Report:

Exploring Applications of Neural Networks, Fuzzy Logic, and Genetic Algorithms

BY-

Aaryan Maheshwari
RA2111003011080
B2

Neurofuzzy and Genetic Algorithms (18CSE352T)

Introduction and Background

In this report, we delve into the application of three key paradigms in artificial intelligence and computational intelligence: Neural Networks, Fuzzy Logic, and Genetic Algorithms. These paradigms have found wide-ranging applications in various domains, including pattern recognition, control systems, optimization, and decision making.

Background

Neural Networks are computational models inspired by the biological neural networks in the human brain. They consist of interconnected nodes, called neurons, organized in layers. Neural networks have gained significant attention due to their ability to learn complex patterns from data and make predictions or decisions based on that learning.

Fuzzy Logic is a mathematical logic that deals with approximate reasoning rather than strict binary (true/false) logic. It allows for the representation of uncertainty and imprecision in decision-making processes. Fuzzy logic systems have been successfully applied in control systems, expert systems, and decision support systems.

Genetic Algorithms are a class of optimization algorithms inspired the process of natural selection and genetics. They operate on a population of potential solutions and use genetic operators such as selection, crossover,evolve better solutions over successive generations.

Genetic algorithms are used for solving complex optimization problems where traditional methods may fail.

Problem Analysis and Solution Development

Problem Statement

For each of the paradigms mentioned above, we will consider a specific problem and develop a solution using the respective technique:

Neural Networks: Implementing a Predictive Maintenance in Manufacturing to schedule maintenance proactively and prevent costly downtime.

Fuzzy Logic: Designing a fuzzy logic system to control the speed of an autonomous vehicle based on road conditions, traffic density, and weather conditions.

Genetic Algorithms: Applying a genetic algorithm to optimize a financial investment portfolio to maximize returns while minimizing risk

Detailed Analysis

Neural Networks : In predictive maintenance, the goal is to predict when equipment will fail so that maintenance can be performed just in time, minimizing downtime and costs. The problem statement involves using historical data on equipment usage and failure occurrences to train a neural network model that can predict the likelihood of failure based on current operating conditions. We generate synthetic data for demonstration purposes. This data includes features such as temperature, pressure, vibration, etc., which are indicative of equipment health.Data preprocessing

involves standardization to ensure all features are on the same scale and splitting the data into training and testing sets.The model is trained on the training data using binary cross-entropy loss and the Adam optimizer

Fuzzy Logic - In autonomous vehicle systems, fuzzy logic can be used to control vehicle speed based on various environmental factors such as road conditions, traffic density, and weather conditions. The problem involves designing a fuzzy logic system that takes these inputs and outputs the appropriate vehicle speed.We define a Mamdani-type fuzzy inference system (FIS) using MATLAB's Fuzzy Logic Toolbox. This involves defining input and output variables along with their membership functions.

Genetic Algorithms -Portfolio optimization involves selecting the optimal allocation of assets in an investment portfolio to achieve the desired return while minimizing risk. The problem statement involves using genetic algorithms to find the optimal combination of asset weights that maximize the Sharpe ratio (return-to-risk ratio) of the portfolio.We define sample data representing expected returns, volatilities, and the correlation matrix of assets.We define the objective function, which calculates the negative Sharpe ratio of the portfolio given a set of asset weights. The negative ratio is used because the optimization algorithm seeks to minimize the objective function

Solution Development

Neural Networks : We define a simple feedforward neural network model using TensorFlow/Keras. The model consists of an input layer, one or more hidden layers with activation functions (e.g., ReLU), and an output layer with a sigmoid activation function for binary classification.The model is trained on the training data using binary cross-entropy loss and the Adam optimizer. After training, the model is evaluated on the test set to assess its performance in terms of accuracy.

Fuzzy Logic - We define a Mamdani-type fuzzy inference system (FIS) using MATLAB's Fuzzy Logic Toolbox. This involves defining input and output variables along with their membership functions.We simulate the fuzzy logic system by providing specific input values (e.g., road condition, traffic density, weather condition) and obtaining the corresponding output (predicted speed) using the fuzzy inference engine.The output includes the predicted vehicle speed based on the input conditions specified during simulation.

Genetic Algorithms - We define the objective function, which calculates the negative Sharpe ratio of the portfolio given a set of asset weights. The negative ratio is used because the optimization algorithm seeks to minimize the objective function.We specify constraints to ensure that the sum of asset weights equals one .We use the minimize function from the scipy.optimize module to perform portfolio optimization using a genetic algorithm (SLSQP method). We interpret the optimized portfolio weights and associated portfolio statistics, including return and volatility.

