**Assignment No. 1**

**Code:**

import numpy as np

import matplotlib.pyplot as plt

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def relu(x):

return np.maximum(0, x)

def tanh(x):

return np.tanh(x)

def softmax(x):

return np.exp(x) / np.sum(np.exp(x))

# Create x values

x = np.linspace(-10, 10, 100)

# Create plots for each activation function fig, axs = plt.subplots(2, 2, figsize=(8, 8)) axs[0, 0].plot(x, sigmoid(x))

axs[0, 0].set\_title('Sigmoid') axs[0, 1].plot(x, relu(x))

axs[0, 1].set\_title('ReLU')

axs[1, 0].plot(x, tanh(x))

axs[1, 0].set\_title('Tanh')

axs[1, 1].plot(x, softmax(x)) axs[1, 1].set\_title('Softmax')

# Add common axis labels and titles fig.suptitle('Common Activation Functions') for ax in axs.flat:

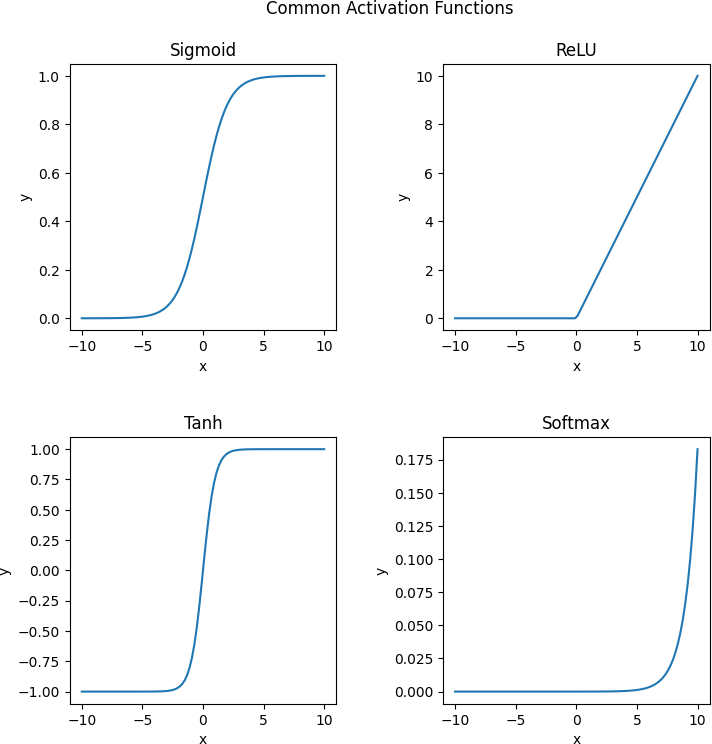
ax.set(xlabel='x', ylabel='y')

# Adjust spacing between subplots

plt.subplots\_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.4, hspace=0.4)

# Show the plot plt.show()

# Output:



**Assignment No. 2**

# Code:

# importing libraries import numpy as np

# function of checking thresold value def linear\_threshold\_gate(dot, T):

'''Returns the binary threshold output''' if dot >= T:

return 1 else:

return 0

# matrix of inputs input\_table = np.array([

[0,0], # both no

[0,1], # one no, one yes [1,0], # one yes, one no [1,1] # bot yes

])

print(f'input table:\n{input\_table}') weights = np.array([1,-1]) dot\_products = input\_table @ weights

T = 1

for i in range(0,4):

activation = linear\_threshold\_gate(dot\_products[i], T) print(f'Activation: {activation}')

**Output:** input table: [[0 0]

[0 1]

[1 0]

[1 1]]

