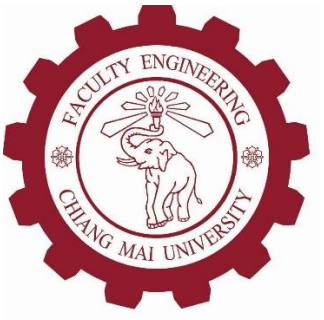


รายงานการทดลอง

Computer Assignment 4

261456 (Introduction to Computational Intelligence)

โดย

ปิยะนันท์ ปิยะวรรณ์โณ 650610845

เสนอ

รศ.ดร. ศันสนีย์ เอื้อพันธ์วิริยะกุล

ภาคเรียนที่ 1 ปีการศึกษา 2567

มหาวิทยาลัยเชียงใหม่

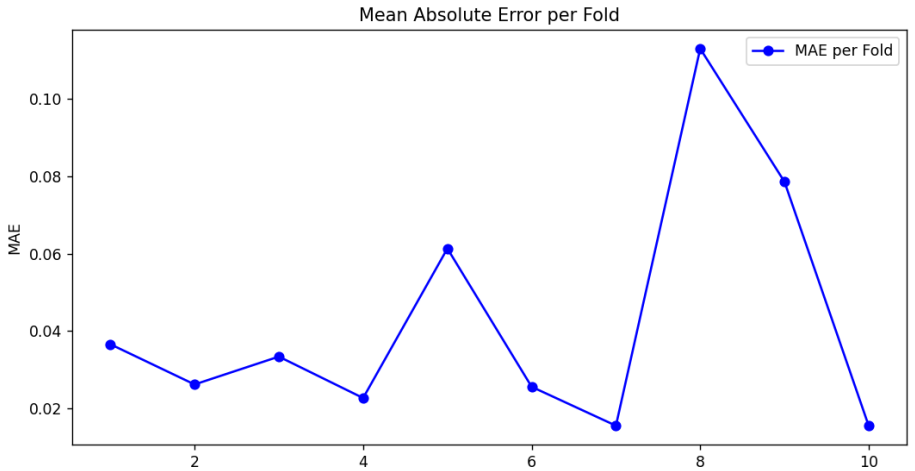
การ Train Multi-layers Perceptron โดยใช้ Particle Swarm Optimization

วิธีการทำงานของโปรแกรม

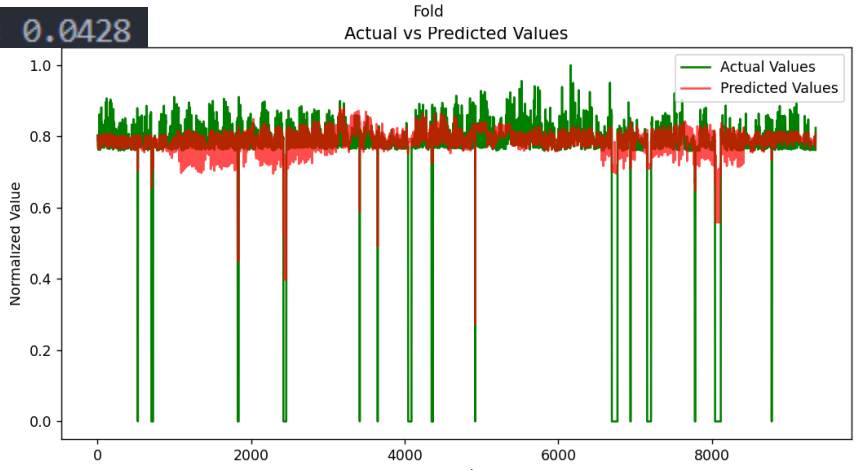
1. นำข้อมูลเข้าจากไฟล์ AirQualityUCI
2. ทำการ Normalization ข้อมูล
3. ทำ 10 % Cross validation โดยการแบ่งข้อมูลเป็น 10 ชุด
4. สุ่ม weight และ Bias ของข้อมูลแต่ละชุด
5. Feed Forward และหาค่า best position
6. หาค่า loss จาก Mean Absolute Error

