Hints for Exercises in **Chapter 3**

1. Backpropagation in neural networks

Answer:
Backpropagation is an algorithm used to train neural networks by adjusting weights based on the error between predicted and actual outputs. It uses gradient descent to minimize the loss function by propagating errors backward through the network.

Hint:
Think of it as teaching the network by showing it what it got wrong and how to improve.

---

2. ReLU vs. Sigmoid activation functions

Answer:

- **ReLU (Rectified Linear Unit):** Outputs zero for negative inputs and the input itself for positive values. It's fast and helps avoid vanishing gradients.
- **Sigmoid:** Outputs values between 0 and 1, useful for probabilities but can suffer from vanishing gradients.

Hint:
Try plotting both functions and observe how they behave for large and small inputs.

---

3. Fuzzy sets and membership functions

Answer:
Fuzzy sets allow partial membership, unlike classical sets. A **membership function** assigns a degree (0 to 1) to indicate how strongly an element belongs to a fuzzy set.

Hint:
Imagine describing temperature as "hot", how hot is hot?

---

4. Key components of a fuzzy control system

Answer:

1. **Fuzzification:** Converts inputs into fuzzy values
2. **Rule base:** Contains fuzzy IF-THEN rules
3. **Inference engine:** Applies rules to fuzzy inputs
4. **Defuzzification:** Converts fuzzy output to a crisp value

Hint:
Think of how a thermostat might decide to heat based on “cold” or “very cold.”

---

5. Fuzzy logic vs. Boolean logic

Answer:
Boolean logic uses binary values (true/false), while fuzzy logic allows degrees of truth. Fuzzy logic is better for handling uncertainty and imprecision.

Hint:
Consider how you’d describe someone’s mood, not just “happy” or “sad.”

---

6. Genetic algorithms: selection, crossover, mutation

Answer:
Genetic algorithms mimic natural evolution.

- **Selection:** Chooses the best individuals
- **Crossover:** Combines parts of two individuals
- **Mutation:** Randomly alters genes to maintain diversity

Hint:
Think of breeding animals with desired traits.

---

7. Pros and cons of genetic algorithms

Answer:
Advantages: Good for complex, non-linear problems; doesn’t require gradient info
Disadvantages: Can be slow; may not guarantee optimal solution

Hint:
Compare with gradient descent, what happens when the landscape is rugged?

8. Using genetic algorithms for optimization

Answer:
They explore large solution spaces by evolving candidate solutions. Useful in scheduling, design, and tuning parameters.

Hint:
Try solving a maze or optimizing a travel route using evolution.

---

9. Hybrid systems: neural networks + fuzzy logic

Answer:
Combining neural networks (learning) with fuzzy logic (reasoning) creates systems that can learn and explain decisions called neuro-fuzzy systems.

Hint:
Imagine a robot that learns how to walk and explains why it slowed down.

---

10. Optimizing fuzzy controllers or neural networks with genetic algorithms

Answer:
Genetic algorithms can tune fuzzy rules or neural network weights and architectures, improving performance without manual tweaking.

Hint:
Think of evolving better decision rules or network designs over time.

---

11. Feedforward neural network for Iris dataset

Answer:
A simple neural network with one hidden layer was trained on the Iris dataset using scikit-learn. It achieved **100% accuracy** on the test set.

Hint:
Try changing the number of neurons or layers and observe the effect.

---

12. Experimenting with neural network hyperparameters

Answer:
We tested different learning rates and hidden layer configurations on the Iris dataset. All configurations achieved **100% accuracy**, showing that the dataset is simple enough for various architectures to perform well.

Hint:
Try more complex datasets to see how hyperparameters affect performance.

---

13. Designing a fuzzy logic controller

Answer:
For a temperature control system:

- **Inputs:** Temperature (e.g., cold, warm, hot)
- **Output:** Heater level (e.g., low, medium, high)
- **Rules:** IF temperature is cold THEN heater is high
  Use fuzzification, rule evaluation, and defuzzification to control output.

Hint:
Sketch a rule table and membership functions to visualize the system.

---

14. Defuzzification methods

Answer:

- **Centroid:** Calculates the center of gravity of the fuzzy set
- **Mean of Maxima:** Averages the values with maximum membership
  Centroid is more accurate; mean of maxima is simpler.

Hint:
Try both methods on the same fuzzy output and compare results.

---

15. Genetic algorithm for optimization

Answer:
A **Genetic Algorithm (GA)** is a search heuristic inspired by the process of **natural selection**, a concept from evolutionary biology. It's often used to find high-quality

solutions to optimization and search problems by mimicking the biological processes of evolution, namely: **selection, crossover, and mutation**.

Here is a step-by-step elaboration of how a Genetic Algorithm works, using the example of maximizing a function like $f(x)=\sin(x)+\cos(x)$ over a given domain (e.g., $x\in[0,2\pi]$).

Genetic Algorithm for Optimization:

1. Initialization (Creating the First Population)

The process begins by randomly generating a **population** of candidate solutions. Each solution is called an **individual** or **chromosome**.

- **Representation:** The variable x in our function f(x) needs to be encoded into a format the algorithm can manipulate, typically a binary string (the "chromosome").
    - *Example:* If we want a precision of 4 decimal places for x in $[0,2\pi]$, we'd convert the floating-point number x into its binary representation.
- **Initial Population:** A set of these binary strings (say, 50 of them) are randomly created to form the first generation.

2. Evaluation (Fitness Function)

Next, the quality of each individual solution in the population is assessed using a **fitness function**. The fitness function is the objective function we want to maximize.

