# EXPERIMENT 3.1

| | |
|---|---|
| **Student Name: Samuel** | **UID: 24MAI10018** |
| **Branch: AIML** | **Section/Group: 24MAI-1** |
| **Semester: 1** | **Date of Performance: / /2025** |
| **Subject Name: SC Lab** | **Subject Code: 24CSH–668** |

**AIM:** Study of ANFIS Architecture

**SOFTWARE REQUIED:** jupyter lab

**THEORY:** ANFIS (Adaptive Neuro-Fuzzy Inference System) combines fuzzy logic and neural networks to create a model capable of handling complex relationships between inputs and outputs. It's particularly useful when you have uncertain, imprecise, or non-linear data.

In ANFIS:

- Fuzzy Logic is used to model the system in linguistic terms (such as "low", "medium", "high").
- Neural Networks are used to optimize the parameters (both membership functions and fuzzy rule parameters) based on data.

The process can be broken down into Fuzzification, Rule Evaluation, Defuzzification, and Learning.

## ALGORITHM:

Layer 1: Fuzzification Layer (Input Layer)

- Objective: This layer assigns membership values to each input. It converts crisp inputs (e.g., 30, 45) into fuzzy values (e.g., "low", "medium", "high").
- Process: Each node represents a membership function for an input variable. For example, for an input xx, you may define three fuzzy sets (Low, Medium, High), and the node will calculate the membership degree of xx in these fuzzy sets.

Membership Functions (MFs): These are used in layer 1 to quantify the fuzziness. A membership function defines how each point in the input space is mapped to a membership value in the fuzzy set.

- Gaussian MF: μ(x)=e−(x−c)2/(2σ2)\mu(x) = e^{-(x — c)² / (2 \sigma²)}
- Triangular MF: A triangular shape, defined by three points (left, peak, right).

- Trapezoidal MF: A trapezoid with two sides having the same slope.

Layer 2: Rule Layer (Fuzzy Rule Evaluation)

- Objective: Each node in this layer corresponds to a fuzzy rule. The nodes perform the fuzzy AND operation (typically minimum) to combine the fuzzy sets from Layer 1.
- Process: The rule node calculates the firing strength of each rule. In a typical fuzzy system, rules are of the form:
- Rule 1: IF xx is Low, THEN yy is Medium
- Rule 2: IF xx is High, THEN yy is High

Each node computes the firing strength based on the fuzzy membership values from Layer 1. The firing strength is calculated by taking the product of the membership values of the input variables.

Layer 3: Normalization Layer

- Objective: This layer normalizes the firing strengths calculated in Layer 2. The normalization ensures that the sum of the firing strengths equals 1, making the system a probability distribution.
- Process: Each node calculates the normalized firing strength.

Layer 4: Defuzzification Layer

- Objective: This layer calculates the output of each rule. The output is weighted by the normalized firing strength calculated in Layer 3.
- Process: Each node multiplies the normalized firing strength from Layer 3 by the consequent parameters of the fuzzy rule. The consequent part is typically linear or constant (e.g., z=ax+bz = a x + b).

For a fuzzy rule of the form:

- Rule: IF xx is Low AND yy is High THEN z=ax+bz = a x + b

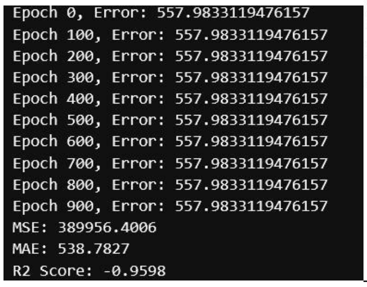
Layer 5: Output Layer

- Objective: The final output is calculated by summing the outputs of all the rules.
- Process: This layer calculates the overall output by summing up the weighted outputs of all rules

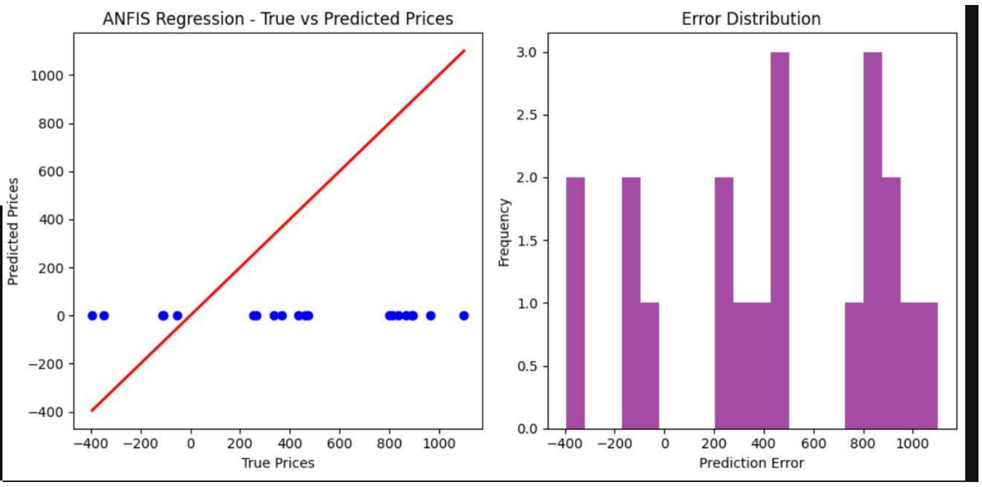
## SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