ผลการทำงาน

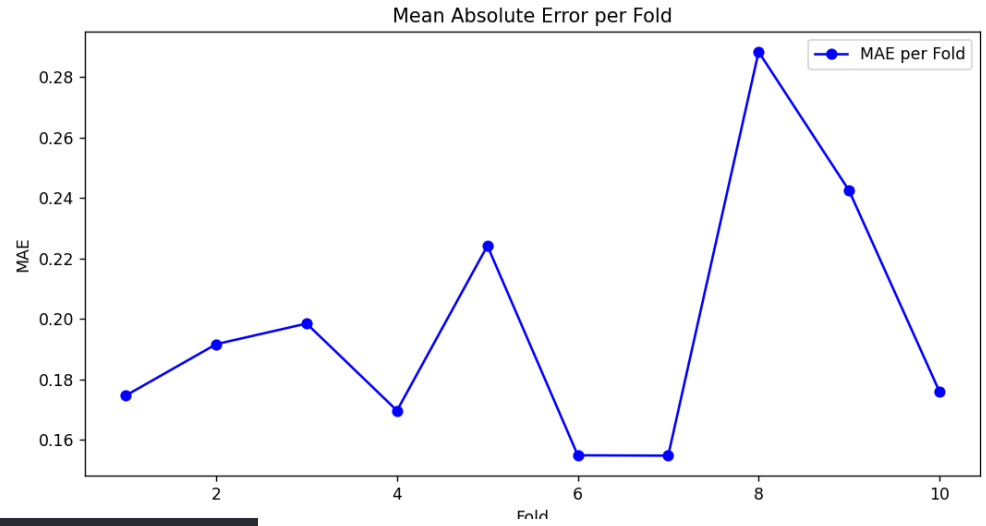
1. ใช้ค่า Hidden layer = 10 , Particle = 50 , iteration = 30