- **Fitness Calculation:** For each chromosome (binary string), it is first decoded back into a value for x. This x value is then plugged into the objective function.
    - *Example:* For a given chromosome representing $x1=1.05$, the fitness is $f(1.05)=\sin(1.05)+\cos(1.05)$. The higher the function value, the **fitter** the individual.

3. Selection (Reproduction)

Based on their fitness, individuals are selected to become "parents" for the next generation. This step ensures that **fitter individuals have a higher probability of being chosen** to pass on their genetic material (their encoded x values).

- **Method:** A common technique is **Roulette Wheel Selection**, where the probability of an individual being selected is proportional to its fitness score relative to the total fitness of the population. Individuals with better f(x) values get a larger "slice" of the wheel.

4. Crossover (Recombination)

This step introduces genetic diversity and exploration. Selected parents exchange parts of their chromosomes to create new offspring.

- **Mechanism:** Two selected parent chromosomes are chosen, a **crossover point** is randomly selected, and the parts of the chromosomes after that point are swapped.
  - *Example:*
    - Parent 1: 101**1001**
    - Parent 2: 010**0110**
    - (Crossover point after the third digit)
    - Child 1: 101**0110**
    - Child 2: 010**1001**
  - The resulting children are new candidate solutions that inherit traits from both parents.

5. Mutation

Mutation introduces small, random changes into the chromosomes, preventing the algorithm from getting stuck in a local optimum and ensuring complete exploration of the search space.

- **Mechanism:** With a very small **mutation probability** (e.g., 1%), a randomly selected bit (a '0' or '1') in a chromosome is flipped.
  - *Example:* If a chromosome is 1010110, a mutation might flip the fourth bit to produce 101**1**110.

6. Replacement (New Generation)

The new population of offspring (created via crossover and mutation) replaces the old population. This new set of individuals forms the next generation.

7. Termination

The entire process (Steps 2-6) repeats for many generations. The algorithm stops when a **termination condition** is met:

- A predefined **maximum number of generations** has been reached.
- A satisfactory solution (a specific f(x) value) has been found.
- The **population's fitness stops improving** (convergence).

The fittest individual in the final generation is presented as the optimal (or near-optimal) solution to the optimization problem. For $f(x)=\sin(x)+\cos(x)$, the GA would converge toward $x\approx\pi/4$, which gives the maximum value of $f(x)\approx1.414$.

Hint:
Plot the function and visualize how the algorithm searches for the peak.

---

16. Experimenting with genetic operators

Answer:
Try different selection methods (roulette, tournament), crossover types (single-point, uniform), and mutation rates. Each affects convergence speed and solution quality.

Hint:
Track performance over generations to see which combination works best.

---

17. Optimizing fuzzy logic controllers with genetic algorithms

Answer:
Use genetic algorithms to evolve membership function parameters and rule weights. This improves controller performance without manual tuning.

Hint:
Think of each fuzzy rule as a gene, how would you breed better rules?

---

18. Training neural networks with genetic algorithms

Answer:
Instead of backpropagation, use genetic algorithms to evolve weights. This is useful when gradients are hard to compute or the loss surface is complex.

Hint:
Compare training speed and accuracy with traditional methods.

---

19. Optimizing neural network weights with genetic algorithms

Answer:
Encode weights as chromosomes. Use fitness based on prediction accuracy. Apply genetic operations to evolve better weight sets.

Hint:
Try this on a small network first, visualize how weights change.

---

20. Modeling decision-making with fuzzy logic

Answer:
In medical diagnosis:

- Inputs: Symptoms (e.g., fever, cough)
- Output: Diagnosis likelihood
- Rules: IF fever is high AND cough is severe THEN flu is likely

Hint:
List fuzzy variables and rules, how would a doctor reason fuzzily?

---

21. (Project) Solving a practical problem with fuzzy logic

Answer:
Choose a real-world problem like energy management or irrigation control. Define fuzzy inputs, outputs, and rules. Implement and test the system.

Here's a complete Python example implementing a **fuzzy logic system for irrigation control** — a practical real-world problem. The system takes two fuzzy inputs:

- **Soil Moisture** (how wet/dry the soil is)
- **Temperature** (hot/cold environment)

and calculates a fuzzy output:

- **Watering Time** (how long to irrigate the plants)

We will:

