INDEX

Experiment 1: Implement McCulloch-Pitts Model

Code:

```
# Write a program to implement McCulloch Pitts Model
def mc_pitts(inputs, weights, threshold):
    if sum(i * w for i, w in zip(inputs, weights)) >= threshold:
        return 1
    else:
        return 0

inputs = [1, 0, 1]
weights = [1, 1, 1]
threshold = 2

print(mc_pitts(inputs, weights, threshold))
```

Output:

```
1
```

Experiment 2: Solve linearly separable problem using Perceptron Model

Code:

```
# Write a program for solving linearly separable problem using Perceptron Model.
import numpy as np

def activation(x): return 1 if x >= 0 else 0

def train(X, y, lr=0.1, epochs=10):
    w, b = np.zeros(X.shape[1]), 0
    for _ in range(epochs):
        for xi, yi in zip(X, y):
            error = yi - activation(np.dot(xi, w) + b)
            w += lr * error * xi
            b += lr * error
    return w, b

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

print(f'Weights: {w}, Bias: {b}')
```

Output:

```
Weights: [0.2 0.1], Bias: -0.20000000000000004
```

Experiment 3: Pattern classification using Perceptron Model

Code:

```
# Write a program for pattern classification using Perceptron Model.
import numpy as np

def activation(x): return 1 if x >= 0 else 0

def perceptron_train(X, y, lr=0.1, epochs=10):
    w, b = np.zeros(X.shape[1]), 0
    for _ in range(epochs):
        for xi, yi in zip(X, y):
            error = yi - activation(np.dot(xi, w) + b)
            w += lr * error * xi
            b += lr * error
    return w, b

def classify(X, w, b):
    return [activation(np.dot(xi, w) + b) for xi in X]

# Example usage
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Inputs
y = np.array([0, 0, 0, 1])  # Labels for AND operation
w, b = perceptron_train(X, y)
```

```
print(f'Weights: {w}, Bias: {b}')
print('Classifications:', classify(X, w, b))
```

Output:

Weights: [0.2 0.1], Bias: -0.20000000000000004

Classifications: [0, 0, 0, 1]

Experiment 4: XOR function program with Backpropagation algorithm

Code:

Write a program for XOR function (binary input and output) with momentum factor using backpropagation algorithm.

```
import math

def sigmoid(x): return 1 / (1 + math.exp(-x))
def sig_deriv(x): return x * (1 - x)

def train_xor(epochs=10000, lr=0.5, momentum=0.9):
    X, y = [[0, 0], [0, 1], [1, 0], [1, 1]], [0, 1, 1, 0]
    w1, w2, b1, b2, dw1, dw2, db1, db2 = [0.5, -0.5], [0.5, -0.5], [0.5, -0.5], 0.5, 0, 0, 0, 0

    for _ in range(epochs):
        for xi, yi in zip(X, y):
            h = [sigmoid(sum(x * w for x, w in zip(xi, w1[i::2])) + b1[i]) for i in range(2)]
            o = sigmoid(sum(h[i] * w2[i] for i in range(2)) + b2)

            error, d_out = yi - o, (yi - o) * sig_deriv(o)
            d_hid = [d_out * w2[i] * sig_deriv(h[i]) for i in range(2)]

            dw2, db2 = momentum * dw2 + lr * d_out * h[0], momentum * db2 + lr * d_out
```

```
        w1 = [momentum * dw1 + lr * d_hid[i] * x for i in range(2) for x in xi]
        b1 = [momentum * db1 + lr * d_hid[i] for i in range(2)]

    return w1, w2, b1, b2

w1, w2, b1, b2 = train_xor()
print(f"Weights: {w1}, {w2}, Biases: {b1}, {b2}")
```

Output:

Weights: [-0.009144119467964158, -0.009144119467964158, 0.009144119465796817, 0.009144119465796817], [0.5, -0.5], Biases: [-0.009144119467964158, 0.009144119465796817], 0.5

Experiment 5: Implement Discrete Hopfield Network

Code:

```
# Write a program to store a pattern (1 1 1 0). Test the network using Discrete Hopfield Net by giving the input with mistakes in First and Second position.

import numpy as np


def train_hopfield(pattern):
    size = len(pattern)
    return np.array([[0 if i == j else (2 * pattern[i] - 1) * (2 * pattern[j] - 1) for j in range(size)] for i in range(size)])


def hopfield_update(weights, input_pattern):
    return [1 if np.dot(weights[i], input_pattern) > 0 else 0 for i in range(len(input_pattern))]


pattern = [1, 1, 1, 0]
weights = train_hopfield(pattern)
test_pattern = [0, 0, 1, 0]  # Mistakes in first and second positions
for _ in range(10):  # Maximum iterations for convergence
    test_pattern = hopfield_update(weights, test_pattern)


print("Recovered Pattern:", test_pattern)
```

