รายงาน

เรื่อง

การ train backpropagation (the generalized delta rule) สำหรับ Multi-Layer Perceptron (MLP) :

การทดลองการพยากรณ์ระดับน้ำที่สะพานนวรัตน์

โดย

น.ส. ภัทรจาริน ผดุงกิจเจริญ รหัสนักศึกษา 650610851

เสนอ

รศ.ดร. ศันสนีย์ เอื้อพันธ์วิริยะกุล

รายงานนี้เป็นส่วนหนึ่งของรายวิชา CPE 261456

Introduction to Computational Intelligence

สาขาวิชาวิศวกรรมหุ่นยนต์และปัญญาประดิษฐ์

ภาคเรียนที่ 1 ปีการศึกษา 2567

มหาวิทยาลัยเชียงใหม่

สารบัญ

บทที่ 1 บทนำ

1.1 ความเป็นมาและความสำคัญ

ในรายวิชา "INTRO COMP INTEL FOR CPE" นักศึกษาจะได้เรียนรู้การพัฒนาและฝึกโมเดล Multi-Layer Perceptron (MLP) สำหรับการทำนายระดับน้ำ ซึ่งเป็นการประยุกต์ใช้ปัญญาประดิษฐ์ในการแก้ปัญหาจริง การทดลองนี้จะช่วยให้นักศึกษาเข้าใจเทคนิค backpropagation และการปรับแต่งพารามิเตอร์ของโมเดล เพื่อเพิ่มความแม่นยำและประสิทธิภาพในการทำนาย ทำให้นักศึกษาเตรียมความพร้อมสำหรับการทำงานในด้านปัญญาประดิษฐ์ในอนาคต

1.2 วัตถุประสงค์ของการศึกษา

1. พัฒนาและฝึก Multi-Layer Perceptron (MLP) สำหรับการทำนายระดับน้ำในอนาคตโดยใช้ข้อมูลย้อนหลัง
2. ตรวจสอบประสิทธิภาพของโมเดลด้วยการใช้ 10% cross-validation และการวัดผลด้วย confusion matrix
3. วิเคราะห์ผลกระทบของการเปลี่ยนแปลงจำนวน hidden nodes, learning rate, และ momentum rate ต่อความแม่นยำและการ converge ของโมเดล

1.3 ขอบเขตของการศึกษา

1. ข้อมูล: ใช้ข้อมูลระดับน้ำจากสถานี 1 และสถานี 2 และข้อมูล cross.pat
2. การสร้างโมเดล: พัฒนาและฝึก Multi-Layer Perceptron (MLP) โดยใช้ backpropagation
3. การทำนาย: ทำนายระดับน้ำในอนาคต 7 ชั่วโมง โดยใช้ข้อมูลย้อนหลัง 3 ชั่วโมง
4. การทดสอบ: ใช้ 10% cross-validation ในการทดสอบโมเดล
5. การปรับพารามิเตอร์: เปลี่ยนแปลงจำนวน hidden nodes, learning rate, และ momentum rate เพื่อตรวจสอบผลกระทบต่อความแม่นยำและการ converge
6. การประเมินผล: ใช้ confusion matrix เพื่อประเมินความแม่นยำและประสิทธิภาพของโมเดล

บทที่ 2 ทฤษฎีและหลักการที่เกี่ยวข้อง

2.1 Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron (MLP) คือโครงข่ายประสาทเทียมที่มีหลายชั้น ประกอบด้วย Input Layer , Hidden Layer(s) และ Output Layer ใช้สำหรับการจำแนกและพยากรณ์ข้อมูล โดยการปรับน้ำหนักของการเชื่อมต่อระหว่างหน่วยประสาทผ่านกระบวนการ backpropagation เพื่อให้ได้ผลลัพธ์ที่แม่นยำ และการเพิ่ม Hidden Layer(s) ช่วยให้โมเดลสามารถเรียนรู้ความสัมพันธ์ที่ซับซ้อนจากข้อมูลได้

2.2 Backpropagation Algorithm

Backpropagation คือกระบวนการที่ใช้ในการฝึกสอนโครงข่ายประสาทเทียม โดยมีขั้นตอนหลักๆ ดังนี้:

