# Assignment No 01

**Name-** Thorve Avishkar Shrikrushna

## Roll No- 63

**Title-** Activation functions that are being used in neural networks.

**Program:**

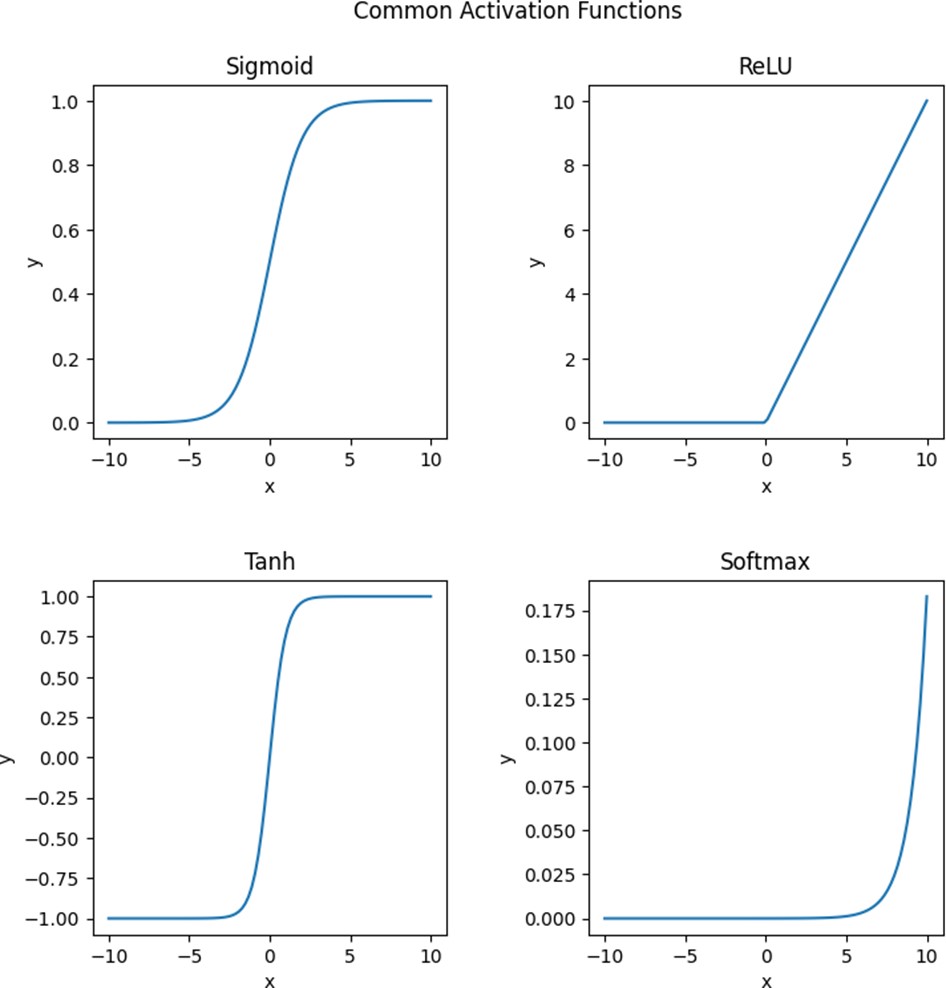
import numpy as np import matplotlib.pyplot as plt def sigmoid(x): return 1 / (1 + np.exp(-x)) def relu(x): return np.maximum(0, x) def tanh(x): return np.tanh(x) def softmax(x):

return np.exp(x) / np.sum(np.exp(x)) x = np.linspace(-10, 10, 100) 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') fig.suptitle('Common Activation Functions') for ax in axs.flat:

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

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

**Output:**



# 

# Assignment No 02

**Name-** Thorve Avishkar Shrikrushna

## Roll No- 63

## Title- AND NOT function using Mc Culloch-Pitts Neural Net.

**Program:**

import numpy as np def linear\_threshold\_gate(dot, T):

'''Returns the binary threshold output''' if dot >= T: return 1 else:

return 0 input\_table = np.array([

[0,0],

[0,1],

[1,0],

[1,1]

])

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]

1. 1]
2. 0]

[1 1]]

Activation: 0

Activation: 0

Activation: 1

Activation: 0

# Assignment No 03

**Name-** Thorve Avishkar Shrikrushna

## Roll No- 63

**Title-** Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII from 0 to 9.

**Program:**

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)

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 04

**Name-** Thorve Avishkar Shrikrushna

## Roll No- 63

**Title-** Demonstrate the Perceptron learning law with its decision regions.

**Program**:

import numpy as np import matplotlib.pyplot as plt from sklearn.datasets import load\_iris iris = load\_iris()

X = iris.data[:, [0, 2]] y = iris.target y = np.where(y == 0, 0, 1) w = np.zeros(2) b = 0 lr = 0.1 epochs = 50 def perceptron(x, w, b): z = np.dot(x, w) + b return np.where(z >= 0, 1, 0)

