  0.075637 -0.391988

[10000 rows x 2 columns]
```

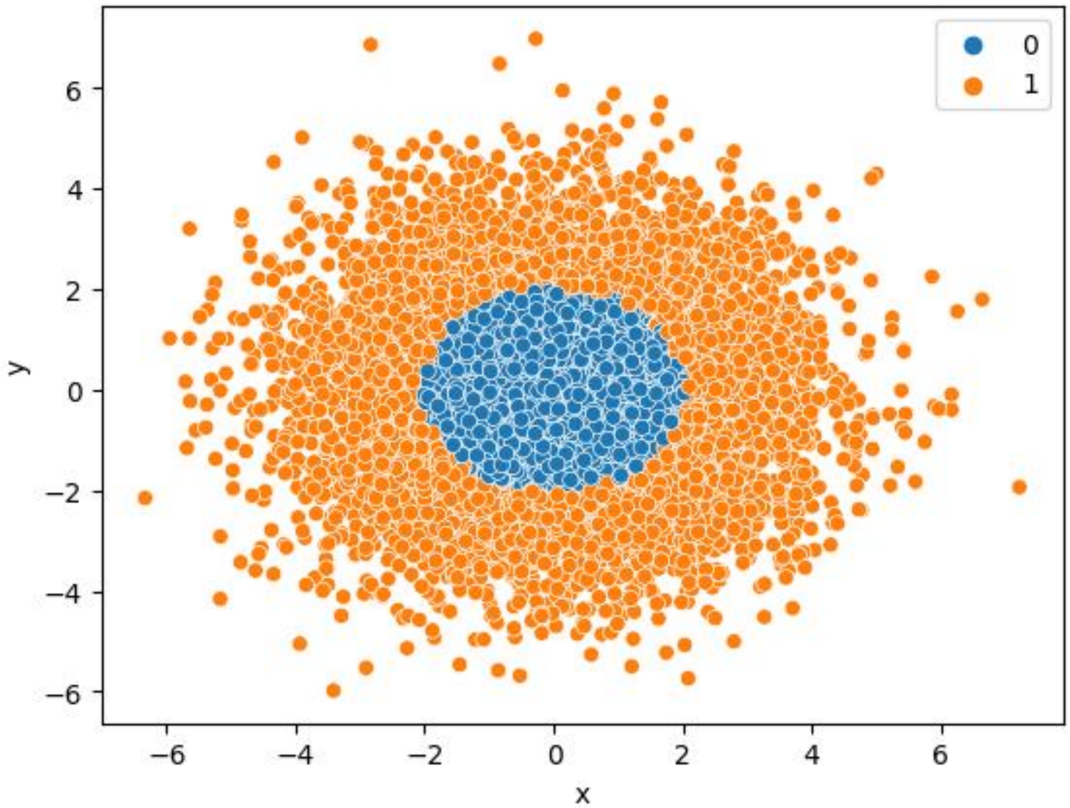
```
y1
```

```
0       0
1       1
2       1
3       0
4       1
       ..
9995    0
9996    1
9997    0
9998    1
```

```
9999    0
Length: 10000, dtype: int64
```

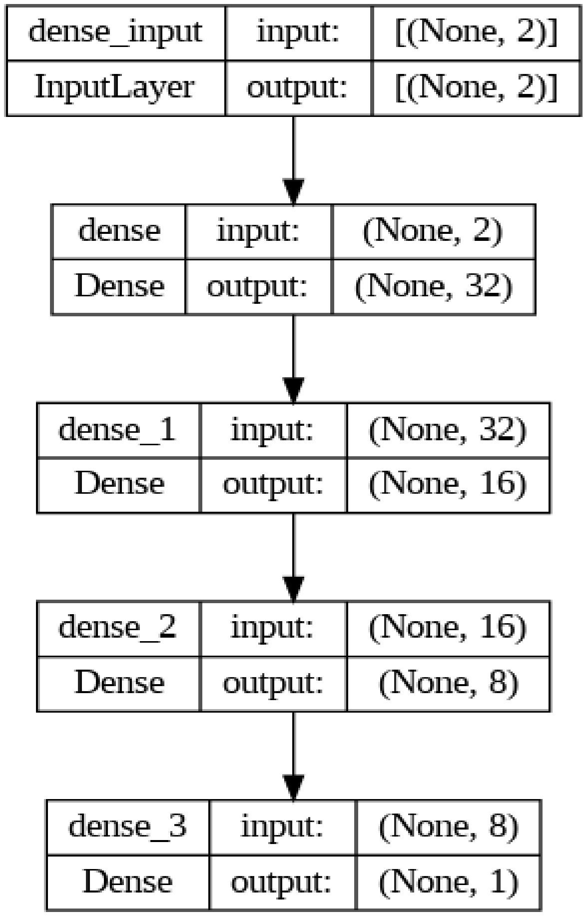
```
sns.scatterplot(x=X1.iloc[:,0], y=X1.iloc[:,1], hue=y1)
```