1. การส่งไปข้างหน้า (Forward Propagation): ข้อมูลถูกส่งผ่านโครงข่ายจาก Input Layer ผ่าน Hidden Layer(s) ไปยัง Output Layer เพื่อคำนวณผลลัพธ์
2. การคำนวณข้อผิดพลาด (Error Calculation): เปรียบเทียบผลลัพธ์ที่คำนวณได้กับค่าจริงเพื่อหาค่าความผิดพลาด
3. การส่งกลับ (Backward Propagation): ข้อผิดพลาดจะถูกส่งย้อนกลับจาก Output Layer ไปยัง Hidden Layer(s) และ Input Layer เพื่อต้องการหาค่าของการเปลี่ยนแปลงในน้ำหนักของการเชื่อมต่อ
4. การปรับน้ำหนัก (Weight Update): ปรับน้ำหนักของการเชื่อมต่อในโครงข่ายประสาทเพื่อให้ค่าความผิดพลาดลดลง

กระบวนการนี้จะทำซ้ำหลายรอบจนกระทั่งข้อผิดพลาดลดลงและโมเดลมีความแม่นยำมากขึ้น

2.3 Cross-Validation

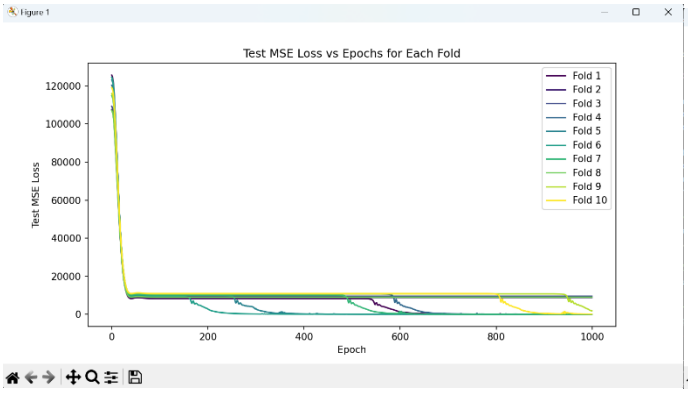
Cross-Validation เป็นเทคนิคที่ใช้ในการประเมินประสิทธิภาพของโมเดลโดยการแบ่งข้อมูลออกเป็นส่วนย่อยๆ หลายๆ ส่วน การใช้ 10% cross-validation หมายถึงการแบ่งข้อมูลเป็น 10 ส่วนเท่าๆ กัน โดยใช้ 9 ส่วนสำหรับการฝึกและ 1 ส่วนสำหรับการทดสอบ และทำซ้ำ 10 รอบเพื่อให้แน่ใจว่าทุกส่วนของข้อมูลได้รับการทดสอบ