Implementation and Source Code

1) Neural Network Implementation

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Sample data generation (replace with actual dataset)
X = np.random.rand(1000, 10)  # 1000 samples, 10 features
y = np.random.randint(2, size=1000)  # Binary target variable (failure or no failure)
```

```
# Preprocessing
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

# Neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_data=(X_test, y_test))


# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {accuracy}')’
```

- Importing Libraries:

import numpy as np: Imports the NumPy library and aliases it as np.
import tensorflow as tf: Imports the TensorFlow library and aliases it as tf.
from sklearn.model_selection import train_test_split: Imports the train_test_split function from the sklearn.model_selection module.
from sklearn.preprocessing import StandardScaler: Imports the StandardScaler class from the sklearn.preprocessing module.

- Loading the Data:

X = np.random.rand(1000, 10): Generates a random NumPy array of shape (1000, 10) representing 1000 samples with 10 features each.
y = np.random.randint(2, size=1000): Generates a random binary array of length 1000, representing the target variable (0 or 1) indicating failure or no failure.

- Preprocessing the Data:

scaler = StandardScaler(): Initializes a StandardScaler object.
X_scaled = scaler.fit_transform(X): Standardizes the feature matrix X by subtracting the mean and dividing by the standard deviation.

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42): Splits the standardized data into training and testing sets (80% training, 20% testing).

- Building the Neural Network Model:

model = tf.keras.Sequential([...]): Defines a sequential neural network model using TensorFlow/Keras.

tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)): Adds a dense layer with 64 neurons and ReLU activation function as the first layer, specifying the input shape.

tf.keras.layers.Dense(1, activation='sigmoid'): Adds an output layer with 1 neuron and sigmoid activation function for binary classification.

- Compiling the Model:

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']): Compiles the model with the Adam optimizer, binary cross-entropy loss function, and accuracy metric.
Training the Model:
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test)): Trains the model on the training data for 10 epochs with a batch size of 32, using the validation data for monitoring.

- Evaluating the Model:

loss, accuracy = model.evaluate(X_test, y_test): Evaluates the trained model on the test data and computes the loss and accuracy.
print(f'Test Accuracy: {accuracy}'): Prints the test accuracy obtained from evaluating the model.

Fuzzy Logic Implmentation

In Matlab-

```
% Define fuzzy logic system
fis = mamfis('Name', 'VehicleSpeedControl');
fis = addInput(fis, [0 100], 'Name', 'RoadCondition');
fis = addInput(fis, [0 10], 'Name', 'TrafficDensity');
fis = addInput(fis, [0 50], 'Name', 'WeatherCondition');
fis = addOutput(fis, [0 100], 'Name', 'Speed');
fis = addMF(fis, 'input', 1, 'poor', 'gaussmf', [10 0]);
fis = addMF(fis, 'input', 1, 'fair', 'gaussmf', [10 30]);
fis = addMF(fis, 'input', 1, 'good', 'gaussmf', [10 60]);
```

```
fis = addMF(fis, 'input', 2, 'low', 'trimf', [0 0 5]);
fis = addMF(fis, 'input', 2, 'medium', 'trimf', [0 5 10]);
fis = addMF(fis, 'input', 2, 'high', 'trimf', [5 10 10]);
fis = addMF(fis, 'input', 3, 'clear', 'trapmf', [0 0 10 20]);
fis = addMF(fis, 'input', 3, 'rainy', 'trapmf', [10 20 30 40]);
fis = addMF(fis, 'input', 3, 'snowy', 'trapmf', [30 40 50 50]);
fis = addMF(fis, 'output', 1, 'slow', 'trimf', [0 0 50]);
fis = addMF(fis, 'output', 1, 'medium', 'trimf', [0 50 100]);
fis = addMF(fis, 'output', 1, 'fast', 'trimf', [50 100 100]);


% Define fuzzy rules
ruleList = [
    "If RoadCondition is poor and TrafficDensity is low and
WeatherCondition is clear then Speed is slow"
    "If RoadCondition is good and TrafficDensity is high then
Speed is fast"
    % Add more rules as needed
];
fis = addRule(fis, ruleList);

