Experiment No 01

Program to implement logical AND using McCulloch Pitts neuron model

```
class McCullochPittsNeuron:
    def __init__(self, threshold):
        self.threshold = threshold
    def activate(self, inputs):
        total_input = sum(inputs)
        output = 1 if total_input >= self.threshold else 0
        return output
def logical_and(input1, input2):
    neuron = McCullochPittsNeuron(threshold=2)
    result = neuron.activate([input1, input2])
    return result
print("Logical AND of 0 and 0:", logical_and(0, 0))
print("Logical AND of 0 and 1:", logical_and(0, 1))
print("Logical AND of 1 and 0:", logical_and(1, 0))
print("Logical AND of 1 and 1:", logical_and(1, 1))
```

Output

```
Logical AND of 0 and 0: 0
Logical AND of 0 and 1: 0
Logical AND of 1 and 0: 0
Logical AND of 1 and 1: 1
```

Experiment No 02

Program to implement logical XOR using McCulloch Pitts neuron model

```
class McCullochPittsNeuron:
    def __init__(self, weights, threshold):
        self.weights = weights
        self.threshold = threshold

    def activate(self, inputs):
        total_input = sum([w * x for w, x in zip(self.weights, inputs)])
        output = 1 if total_input >= self.threshold else 0
        return output

def logical_xor(input1, input2):
    and_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=2)
    or_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=1)
    not_neuron = McCullochPittsNeuron(weights=[-1], threshold=0)

    and_result = and_neuron.activate([input1, input2])
    or_result = or_neuron.activate([input1, input2])
    not_result = not_neuron.activate([and_result])

    result = or_result and not_result
    return result
```

```
# Test the logical XOR function
print("Logical XOR of 0 and 0:", logical_xor(0, 0))
print("Logical XOR of 0 and 1:", logical_xor(0, 1))
print("Logical XOR of 1 and 0:", logical_xor(1, 0))
print("Logical XOR of 1 and 1:", logical_xor(1, 1))
```

Output

```
Logical XOR of 0 and 0: 0
Logical XOR of 0 and 1: 1
Logical XOR of 1 and 0: 1
Logical XOR of 1 and 1: 0
```

Experiment No 03

Write a program to implement logical AND using the Perceptron network

model

```
import numpy as np
class Perceptron:
    def __init__(self):
        self.weights = np.array([1, 1])
        self.bias = -1.5
   def predict(self, inputs):
        weighted_sum = np.dot(self.weights, inputs) + self.bias
        return 1 if weighted_sum >= 0 else 0
def logical_and(input1, input2):
    perceptron = Perceptron()
    return perceptron.predict(np.array([input1, input2]))
print("Logical AND of 0 and 0:", logical_and(0, 0))
print("Logical AND of 0 and 1:", logical_and(0, 1))
print("Logical AND of 1 and 0:", logical_and(1, 0))
print("Logical AND of 1 and 1:", logical_and(1, 1))
```

Output

```
Logical AND of 0 and 0: 0
Logical AND of 0 and 1: 0
Logical AND of 1 and 0: 0
Logical AND of 1 and 1: 1
```

Experiment No 04

Write a program to implement logical OR using the Perceptron network

model

```
import numpy as np
class Perceptron:
    def __init__(self):
        self.weights = np.array([1, 1])
        self.bias = -0.5
    def predict(self, inputs):
        weighted_sum = np.dot(self.weights, inputs) + self.bias
        return 1 if weighted_sum >= 0 else 0
def logical_or(input1, input2):
    perceptron = Perceptron()
    return perceptron.predict(np.array([input1, input2]))
print("Logical OR of 0 and 0:", logical_or(0, 0))
print("Logical OR of 0 and 1:", logical_or(0, 1))
print("Logical OR of 1 and 0:", logical_or(1, 0))
print("Logical OR of 1 and 1:", logical_or(1, 1))
```

Output

```
Logical OR of 0 and 0: 0
Logical OR of 0 and 1: 1
Logical OR of 1 and 0: 1
Logical OR of 1 and 1: 1
```

Experiment No 05

Write a program to implement logical AND using the Adaline network model

```
import numpy as np
class Adaline:
    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand(1)
        self.lr = 0.1
    def activation(self, x):
        return 1 if x >= 0 else 0
    def predict(self, inputs):
        net_input = np.dot(inputs, self.weights) + self.bias
        return self.activation(net_input)
    def train(self, inputs, targets, epochs):
        for _ in range(epochs):
            for i in range(len(inputs)):
                output = self.predict(inputs[i])
                error = targets[i] - output
                self.weights += self.lr * error * inputs[i]
                self.bias += self.lr * error
# Input data
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# Target outputs
targets = np.array([0, 0, 0, 1])
```

```
# Create and train Adaline network
adaline = Adaline(num_inputs=2)
adaline.train(inputs, targets, epochs=100)

# Test the network
print("Logical AND of 0 and 0:", adaline.predict([0, 0]))
print("Logical AND of 0 and 1:", adaline.predict([0, 1]))
print("Logical AND of 1 and 0:",adaline.predict([1, 0]))
print("Logical AND of 1 and 1:", adaline.predict([1, 1]))
```