class ANFIS:
    def __init__(self, X, y, n_rules):
        self.X = X
        self.y = y.reshape(-1, 1)
        self.n_rules = n_rules
        self.n_inputs = X.shape[1]
        self.mf_params = np.random.rand(self.n_inputs, n_rules, 2) * 5  # Adjusting init scale
        self.rule_params = np.random.rand(n_rules, self.n_inputs + 1)

    def gaussian_mf(self, x, c, sigma):
        sigma = max(sigma, 1e-3)  # Prevent division by very small values
        return np.exp(-((x - c) ** 2) / (2 * sigma ** 2))

    def fuzzification(self, X):
        fuzzified = np.zeros((X.shape[0], self.n_rules))
        for j in range(self.n_rules):
            product = np.ones(X.shape[0])
            for i in range(self.n_inputs):
                c, sigma = self.mf_params[i, j]
                product *= self.gaussian_mf(X[:, i], c, sigma)
            fuzzified[:, j] = product
        return fuzzified

    def rule_evaluation(self, fuzzified):
        epsilon = 1e-6
        return fuzzified / (np.sum(fuzzified, axis=1, keepdims=True) + epsilon)

    def defuzzify(self, normalized_w, X):
        X_aug = np.hstack([X, np.ones((X.shape[0], 1))])
        return np.sum(normalized_w[:, :, np.newaxis] * self.rule_params[np.newaxis, :, :] @ X_aug[:, :,
np.newaxis], axis=1)

    def forward_pass(self, X):
        fuzzified = self.fuzzification(X)
        normalized_w = self.rule_evaluation(fuzzified)
        return self.defuzzify(normalized_w, X).squeeze()

    def train(self, epochs=1000, lr=0.01):
        for epoch in range(epochs):
            y_pred = self.forward_pass(self.X)
            error = self.y - y_pred.reshape(-1, 1)

            if np.isnan(error).any():
                print("NaN encountered! Stopping training.")
                break

            for i in range(self.n_inputs):
                diff = self.X[:, i][:, np.newaxis] - self.mf_params[i, :, 0]
                grad = np.sum(error * diff, axis=0, keepdims=True)
                grad = np.clip(grad, -1, 1)  # Gradient Clipping
            self.mf_params[i, :, 0] -= lr * grad.squeeze()
            self.mf_params[i, :, 1] -= lr * np.abs(grad.squeeze())

        self.rule_params -= lr * (error.T @ np.hstack([self.X, np.ones((self.X.shape[0], 1))])) /
self.X.shape[0]

        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Error: {np.mean(np.abs(error))}')

    def predict(self, X):
        return self.forward_pass(X)

np.random.seed(42)
size = np.random.uniform(500, 5000, 100)
rooms = np.random.randint(1, 10, 100)
price = 50 + 0.1 * size + 20 * rooms + np.random.normal(0, 500, 100)

X = np.vstack([size, rooms]).T
y = price
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

anfis = ANFIS(X_train, y_train, n_rules=4)
anfis.train(epochs=1000, lr=0.01)

y_pred = anfis.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'MSE: {mse:.4f}')
print(f'MAE: {mae:.4f}')
print(f'R2 Score: {r2:.4f}')

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred, color='blue')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linewidth=2)
plt.xlabel('True Prices')
plt.ylabel('Predicted Prices')
plt.title('ANFIS Regression - True vs Predicted Prices')

plt.subplot(1, 2, 2)
errors = y_test - y_pred
plt.hist(errors, bins=20, color='purple', alpha=0.7)
plt.xlabel('Prediction Error')
plt.ylabel('Frequency')
plt.title('Error Distribution')
plt.tight_layout()
plt.show()
```

## SCREENSHOT OF OUTPUTS:



```
Epoch 0, Error: 557.9833119476157
Epoch 100, Error: 557.9833119476157
Epoch 200, Error: 557.9833119476157
Epoch 300, Error: 557.9833119476157
Epoch 400, Error: 557.9833119476157
Epoch 500, Error: 557.9833119476157
Epoch 600, Error: 557.9833119476157
Epoch 700, Error: 557.9833119476157
Epoch 800, Error: 557.9833119476157
Epoch 900, Error: 557.9833119476157
MSE: 389956.4006
MAE: 538.7827
R2 Score: -0.9598
```



LEARNING OUTCOMES:
Here are four small learning outcomes from your ANFIS regression implementation:

1. Fuzzy Membership Functions Improve Interpretability
2. Gradient Clipping Enhances Stability
3. Normalization of Rule Weights is Essential
4. Regularization and Learning Rate Tuning Impact Performance