List of Programs

Exercise 1

Multiple Perceptron

Question

1. Write a Program to implement Multiple Perceptron Model.

Code

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Define the MLP model
def create_mlp(input_size, hidden_sizes, output_size):
    model = models.Sequential()

    # Add the input layer
    model.add(layers.InputLayer(input_shape=(input_size,)))

    # Add hidden layers
    for hidden_size in hidden_sizes:
        model.add(layers.Dense(hidden_size, activation='relu'))

    # Add the output layer
    model.add(layers.Dense(output_size, activation='softmax'))

    return model

# Example usage
if __name__ == "__main__":
    # Define the model architecture
    input_size = 10  # Example: Number of input features
    hidden_sizes = [64, 32]  # Example: Two hidden layers with 64 and 32 neurons,
    respectively
    output_size = 2  # Example: Number of output classes

    # Create the MLP model
    mlp_model = create_mlp(input_size, hidden_sizes, output_size)

    # Display the model summary
    mlp_model.summary()
```

Exercise 2

XOR with backpropagation algorithm

Question

2. Write a Program to implement XOR with backpropagation algorithm.

Code

```
import numpy as np

class SimplePerceptron:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases with random values
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = 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(self, inputs):
        # Forward pass through the network
        self.hidden_layer_activation = self.sigmoid(np.dot(inputs, self.weights_input_hidden
    ) + self.bias_hidden)
        self.output_layer_activation = self.sigmoid(np.dot(self.hidden_layer_activation,
    self.weights_hidden_output) + self.bias_output)

        return self.output_layer_activation

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network

        # Calculate output layer error and delta
        output_layer_error = targets - self.output_layer_activation
        output_layer_delta = output_layer_error * self.sigmoid_derivative(self.
    output_layer_activation)

        # Calculate hidden layer error and delta
        hidden_layer_error = output_layer_delta.dot(self.weights_hidden_output.T)
        hidden_layer_delta = hidden_layer_error * self.sigmoid_derivative(self.
    hidden_layer_activation)

        # Update weights and biases
        self.weights_hidden_output += self.hidden_layer_activation.T.dot(output_layer_delta)
     * learning_rate
        self.bias_output += np.sum(output_layer_delta, axis=0, keepdims=True) *
    learning_rate

        self.weights_input_hidden += inputs.T.dot(hidden_layer_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_layer_delta, axis=0, keepdims=True) *
    learning_rate
```

```
    def train(self, inputs, targets, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward and backward pass for each training example
            for i in range(len(inputs)):
                input_data = np.array([inputs[i]])
                target_data = np.array([targets[i]])

                # Forward pass
                output = self.forward(input_data)

                # Backward pass
                self.backward(input_data, target_data, learning_rate)

            # Print the mean squared error for every 100 epochs
            if epoch % 100 == 0:
                mse = np.mean(np.square(targets - self.forward(inputs)))
                print(f"Epoch {epoch}, Mean Squared Error: {mse}")

# Example usage
if __name__ == "__main__":
    # Define the dataset (XOR problem)
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([[0], [1], [1], [0]])

    # Create a simple perceptron model
    model = SimplePerceptron(input_size=2, hidden_size=4, output_size=1)

    # Train the model
    model.train(inputs, targets, epochs=1000, learning_rate=0.1)

    # Test the trained model
    test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    predictions = model.forward(test_inputs)
    print("Predictions:")
    print(predictions)
```

Exercise 3

Union, Intersection and Complement operations

Question

3. Write a Program to implement Union, Intersection and Complement operations in fuzzy set.

Code

```
Z = {}

def union(A, B):
    for x in A and B:
        Z[x] = max(A[x], B[x])
    return Z

def intersection(A, B):
    for x in A and B:
        Z[x] = min(A[x], B[x])
    return Z

def compliment(X):
    for x in X:
        Z[x] = round(1 - X[x],2)
    return Z

def get_membership_value(key):
    while True:
        value = float(input("Enter the Membership value (between 0 and 1) for " + key))
        if 0 <= value <= 1:
            return value
        else:
            print("Invalid input. Please enter a value between 0 and 1.")