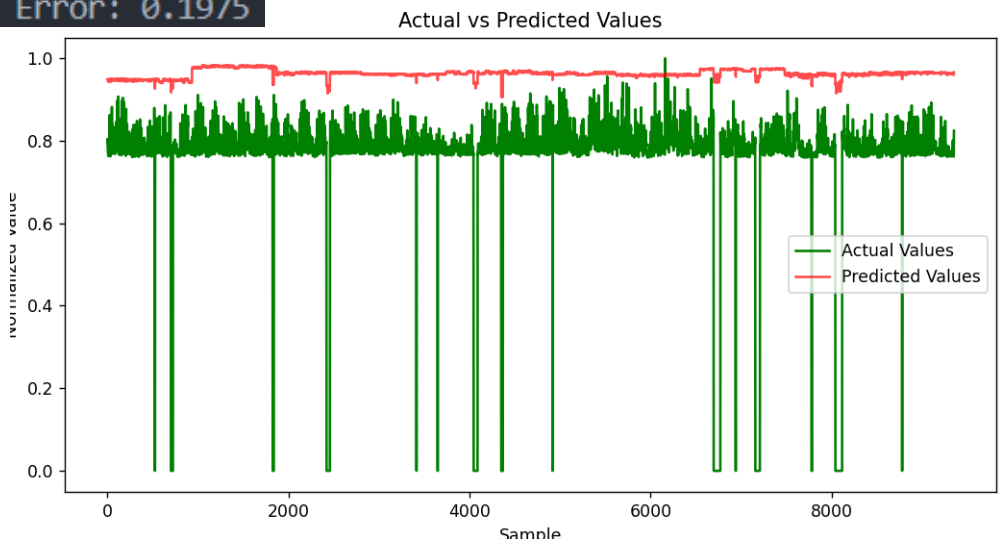

```
Average Error: 0.0428
```



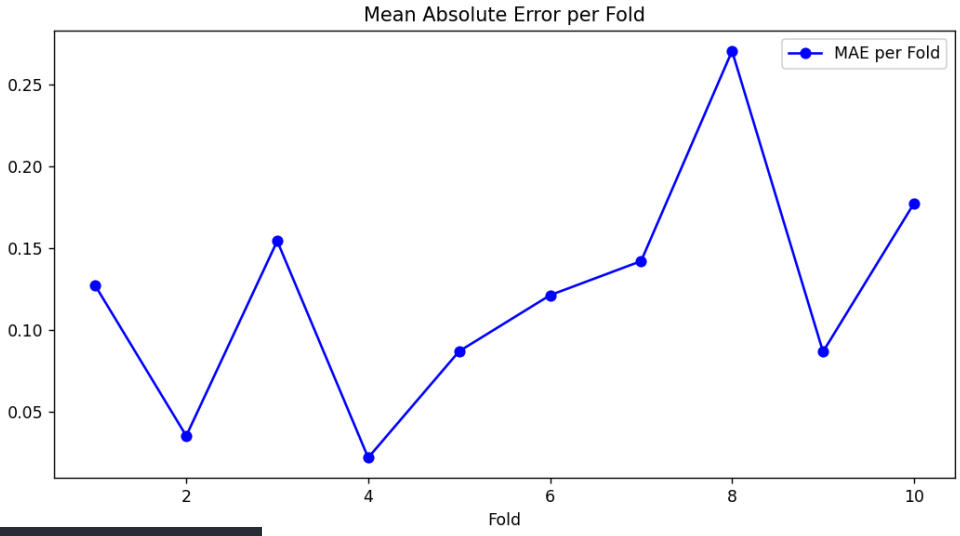
2. ใช้ค่า Hidden layer = 10 , Particle = 50 , iteration = 1



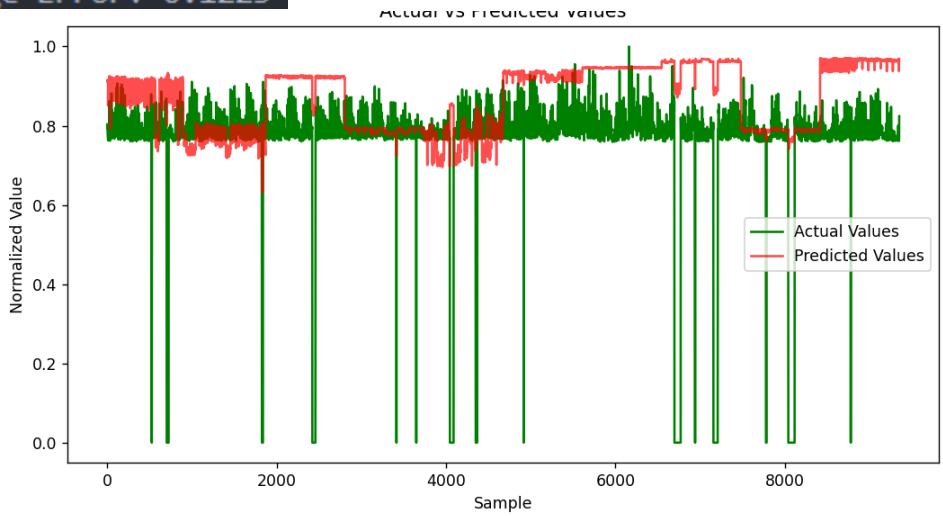

```
Average Error: 0.1975
```



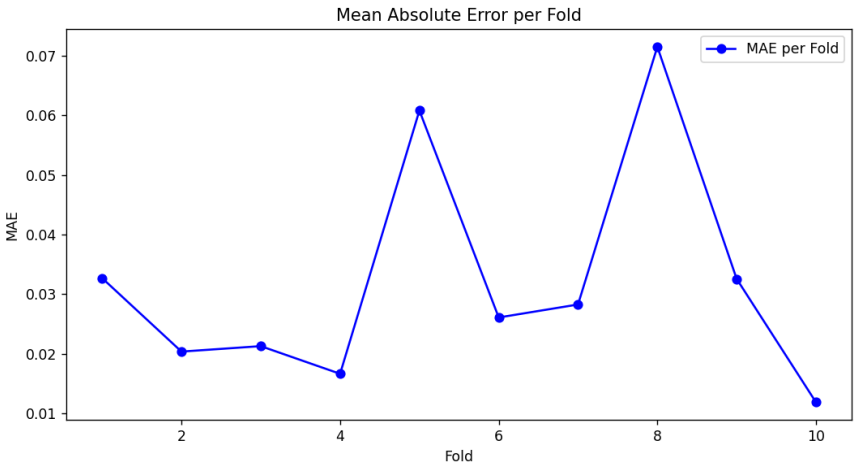
3. ใช้ค่า Hidden layer = 10 , Particle = 50 , iteration = 30