```
<Axes: xlabel='x', ylabel='y'>
```




```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=2))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
from keras.utils import plot_model
plot_model(model, show_shapes=True)
```




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

```
model.fit(X1, y1, epochs=50, verbose=1)
```

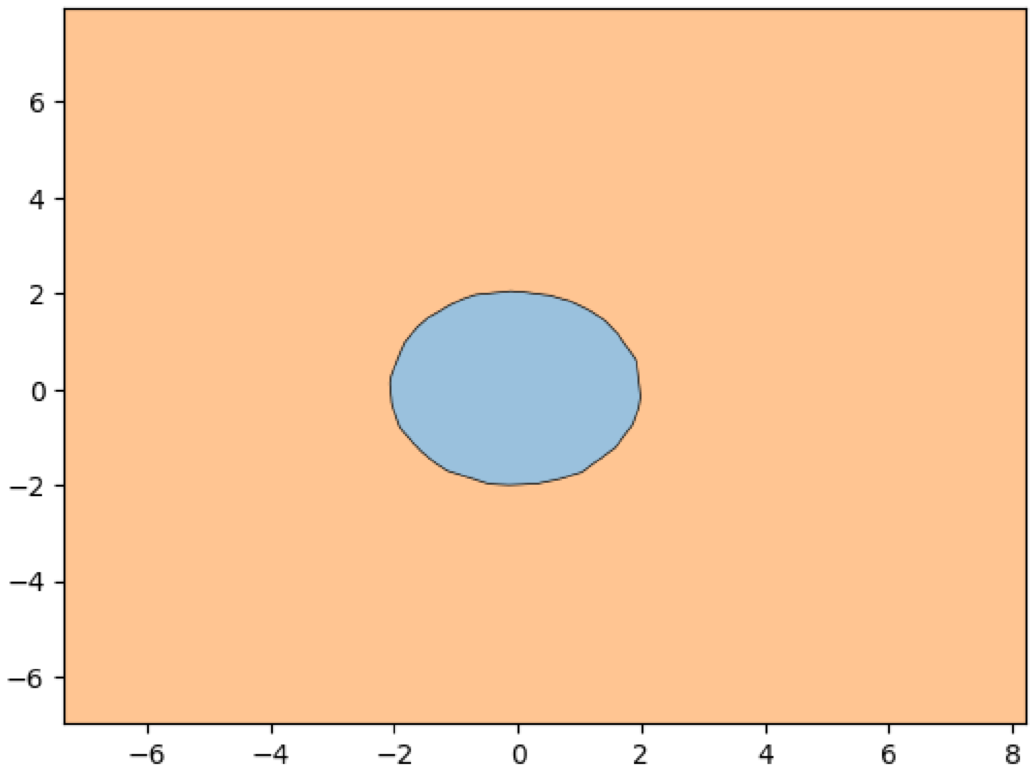
Output:

```
Epoch 1/50
313/313 [==============================] - 5s 5ms/step - loss: 0.4545 -
accuracy: 0.7580
Epoch 2/50
313/313 [==============================] - 2s 5ms/step - loss: 0.0947 -
accuracy: 0.9822
Epoch 3/50
313/313 [==============================] - 1s 4ms/step - loss: 0.0521 -
accuracy: 0.9877
Epoch 4/50
313/313 [==============================] - 2s 6ms/step - loss: 0.0415 -
accuracy: 0.9879
Epoch 5/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0345 -
accuracy: 0.9903
```