Output

```
Logical AND of 0 and 0: 0
Logical AND of 0 and 1: 0
Logical AND of 1 and 0: 0
Logical AND of 1 and 1: 1
```

Experiment No 06

Write a program to implement logical OR using the Adaline network model

```
import numpy as np
class Adaline:
    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand(1)
        self.lr = 0.1
    def activation(self, x):
        return 1 if x >= 0 else 0
    def predict(self, inputs):
        net_input = np.dot(inputs, self.weights) + self.bias
        return self.activation(net_input)
    def train(self, inputs, targets, epochs):
        for _ in range(epochs):
            for i in range(len(inputs)):
                output = self.predict(inputs[i])
                error = targets[i] - output
                self.weights += self.lr * error * inputs[i]
                self.bias += self.lr * error
# Input data for logical OR
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# Target outputs for logical OR
targets = np.array([0, 1, 1, 1])
```

```python
# Create and train Adaline network

adaline = Adaline(num_inputs=2)

adaline.train(inputs, targets, epochs=100)


# Test the network

print("Logical OR of 0 and 0:", adaline.predict([0, 0]))

print("Logical OR of 0 and 1:", adaline.predict([0, 1]))

print("Logical OR of 1 and 0:",adaline.predict([1, 0]))

print("Logical OR of 1 and 1:", adaline.predict([1, 1]))
```

Output

```
Logical OR of 0 and 0: 0
Logical OR of 0 and 1: 1
Logical OR of 1 and 0: 1
Logical OR of 1 and 1: 1
```

Experiment No 07

Write a program to implement logical XOR using the Madaline network

model

```
import numpy as np
def activation_function(x):
    return np.where(x >= 0, 1, -1)
class MADALINE:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.randn(hidden_size, input_size)
        self.weights_hidden_output = np.random.randn(output_size, hidden_size)
    def train(self, X, y, learning_rate=0.1, epochs=100):
        for _ in range(epochs):
            for inputs, target in zip(X, y):
                # Forward pass
                hidden_net = np.dot(self.weights_input_hidden, inputs)
                hidden_output = activation_function(hidden_net)
                output_net = np.dot(self.weights_hidden_output, hidden_output)
                output = activation_function(output_net)
                # Backpropagation
                output_error = target - output
                hidden_error = np.dot(self.weights_hidden_output.T, output_error)
```

```
            # Weight updates
            self.weights_hidden_output += learning_rate * np.outer(output_error, hidden_output)
            self.weights_input_hidden += learning_rate * np.outer(hidden_error, inputs)
    def predict(self, inputs):
        hidden_net = np.dot(self.weights_input_hidden, inputs)
        hidden_output = activation_function(hidden_net)
        output_net = np.dot(self.weights_hidden_output, hidden_output)
        return activation_function(output_net)
# Define the XOR inputs and targets
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[1], [-1], [-1], [1]])
# Create and train the MADALINE network
madaline = MADALINE(input_size=2, hidden_size=3, output_size=1)
madaline.train(X, y)
# Predict XOR outputs
for inputs in X:
    print("Inputs:", inputs, "Prediction:", madaline.predict(inputs))
```

Output

```
Inputs: [0 0] Prediction: [1]
Inputs: [0 1] Prediction: [-1]
Inputs: [1 0] Prediction: [-1]
Inputs: [1 1] Prediction: [1]
```

Experiment No 08

Write a program to implement a Backpropagation network

```
import numpy as np

# Define the sigmoid activation function and its derivative

def sigmoid(x):

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

def sigmoid_derivative(x):

    return x * (1 - x)

# Define the XOR dataset

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

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

# Initialize random weights and biases

np.random.seed(1)

input_neurons = 2

hidden_neurons = 2

output_neurons = 1

weights_input_hidden = np.random.uniform(size=(input_neurons,
hidden_neurons))

biases_input_hidden = np.random.uniform(size=(1, hidden_neurons))

weights_hidden_output = np.random.uniform(size=(hidden_neurons,
output_neurons))

biases_hidden_output = np.random.uniform(size=(1, output_neurons))

# Set hyperparameters

learning_rate = 0.1

epochs = 100000
```

```
# Training loop
for epoch in range(epochs):
    # Forward propagation
    hidden_input = np.dot(X, weights_input_hidden) + biases_input_hidden
    hidden_output = sigmoid(hidden_input)
    output = sigmoid(np.dot(hidden_output, weights_hidden_output) +
biases_hidden_output)

    # Backpropagation
    output_error = y - output
    d_output = output_error * sigmoid_derivative(output)

    hidden_error = d_output.dot(weights_hidden_output.T)
    d_hidden = hidden_error * sigmoid_derivative(hidden_output)

    # Update weights and biases
    weights_hidden_output += hidden_output.T.dot(d_output) * learning_rate
    biases_hidden_output += np.sum(d_output, axis=0, keepdims=True) *
learning_rate
    weights_input_hidden += X.T.dot(d_hidden) * learning_rate
    biases_input_hidden += np.sum(d_hidden, axis=0, keepdims=True) *
learning_rate

# Print final output after training
print("Final Output after training:")
print(np.round(output))
```