Output:

```
Recovered Pattern: [1, 1, 1, 0]
```

Experiment 6: Pattern storage of 10 digits with DH Network

Code:

```
# Program for Pattern storage of 10 digits with Discrete Hopfield Network.

import numpy as np

# Train Hopfield Network to store multiple patterns
def train_hopfield(patterns):
    size = len(patterns[0])
    weights = np.zeros((size, size))

    for p in patterns:
        p = [2 * x - 1 for x in p]  # Convert binary 0/1 to bipolar -1/1

        for i in range(size):
            for j in range(size):
                if i != j:
                    weights[i][j] += p[i] * p[j]
    return weights

# Update the input pattern
def hopfield_update(weights, input_pattern):
    return [1 if np.dot(weights[i], input_pattern) > 0 else 0 for i in
range(len(input_pattern))]
```

```
# Example patterns (10 binary digits)
patterns = [
    [1, 0, 0, 1, 1, 0, 0, 1, 1, 0],
    [0, 1, 1, 0, 0, 1, 1, 0, 0, 1],
    [1, 1, 0, 0, 1, 1, 0, 1, 0, 0],
    [0, 0, 1, 1, 0, 1, 0, 0, 1, 1],
    [1, 0, 1, 0, 0, 1, 1, 1, 0, 0],
    [0, 1, 0, 1, 1, 0, 0, 1, 1, 1],
    [1, 1, 1, 0, 0, 1, 0, 0, 1, 1],
    [0, 0, 0, 1, 1, 0, 1, 1, 0, 0],
    [1, 0, 0, 1, 1, 1, 0, 0, 1, 0],
    [0, 1, 1, 0, 1, 0, 1, 1, 0, 1]
]
weights = train_hopfield(patterns)

# Test the network with a noisy pattern
test_pattern = [1, 0, 0, 1, 1, 0, 0, 1, 0, 0]  # Example with some noise

for _ in range(10):  # Update until convergence
    test_pattern = hopfield_update(weights, test_pattern)

print("Recovered Pattern:", test_pattern)
```

Output:

```
Recovered Pattern: [0, 0, 0, 1, 1, 0, 0, 1, 1, 0]
```

Experiment 7: Fuzzy control algorithm.

Fuzzy Control is a control system that uses fuzzy logic to handle imprecision and uncertainty in real-world systems. Unlike traditional control systems that rely on binary logic (true/false), fuzzy control systems use fuzzy sets and membership functions to represent the degrees of truth.

Principles of Fuzzy Control Design:

Fuzzification: Converts crisp inputs into fuzzy values using membership functions. Each input belongs to a fuzzy set with a degree of membership.

Rule-Based Inference: The fuzzy system operates based on IF-THEN rules. Each rule maps fuzzy inputs to fuzzy outputs. These rules are created based on expert knowledge or heuristic reasoning.

Inference Engine: The engine applies the fuzzy rules to the fuzzified inputs and calculates fuzzy outputs. Common inference methods are Mamdani (fuzzy outputs) and Sugeno (crisp outputs).

Defuzzification: Converts the fuzzy output back into a crisp value, typically using methods like the centroid method.

Rule-Based Fuzzy Inference System (FIS):

A Fuzzy Inference System (FIS) uses fuzzy logic to perform reasoning. The system consists of:

Fuzzification Interface: Converts input values into fuzzy values.

Rule Base: A collection of fuzzy rules that map fuzzy inputs to fuzzy outputs.

Inference Engine: Calculates fuzzy output by applying the rules.

Defuzzification Interface: Converts fuzzy output into a crisp value.

Example of Rule-Based Fuzzy Inference System:

A fuzzy system can be designed to control the speed of a fan based on temperature and humidity:

Temperature: Cold, Warm, Hot

Humidity: Low, Medium, High

Fan Speed: Slow, Medium, Fast

Fuzzification involves determining how much each input (temperature and humidity) belongs to each fuzzy set. For example, a temperature of 30°C might be "Warm" with a membership degree of 0.8 and "Hot" with 0.2.

Inference Rules could be:

IF Temperature is Hot AND Humidity is Low THEN Fan Speed is Fast

IF Temperature is Warm AND Humidity is Medium THEN Fan Speed is Medium

IF Temperature is Cold AND Humidity is High THEN Fan Speed is Slow

The system then calculates the fan speed by applying the appropriate rules and uses defuzzification to get a crisp output, such as a fan speed in RPM.

