DNN - Experiment 03

- **SIA VASHIST**
- PRN: 20190802107

In [10]:

```
#INPUT LAYER

import numpy as np

class Perceptron:
    def __init__(self, inputs, targets, learning_rate=0.1, epochs=100):
        self.inputs = inputs
        self.targets = targets
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(inputs.shape[1])
        self.bias = 0

    def activation_function(self, x):
        # Step function
        if x >= 0:
            return 1
        else:
            return 0

    def train(self):
        for epoch in range(self.epochs):
            for i, x in enumerate(self.inputs):
                y = self.activation_function(np.dot(x, self.weights) +
                self.bias)
            error = self.targets[i] - y
            self.weights += self.learning_rate * error * x
            self.bias += self.learning_rate * error

    def predict(self, x):
        y = self.activation_function(np.dot(x, self.weights) + self.bias)
        return y

# Example usage:
inputs = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
targets = np.array([1, 0, 0, 0])
perceptron = Perceptron(inputs, targets)
perceptron.train()
print(perceptron.predict([1, 1])) # Output: 1
print(perceptron.predict([0, 0])) # Output: 0
```

```
0
0
```

In [16]:

```
#HIDDEN LAYER

import numpy as np

class Perceptron:
    def __init__(self, num_inputs, num_hidden, num_outputs):
        self.input_weights = np.random.randn(num_inputs, num_hidden)
        self.hidden_weights = np.random.randn(num_hidden, num_outputs)
        self.bias_input = np.zeros((1, num_hidden))
        self.bias_hidden = np.zeros((1, num_outputs))

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

    def predict(self, inputs):
        hidden_layer = np.dot(inputs, self.input_weights) + self.bias_input
        hidden_layer = self.sigmoid(hidden_layer)
        output = np.dot(hidden_layer, self.hidden_weights) + self.bias_hidden
        output = self.sigmoid(output)
        return output

perceptron = Perceptron(2, 4, 1)
inputs = np.array([[0.5, 0.5]])
output = perceptron.predict(inputs)
print(output)
```

```
[[0.58253405]]
```

In [17]:

```
#WEIGHTS

import numpy as np

class Perceptron:
    def __init__(self, weights, bias):
        self.weights = np.array(weights)
        self.bias = bias

    def activation(self, x):
        # Step function as activation
        return 1 if np.dot(self.weights, x) + self.bias >= 0 else 0

    def predict(self, inputs):
        return [self.activation(x) for x in inputs]

# Example usage
p = Perceptron([1, 2], -3)
print(p.predict([[1, 2], [2, 1], [-1, -2]])) # [1, 1, 0]
```

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

In [19]:

```
#ACTIVATION FUNCTION

import numpy as np

class Perceptron:
    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand(1)

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

    def predict(self, inputs):
        linear_combination = np.dot(inputs, self.weights) + self.bias
        return self.sigmoid(linear_combination)
```

In [20]:

```
#BIAS

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        self.weights = np.zeros(input_size+1) # Adding 1 for the bias
        self.learning_rate = learning_rate

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights[1:]) + self.weights[0] # bias is self.weights[0]
        prediction = 1 if weighted_sum >= 0 else 0
        return prediction

    def train(self, inputs, label):
        prediction = self.predict(inputs)
        error = label - prediction
        self.weights[1:] += self.learning_rate * error * inputs
        self.weights[0] += self.learning_rate * error # updating the bias
```

In [22]:

```
#OUTPUT LAYER

import numpy as np

class Perceptron:
    def __init__(self, inputs, targets):
        self.inputs = inputs
        self.targets = targets
        self.weights = np.random.uniform(-0.5, 0.5, (inputs.shape[1], 1))
        self.bias = np.random.uniform(-0.5, 0.5)

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def train(self, epochs, learning_rate):
        for epoch in range(epochs):
            output = self.sigmoid(np.dot(self.inputs, self.weights) + self.bias)
            error = self.targets - output
            d_weights = np.dot(self.inputs.T, error * self.sigmoid_derivative(output))
            d_bias = np.sum(error * self.sigmoid_derivative(output))
            self.weights += learning_rate * d_weights
            self.bias += learning_rate * d_bias

    def predict(self, inputs):
        return np.round(self.sigmoid(np.dot(inputs, self.weights) + self.bias))
```

In [23]:

```
import numpy as np

class SingleLayerPerceptron:
    def __init__(self, num_inputs, activation_function='sigmoid'):
        self.num_inputs = num_inputs
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand()
        self.activation_function = activation_function

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

    def relu(self, x):
        return np.maximum(0, x)

    def activate(self, x):
        if self.activation_function == 'sigmoid':
            return self.sigmoid(x)
        elif self.activation_function == 'relu':
            return self.relu(x)

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

    def train(self, inputs, targets, learning_rate):
        predictions = self.predict(inputs)
        errors = targets - predictions
        self.weights += learning_rate * np.dot(inputs.T, errors)
        self.bias += learning_rate * np.sum(errors)

perceptron = SingleLayerPerceptron(2)
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([0, 1, 1, 0])

for i in range(1000):
    for j in range(inputs.shape[0]):
        perceptron.train(inputs[j], targets[j], learning_rate=0.1)

print(perceptron.predict(inputs))
```

```
[0.51281762 0.5        0.48718237 0.47438158]
```