Output

```
Final Output after training:
[[0.]
 [1.]
 [1.]
 [0.]]
```

Experiment No 09

Write a program to implement the various primitive operations of classical sets

```
class ClassicalSet:
    def __init__(self, elements):
        self.elements = elements
    def union(self, other_set):
        return ClassicalSet(list(set(self.elements) | set(other_set.elements)))
    def intersection(self, other_set):
        return ClassicalSet(list(set(self.elements) & set(other_set.elements)))
    def difference(self, other_set):
        return ClassicalSet(list(set(self.elements) - set(other_set.elements)))
    def is_subset(self, other_set):
        return set(self.elements).issubset(set(other_set.elements))
    def is_superset(self, other_set):
        return set(self.elements).issuperset(set(other_set.elements))
    def is_disjoint(self, other_set):
        return set(self.elements).isdisjoint(set(other_set.elements))
    def is_equal(self, other_set):
        return set(self.elements) == set(other_set.elements)
    def is_empty(self):
        return len(self.elements) == 0
    def get_elements(self):
        return self.elements
set1 = ClassicalSet([1, 2, 3, 4])
set2 = ClassicalSet([3, 4, 5, 6])
```

```python
# Union
print("Union:", set1.union(set2).get_elements())
# Intersection
print("Intersection:", set1.intersection(set2).get_elements())
# Difference
print("Difference:", set1.difference(set2).get_elements())
# Subset
print("Is set1 a subset of set2?", set1.is_subset(set2))
# Superset
print("Is set1 a superset of set2?", set1.is_superset(set2))
# Disjoint
print("Are set1 and set2 disjoint?", set1.is_disjoint(set2))
# Equality
print("Are set1 and set2 equal?", set1.is_equal(set2))
# Empty set
empty_set = ClassicalSet([])
print("Is the set empty?", empty_set.is_empty())
```

Output

```
Union: [1, 2, 3, 4, 5, 6]
Intersection: [3, 4]
Difference: [1, 2]
Is set1 a subset of set2? False
Is set1 a superset of set2? False
Are set1 and set2 disjoint? False
Are set1 and set2 equal? False
Is the set empty? True
```

Experiment No 10

Write a program to implement various primitive operations on fuzzy sets with Dynamic Components

```
class FuzzySet:
    def __init__(self, elements):
        self.elements = elements
        self.membership_degrees = {element: 0.0 for element in elements}
    def set_membership_degree(self, element, degree):
        if element in self.elements:
            self.membership_degrees[element] = degree
        else:
            print(f"Element '{element}' not in the set.")
    def get_membership_degree(self, element):
        if element in self.elements:
            return self.membership_degrees[element]
        else:
            print(f"Element '{element}' not in the set.")
            return 0.0  # Return a default value
    def union(self, other_set):
        new_set = FuzzySet(list(set(self.elements) | set(other_set.elements)))
        for element in new_set.elements:
            degree_self = self.get_membership_degree(element)
            degree_other = other_set.get_membership_degree(element)
            new_set.set_membership_degree(element, max(degree_self,
degree_other))
        return new_set
```

```
    def intersection(self, other_set):
        new_set = FuzzySet(list(set(self.elements) & set(other_set.elements)))
        for element in new_set.elements:
            degree_self = self.get_membership_degree(element)
            degree_other = other_set.get_membership_degree(element)
            new_set.set_membership_degree(element, min(degree_self,
degree_other))
        return new_set
    def complement(self):
        new_set = FuzzySet(self.elements)
        for element in new_set.elements:
            new_set.set_membership_degree(element, 1 -
self.get_membership_degree(element))
        return new_set
    def print_set(self):
        print("Elements and Membership Degrees:")
        for element in self.elements:
            print(f"{element}: {self.get_membership_degree(element)}")
# Example usage
set1 = FuzzySet(['a', 'b', 'c', 'd'])
set2 = FuzzySet(['c', 'd', 'e', 'f'])
# Set membership degrees
set1.set_membership_degree('a', 0.6)
set1.set_membership_degree('b', 0.8)
set2.set_membership_degree('c', 0.7)
set2.set_membership_degree('d', 0.9)
```