1. Define the fuzzy sets for inputs and output using triangular membership functions.
2. Define the fuzzy rules.

3. Perform fuzzification, inference (rule evaluation), aggregation, and defuzzification to get a crisp output.
4. Test the system on sample inputs.

```
import numpy as np

import matplotlib.pyplot as plt


def triangular_mf(x, a, b, c):

    """Triangular membership function."""

    return np.maximum(np.minimum((x - a) / (b - a), (c - x) / (c - b)), 0)


class FuzzyVariable:

    def __init__(self, name, sets):

        """

        name: str, variable name

        sets: dict of fuzzy set names to their triangular MF parameters (a,
b, c)

        """

        self.name = name

        self.sets = sets


    def fuzzify(self, x):

        """

        Returns degree of membership for all fuzzy sets given crisp input x.

        """

        memberships = {}

        for set_name, (a, b, c) in self.sets.items():

            memberships[set_name] = triangular_mf(x, a, b, c)

        return memberships


class FuzzyRule:
```

```
    def __init__(self, antecedents, consequent):
        """
        antecedents: list of tuples (variable_name, fuzzy_set_name)
        consequent: tuple (variable_name, fuzzy_set_name)
        """
        self.antecedents = antecedents
        self.consequent = consequent

    def evaluate(self, fuzzy_values):
        """
        fuzzy_values: dict of variable_name -> dict of fuzzy_set_name -> membership_degree
        Returns the firing strength of this rule (min of antecedents memberships).
        """
        degrees = []
        for var, fuzzy_set in self.antecedents:
            degree = fuzzy_values[var][fuzzy_set]
            degrees.append(degree)
        return min(degrees)

class FuzzySystem:
    def __init__(self, input_vars, output_var, rules):
        """
        input_vars: dict of variable_name -> FuzzyVariable
        output_var: FuzzyVariable
        rules: list of FuzzyRule
        """
        self.input_vars = input_vars
        self.output_var = output_var
        self.rules = rules
```

```
    def infer(self, inputs):
        """
        inputs: dict of variable_name -> crisp value
        Returns: crisp output obtained by defuzzification (centroid).
        """
        # 1. Fuzzification
        fuzzy_inputs = {}
        for var_name, crisp_val in inputs.items():
            fuzzy_inputs[var_name] =
self.input_vars[var_name].fuzzify(crisp_val)

        # 2. Rule evaluation and aggregation
        # Aggregate rule outputs by max operator for each output fuzzy set
        output_mf = {key: 0 for key in self.output_var.sets}
        for rule in self.rules:
            strength = rule.evaluate(fuzzy_inputs)
            cons_var, cons_set = rule.consequent
            output_mf[cons_set] = max(output_mf[cons_set], strength)

        # 3. Defuzzification by centroid method
        x_vals = np.linspace(min(a for a,_,_ in
self.output_var.sets.values()),
                             max(c for _,_,c in
self.output_var.sets.values()), 1000)
        aggregated_mf = np.zeros_like(x_vals)
        for set_name, mf_params in self.output_var.sets.items():
            mf = np.array([triangular_mf(x, *mf_params) for x in x_vals])
            aggregated_mf = np.maximum(aggregated_mf,
np.minimum(output_mf[set_name], mf))

        if aggregated_mf.sum() == 0:
```

```
            return 0  # Avoid division by zero
        crisp_output = (x_vals * aggregated_mf).sum() / aggregated_mf.sum()
        return crisp_output

# Define fuzzy variables and sets
soil_moisture = FuzzyVariable('Soil Moisture', {
    'Dry': (0, 0, 50),
    'Moderate': (25, 50, 75),
    'Wet': (50, 100, 100)
})

temperature = FuzzyVariable('Temperature', {
    'Cold': (0, 0, 20),
    'Warm': (15, 25, 35),
    'Hot': (30, 50, 50)
})

watering_time = FuzzyVariable('Watering Time', {
    'Short': (0, 0, 10),
    'Medium': (5, 15, 25),
    'Long': (20, 30, 30)
})

# Define fuzzy rules
rules = [
    # If soil is Dry and temperature is Hot, then watering time is Long
    FuzzyRule([('Soil Moisture', 'Dry'), ('Temperature', 'Hot')], ('Watering
Time', 'Long')),
    # If soil is Dry and temperature is Warm, then watering time is Medium
    FuzzyRule([('Soil Moisture', 'Dry'), ('Temperature', 'Warm')], ('Watering
Time', 'Medium')),
```

```
    # If soil is Dry and temperature is Cold, then watering time is Medium
    FuzzyRule([('Soil Moisture', 'Dry'), ('Temperature', 'Cold')], ('Watering
Time', 'Medium')),
    # If soil is Moderate and temperature is Hot, then watering time is
Medium
    FuzzyRule([('Soil Moisture', 'Moderate'), ('Temperature', 'Hot')],
('Watering Time', 'Medium')),
    # If soil is Moderate and temperature is Warm, then watering time is
Short
    FuzzyRule([('Soil Moisture', 'Moderate'), ('Temperature', 'Warm')],
('Watering Time', 'Short')),
    # If soil is Moderate and temperature is Cold, then watering time is
Short
    FuzzyRule([('Soil Moisture', 'Moderate'), ('Temperature', 'Cold')],
('Watering Time', 'Short')),
    # If soil is Wet, watering time is Short regardless of temperature
    FuzzyRule([('Soil Moisture', 'Wet')], ('Watering Time', 'Short')),
]


# Create fuzzy system
fuzzy_system = FuzzySystem(
    input_vars={'Soil Moisture': soil_moisture, 'Temperature': temperature},
    output_var=watering_time,
    rules=rules
)


# Test system
test_inputs = [
    {'Soil Moisture': 10, 'Temperature': 40},  # Very dry, hot
    {'Soil Moisture': 40, 'Temperature': 20},  # Moderate soil, warm temp
    {'Soil Moisture': 80, 'Temperature': 10},  # Wet soil, cold temp
    {'Soil Moisture': 30, 'Temperature': 35},  # Moderate soil, hot temp
]
```

```
for i, inputs in enumerate(test_inputs):
    output = fuzzy_system.infer(inputs)
    print(f"Test case {i+1}: Inputs={inputs}, Recommended Watering Time = {output:.2f} minutes")


# Optional: Plot membership functions for Watering Time with an example output
x = np.linspace(0, 30, 300)
plt.figure(figsize=(8,4))
for set_name, (a,b,c) in watering_time.sets.items():
    y = np.array([triangular_mf(xx, a, b, c) for xx in x])
    plt.plot(x, y, label=set_name)
plt.title("Watering Time Membership Functions")
plt.xlabel("Minutes")
plt.ylabel("Membership Degree")
plt.legend()
plt.show()
```

- Fuzzification: Converts crisp input values (soil moisture, temperature) into fuzzy memberships over linguistic categories (Dry, Moderate, Hot, etc.).
- Rules: Evaluate firing strengths based on premise fuzzy memberships.
- Aggregation: Max aggregates output fuzzy sets weighted by rule firing strengths.
- Defuzzification: Uses centroid method to compute a crisp watering time.
- Test cases: Show recommended irrigation durations for different input conditions.