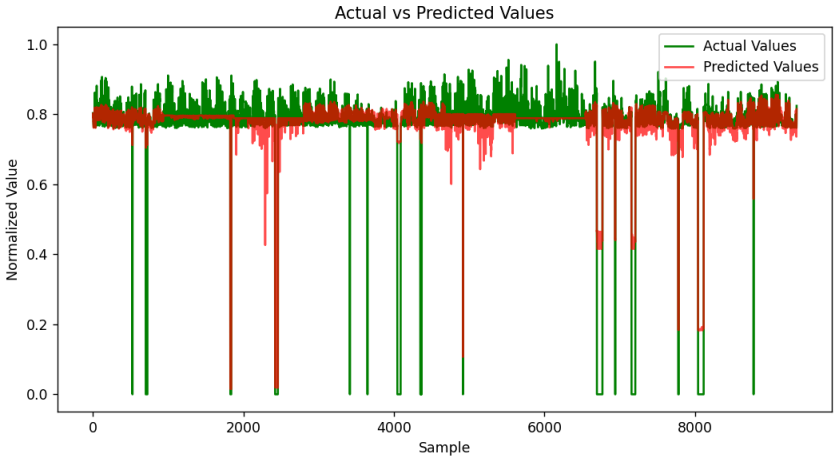

```
Average Error: 0.1223
```



4. ใช้ค่า Hidden layer = 10 , Particle = 50 , iteration = 30




```
Average Error: 0.0322
```



สรุปผลการทำงาน

จากการทดลองทำ MLP โดยใช้ Particle Swarm Optimization จะเห็นว่าเมื่อ input ค่าที่เหมาะสม โปรแกรมจะมีค่า Error น้อย แต่ถ้า Input ค่าใดค่าหนึ่งมากหรือน้อยเกินไป ค่า Error จะเพิ่มขึ้น ค่า Error ที่เพิ่มขึ้นทำให้ค่าที่ Predict ออกมา ไม่มีความแม่นยำ

โปรแกรม

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load Data
def Load_data(file='AirQualityUCI.xlsx'):
    data = pd.read_excel(file)
    inputs = data.iloc[:, [2, 5, 7, 9, 10, 11, 12, 13]].values
    outputs = data.iloc[:, 5].values.reshape(-1, 1)  # Reshape to column vector
    return inputs, outputs

# 2. Normalization Functions
def Normalize(x):
    return (x - np.min(x, axis=0)) / (np.max(x, axis=0) - np.min(x, axis=0))

def Denormalize(normalized_X, x):
    return normalized_X * (np.max(x, axis=0) - np.min(x, axis=0)) + np.min(x, axis=0)

# 3. Activation Function
def Sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 4. Weight and Bias Initialization
def Init_WeightandBias(input_size, hidden_size, output_size):
    weight_input_hidden = np.random.randn(hidden_size, input_size) * 0.1
    bias_hidden = np.zeros((hidden_size, 1))
    weight_hidden_output = np.random.randn(output_size, hidden_size) * 0.1
    bias_output = np.zeros((output_size, 1))
    return weight_input_hidden, weight_hidden_output, bias_hidden, bias_output
```

```python
def Feed_Forward(inputs, weight_input_hidden, weight_hidden_output, bias_hidden, bias_output):
    hidden_input = np.dot(weight_input_hidden, inputs) + bias_hidden
    hidden_output = Sigmoid(hidden_input)
    output_input = np.dot(weight_hidden_output, hidden_output) + bias_output
    output = Sigmoid(output_input)
    return output

# 6. Particle Initialization for PSO
def Init_Particle(num_params):
    position = np.random.rand(num_params)
    velocity = np.random.rand(num_params) * 0.1
    best_position = np.copy(position)
    best_value = float('inf')
    return position, velocity, best_position, best_value

# 7. Loss Function (Mean Absolute Error)
def Mean_Absolute_Error(pred, actual):
    return np.mean(np.abs(pred - actual))
```