Experiment 8: Multi-layer Feed Forward Neural Network

Code:

```
# Program to implement Multi layer feed forward neural networks.

import numpy as np

def sigmoid(x): return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x): return x * (1 - x)

def train_network(X, y, epochs=10000, lr=0.1):
    input_size, hidden_size, output_size = X.shape[1], 4, 1
    w1, w2 = np.random.rand(input_size, hidden_size),
np.random.rand(hidden_size, output_size)
    b1, b2 = np.zeros((1, hidden_size)), np.zeros((1, output_size))

    for _ in range(epochs):
        h = sigmoid(np.dot(X, w1) + b1)
        y_pred = sigmoid(np.dot(h, w2) + b2)
        error = y - y_pred

        d_output = error * sigmoid_derivative(y_pred)
        d_hidden = d_output.dot(w2.T) * sigmoid_derivative(h)

        w2 += h.T.dot(d_output) * lr
        b2 += np.sum(d_output, axis=0, keepdims=True) * lr
```

```
        w1 += X.T.dot(d_hidden) * lr
        b1 += np.sum(d_hidden, axis=0, keepdims=True) * lr


    return w1, w2, b1, b2

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

w1, w2, b1, b2 = train_network(X, y)
output = sigmoid(sigmoid(np.dot(X, w1) + b1).dot(w2) + b2)
print(output)
```

Output:

```
[[0.05433772]
 [0.95135778]
 [0.9512438 ]
 [0.05091949]]
```

Experiment 9: Competitive learning Neural Networks for pattern clustering

Code:

```
# Program to implement Competitive learning neural networks for pattern clustering.

import numpy as np

def competitive_learning(X, num_neurons, epochs=1000, learning_rate=0.1):
    # Initialize weights randomly for each neuron
    weights = np.random.rand(num_neurons, X.shape[1])

    for _ in range(epochs):
        # Pick a random pattern from input
        pattern = X[np.random.choice(X.shape[0])]

        # Compute the distance from each neuron weight to the input pattern
        distances = np.linalg.norm(weights - pattern, axis=1)

        # Find the winning neuron (the one with the smallest distance)
        winner = np.argmin(distances)

        # Update the winning neuron's weights
        weights[winner] += learning_rate * (pattern - weights[winner])

    return weights
```

```
# Example usage (pattern clustering)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1], [2, 2], [3, 3]])  # Sample data
num_neurons = 2  # Number of neurons for clustering

weights = competitive_learning(X, num_neurons)

print("Cluster centers (neurons' weights):")
print(weights)
```

Output:

```
Cluster centers (neurons' weights):
[[2.5154926  2.5154926 ]
 [0.42399146 0.46258807]]
```

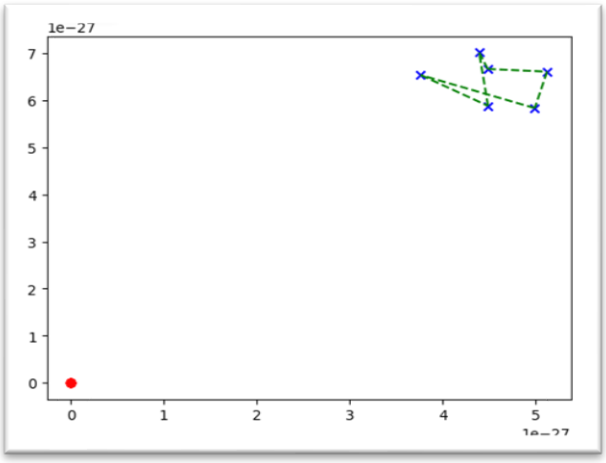
Experiment 10: Travelling salesman problem using self-organizing maps

Code:

```
# Program to implement Solution to travelling salesman problem using self organizing maps

import numpy as np

import random

import matplotlib.pyplot as plt


# Initialize cities (coordinates)

cities = np.array([[0, 0], [1, 2], [2, 4], [3, 1], [4, 3], [5, 5]])


# SOM initialization

som = np.random.rand(len(cities), 2)

lr, radius, epochs = 0.1, 1, 1000


# Training the SOM

for _ in range(epochs):
    random.shuffle(cities)
    for city in cities:
        dist = np.linalg.norm(som - city, axis=1)
        winner_idx = np.argmin(dist)
        som += lr * (city - som) * (np.linalg.norm(som - som[winner_idx], axis=1) < radius).reshape(-1, 1)
    lr *= 0.99
    radius *= 0.99
```

```
# Plot the results
plt.scatter(cities[:, 0], cities[:, 1], c='red')
plt.scatter(som[:, 0], som[:, 1], c='blue', marker='x')
for i in range(len(som)):
    plt.plot([som[i, 0], som[(i+1) % len(som), 0]], [som[i, 1], som[(i+1) % len(som), 1]], 'g--')
plt.show()

# Final path
print("Approximate path:", som)
```

Output:




```
Approximate path: [[4.48288421e-27 6.65977056e-27]
 [5.11994047e-27 6.60378888e-27]
 [4.98791093e-27 5.83035847e-27]
 [3.76240034e-27 6.53119982e-27]
 [4.48578361e-27 5.87767727e-27]
 [4.38959412e-27 7.01064174e-27]]
```