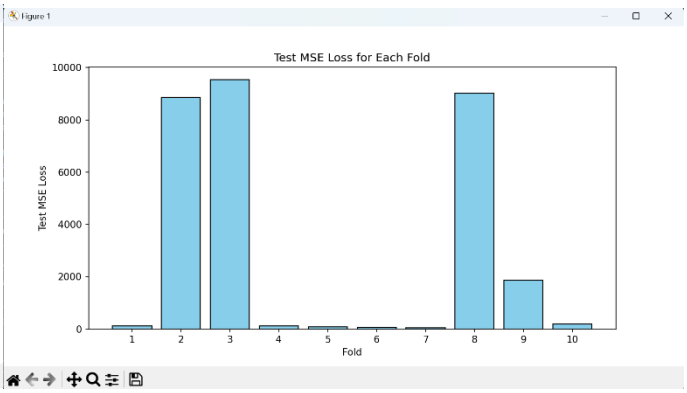
บทที่ 3 วิธีการทดลอง

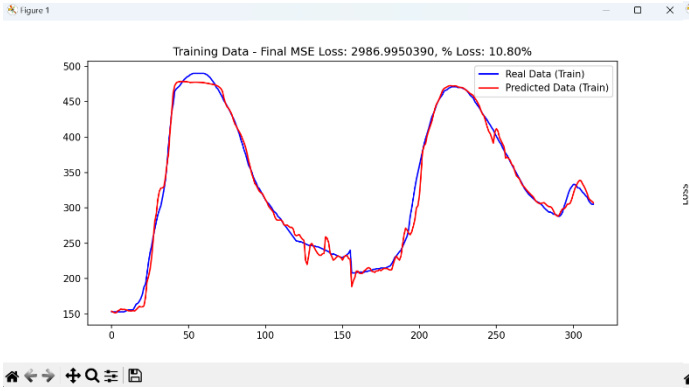
1. การเตรียมข้อมูล:
   - ข้อมูลน้ำท่วม: รวมข้อมูลระดับน้ำจากสถานี 1 และสถานี 2 ที่มีข้อมูลย้อนหลัง 3 ชั่วโมง เพื่อใช้ในการทำนายระดับน้ำในอนาคต 7 ชั่วโมง
   - ข้อมูล cross.pat: ข้อมูลที่มี 2 คลาสและ 2 ฟีเจอร์สำหรับการทดสอบ
2. การออกแบบและสร้าง MLP:
   - กำหนดจำนวนเลเยอร์และจำนวนโหนดในแต่ละเลเยอร์
3. การฝึก MLP:
   - Forward Propagation: ส่งข้อมูลผ่านเครือข่ายเพื่อคำนวณผลลัพธ์
   - Backward Propagation: คำนวณค่าความผิดพลาดและปรับปรุงน้ำหนักของเครือข่ายตามค่าความผิดพลาด
   - การฝึกโมเดล: ใช้ข้อมูลสำหรับการฝึกและปรับน้ำหนักตาม learning rate และ momentum rate ที่กำหนด
4. การทดสอบและประเมินผล:
   - การทำนายระดับน้ำ: ใช้โมเดลที่ฝึกแล้วในการทำนายระดับน้ำในอนาคต 7 ชั่วโมง
   - Cross-Validation: แบ่งข้อมูลออกเป็น 10 ชุด ใช้ 90% สำหรับการฝึกและ 10% สำหรับการทดสอบ
   - Confusion Matrix: ใช้ในการประเมินผลลัพธ์สำหรับข้อมูล cross.pat และวัดความแม่นยำ
5. การปรับปรุงและวิเคราะห์ผล:
   - เปลี่ยนแปลงจำนวน hidden nodes, learning rate, และ momentum rate เพื่อทดสอบผลกระทบต่อความแม่นยำและการ converge ของโมเดล
   - วิเคราะห์ผลลัพธ์และเปรียบเทียบความแม่นยำและประสิทธิภาพของโมเดลที่แตกต่างกัน

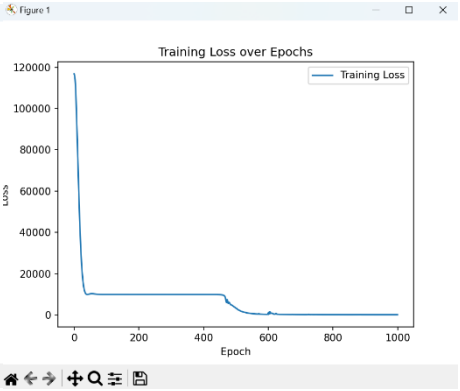
บทที่ 4 ผลการทดลอง

hidden_layers = [8, 5] , epochs = 1000 , learning_rate = 0.01 , momentum_rate = 0.9