Hint:
Look for problems with vague or overlapping conditions.

---

22. (Project) Hybrid system for autonomous robot navigation

Answer:
Combine:

- **Neural networks** for sensor data interpretation
- **Fuzzy logic** for decision-making
- **Genetic algorithms** for optimizing behavior
  This creates a robust, adaptive navigation system.

Here is a conceptual but functional Python code demonstrating a **hybrid system** for autonomous robot navigation combining:

- The **Neural Network** simulates sensor fusion, interpreting raw sensor readings into meaningful indicators (like obstacle proximity likelihood).
- The **Fuzzy Logic controller** takes NN outputs plus other inputs (e.g., current speed) to decide steering and acceleration.
- The **Genetic Algorithm** optimizes parameters of the fuzzy inference system to maximize a fitness function (e.g., successful navigation steps).

```
import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

import random


# ----------------------------

# 1. Neural Network for Sensor Interpretation

# ----------------------------

class SensorNN:

    def __init__(self, input_dim=5, output_dim=2):

        # Simple NN: Input sensor signals, output interpreted values

        self.model = models.Sequential([

            layers.Dense(16, activation='relu', input_shape=(input_dim,)),

            layers.Dense(16, activation='relu'),
```

```
            layers.Dense(output_dim, activation='sigmoid')  # outputs between 0 and 1
        ])
        self.model.compile(optimizer='adam', loss='mse')

    def predict(self, sensor_data):
        # sensor_data: np.array shape (input_dim,)
        return self.model.predict(sensor_data.reshape(1, -1))[0]

    def train(self, X, y, epochs=20):
        self.model.fit(X, y, epochs=epochs, verbose=0)


# ----------------------------
# 2. Fuzzy Logic Controller for Decision Making
# ----------------------------
def triangular_mf(x, a, b, c):
    return np.maximum(np.minimum((x - a) / (b - a + 1e-6), (c - x) / (c - b + 1e-6)), 0)

class FuzzyNavigationController:
    def __init__(self, fuzzy_params):
        # fuzzy_params is a dict tuning membership function centers / widths
        self.params = fuzzy_params

    def fuzzify_proximity(self, x):
        params = self.params['proximity']
        return {
            'Near': triangular_mf(x, params[0], params[1], params[2]),
            'Medium': triangular_mf(x, params[3], params[4], params[5]),
            'Far': triangular_mf(x, params[6], params[7], params[8])
```

```
        }

    def fuzzify_speed(self, x):
        params = self.params['speed']
        return {
            'Slow': triangular_mf(x, params[0], params[1], params[2]),
            'Medium': triangular_mf(x, params[3], params[4], params[5]),
            'Fast': triangular_mf(x, params[6], params[7], params[8])
        }

    def infer(self, proximity_val, speed_val):
        # Fuzzify inputs
        prox = self.fuzzify_proximity(proximity_val)
        speed = self.fuzzify_speed(speed_val)

        # Define fuzzy rules (weights modulated by fuzzy_params)
        rules = []
        # Example rule: IF proximity is Near AND speed is Fast THEN action is
"Turn Sharp"
        turn_sharp_weight = min(prox['Near'], speed['Fast']) *
self.params['weights']['turn_sharp']
        turn_slight_weight = min(prox['Medium'], speed['Medium']) *
self.params['weights']['turn_slight']
        go_straight_weight = min(prox['Far'], speed['Slow']) *
self.params['weights']['go_straight']

        # Aggregate and defuzzify action output to steering angle [-30°, 30°]
        # Here represented simply as weighted average:
        actions = {
            'Turn_Sharp': (-30, turn_sharp_weight),
            'Turn_Slight': (-10, turn_slight_weight),
            'Straight': (0, go_straight_weight),
```

```
        }

        numerator = sum(angle * weight for angle, weight in actions.values())
        denominator = sum(weight for _, weight in actions.values())
        steering_angle = numerator / (denominator + 1e-6)

        # Similarly produce acceleration control [-1 brake, 1 accel]
        accel = self.params['weights']['accel'] * (1 - prox['Near'])  # accelerate more when far

        return steering_angle, accel


# ----------------------------
# 3. Genetic Algorithm for Optimization
# ----------------------------
class GAOptimizer:
    def __init__(self, pop_size=10):
        self.pop_size = pop_size
        # Parameter bounds for fuzzy membership functions (proximity and speed)
        self.param_bounds = {
            'proximity': [(0, 1), (1, 3), (2, 5), (4, 6), (5, 7), (6, 9), (8, 10), (9, 12), (11, 15)],
            'speed': [(0, 1), (1, 3), (2, 5), (4, 6), (5, 7), (6, 9), (8, 10), (9, 12), (11, 15)],
            'weights': {
                'turn_sharp': (0, 1),
                'turn_slight': (0, 1),
                'go_straight': (0, 1),
                'accel': (0, 1)
            }
```

```
        }


    def create_individual(self):
        individual = {}
        individual['proximity'] = [random.uniform(a, b) for a,b in
self.param_bounds['proximity']]
        individual['speed'] = [random.uniform(a, b) for a,b in
self.param_bounds['speed']]
        individual['weights'] = {
            k: random.uniform(v[0], v[1]) for k,v in
self.param_bounds['weights'].items()
        }
        return individual


    def mutate(self, individual, mutation_rate=0.1):
        # Small mutations in parameters
        for key in ['proximity', 'speed']:
            for i in range(len(individual[key])):
                if random.random() < mutation_rate:
                    a,b = self.param_bounds[key][i]
                    individual[key][i] = np.clip(individual[key][i] +
random.uniform(-0.5,0.5), a, b)
        for k in individual['weights']:
            if random.random() < mutation_rate:
                v_min, v_max = self.param_bounds['weights'][k]
                individual['weights'][k] = np.clip(individual['weights'][k] +
random.uniform(-0.1,0.1), v_min, v_max)
        return individual


    def crossover(self, parent1, parent2):
        # Single-point crossover for lists and averaging for weights
        child = {}
```

```
        child['proximity'] =
parent1['proximity'][:len(parent1['proximity'])//2] +
parent2['proximity'][len(parent2['proximity'])//2:]