A = {}
B = {}

no_items = int(input("Enter the numbers: "))
for _ in range(no_items):
    key = input("Enter the crispy set elements: ")
    value = get_membership_value(key)
    A[key] = value
    value = get_membership_value(key)
    B[key] = value

print(f"{A=} UNION {B=} Is {union(A,B)}")
print(f"{A=} Intersection {B=} Is {intersection(A,B)}")
print(f"COMPLIMENT OF {A=} IS {compliment(A)}")
print(f"COMPLIMENT OF {B=} IS {compliment(B)}")
```

Exercise 4

De-Morgan's Law

Question

4. Write a program to implement De-Morgan's Law in Fuzzy sets.

Code

```
Z = {}

def union(A, B):
    for x in A and B:
        Z[x] = max(A[x], B[x])
    return Z

def compliment(X):
    for x in X:
        Z[x] = round(1 - X[x],2)
    return Z

input("Program to prove DeMorgans Theorem!")

def get_membership_value(key):
    while True:
        value = float(input("Enter the Membership value (between 0 and 1) for " + key))
        if 0 <= value <= 1:
            return value
        else:
            print("Invalid input. Please enter a value between 0 and 1.")

A = {}
B = {}

no_items = int(input("Enter the numbers: "))
for _ in range(no_items):
    key = input("Enter the crispy set elements: ")
    value = get_membership_value(key)
    A[key] = value
    value = get_membership_value(key)
    B[key] = value

LHS = compliment(union(A, B))
RHS = union(compliment(A), compliment(B))

print(f"Fuzzy sets given {A=} and {B=}")
if LHS == RHS:
    print(f"DeMorgans is proved \n Since {LHS} = {RHS}")
```

Exercise 5

Max-Min composition

Question

5. Write a program to implement Fuzzy Relations(Max-Min composition)

Code

```
import numpy as np

def max_min_composition(R, S):
    result = np.zeros((R.shape[0], S.shape[1]))

    for i in range(R.shape[0]):
        for j in range(S.shape[1]):
            max_min = 0
            for k in range(R.shape[1]):
                max_min = max(max_min, min(R[i, k], S[k, j]))
            result[i, j] = max_min

    return result

R = np.array([[0.7, 0.6], [0.8, 0.3]])
S = np.array([[0.8, 0.1, 0.4], [0.5, 0.6, 0.7]])

result = max_min_composition(R, S)
print("Result of Max-min Composition:")
print(result)
```

Exercise 6

Washing Machine

Question

6. Write a program to implement Fuzzy Controller(Washing Machine)

Code

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Create fuzzy variables
dirtiness = ctrl.Antecedent(np.arange(0, 11, 1), 'dirtiness')
stain_type = ctrl.Antecedent(np.arange(0, 11, 1), 'stain_type')
washing_time = ctrl.Consequent(np.arange(0, 61, 1), 'washing_time')

# Define membership functions
dirtiness['low'] = fuzz.trimf(dirtiness.universe, [0, 0, 5])
dirtiness['medium'] = fuzz.trimf(dirtiness.universe, [0, 5, 10])
dirtiness['high'] = fuzz.trimf(dirtiness.universe, [5, 10, 10])

stain_type['low'] = fuzz.trimf(stain_type.universe, [0, 0, 5])
stain_type['medium'] = fuzz.trimf(stain_type.universe, [0, 5, 10])
stain_type['high'] = fuzz.trimf(stain_type.universe, [5, 10, 10])

washing_time['short'] = fuzz.trimf(washing_time.universe, [0, 0, 30])
washing_time['medium'] = fuzz.trimf(washing_time.universe, [0, 30, 60])
washing_time['long'] = fuzz.trimf(washing_time.universe, [30, 60, 60])

# Define rules
rule1 = ctrl.Rule(dirtiness['low'] & stain_type['low'], washing_time['short'])
rule2 = ctrl.Rule(dirtiness['medium'] & stain_type['medium'], washing_time['medium'])
rule3 = ctrl.Rule(dirtiness['high'] & stain_type['high'], washing_time['long'])

# Create control system
washing_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
washing_machine = ctrl.ControlSystemSimulation(washing_ctrl)

# Example usage
if __name__ == "__main__":
    # Set input values
    washing_machine.input['dirtiness'] = 7
    washing_machine.input['stain_type'] = 8

    # Compute the result
    washing_machine.compute()

    # Print the output
    print("Washing Time:", washing_machine.output['washing_time'])

    # Visualize the result
    washing_time.view(sim=washing_machine)
```