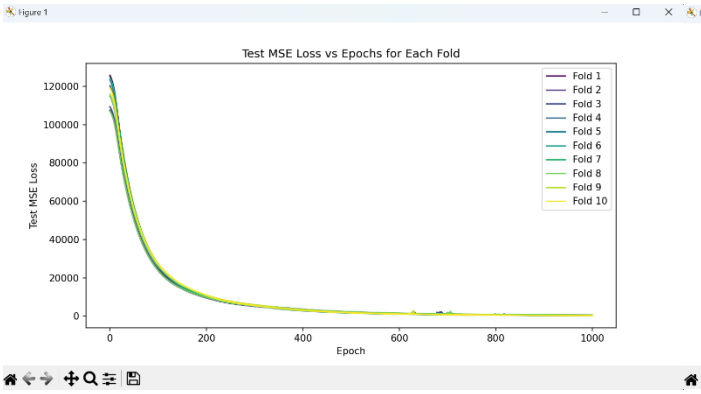


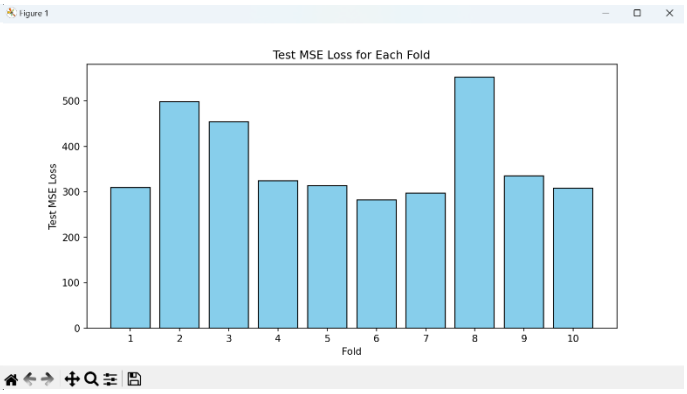


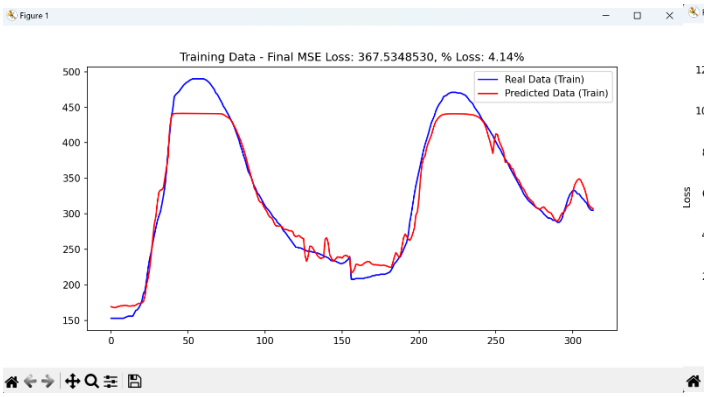


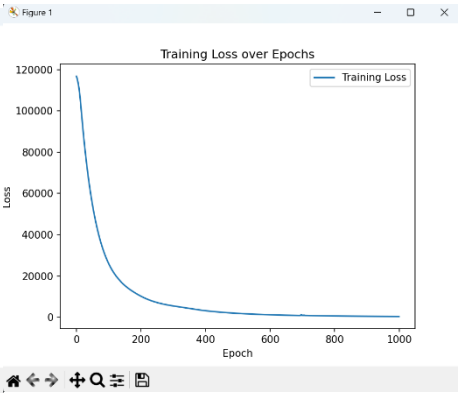


hidden_layers = [4, 9] , epochs = 1000 , learning_rate = 0.001 , momentum_rate = 0.5









ภาคผนวก

GITHUB : https://github.com/Pattharajrin/261456_CI_HW1_Y3-1.git

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

# ฟังก์ชันสำหรับอ่านข้อมูลจากไฟล์ Flood_dataset.txt
def Read_Flood_data():
    with open("Flood_dataset.txt", "r") as f:
        data = [line.split() for line in f]

    data = np.array(data[2:])  # ข้ามสองบรรทัดแรก
    input_train = np.array([i[:8] for i in data], dtype=np.float32)  # ใช้ list comprehension เพื่อสร้าง input_train
    output_train = np.array([i[-1] for i in data], dtype=np.int64)  # ใช้ list comprehension เพื่อสร้าง output_train
    output_train = output_train.reshape(-1, 1)  # แปลง output_train ให้เป็นสองมิติ

    return input_train, output_train  # คืนค่า input_train และ output_train

# ฟังก์ชันสำหรับอ่านข้อมูลจากไฟล์ cross.txt
def Read_Cross_data(filename='cross.txt'):
    data = []
    with open(filename) as f:
        lines = f.readlines()
        for line in range(1, len(lines), 3):
            cross1 = np.array([float(element) for element in lines[line].strip().split()])
            cross2 = np.array([float(element) for element in lines[line + 1].strip().split()])
            data.append(np.hstack((cross1, cross2)))  # รวม cross1 และ cross2 ด้วย hstack

    data = np.array(data)
    input_data = data[:, :-2]  # แยก input
    output_data = data[:, -2:]  # แยก output
    return input_data, output_data  # คืนค่า input_data และ output_data

# ฟังก์ชันสำหรับการปรับขนาดข้อมูล (Normalization)
def Normalize(X):
    mean = np.mean(X, axis=0)  # คำนวณค่าเฉลี่ย
    std = np.std(X, axis=0)  # คำนวณส่วนเบี่ยงเบนมาตรฐาน
    X_normalized = (X - mean) / std  # ปรับขนาดข้อมูล
    return X_normalized, mean, std  # คืนค่า X_normalized, mean และ std

# ฟังก์ชันสำหรับฟังก์ชัน Sigmoid
def Sigmoid(x):
    return 1 / (1 + np.exp(-x))  # ฟังก์ชัน Sigmoid

# ฟังก์ชันสำหรับคำนวณเอนุพันธ์ของ Sigmoid
def Sigmoid_Derivative(x):
    return x * (1 - x)  # คำนวณเอนุพันธ์ของ Sigmoid

# ฟังก์ชันสำหรับเริ่มต้นพารามิเตอร์
def Initialize_Parameters(input_layers, hidden_layers, output_layers):
    parameters = {}
    layer_dims = [input_layers] + hidden_layers + [output_layers]