```
Epoch 6/50
313/313 [==============================] - 1s 4ms/step - loss: 0.0313 -
accuracy: 0.9911
Epoch 7/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0284 -
accuracy: 0.9900
Epoch 8/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0268 -
accuracy: 0.9912
Epoch 9/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0243 -
accuracy: 0.9912
Epoch 10/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0241 -
accuracy: 0.9907
Epoch 11/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0230 -
accuracy: 0.9918
Epoch 12/50
313/313 [==============================] - 1s 5ms/step - loss: 0.0210 -
accuracy: 0.9920
Epoch 13/50
313/313 [==============================] - 2s 6ms/step - loss: 0.0211 -
accuracy: 0.9924
Epoch 14/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0223 -
accuracy: 0.9920
Epoch 15/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0187 -
accuracy: 0.9935
Epoch 16/50
313/313 [==============================] - 1s 4ms/step - loss: 0.0176 -
accuracy: 0.9935
Epoch 17/50
313/313 [==============================] - 2s 5ms/step - loss: 0.0196 -
accuracy: 0.9930
Epoch 18/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0193 -
accuracy: 0.9921
Epoch 19/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0171 -
accuracy: 0.9936
Epoch 20/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0164 -
accuracy: 0.9937
Epoch 21/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0183 -
accuracy: 0.9929
Epoch 22/50
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0177 -
accuracy: 0.9924
Epoch 23/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0195 -
accuracy: 0.9920
Epoch 24/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0174 -
accuracy: 0.9927
Epoch 25/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0144 -
accuracy: 0.9948
Epoch 26/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0169 -
accuracy: 0.9936
Epoch 27/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0171 -
accuracy: 0.9928
Epoch 28/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0148 -
accuracy: 0.9946
Epoch 29/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0174 -
accuracy: 0.9930
Epoch 30/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0137 -
accuracy: 0.9948
Epoch 31/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0170 -
accuracy: 0.9935
Epoch 32/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0146 -
accuracy: 0.9943
Epoch 33/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0160 -
accuracy: 0.9935
Epoch 34/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0145 -
accuracy: 0.9938
Epoch 35/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0177 -
accuracy: 0.9928
Epoch 36/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0173 -
accuracy: 0.9927
Epoch 37/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0166 -
accuracy: 0.9924
Epoch 38/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0139 -
```

```
accuracy: 0.9942
Epoch 39/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0133 -
accuracy: 0.9944
Epoch 40/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0140 -
accuracy: 0.9942
Epoch 41/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0123 -
accuracy: 0.9957
Epoch 42/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0155 -
accuracy: 0.9932
Epoch 43/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0145 -
accuracy: 0.9937
Epoch 44/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0156 -
accuracy: 0.9927
Epoch 45/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0148 -
accuracy: 0.9938
Epoch 46/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0137 -
accuracy: 0.9948
Epoch 47/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0158 -
accuracy: 0.9931
Epoch 48/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0174 -
accuracy: 0.9924
Epoch 49/50
313/313 [==============================] - 1s 3ms/step - loss: 0.0137 -
accuracy: 0.9940
Epoch 50/50
313/313 [==============================] - 1s 2ms/step - loss: 0.0170 -
accuracy: 0.9925

<keras.callbacks.History at 0x7f2618d1c5b0>
```




```
model.predict(X1)
```

```
313/313 [==============================] - 0s 1ms/step
```

```
array([[2.0704062e-08],
       [1.0000000e+00],
       [1.0000000e+00],
       ...,
       [4.4519486e-14],
       [1.0000000e+00],
       [8.1487582e-16]], dtype=float32)
```

```
loss, accuracy = model.evaluate(x=X1, y=y1)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0201 -
accuracy: 0.9911
```

```
print(f"Model having Loss of {loss} and accuracy with {accuracy}")
```

```
Model having Loss of 0.020140156149864197 and accuracy with 0.991100013256073
```

Lab Assignment No. 12

Code:

```
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Conv2D, Dense, MaxPooling2D, Flatten, Dropout, Input
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from keras.utils import plot_model
from keras.datasets import mnist

num_classes = 10
input_shape = (28, 28, 1)
(x_train, y_train), (x_test, y_test)= mnist.load_data()
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

```
print("len of x_train: ", len(x_train), "Len of y_train: ", len(y_train), " len of x_test: ", len(x_test), " len y_test: ", len(y_test))
```

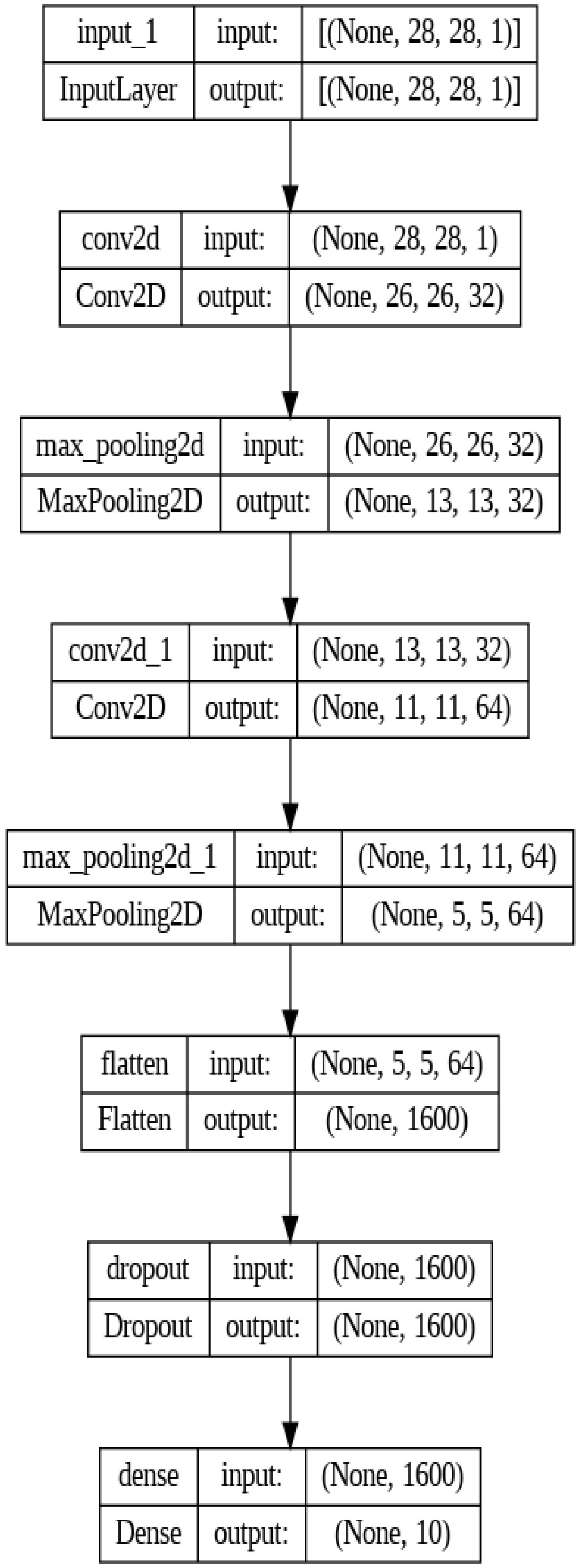
```
len of x_train:  60000 Len of y_train:  60000  len of x_test:  10000  len y_test:  10000
```

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        Conv2D(32, kernel_size=(3, 3), activation="relu"),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(64, kernel_size=(3, 3), activation="relu"),
```

```
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dropout(0.5),
        Dense(num_classes, activation="softmax"),
    ]
)

model.summary()