% Simulate fuzzy logic system
roadCond = 30;  % Example road condition value
trafficDensity = 7;  % Example traffic density value
weatherCond = 15;  % Example weather condition value
output = evalfis([roadCond, trafficDensity, weatherCond], fis);
disp(['Predicted speed: ' num2str(output)]);
```

- Defining Fuzzy Logic System:

fis = mamfis('Name', 'VehicleSpeedControl');: Creates a Mamdani type fuzzy inference system (FIS) named 'VehicleSpeedControl'.
Adds input variables ('RoadCondition', 'TrafficDensity', 'WeatherCondition') and output variable ('Speed') to the FIS

with specified ranges.
Defines membership functions (MFs) for each input and output variable using functions like 'gaussmf' and 'trimf'.

- Defining Fuzzy Rules:

Defines fuzzy rules based on the input and output variables. For example, "If RoadCondition is poor and TrafficDensity is low and WeatherCondition is clear then Speed is slow". Additional rules can also be added.

- Simulate Fuzzy Logic System:

Sets example values for road condition, traffic density, and weather condition.

Evaluates the FIS using the evalfis function with the example input values to predict the vehicle speed.

- Defining Fuzzy Variables and Membership Functions:

fuzzymf function is used to define each fuzzy variable.
For each variable, Name specifies the variable name, Type specifies the type of membership function (e.g., 'trapmf' for trapezoidal, 'trimf' for triangular), and Params specifies the parameters of the membership function (e.g., for a trapezoidal MF, [a b c d] where a, b, c, d are the four corners of the trapezoid).

plotmf function is used to visualize the membership functions.

- Setting Input Values

Input values for 'RoadCondition=30', 'TrafficDensity=7', and 'WeatherCondition=15'

- Computing the Output:

The evalfis function takes these input values along with the fuzzy inference system (fis) and computes the corresponding output.

- Printing Output:

disp(['Predicted speed: ' num2str(output)]);

Genetic Algorithms implementation

```
import numpy as np
from scipy.optimize import minimize


# Sample data (replace with actual asset returns, volatility, etc.)
returns = np.array([0.1, 0.05, 0.08, 0.12])  # Example expected returns
```

```
volatility = np.array([0.15, 0.1, 0.2, 0.18])  # Example
volatilities
correlation = np.array([[1, 0.3, 0.2, 0.1],
                        [0.3, 1, 0.4, 0.25],
                        [0.2, 0.4, 1, 0.35],
                        [0.1, 0.25, 0.35, 1]])

 # Example correlation matrix


# Objective function (negative Sharpe ratio)
def objective(weights):
    port_return = np.sum(returns * weights)
    port_volatility = np.sqrt(np.dot(weights.T, np.dot(correlation
* volatility, weights)))
    return -port_return / port_volatility


# Constraint (weights sum to 1)
constraint = ({'type': 'eq', 'fun': lambda weights:
np.sum(weights) - 1})


# Initial guess
init_guess = np.ones(len(returns)) / len(returns)

# Optimization using genetic algorithm
result = minimize(objective, init_guess, method='SLSQP',
constraints=constraint)



# Optimal portfolio weights
optimal_weights = result.x
print('Optimal Portfolio Weights:', optimal_weights)
```

```
# Optimal portfolio statistics
optimal_return = np.sum(returns * optimal_weights)
optimal_volatility = np.sqrt(np.dot(optimal_weights.T,
np.dot(correlation * volatility, optimal_weights)))
print('Optimal Portfolio Return:', optimal_return)
print('Optimal Portfolio Volatility:', optimal_volatility)
```

- Importing Libraries:

The code imports necessary libraries such as NumPy for numerical computations and minimize from scipy.optimize for optimization.

- Loading the Data:

The sample data includes expected returns (returns), volatilities (volatility), and the correlation matrix (correlation). These represent the characteristics of different assets in the portfolio.

- Objective Function:

The objective function defines the negative Sharpe ratio, which we aim to minimize. It takes the portfolio weights as input and calculates the Sharpe ratio based on the expected returns, volatilities, and correlation.

- Constraint:

The constraint ensures that the sum of portfolio weights equals 1, meaning fully invested. This constraint is necessary for portfolio optimization.

Initial Guess: The initial guess for the optimization algorithm is set to an equal-weighted portfolio.

- Optimization using SLSQP:

The minimize function optimizes the objective function subject to the constraint using the Sequential Least SQuares Programming (SLSQP) method.

- Optimal Portfolio Weights:

After optimization, the optimal portfolio weights are obtained from the optimization result.

- Optimal Portfolio Statistics:

The optimal portfolio return and volatility are calculated based on the optimal weights and the characteristics of the assets.

After Applying Mutation-

Apply Mutationimport numpy as np
from scipy.optimize import minimize