Activation: 0

Activation: 0

Activation: 1

Activation: 0

# Assignment No.3

**Code:**

# import numpy as np

# class Perceptron:

# def \_\_init\_\_(self, input\_size, lr=0.1):

# self.W = np.zeros(input\_size + 1)

# self.lr = lr

# def activation\_fn(self, x):

# return 1 if x >= 0 else 0

# def predict(self, x):

# x = np.insert(x, 0, 1)

# z = self.W.T.dot(x)

# a = self.activation\_fn(z)

# return a

# def train(self, X, Y, epochs):

# for \_ in range(epochs):

# for i in range(Y.shape[0]):

# x = X[i]

# y = self.predict(x)

# e = Y[i] - y

# self.W = self.W + self.lr \* e \* np.insert(x, 0, 1)

# # Define the input data and labels

# X = np.array([

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

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

# [0,0,0,0,0,0,0,0,1,0], # 2

# [0,0,0,0,0,0,0,0,0,1], # 3

# [0,0,0,0,0,0,1,1,0,0], # 4

# [0,0,0,0,0,0,1,0,1,0], # 5

# [0,0,0,0,0,0,1,1,1,0], # 6

# [0,0,0,0,0,0,1,1,1,1], # 7

# [0,0,0,0,0,0,1,0,1,1], # 8

# [0,0,0,0,0,0,0,1,1,1], # 9

# ])

# Y = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])

# # Create the perceptron and train it

# perceptron = Perceptron(input\_size=10)

# perceptron.train(X, Y, epochs=100)

# # Test the perceptron on some input data

# test\_X = np.array([

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

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

# [0,0,0,0,0,0,0,0,1,0], # 2

# [0,0,0,0,0,0,0,0,0,1], # 3

# [0,0,0,0,0,0,1,1,0,0], # 4

# [0,0,0,0,0,0,1,0,1,0], # 5

# [0,0,0,0,0,0,1,1,1,0], # 6

# [0,0,0,0,0,0,1,1,1,1], # 7

# [0,0,0,0,0,0,1,0,1,1], # 8

# [0,0,0,0,0,0,0,1,1,1], # 9

# ])

# for i in range(test\_X.shape[0]):

# x = test\_X[i]

# y = perceptron.predict(x)

# print(f'{x} is {"even" if y == 0 else "odd"}')

# Output:

[0 0 0 0 0 0 1 0 0 0] is even

[0 0 0 0 0 0 0 1 0 0] is odd

[0 0 0 0 0 0 0 0 1 0] is even

[0 0 0 0 0 0 0 0 0 1] is odd

[0 0 0 0 0 0 1 1 0 0] is even

[0 0 0 0 0 0 1 0 1 0] is even

[0 0 0 0 0 0 1 1 1 0] is even

[0 0 0 0 0 0 1 1 1 1] is even

[0 0 0 0 0 0 1 0 1 1] is even

[0 0 0 0 0 0 0 1 1 1] is odd

# Assignment No. 4

**Code**:

# import numpy as np

# import matplotlib.pyplot as plt

# import pandas as pd

# # Load iris dataset

# iris = pd.read\_csv("iris.csv")

# # Extract features and target variable

# X = iris[['sepal\_length', 'petal\_length']].values

# y = iris['species'].apply(lambda x: 0 if x == 'setosa' else 1).values

# # Initialize weights and bias

# w = np.zeros(2)

# b = 0

# # Set learning rate and number of epochs

# lr = 0.1

# epochs = 50

# # Define perceptron function

# def perceptron(x, w, b):

# z = np.dot(x, w) + b

# return np.where(z >= 0, 1, 0)

# # Train the perceptron

# for epoch in range(epochs):

# for i in range(len(X)):

# x = X[i]

# target = y[i]

# output = perceptron(x, w, b)

# error = target - output

# w += lr \* error \* x

# b += lr \* error

# # Plot decision boundary

# x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

# y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

# xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02), np.arange(y\_min, y\_max, 0.02))

# Z = perceptron(np.c\_[xx.ravel(), yy.ravel()], w, b)

# Z = Z.reshape(xx.shape)

# plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

# # Plot data points

# plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

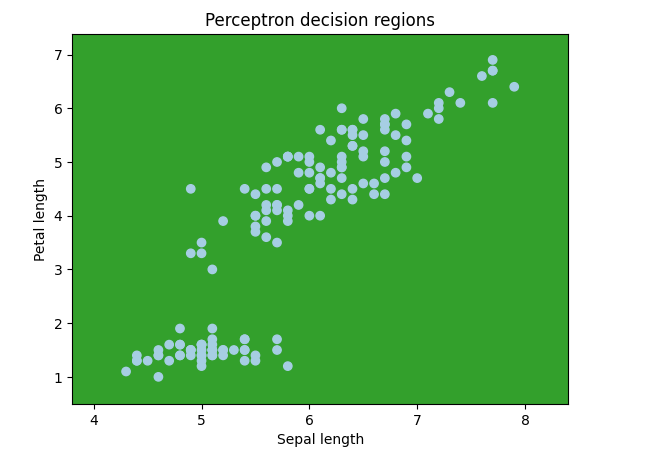