Union

union_set = set1.union(set2)

print("Union:")

union_set.print_set()

Intersection

intersection_set = set1.intersection(set2)

print("\nIntersection:")

intersection_set.print_set()

Complement

complement_set = set1.complement()

print("\nComplement:")

complement_set.print_set()

Output

```
Element 'e' not in the set.
Element 'a' not in the set.
Element 'b' not in the set.
Element 'f' not in the set.
Union:
Elements and Membership Degrees:
e: 0.0
a: 0.6
b: 0.8
f: 0.0
d: 0.9
c: 0.7

Intersection:
Elements and Membership Degrees:
c: 0.0
d: 0.0

Complement:
Elements and Membership Degrees:
a: 0.4
b: 0.19999999999999996
c: 1.0
d: 1.0
```

Experiment No 11

Write a program to maximize f(x1+x2)=4x1+3x2 using a genetic algorithm

```
import numpy as np
def objective_function(x1, x2):
    return 4 * x1 + 3 * x2


def fitness_function(x1, x2):
    return objective_function(x1, x2)


# Genetic Algorithm parameters
population_size = 100
num_generations = 1000
mutation_rate = 0.1


# Initialize population
population = np.random.uniform(low=0, high=10, size=(population_size, 2))


# Main loop
for generation in range(num_generations):
    # Evaluate fitness of each individual
    fitness_values = fitness_function(population[:, 0], population[:, 1])


    # Selection: Roulette wheel selection
    probabilities = fitness_values / np.sum(fitness_values)
```

```
    selected_indices = np.random.choice(range(population_size),
size=population_size, p=probabilities)
    selected_population = population[selected_indices]

    # Crossover: Single-point crossover
    crossover_point = np.random.randint(1, 2)  # Single-point crossover for 2D
representation
    offspring = []
    for i in range(population_size // 2):
        parent1, parent2 = selected_population[i],
selected_population[population_size - i - 1]
        child1 = np.concatenate((parent1[:crossover_point],
parent2[crossover_point:]))
        child2 = np.concatenate((parent2[:crossover_point],
parent1[crossover_point:]))
        offspring.extend([child1, child2])

    # Mutation: Random mutation
    for i in range(population_size):
        if np.random.rand() < mutation_rate:
            offspring[i] = np.random.uniform(low=0, high=10, size=2)

    # Replacement: Elitism (replace the worst individuals with the best ones)
    combined_population = np.vstack((population, np.array(offspring)))
    combined_fitness = fitness_function(combined_population[:, 0],
combined_population[:, 1])
    sorted_indices = np.argsort(-combined_fitness)[:population_size]
    population = combined_population[sorted_indices]
```

Print the best individual and its fitness value

best_individual = population[0]

best_fitness = fitness_function(best_individual[0], best_individual[1])

print("Best individual:", best_individual)

print("Best fitness value:", best_fitness)

Output

```
Best individual: [9.98281496 9.93199002]
Best fitness value: 69.72722991912372
```

Experiment No 12

Write a program to minimize f(x)=x2 using a genetic algorithm

```
import numpy as np
# Define the objective function
def objective_function(x):
    return x**2
# Define the fitness function (minimization problem)
def fitness_function(x):
    return -objective_function(x)  # Negate the objective function for
minimization
# Genetic Algorithm parameters
population_size = 100
num_generations = 1000
mutation_rate = 0.1
# Initialize population
population = np.random.uniform(low=-10, high=10, size=(population_size,))
# Main loop
for generation in range(num_generations):
    # Evaluate fitness of each individual
    fitness_values = fitness_function(population)
    # Selection: Roulette wheel selection
    probabilities = fitness_values / np.sum(fitness_values)
    selected_indices = np.random.choice(range(population_size),
size=population_size, p=probabilities)
```

```
    selected_population = population[selected_indices]


    # Mutation: Random mutation
    for i in range(population_size):
        if np.random.rand() < mutation_rate:
            population[i] = np.random.uniform(low=-10, high=10)
    # Replacement: Elitism (replace the worst individuals with the best ones)
    combined_population = np.concatenate((population, selected_population))
    combined_fitness = fitness_function(combined_population)
    sorted_indices = np.argsort(combined_fitness)
    population = combined_population[sorted_indices[:population_size]]


# Print the best individual and its fitness value
best_individual = population[0]
best_fitness = fitness_function(best_individual)
print("Best individual:", best_individual)
print("Best fitness value:", -best_fitness)  # Negate back to get the actual fitness value (minimization)
```

Output

```
Best individual: 9.999520833108622
Best fitness value: 99.99041689177336
```