```python
def Train(inputs, outputs, input_size=8, hidden_size=1, output_size=1, num_particles=50, max_iter=30):
    num_params = (input_size * hidden_size + hidden_size * output_size + hidden_size + output_size)
    particles = [Init_Particle(num_params) for _ in range(num_particles)]

    global_best_position = np.zeros(num_params)
    global_best_value = float('inf')

    for iteration in range(max_iter):
        for i, (position, velocity, best_position, best_value) in enumerate(particles):
            w_ih = position[:input_size * hidden_size].reshape(hidden_size, input_size)
            w_ho = position[input_size * hidden_size:(input_size * hidden_size + hidden_size * output_size)].reshape(output_size, hidden_size)
            b_h = position[(input_size * hidden_size + hidden_size * output_size):-output_size].reshape(hidden_size, 1)
            b_o = position[-output_size:].reshape(output_size, 1)

            predictions = Feed_Forward(inputs.T, w_ih, w_ho, b_h, b_o)
            loss = Mean_Absolute_Error(predictions, outputs.T)

            if loss < best_value:
                best_position = np.copy(position)
                best_value = loss

            if loss < global_best_value:
                global_best_position = np.copy(position)
                global_best_value = loss

            inertia = 0.5
            cognitive = 1.5 * np.random.rand() * (best_position - position)
            social = 1.5 * np.random.rand() * (global_best_position - position)
            velocity = inertia * velocity + cognitive + social
            position += velocity

            particles[i] = (position, velocity, best_position, best_value)

        print(f"Iteration {iteration+1}/{max_iter}, Best Loss: {global_best_value:.4f}")

    w_ih = global_best_position[:input_size * hidden_size].reshape(hidden_size, input_size)
    w_ho = global_best_position[input_size * hidden_size:(input_size * hidden_size + hidden_size * output_size)].reshape(output_size, hidden_size)
    b_h = global_best_position[(input_size * hidden_size + hidden_size * output_size):-output_size].reshape(hidden_size, 1)
    b_o = global_best_position[-output_size:].reshape(output_size, 1)

    return w_ih, w_ho, b_h, b_o
```

```python
def Cross_Validation(inputs, outputs, k=10):
    fold_size = len(inputs) // k
    errors = []
    all_predictions = []
    all_actuals = []

    for i in range(k):
        val_start = i * fold_size
        val_end = val_start + fold_size

        X_val = inputs[val_start:val_end]
        y_val = outputs[val_start:val_end]
        X_train = np.concatenate((inputs[:val_start], inputs[val_end:]), axis=0)
        y_train = np.concatenate((outputs[:val_start], outputs[val_end:]), axis=0)

        w_ih, w_ho, b_h, b_o = Train(X_train, y_train)

        predictions = Feed_Forward(X_val.T, w_ih, w_ho, b_h, b_o)
        error = Mean_Absolute_Error(predictions, y_val.T)
        errors.append(error)

        all_predictions.extend(predictions.flatten())
        all_actuals.extend(y_val.flatten())

        print(f"Fold {i+1}, Error: {error:.4f}")

    print(f"Average Error: {np.mean(errors):.4f}")
```

```
    # Plot MAE per Fold
    plt.figure(figsize=(10, 5))
    plt.plot(range(1, k+1), errors, marker='o', linestyle='-', color='b', label='MAE per Fold')
    plt.xlabel('Fold')
    plt.ylabel('MAE')
    plt.title('Mean Absolute Error per Fold')
    plt.legend()
    plt.show()

    # Plot Actual vs Predicted Values
    plt.figure(figsize=(10, 5))
    plt.plot(all_actuals, label='Actual Values', color='g')
    plt.plot(all_predictions, label='Predicted Values', color='r', alpha=0.7)
    plt.xlabel('Sample')
    plt.ylabel('Normalized Value')
    plt.title('Actual vs Predicted Values')
    plt.legend()
    plt.show()

# Example Usage
inputs, outputs = Load_data()
inputs = Normalize(inputs)
outputs = Normalize(outputs)
Cross_Validation(inputs, outputs)
```

SPHSTR/Computer_Intelligence_Particle_Swarm_Optimization