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

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dropout (Dropout)           (None, 1600)              0

 dense (Dense)               (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model, show_shapes=True)
```


input_1 | input: | [(None, 28, 28, 1)]
InputLayer | output: | [(None, 28, 28, 1)]
conv2d | input: | (None, 28, 28, 1)
Conv2D | output: | (None, 26, 26, 32)
max_pooling2d | input: | (None, 26, 26, 32)
MaxPooling2D | output: | (None, 13, 13, 32)
conv2d_1 | input: | (None, 13, 13, 32)
Conv2D | output: | (None, 11, 11, 64)
max_pooling2d_1 | input: | (None, 11, 11, 64)
MaxPooling2D | output: | (None, 5, 5, 64)
flatten | input: | (None, 5, 5, 64)
Flatten | output: | (None, 1600)
dropout | input: | (None, 1600)
Dropout | output: | (None, 1600)
dense | input: | (None, 1600)
Dense | output: | (None, 10)

```
x_train.shape
```

```
(60000, 28, 28, 1)
```

```
y_train.shape
```

```
(60000, 10)
```

```
epochs=10
batch_size = 128
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_split=0.1)
```

Output:

```
Epoch 1/10
422/422 [==============================] - 57s 133ms/step - loss: 0.3786 -
accuracy: 0.8882 - val_loss: 0.0825 - val_accuracy: 0.9783
Epoch 2/10
422/422 [==============================] - 49s 117ms/step - loss: 0.1133 -
accuracy: 0.9661 - val_loss: 0.0569 - val_accuracy: 0.9850
Epoch 3/10
422/422 [==============================] - 50s 117ms/step - loss: 0.0847 -
accuracy: 0.9744 - val_loss: 0.0498 - val_accuracy: 0.9865
Epoch 4/10
422/422 [==============================] - 48s 114ms/step - loss: 0.0713 -
accuracy: 0.9777 - val_loss: 0.0415 - val_accuracy: 0.9890
Epoch 5/10
422/422 [==============================] - 47s 111ms/step - loss: 0.0638 -
accuracy: 0.9797 - val_loss: 0.0436 - val_accuracy: 0.9885
Epoch 6/10
422/422 [==============================] - 48s 114ms/step - loss: 0.0574 -
accuracy: 0.9824 - val_loss: 0.0360 - val_accuracy: 0.9910
Epoch 7/10
422/422 [==============================] - 51s 120ms/step - loss: 0.0512 -
accuracy: 0.9841 - val_loss: 0.0384 - val_accuracy: 0.9883
Epoch 8/10
422/422 [==============================] - 48s 114ms/step - loss: 0.0476 -
accuracy: 0.9846 - val_loss: 0.0321 - val_accuracy: 0.9905
Epoch 9/10
422/422 [==============================] - 53s 125ms/step - loss: 0.0439 -
accuracy: 0.9861 - val_loss: 0.0302 - val_accuracy: 0.9918
Epoch 10/10
422/422 [==============================] - 50s 117ms/step - loss: 0.0434 -
accuracy: 0.9862 - val_loss: 0.0315 - val_accuracy: 0.9908
```