        child['speed'] = parent1['speed'][:len(parent1['speed'])//2] +
parent2['speed'][len(parent2['speed'])//2:]

        child['weights'] = {k: (parent1['weights'][k] +
parent2['weights'][k]) / 2 for k in parent1['weights']}

        return child


    def fitness(self, individual):

        """

        Simulate navigation controlling for a few steps and evaluate
performance:

        Higher fitness = better navigation (less collision, faster progress).

        """

        fuzzy_controller = FuzzyNavigationController(individual)


        # For simplicity, simulate 10 navigation steps:

        # Random simulated "true" proximity and speed sensor values

        total_score = 0

        for _ in range(10):

            # Random sensor input: proximity [0-15], speed [0-15]

            sensor_data = np.array([

                random.uniform(0,15),

                random.uniform(0,15),

                random.uniform(0,15),

                random.uniform(0,15),

                random.uniform(0,15),

            ])


            # NN interprets sensor data to proximity & speed-like values
(simplified here by average)

            proximity_val = np.mean(sensor_data[:3])
```

```
            speed_val = np.mean(sensor_data[3:])

            steering, accel = fuzzy_controller.infer(proximity_val,
speed_val)

            # Evaluate performance: penalize high steering angles near
obstacles
            if proximity_val < 3 and abs(steering) > 20:
                score = -1  # collision risk
            else:
                score = accel * (15 - proximity_val)  # encourage
acceleration when safe

            total_score += score

        return total_score

    def run(self, generations=10):
        # Initialize population
        population = [self.create_individual() for _ in range(self.pop_size)]

        for gen in range(generations):
            # Evaluate fitness
            fitness_scores = [self.fitness(ind) for ind in population]

            # Sort individuals by fitness descending
            sorted_pairs = sorted(zip(fitness_scores, population), key=lambda
x: x[0], reverse=True)
            fitness_scores, population = zip(*sorted_pairs)

            print(f"Gen {gen+1}, best fitness: {fitness_scores[0]:.3f}")
```

```
            # Select top 50% for crossover
            survivors = population[:self.pop_size//2]

            # Generate offspring
            offspring = []
            while len(offspring) < self.pop_size // 2:
                p1, p2 = random.sample(survivors, 2)
                child = self.crossover(p1, p2)
                child = self.mutate(child, mutation_rate=0.2)
                offspring.append(child)

            population = list(survivors) + offspring

        return population[0]

# ----------------------------
# Main Program to Connect All
# ----------------------------
def main():
    # Instantiate sensor NN and train with dummy data (for demonstration)
    sensor_nn = SensorNN()
    # Dummy training: input random sensor readings, output proximity+speed-
like labels
    X_train = np.random.uniform(0, 15, (100,5))
    y_train = np.hstack([
        np.mean(X_train[:, :3], axis=1, keepdims=True),  # proximity approx
        np.mean(X_train[:, 3:], axis=1, keepdims=True)   # speed approx
    ])
    sensor_nn.train(X_train, y_train, epochs=5)
    print("Sensor NN trained (dummy)")
```

```
    # Run genetic algorithm to optimize fuzzy controller params
    optimizer = GAOptimizer(pop_size=10)
    best_params = optimizer.run(generations=10)
    print("Optimized fuzzy parameters found:")

    # Display best fuzzy params (some truncated for neatness)
    for key in best_params:
        print(f"{key}: {best_params[key] if isinstance(best_params[key], dict) else best_params[key][:5]}...")

    # Create fuzzy controller with best params
    fuzzy_controller = FuzzyNavigationController(best_params)

    # Simulate one input using SensorNN + Fuzzy controller
    sample_sensor_data = np.array([5, 2, 1, 10, 12])  # hypothetical sensor readings
    interpreted = sensor_nn.predict(sample_sensor_data)
    proximity_val, speed_val = interpreted

    steering, accel = fuzzy_controller.infer(proximity_val, speed_val)
    print(f"Final Decision: Steering Angle = {steering:.2f}°, Acceleration = {accel:.2f}")


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

- **SensorNN**: A small neural network learns to interpret raw sensor vector inputs into proximity and speed estimates (simplified here as training on synthetic data).
- **FuzzyNavigationController**: Uses triangular membership functions with tunable parameters, applies fuzzy inference to decide steering and acceleration.

- **GAOptimizer**: Evolves the fuzzy system parameters (membership function points and rule weights) over generations to maximize a navigation fitness function that simulates sensor readings and penalizes unsafe navigation.
- At the end, optimized fuzzy controller parameters are applied with sensor NN interpretations to decide navigation commands.

For a real robot, sensor data, training data, and fitness evaluations would be based on real or simulated environments.

Hint:
Break the system into modules, how do they interact?

---

23. (Project) Adaptive system using genetic algorithms

Answer:
Design a system where genetic algorithms evolve neural or fuzzy components over time. Useful in dynamic environments like stock trading or robotics.

Hint:
Think of evolution as continuous learning, how does the system adapt?

---

24. (Project) Genetic algorithm for abstract art or music

Answer:
Encode artistic elements (colors, shapes, notes) as genes. Use fitness based on aesthetic rules or user feedback. Evolve generations of creative outputs.

Below is a Python example of a **Genetic Algorithm (GA)** framework to evolve simple abstract art images. The genetic encoding represents a set of colored circles (positions, radii, colors) as genes.

- Each individual is a "painting" composed of multiple circles.
- The fitness function is a simple aesthetic heuristic: diversity of colors + coverage of canvas.
- You can replace or augment fitness with user feedback by integrating an interface or manual scoring.
- The code evolves a population over generations and displays the best painting.