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

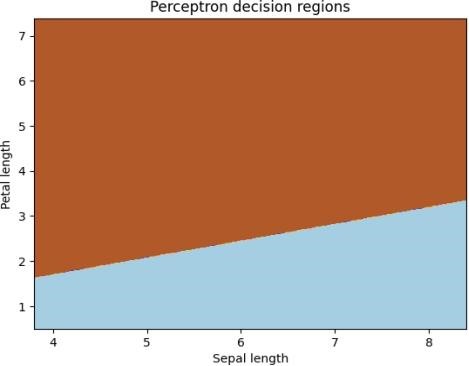
b += lr \* error x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

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) 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 05

**Name-** Thorve Avishkar Shrikrushna

### Roll No-63

**Title-**Bidirectional Associative Memory with two pairs of vectors.

**Program:**

import numpy as np x1=np.array([1,1,1,-1]) y1=np.array([1,-1]) x2=np.array([-1,-1,1,1]) y2=np.array([-1,1])

W=np.outer(y1,x1)+np.outer(y2,x2) def bam(x): y=np.dot(W,x) y=np.where(y>=0,1,-1) return y x\_test=np.array([1,-1,-1,-1]) y\_test=bam(x\_test) print("Input x:",x\_test)

print("Output:",y\_test)

**Output:**

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

Output: [ 1 -1]

## 

## Assignment No 06

**Name-** Thorve Avishkar Shrikrushna

### Roll No-63

**Title-** Artificial Neural Network trining process of Forward Propagation, Back Propagation.

**Program:**

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

return 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=0.1) for i in range(epochs):

y\_hat = self.forward\_propagation(X) self.backward\_propagation(X, y, y\_hat) if i % 1000 == 0:

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

return self.forward\_propagation(X) X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(input\_size=2, hidden\_size=4, output\_size=1) nn.train(X, y, epochs=10000) predictions = nn.predict(X) print("\nPredictions:\n", predictions)

**Output:**

[[5.55111512e-16]

[6.66666667e-63]

[6.66666667e-63]

[6.66666667e-63]]

## Assignment No 07

**Name-** Thorve Avishkar Shrikrushna

### Roll No- 63

**Title:** Python program to show back propagationnetwork for XOR function with Binary Input and Output.

**Program:**

import numpy as np class XORNetwork: def \_\_init\_\_(self): self.W1 = np.random.randn(2, 2) self.b1 = np.random.randn(2) self.W2 = np.random.randn(2, 1) self.b2 = np.random.randn(1) def sigmoid(self, x): return 1 / (1 + np.exp(-x)) def sigmoid\_derivative(self, x):

return x \* (1 - x) def forward(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.a2 = self.sigmoid(self.z2) return self.a2 def backward(self, X, y, output): self.output\_error = y - output

self.output\_delta = self.output\_error \* self.sigmoid\_derivative(output) self.z1\_error = self.output\_delta.dot(self.W2.T)

self.z1\_delta = self.z1\_error \* self.sigmoid\_derivative(self.a1) self.W1 += X.T.dot(self.z1\_delta) self.b1 += np.sum(self.z1\_delta, axis=0) self.W2 += self.a1.T.dot(self.output\_delta) self.b2 += np.sum(self.output\_delta, axis=0) def train(self, X, y, epochs): for \_ in range(epochs):

output = self.forward(X) self.backward(X, y, output) def predict(self, X): return self.forward(X) xor\_nn = XORNetwork()

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) y = np.array([[0], [1], [1], [0]]) xor\_nn.train(X, y, epochs=10000) predictions = xor\_nn.predict(X) print(predictions)

**Output:**

[[0.63075848]

[0.98777524]

[0.9877705 ]

[0.63148649]]

### Assignment No 08

**Name-** Thorve Avishkar Shrikrushna

#### Roll No-63

**Title-** Program for creating a back propagation feed-forward neural network.

**Program:**

import numpy as np def sigmoid(x):

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

learning\_rate = 0.1

num\_epochs = 100000

hidden\_weights = 2 \* np.random.random((2, 2)) - 1

output\_weights = 2 \* np.random.random((2, 1)) - 1

for \_ in range(num\_epochs):

hidden\_layer = sigmoid(np.dot(X, hidden\_weights))

output\_layer = sigmoid(np.dot(hidden\_layer, output\_weights))

output\_error = y - output\_layer

output\_delta = output\_error \* sigmoid\_derivative(output\_layer)

hidden\_error = output\_delta.dot(output\_weights.T)

hidden\_delta = hidden\_error \* sigmoid\_derivative(hidden\_layer)

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

print("Input:") print(X) print("Output:")

print(output\_layer)

**Output:**

Input:

[[00]

1. 1]
2. 0]

[1 1]]

Output:

[[0 61385986]

[0.63944088]

[0.8569871] [0.11295854]]