Exercise 7

Genetic algorithm

Question

7. Write a program to implement simple genetic algorithm

Code

```
import random

# Genetic Algorithm parameters
population_size = 10
chromosome_length = 6
mutation_rate = 0.1
target_chromosome = "110110"

def generate_population(population_size, chromosome_length):
    return [''.join(random.choice('01') for _ in range(chromosome_length)) for _ in range(
    population_size)]

def fitness(chromosome):
    return sum(bit == target_bit for bit, target_bit in zip(chromosome, target_chromosome))

def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

def mutate(chromosome, mutation_rate):
    mutated_chromosome = ''.join(
        bit if random.random() > mutation_rate else random.choice('01') for bit in
    chromosome
    )
    return mutated_chromosome

def select_parents(population):
    parents = random.choices(population, weights=[fitness(chromosome) for chromosome in
    population], k=2)
    return parents[0], parents[1]

def genetic_algorithm(population_size, chromosome_length, mutation_rate, generations):
    population = generate_population(population_size, chromosome_length)

    for generation in range(generations):
        # Evaluate fitness
        fitness_scores = [fitness(chromosome) for chromosome in population]

        # Select parents and create offspring
        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
```

```
            new_population.extend([child1, child2])

        # Replace the old population with the new one
        population = new_population

        # Display the best chromosome in each generation
        best_chromosome = max(population, key=fitness)
        print(f"Generation {generation + 1}: Best Chromosome - {best_chromosome}, Fitness -
    {fitness(best_chromosome)}")

        # Check for convergence
        if fitness(best_chromosome) == len(target_chromosome):
            print("Target chromosome reached!")
            break

if __name__ == "__main__":
    genetic_algorithm(population_size, chromosome_length, mutation_rate, generations=50)
```

Exercise 8

Genetic algorithm to maximize f(x)=x2

Question

8. Write a program for Genetic algorithm to maximize f(x)=x2

Code

```
import numpy as np

# Define the objective function to maximize
def objective_function(x):
    return x**2

# Genetic Algorithm parameters
population_size = 50
generations = 100
mutation_rate = 0.1

# Initialization: Generate a random population
population = np.random.uniform(-10, 10, size=population_size)

# Main loop
for generation in range(generations):
    # Evaluate the fitness of each individual in the population
    fitness = objective_function(population)

    # Select the top individuals based on fitness
    sorted_indices = np.argsort(fitness)[::-1]
    selected_population = population[sorted_indices[:population_size]]

    # Crossover: Create new individuals by combining pairs of selected individuals
    crossover_population = np.random.choice(selected_population, size=population_size)

    # Mutation: Introduce random changes to some individuals
    mutation_mask = np.random.rand(population_size) < mutation_rate
    mutation_population = np.random.uniform(-1, 1, size=population_size)
    crossover_population[mutation_mask] += mutation_population[mutation_mask]

    # Replace the old population with the new one
    population = crossover_population

# Find the best individual in the final population
best_individual_index = np.argmax(objective_function(population))
best_x = population[best_individual_index]

# Print the result
print(f"Optimal x: {best_x}")
print(f"Optimal f(x): {objective_function(best_x)}")
```