```
prediction = model.predict(x_test)
```

```
313/313 [==============================] - 3s 9ms/step
```

```
prediction.shape
```

```
(10000, 10)
```

```
y_test.shape
```

```
(10000, 10)
```

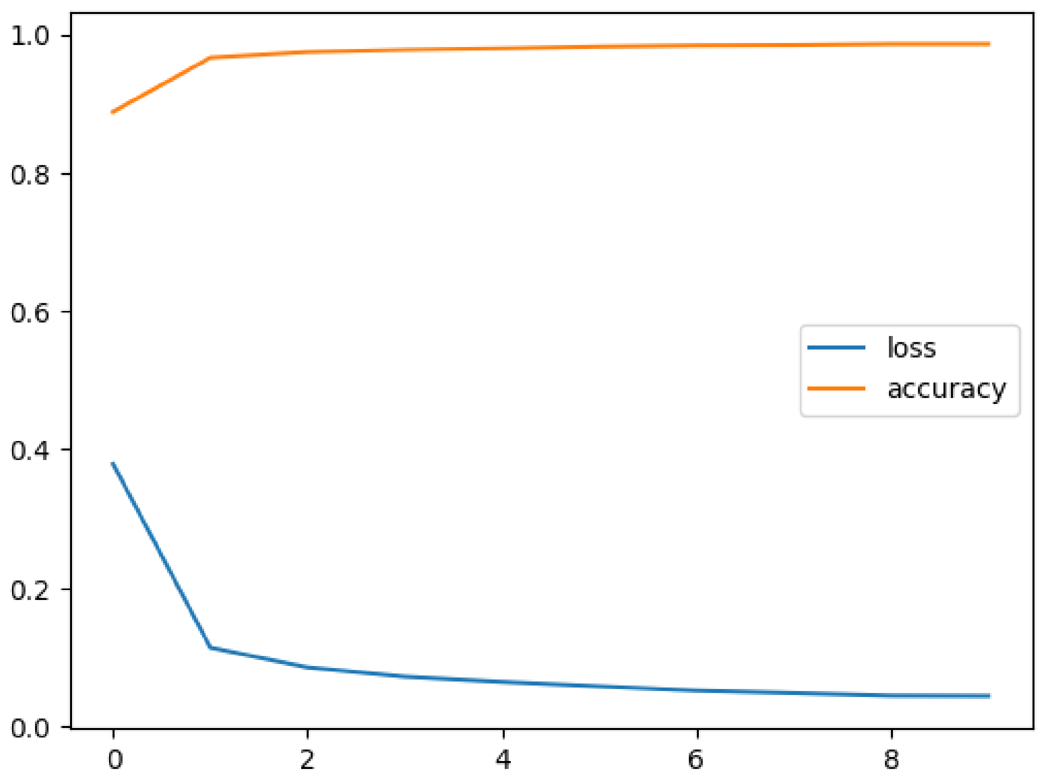
```
loss, accuracy = model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.0276 -
accuracy: 0.9908
```

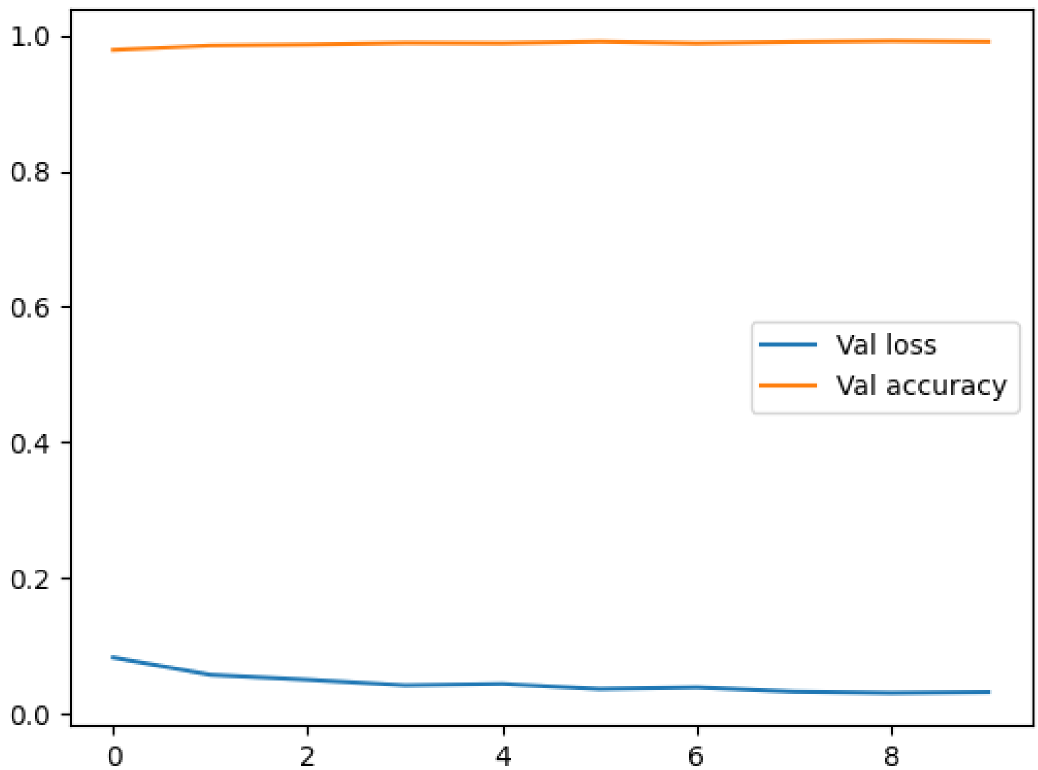
```
print(f"Loss of model is on testing data: {loss} and accuracy of model is on
testing data: {accuracy}")
```