# plt.xlabel('Sepal length')

# plt.ylabel('Petal length')

# plt.title('Perceptron decision regions')

# plt.show()

# Output:



**Assignment No. 5**

# Code:

import numpy as np

# define two pairs of vectors x1 = np.array([1, 1, 1, -1])

y1 = np.array([1, -1])

x2 = np.array([-1, -1, 1, 1]) y2 = np.array([-1, 1])

# compute weight matrix W

W = np.outer(y1, x1) + np.outer(y2, x2)

# define BAM function def bam(x):

y = np.dot(W, x)

y = np.where(y >= 0, 1, -1) return y

# test BAM with inputs

x\_test = np.array([1, -1, -1, -1]) y\_test = bam(x\_test)

# print output print("Input x: ", x\_test) print("Output y: ", y\_test)

# Output:

Input x: [ 1 -1 -1 -1]

Output y: [ 1 -1]

# Assignment No.6

**Code:**

# import numpy as np

# class NeuralNetwork:

# def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

# self.W1 = np.random.randn(input\_size, hidden\_size)

# self.b1 = np.zeros((1, hidden\_size))

# self.W2 = np.random.randn(hidden\_size, output\_size)

# self.b2 = np.zeros((1, output\_size))

# def sigmoid(self, x):

# return 1 / (1 + np.exp(-x))

# def sigmoid\_derivative(self, x):

# return x \* (1 - x)

# def forward\_propagation(self, X):

# self.z1 = np.dot(X, self.W1) + self.b1

# self.a1 = self.sigmoid(self.z1)

# self.z2 = np.dot(self.a1, self.W2) + self.b2

# self.y\_hat = self.sigmoid(self.z2)

# return self.y\_hat

# def backward\_propagation(self, X, y, y\_hat):

# self.error = y - y\_hat

# self.delta2 = self.error \* self.sigmoid\_derivative(y\_hat)

# self.a1\_error = self.delta2.dot(self.W2.T)

# self.delta1 = self.a1\_error \* self.sigmoid\_derivative(self.a1)

# self.W2 += self.a1.T.dot(self.delta2)

# self.b2 += np.sum(self.delta2, axis=0, keepdims=True)

# self.W1 += X.T.dot(self.delta1)

# self.b1 += np.sum(self.delta1, axis=0)

# def train(self, X, y, epochs, learning\_rate):

# for i in range(epochs):

# y\_hat = self.forward\_propagation(X)

# self.backward\_propagation(X, y, y\_hat)

# if i % 100 == 0:

# print("Error at epoch", i, ":", np.mean(np.abs(self.error)))

# # Define the input and output datasets

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

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

# # Create a neural network with 2 input neurons, 4 neurons in the hidden layer, and 1 output neuron

# nn = NeuralNetwork(input\_size=2, hidden\_size=4, output\_size=1)

# # Train the neural network on the input and output datasets for 10000 epochs with a learning rate of 0.1

# nn.train(X, y, epochs=10000, learning\_rate=0.1)

# # Use the trained neural network to make predictions on the same input dataset

# predictions = nn.forward\_propagation(X)

# # Print the predictions

# print(predictions)

# Output:

[[5.55111512e-16] [6.66666667e-01] [6.66666667e-01] [6.66666667e-01]]

--------------------------------------------------------------------------------------------------------------------------