```
import numpy as np
import matplotlib.pyplot as plt
import random

# Parameters
CANVAS_SIZE = 100
NUM_CIRCLES = 10
POPULATION_SIZE = 20
GENERATIONS = 30
MUTATION_RATE = 0.1

# Gene encoding:
# Each circle: (x_pos, y_pos, radius, r, g, b)
# x_pos, y_pos in [0, CANVAS_SIZE]
# radius in [5, 30]
# r,g,b in [0,1]

def create_individual():
    """Create a random individual (list of circles)."""
    individual = []
    for _ in range(NUM_CIRCLES):
        circle = {
            'x': random.uniform(0, CANVAS_SIZE),
            'y': random.uniform(0, CANVAS_SIZE),
            'r': random.uniform(5, 30),
            'color': (random.random(), random.random(), random.random())
        }
        individual.append(circle)
    return individual

def draw_individual(individual, ax=None):
```

```
    """Draw the painting represented by an individual."""
    if ax is None:
        fig, ax = plt.subplots(figsize=(5,5))
    ax.clear()
    ax.set_xlim(0, CANVAS_SIZE)
    ax.set_ylim(0, CANVAS_SIZE)
    ax.set_aspect('equal')
    ax.axis('off')
    for circle in individual:
        c = plt.Circle((circle['x'], circle['y']), circle['r'],
color=circle['color'], alpha=0.6)
        ax.add_patch(c)
    plt.tight_layout()

def fitness(individual):
    """
    Simple aesthetic fitness:
    - Color diversity: higher is better
    - Canvas coverage (sum of circle areas normalized): higher is better
    """
    colors = np.array([circle['color'] for circle in individual])
    # Color diversity: mean pairwise Euclidean distance in RGB space
    dist_sum = 0
    count = 0
    for i in range(len(colors)):
        for j in range(i+1, len(colors)):
            dist_sum += np.linalg.norm(colors[i] - colors[j])
            count += 1
    color_diversity = dist_sum / count if count > 0 else 0

    # Coverage: sum of circle areas normalized by canvas area
```

```
    canvas_area = CANVAS_SIZE * CANVAS_SIZE
    coverage = sum(np.pi * c['r']**2 for c in individual) / canvas_area
    coverage = min(coverage, 1)  # cap at 1

    # Combine with weights
    return 0.7 * color_diversity + 0.3 * coverage


def crossover(parent1, parent2):
    """Single point crossover of circles."""
    point = random.randint(1, NUM_CIRCLES - 1)
    child = parent1[:point] + parent2[point:]
    return child


def mutate(individual):
    """Randomly mutate circles' parameters."""
    for circle in individual:
        if random.random() < MUTATION_RATE:
            circle['x'] = np.clip(circle['x'] + random.uniform(-5, 5), 0,
CANVAS_SIZE)
        if random.random() < MUTATION_RATE:
            circle['y'] = np.clip(circle['y'] + random.uniform(-5, 5), 0,
CANVAS_SIZE)
        if random.random() < MUTATION_RATE:
            circle['r'] = np.clip(circle['r'] + random.uniform(-3, 3), 5, 30)
        if random.random() < MUTATION_RATE:
            r, g, b = circle['color']
            r = np.clip(r + random.uniform(-0.2, 0.2), 0, 1)
            g = np.clip(g + random.uniform(-0.2, 0.2), 0, 1)
            b = np.clip(b + random.uniform(-0.2, 0.2), 0, 1)
            circle['color'] = (r, g, b)
    return individual
```

```
def select(population, fitnesses, num):
    """Select top num individuals by fitness."""
    sorted_pop = [ind for _, ind in sorted(zip(fitnesses, population),
key=lambda x: x[0], reverse=True)]
    return sorted_pop[:num]


def run_ga():
    # Initialize population
    population = [create_individual() for _ in range(POPULATION_SIZE)]

    for gen in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
        best_fit = max(fitnesses)
        print(f"Generation {gen+1}: Best fitness = {best_fit:.3f}")

        # Selection
        selected = select(population, fitnesses, POPULATION_SIZE // 2)

        # Create next generation
        next_generation = selected.copy()
        while len(next_generation) < POPULATION_SIZE:
            p1, p2 = random.sample(selected, 2)
            child = crossover(p1, p2)
            child = mutate(child)
            next_generation.append(child)

        population = next_generation

    # Draw best individual
    fitnesses = [fitness(ind) for ind in population]
```

```
    best_individual = population[np.argmax(fitnesses)]
    fig, ax = plt.subplots(figsize=(6,6))
    draw_individual(best_individual, ax)
    plt.title("Best Abstract Art after GA Evolution")
    plt.show()


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

- Each individual is a set of circles with position, size, and color.
- Fitness encourages colorful diversity and good canvas coverage.
- Genetic operations: crossover combines parts of two parents; mutation randomly tweaks genes.
- Over generations, the population evolves toward more aesthetically diverse and well-covered images.
- Finally, the best individual is displayed.