```
Loss of model is on testing data: 0.027647219598293304 and accuracy of model
is on testing data: 0.9908000230789185
```

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['accuracy'], label='accuracy')
plt.legend()
plt.show()
```



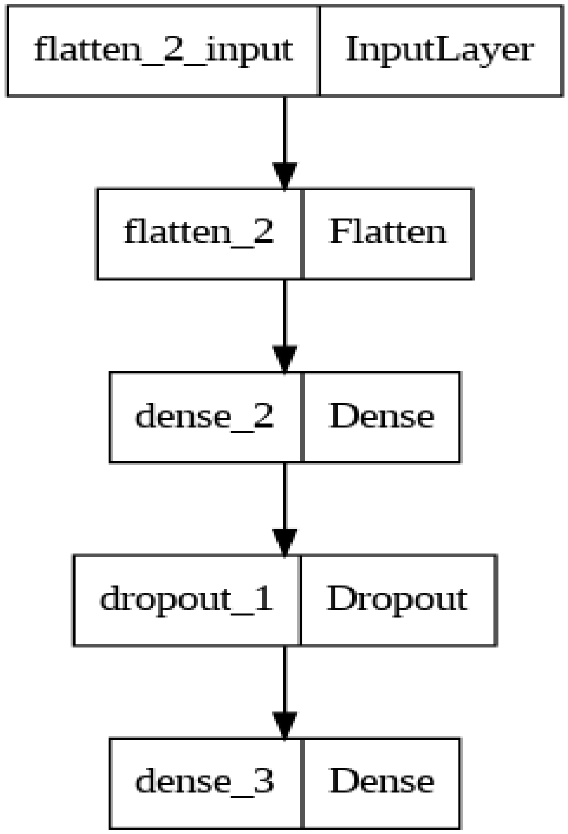
```
plt.plot(history.history['val_loss'], label='Val loss')
plt.plot(history.history['val_accuracy'], label='Val accuracy')
plt.legend()
plt.show()
```



Lab Assignment No. 13

Code:

```
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Dropout, Flatten
import numpy as np
import pandas as pd

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

x_test.shape

(10000, 28, 28)

x_test[0][0].shape

(28,)

model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

from keras.utils import plot_model
plot_model(model)
```




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

```
model.fit(x_train, y_train, epochs=20)
```

```
Epoch 1/20
1875/1875 [==============================] - 10s 5ms/step - loss: 2.4438 -
accuracy: 0.7533
Epoch 2/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.5959 -
accuracy: 0.8427
Epoch 3/20
1875/1875 [==============================] - 8s 4ms/step - loss: 0.4775 -
accuracy: 0.8741
Epoch 4/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.4282 -
accuracy: 0.8868
Epoch 5/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3929 -
accuracy: 0.8992
Epoch 6/20
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3806 -
accuracy: 0.9053
Epoch 7/20
```

```
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3620 - accuracy: 0.9105
Epoch 8/20
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3399 - accuracy: 0.9143
Epoch 9/20
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3321 - accuracy: 0.9180
Epoch 10/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3288 - accuracy: 0.9205
Epoch 11/20
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3168 - accuracy: 0.9228
Epoch 12/20
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3086 - accuracy: 0.9234
Epoch 13/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3144 - accuracy: 0.9246
Epoch 14/20
1875/1875 [==============================] - 15s 8ms/step - loss: 0.2978 - accuracy: 0.9266
Epoch 15/20
1875/1875 [==============================] - 10s 5ms/step - loss: 0.3015 - accuracy: 0.9278
Epoch 16/20
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2903 - accuracy: 0.9278
Epoch 17/20
1875/1875 [==============================] - 13s 7ms/step - loss: 0.2832 - accuracy: 0.9319
Epoch 18/20
1875/1875 [==============================] - 13s 7ms/step - loss: 0.2923 - accuracy: 0.9307
Epoch 19/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2869 - accuracy: 0.9321
Epoch 20/20
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2834 - accuracy: 0.9320

<keras.callbacks.History at 0x7f2103559030>
```

```
prediction = model.predict(x_test)
```

```
313/313 [==============================] - 1s 2ms/step
```

```
prediction[3].shape
```

```
(10,)
```

```
x_test[0][0]
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0], dtype=uint8)
```

```
prediction[0]
```

```
array([0.0000000e+00, 1.3933523e-19, 2.9269228e-15, 7.6539842e-12,
       7.4557314e-31, 4.5651832e-31, 0.0000000e+00, 9.9999994e-01,
       0.0000000e+00, 5.6528885e-29], dtype=float32)
```

```
prediction[0][7]
```

```
0.99999994
```

```
prediction[1][1]
```

```
9.435168e-11
```

```
pd.Series(prediction[2]).idxmax()
```

```
1
```