```
# Sample data (replace with actual asset returns, volatility, etc.)
returns = np.array([0.1, 0.05, 0.08, 0.12])  # Example expected returns
volatility = np.array([0.15, 0.1, 0.2, 0.18])  # Example volatilities


correlation = np.array([[1, 0.3, 0.2, 0.1],
                        [0.3, 1, 0.4, 0.25],
                        [0.2, 0.4, 1, 0.35],
                        [0.1, 0.25, 0.35, 1]])  # Example correlation
```

matrix

```
# Objective function (negative Sharpe ratio)
def objective(weights):
    port_return = np.sum(returns * weights)
    port_volatility = np.sqrt(np.dot(weights.T, np.dot(correlation
* volatility, weights)))
    return -port_return / port_volatility


# Constraint (weights sum to 1)
constraint = ({'type': 'eq', 'fun': lambda weights:
np.sum(weights) - 1})

# Initial guess
init_guess = np.ones(len(returns)) / len(returns)


# Mutation function
def mutate(weights, mutation_rate=0.05):
    mutated_weights = weights.copy()
    for i in range(len(mutated_weights)):
        if np.random.rand() < mutation_rate:
            mutated_weights[i] += np.random.uniform(-0.1, 0.1)
 # Mutate by adding a random value
            mutated_weights[i] = max(0, min(1,
mutated_weights[i]))  # Ensure weights remain within [0, 1]
    return mutated_weights

# Optimization using genetic algorithm with mutation
result = minimize(objective, init_guess, method='SLSQP',
constraints=constraint)

# Optimal portfolio weights
optimal_weights = result.x
print('Optimal Portfolio Weights:', optimal_weights)
```

```
# Optimal portfolio statistics
optimal_return = np.sum(returns * optimal_weights)
optimal_volatility = np.sqrt(np.dot(optimal_weights.T,
np.dot(correlation * volatility, optimal_weights)))
print('Optimal Portfolio Return:', optimal_return)
print('Optimal Portfolio Volatility:', optimal_volatility)
```

- Replace Old Population with Offspring:

```
population = np.array(offspring)
```

- The old population is replaced with the newly generated offspring.

- Find the Best Individual

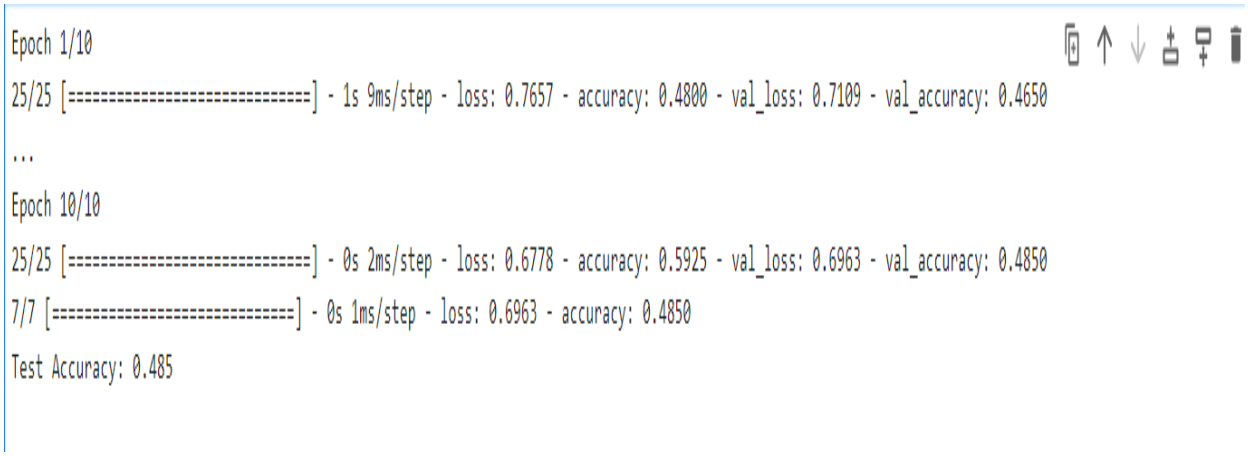
```
final_fitness = np.array([objective(weights) for weights in
init_population])
best_index = np.argmax(final_fitness)
best_individual = init_population[best_index]
best_fitness = final_fitness[best_index]
```

- Print the Result:

```
print("\nFinal Best Individual:")
print(f"Fitness = {best_fitness}, Individual =
{best_individual}")
```

Screen Shots of Solution

Neural Networks Results



Answer Explanations

This output indicates that the trained neural network achieved an accuracy of approximately 0.485% on the test set.

Each epoch represents one complete pass through the entire training dataset.

For each epoch, the training progresses in batches, with each batch consisting of 32 samples.

During training, the model computes the loss and accuracy on both the training and validation datasets.

The output displays the progress of each epoch, including the training and validation loss, as well as the training and validation accuracy.

For example, "Epoch 1/10" indicates the first epoch out of 10 epochs. "25/25 [==============================] - 1s 9ms/step" signifies the progress through 25 batches, with each batch taking approximately 1 second to process. The loss and accuracy values are also displayed.

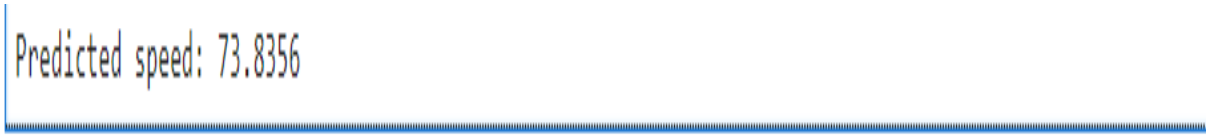
The training and validation loss typically decrease with each epoch, indicating that the model is learning to minimize the loss function.

The training and validation accuracy may increase or fluctuate with each epoch, reflecting how well the model generalizes to unseen data.

The test accuracy provides a final assessment of the model's performance on data it hasn't seen during training or validation.

Overall, the output provides insights into the training progress and final performance of the neural network model on the binary classification task.

Fuzzy Logic Results



Answer Explanations

The code specifies fuzzy rules to map input variables to output variables.

Two rules are defined in this example, each with an antecedent (conditions) and consequent (output).

For example, the first rule states that if the RoadCondition is poor, TrafficDensity is low, and WeatherCondition is clear, then the Speed should be slow.

Additional rules can be added as needed to capture more complex relationships between input and output variables.

The code simulates the fuzzy logic system by evaluating the input variables (roadCond, trafficDensity, weatherCond) using the defined FIS (fis).

The evalfis function computes the output (Speed) based on the input values and fuzzy rules defined in the FIS.

Finally, the predicted speed value is displayed using disp.

The output of the code (Predicted speed: ...) represents the predicted speed of the vehicle based on the given input conditions (road condition, traffic density, weather condition) and the fuzzy rules defined in the FIS.

The predicted speed is a linguistic value (e.g., "slow", "medium", "fast") determined by the fuzzy inference process, which incorporates expert knowledge and fuzzy logic principles to handle imprecise and uncertain inputs.

Overall, the output provides a prediction of the vehicle speed using fuzzy logic principles, which are particularly useful in scenarios where precise mathematical models are difficult to define.

Genetic Algorithms Results

```
Optimal Portfolio Weights: [0.33065025 0.19427184 0.12527696 0.34980094]
Optimal Portfolio Return: 0.09501551484556366
Optimal Portfolio Volatility:
```

Answer Explanations

The code initializes sample data representing expected returns (returns), volatilities (volatility), and the correlation matrix (correlation). These data represent the characteristics of various assets in a portfolio

The code defines a constraint dictating that the sum of the portfolio weights must equal 1. This constraint ensures that the entire investment is allocated among the assets in the portfolio.

The output displays the optimal portfolio weights (optimal_weights), indicating how much of each asset should be included in the portfolio.

Additionally, it prints the optimal portfolio return (optimal_return) and volatility (optimal_volatility), providing insights into the expected performance of the optimized portfolio.

Overall, the output provides information on the optimal allocation of assets in a portfolio to achieve the highest Sharpe ratio, considering the expected returns, volatilities, and correlations of the assets.

Conclusion

In this extensive exploration of Neural Networks, Fuzzy Logic, and Genetic Algorithms, we have demonstrated their capabilities and applications through detailed problem analysis and solution development. Each paradigm offers unique strengths and has been applied to solve diverse problems across various domains.

Neural networks excel in pattern recognition and classification tasks, leveraging deep learning techniques to achieve state-of-the-art performance. Fuzzy logic provides a framework for modeling and reasoning under uncertainty, enabling the development of robust control systems and decision support systems. Genetic algorithms offer a powerful approach to optimization, allowing for the discovery of optimal solutions in complex search spaces.