Extending to Music:

- Encode notes as genes (pitch, duration, velocity).
- Fitness could depend on music theory rules or user rating.
- Generate MIDI or audio from best individuals.

Hint:
Explore how randomness and selection can lead to creativity.

---

25. (Project) Fuzzy logic expert system for customer service

Answer:
is a Python implementation of a simple **Fuzzy Logic Expert System** for customer service with the specified inputs and outputs:

- **Inputs:** Customer Tone, Issue Severity

- **Outputs:** Response Urgency, Escalation Level
- **Rules:** including the example rule:
  IF tone is angry AND issue is severe THEN escalate immediately

The system uses triangular membership functions and follows the fuzzify → infer → defuzzify approach.

```
import numpy as np


def triangular_mf(x, a, b, c):
    """Triangular membership function."""
    return np.maximum(np.minimum((x - a) / (b - a + 1e-6), (c - x) / (c - b + 1e-6)), 0)


class FuzzyVariable:
    def __init__(self, name, sets):
        """
        name: str
        sets: dict mapping fuzzy set names to (a,b,c)
        """
        self.name = name
        self.sets = sets

    def fuzzify(self, x):
        memberships = {}
        for set_name, (a,b,c) in self.sets.items():
            memberships[set_name] = triangular_mf(x, a, b, c)
        return memberships


class FuzzyRule:
    def __init__(self, antecedents, consequents):
        """
        antecedents: list of tuples (variable_name, fuzzy_set_name)
```

```
        consequents: list of tuples (variable_name, fuzzy_set_name)
        """
        self.antecedents = antecedents
        self.consequents = consequents

    def evaluate(self, fuzzified_inputs):
        degrees = []
        for var, fuzzy_set in self.antecedents:
            degree = fuzzified_inputs[var].get(fuzzy_set, 0)
            degrees.append(degree)
        return min(degrees)

class FuzzyExpertSystem:
    def __init__(self, input_vars, output_vars, rules):
        self.input_vars = input_vars
        self.output_vars = output_vars
        self.rules = rules

    def infer(self, inputs):
        fuzzified_inputs = {}
        for var_name, crisp_val in inputs.items():
            fuzzified_inputs[var_name] =
self.input_vars[var_name].fuzzify(crisp_val)

        # Initialize output fuzzy sets memberships zero
        output_mfs = {var: {term:0 for term in self.output_vars[var].sets}
for var in self.output_vars}

        # Evaluate rules and aggregate outputs
        for rule in self.rules:
            firing_strength = rule.evaluate(fuzzified_inputs)
```

```
            for out_var, out_set in rule.consequents:
                output_mfs[out_var][out_set] =
max(output_mfs[out_var][out_set], firing_strength)

        # Defuzzify output variables using centroid method
        outputs = {}
        for var_name, var in self.output_vars.items():
            x_vals = np.linspace(min(a for (a,_,_) in var.sets.values()),
                                 max(c for (_,_,c) in var.sets.values()),
1000)
            agg_mf = np.zeros_like(x_vals)
            for set_name, (a,b,c) in var.sets.items():
                mf = np.array([triangular_mf(x, a,b,c) for x in x_vals])
                agg_mf = np.maximum(agg_mf,
np.minimum(output_mfs[var_name][set_name], mf))
            if agg_mf.sum() == 0:
                outputs[var_name] = 0  # Avoid division by zero
            else:
                outputs[var_name] = (x_vals * agg_mf).sum() / agg_mf.sum()
        return outputs

# Define fuzzy variables
customer_tone = FuzzyVariable('Customer Tone', {
    'Calm': (0, 0, 4),
    'Annoyed': (2, 5, 7),
    'Angry': (6, 10, 10)
})

issue_severity = FuzzyVariable('Issue Severity', {
    'Low': (0, 0, 4),
    'Medium': (2, 5, 7),
    'Severe': (6, 10, 10)
```

```
})

response_urgency = FuzzyVariable('Response Urgency', {
    'Low': (0, 0, 4),
    'Medium': (2, 5, 7),
    'High': (6, 10, 10)
})

escalation_level = FuzzyVariable('Escalation Level', {
    'None': (0, 0, 3),
    'Normal': (2, 5, 7),
    'Immediate': (6, 10, 10)
})

# Define fuzzy rules
rules = [
    # IF tone is angry and issue is severe THEN escalate immediately and high urgency
    FuzzyRule(
        [('Customer Tone', 'Angry'), ('Issue Severity', 'Severe')],
        [('Response Urgency', 'High'), ('Escalation Level', 'Immediate')]
    ),
    # IF tone is annoyed and issue is medium THEN medium urgency and normal escalation
    FuzzyRule(
        [('Customer Tone', 'Annoyed'), ('Issue Severity', 'Medium')],
        [('Response Urgency', 'Medium'), ('Escalation Level', 'Normal')]
    ),
    # IF tone is calm and issue is low THEN low urgency and no escalation
    FuzzyRule(
        [('Customer Tone', 'Calm'), ('Issue Severity', 'Low')],
```

```
        [('Response Urgency', 'Low'), ('Escalation Level', 'None')]
    ),
    # Add more rules as appropriate
]

# Instantiate system
input_vars = {
    'Customer Tone': customer_tone,
    'Issue Severity': issue_severity
}
output_vars = {
    'Response Urgency': response_urgency,
    'Escalation Level': escalation_level
}

fuzzy_system = FuzzyExpertSystem(input_vars, output_vars, rules)

# Test inputs
test_cases = [
    {'Customer Tone': 9, 'Issue Severity': 9},  # Angry and Severe
    {'Customer Tone': 4, 'Issue Severity': 5},  # Annoyed and Medium
    {'Customer Tone': 1, 'Issue Severity': 2},  # Calm and Low
    {'Customer Tone': 7, 'Issue Severity': 3},  # Angry tone but Low severity
]

print("Fuzzy logic system outputs:\n")
for i, case in enumerate(test_cases, 1):
    outputs = fuzzy_system.infer(case)
    print(f"Test case {i}, inputs: {case}")
    print(f"  Response Urgency: {outputs['Response Urgency']:.2f} (0=Low, 10=High)")
```

```
    print(f"  Escalation Level: {outputs['Escalation Level']:.2f}
(0=None,10=Immediate)\n")
```