Output:

```
for i in range(10):
  print(f"Prediction of {prediction[i]} is ",
pd.Series(prediction[i]).idxmax())
```

```
Prediction of [0.0000000e+00 1.3933523e-19 2.9269228e-15 7.6539842e-12
7.4557314e-31
 4.5651832e-31 0.0000000e+00 9.9999994e-01 0.0000000e+00 5.6528885e-29] is  7
Prediction of [0.0000000e+00 9.4351679e-11 9.9999994e-01 1.3839746e-21
0.0000000e+00
 0.0000000e+00 0.0000000e+00 2.9035965e-19 2.7713451e-30 0.0000000e+00] is  2
Prediction of [0.0000000e+00 9.9999994e-01 0.0000000e+00 0.0000000e+00
9.8192167e-27
 8.6102357e-27 1.3392039e-24 6.7198263e-25 3.7141534e-19 0.0000000e+00] is  1
Prediction of [9.99999940e-01 0.00000000e+00 3.97543002e-18 2.87898494e-15
 4.17670492e-24 2.52059403e-17 4.16241627e-19 4.58924653e-15
 8.97595410e-23 1.08165845e-29] is  0
Prediction of [2.8448460e-23 1.3613716e-11 1.5544450e-14 2.5488617e-13
9.9999994e-01
 2.5542689e-13 2.4306238e-10 1.3248903e-12 3.4214355e-23 2.7959052e-08] is  4
Prediction of [0.0000000e+00 9.9999994e-01 0.0000000e+00 0.0000000e+00
2.3557797e-24
 5.3822781e-25 7.0089274e-24 7.2132002e-25 1.9120865e-19 0.0000000e+00] is  1
Prediction of [4.9079178e-32 3.3550220e-12 1.7724985e-10 8.9461727e-17
```

```
9.9999994e-01
 6.9597727e-16 2.2915382e-23 1.8836836e-13 1.2865342e-32 6.4851630e-10] is  4
Prediction of [6.5651999e-14 1.0212370e-12 4.1792095e-08 2.0033461e-03
2.7078322e-05
 9.8896944e-06 2.2923856e-31 2.4588793e-04 1.6237841e-13 9.9771374e-01] is  9
Prediction of [1.9678354e-04 2.5814313e-06 6.8850612e-05 5.4609153e-04
1.9435544e-09
 5.3101850e-01 4.3385461e-01 9.2009661e-10 2.9303946e-02 5.0086309e-03] is  5
Prediction of [8.8292691e-37 9.5649813e-14 4.9173587e-15 2.1960420e-09
8.8311272e-06
 3.7559491e-12 2.3588077e-30 1.2012749e-05 1.6326638e-16 9.9997908e-01] is  9
```

```
loss, accuracy = model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3086 -
accuracy: 0.9480
```

```
print(f"Loss of model is {loss} and accuracy of model is {accuracy}")
```

```
Loss of model is 0.3086441457271576 and accuracy of model is
0.9480000138282776
```