Exercise 9

Genetic algorithm to maximize f(x)=x2+4x-4

Question

9. Write a program for Genetic algorithm to maximize f(x)=x2+4x-4

Code

```
import random

def objective_function(x):
    """Function to maximize: f(x) = -x^2 + 4x - 4"""
    return -x**2 + 4*x - 4

def generate_population(population_size, lower_bound, upper_bound):
    """Generate a random population of potential solutions."""
    return [random.uniform(lower_bound, upper_bound) for _ in range(population_size)]

def fitness(x):
    """Fitness function based on the objective function."""
    return objective_function(x)

def crossover(parent1, parent2):
    """Perform crossover to create new individuals."""
    crossover_point = random.uniform(0, 1)
    child1 = crossover_point * parent1 + (1 - crossover_point) * parent2
    child2 = (1 - crossover_point) * parent1 + crossover_point * parent2
    return child1, child2

def mutate(individual, mutation_rate, mutation_strength):
    """Introduce random mutations to individuals."""
    if random.uniform(0, 1) < mutation_rate:
        return individual + random.uniform(-mutation_strength, mutation_strength)
    return individual

def genetic_algorithm(population_size, lower_bound, upper_bound, mutation_rate,
    mutation_strength, generations):
    """Genetic Algorithm to maximize the objective function."""
    population = generate_population(population_size, lower_bound, upper_bound)

    for generation in range(generations):
        population.sort(key=fitness, reverse=True)
        best_individual = population[0]

        print(f"Generation {generation + 1}: Best Individual - {best_individual}, Fitness -
    {fitness(best_individual)}")

        next_generation = [best_individual]

        while len(next_generation) < population_size:
            parent1, parent2 = random.choices(population[:5], k=2)  # Select top 5
    individuals for crossover
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate, mutation_strength)
            child2 = mutate(child2, mutation_rate, mutation_strength)
```

```
            next_generation.extend([child1, child2])

        population = next_generation

if __name__ == "__main__":
    genetic_algorithm(population_size=10, lower_bound=-10, upper_bound=10, mutation_rate
    =0.1, mutation_strength=2, generations=50)
```

Exercise 10

Genetic Algorithm for TSP

Question

10. Write a program to show Multiobjective optimization in Genetic Algorithm for travelling salesman problem.

Code

```
import random
import numpy as np
from deap import base, creator, tools, algorithms

# Define the problem: Multi-objective TSP
creator.create("FitnessMulti", base.Fitness, weights=(-1.0, -1.0))  # Minimize both distance
     and time
creator.create("Individual", list, fitness=creator.FitnessMulti)

# Define the TSP parameters
num_cities = 10
city_coordinates = {i: (random.uniform(0, 100), random.uniform(0, 100)) for i in range(
    num_cities)}

# Function to calculate distance between two cities
def distance(city1, city2):
    x1, y1 = city_coordinates[city1]
    x2, y2 = city_coordinates[city2]
    return np.sqrt((x1 - x2)**2 + (y1 - y2)**2)

# Function to calculate total distance and time for a tour
def evaluate_tsp(individual):
    total_distance = sum(distance(individual[i], individual[i + 1]) for i in range(len(
    individual) - 1))
    total_time = len(individual)  # Simple representation of time based on the number of
    cities visited
    return total_distance, total_time

# Genetic Algorithm parameters
toolbox = base.Toolbox()
toolbox.register("indices", random.sample, range(num_cities), num_cities)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("mate", tools.cxOrdered)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.2)
toolbox.register("select", tools.selNSGA2)
toolbox.register("evaluate", evaluate_tsp)

def main():
    # Create an initial population of individuals
    population = toolbox.population(n=50)

    # Number of generations
    ngen = 100

    # Statistics to be recorded
    stats = tools.Statistics(lambda ind: ind.fitness.values)
```

```
    stats.register("avg", np.mean, axis=0)
    stats.register("min", np.min, axis=0)
    stats.register("max", np.max, axis=0)

    # Run the Genetic Algorithm
    algorithms.eaMuPlusLambda(population, toolbox, mu=50, lambda_=100, cxpb=0.7, mutpb=0.2,
    ngen=ngen,
                              stats=stats, halloffame=None, verbose=True)

    # Print the Pareto front solutions
    pareto_front = tools.sortNondominated(population, len(population), first_front_only=True
    )[0]
    print("Pareto Front:")
    for ind in pareto_front:
        print(f"Objectives: {ind.fitness.values}, Tour: {ind}")

if __name__ == "__main__":
    main()
```