- Inputs are on scale 0–10 (where 0=very calm/low severity, 10=very angry/severe).
- Outputs are numeric in 0–10 scale representing urgency and escalation levels.
- The example rule: when tone is *Angry* and issue is *Severe*, escalation is immediate and urgency is high.

Run the code above to see crisp outputs for sample customer scenarios.

Hint:
Map fuzzy inputs to actions, how would a human agent respond?

---

26. (Project) Train an RNN for sentiment classification

Answer:
Use an RNN (e.g., LSTM) to classify movie reviews as positive or negative. Preprocess text, tokenize, and train on labeled data.

Here is a complete Python example using TensorFlow/Keras to build and train an LSTM-based RNN for binary sentiment classification on movie reviews. We'll use the **IMDB dataset** included in Keras, which contains labeled movie reviews (positive/negative). The code includes:

- Loading and preprocessing text data
- Tokenizing and padding sequences
- Building an LSTM model
- Training and evaluation

```
import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

from tensorflow.keras.callbacks import EarlyStopping
```

```
# Parameters

VOCAB_SIZE = 10000  # Number of words to consider as features

MAX_LEN = 200       # Cut texts after this number of words (max sequence
length)

EMBEDDING_DIM = 128 # Embedding output dimension

BATCH_SIZE = 64

EPOCHS = 5


# 1. Load IMDB dataset (already tokenized into word indices)

print("Loading dataset...")

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=VOCAB_SIZE)


# 2. Pad sequences to the same length for batch processing

print("Padding sequences to the same length...")

x_train = pad_sequences(x_train, maxlen=MAX_LEN, padding='post',
truncating='post')

x_test = pad_sequences(x_test, maxlen=MAX_LEN, padding='post',
truncating='post')


# 3. Build the LSTM model

print("Building LSTM model...")

model = Sequential([

    Embedding(VOCAB_SIZE, EMBEDDING_DIM, input_length=MAX_LEN),

    LSTM(64, return_sequences=False),

    Dropout(0.5),

    Dense(1, activation='sigmoid')

])


model.compile(

    loss='binary_crossentropy',

    optimizer='adam',
```

```
    metrics=['accuracy']
)

model.summary()

# 4. Train the model with early stopping
print("Training the model...")
early_stop = EarlyStopping(monitor='val_loss', patience=2)

history = model.fit(
    x_train, y_train,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_split=0.2,
    callbacks=[early_stop]
)

# 5. Evaluate the model on test data
print("Evaluating the model...")
loss, accuracy = model.evaluate(x_test, y_test, batch_size=BATCH_SIZE)
print(f"Test Loss: {loss:.4f}, Test Accuracy: {accuracy:.4f}")

# Optional: Predict sentiment on new review examples
word_index = imdb.get_word_index()

def encode_review(text):
    """
    Encodes a text review into a sequence of integers based on IMDB word
index.
    """
    tokens = text.lower().split()
```

```
    encoded = []
    for word in tokens:
        index = word_index.get(word, 2)  # Use 2 for unknown words (oov)
        if index < VOCAB_SIZE:
            encoded.append(index)
    return pad_sequences([encoded], maxlen=MAX_LEN, padding='post',
truncating='post')


sample_reviews = [
    "This movie was fantastic! I really loved it and the acting was great.",
    "Terrible movie. It was boring and I did not enjoy it at all."
]


for review in sample_reviews:
    encoded_review = encode_review(review)
    pred = model.predict(encoded_review)[0][0]
    sentiment = "Positive" if pred >= 0.5 else "Negative"
    print(f"Review: {review}\nPredicted sentiment: {sentiment}
(score={pred:.3f})\n")
```

- `imdb.load_data()`: Loads pre-tokenized IMDB reviews mapped to integer word indices.
- `pad_sequences()`: Pads or truncates sequences to fixed length so they can be batched.
- `Embedding` layer: Learns word embeddings for each token.
- `LSTM` layer: Processes the sequence data capturing temporal dependencies.
- `Dense` layer with sigmoid: Outputs probability of positive sentiment.
- Trained with binary crossentropy loss and optimized by Adam.
- Early stopping prevents overfitting.
- After training, evaluation and prediction on example reviews are shown.

Hint:

Try visualizing word sequences, how does context affect sentiment?