    # เริ่มต้นพารามิเตอร์ weights และ biases สำหรับแต่ละชั้นในเครือข่ายประสาทเทียม
    for l in range(1, len(layer_dims)):
        parameters[f"weights{l}"] = np.random.randn(layer_dims[l-1], layer_dims[l]) * 0.01  # สุ่มค่า weights
        parameters[f"biases{l}"] = np.zeros((1, layer_dims[l]))  # เริ่มต้นค่า biases ด้วย 0

    return parameters  # คืนค่าพารามิเตอร์ทั้งหมด

# ฟังก์ชันสำหรับเริ่มต้นค่า velocity
def Initialize_Velocity(parameters):
    velocity = {}
    L = len(parameters) // 2  # จำนวนชั้นของเครือข่ายประสาทเทียม (ไม่นับ input layer)

    # เริ่มต้นค่าเทอมความเร็ว (velocity) สำหรับแต่ละพารามิเตอร์
    for l in range(1, L + 1):
        velocity[f"dweights{l}"] = np.zeros_like(parameters[f"weights{l}"])  # เริ่มต้น dweights ด้วย 0
        velocity[f"dbiases{l}"] = np.zeros_like(parameters[f"biases{l}"])  # เริ่มต้น dbiases ด้วย 0

    return velocity  # คืนค่า velocity
```

```python
# ฟังก์ชันสำหรับการส่งข้อมูลไปข้างหน้า (Forward Propagation)
def Forward_Propagation(x, parameters, hidden_layers):
    caches = {}  # ตัวแปรเพื่อเก็บค่าที่จำเป็นสำหรับการ backpropagation
    A = x  # ตั้งค่า A เริ่มต้นเป็น input X
    L = len(hidden_layers) + 1  # จำนวนชั้นทั้งหมดในโมเดล (รวม output layer)

    # การส่งข้อมูลไปข้างหน้าในทุกชั้นที่ซ่อน
    for l in range(1, L):
        Z = np.dot(A, parameters[f"weights{l}"]) + parameters[f"biases{l}"]  # คำนวณ Z = W·A + b
        A = Sigmoid(Z)  # ส่ง Z ผ่านฟังก์ชัน Sigmoid
        caches[f"Z{l}"] = Z  # เก็บ Z ใน caches
        caches[f"A{l}"] = A  # เก็บ A ใน caches

    # การส่งข้อมูลไปข้างหน้าสำหรับชั้นสุดท้าย (output layer)
    ZL = np.dot(A, parameters[f"weights{L}"]) + parameters[f"biases{L}"]  # คำนวณ Z สำหรับชั้นสุดท้าย
    AL = ZL  # สำหรับชั้นสุดท้าย A จะเท่ากับ Z (ไม่มีฟังก์ชันการกระตุ้นที่นี่)
    caches[f"Z{L}"] = ZL  # เก็บ Z สำหรับชั้นสุดท้ายใน caches
    caches[f"A{L}"] = AL  # เก็บ A สำหรับชั้นสุดท้ายใน caches

    return AL, caches  # คืนค่าผลลัพธ์สุดท้าย (AL) และ caches สำหรับการ backpropagation

# ฟังก์ชันคำนวณค่า Mean Squared Error (MSE) loss
def MSE_Loss(Y, AL):
    return np.mean((Y - AL)**2)  # คำนวณค่า MSE ระหว่างค่าจริง (Y) กับค่าที่ทำนายได้ (AL)

# ฟังก์ชันคำนวณเปอร์เซ็นต์ความสูญเสีย
def Percentage_Loss(Y, AL):
    return np.mean(np.abs((Y - AL) / Y)) * 100  # คำนวณเปอร์เซ็นต์ความสูญเสีย

# ฟังก์ชันสำหรับการย้อนกลับ (Backward Propagation)
def Backward_Propagation(X, Y, parameters, caches, hidden_layers):
    grads = {}  # สร้างดิกชันนารีเพื่อเก็บค่าเกรดต่างๆ
    m = X.shape[0]  # จำนวนตัวอย่างในชุดข้อมูล
    L = len(hidden_layers) + 1  # จำนวนชั้นทั้งหมดในโมเดล (รวม output layer)

    # สำหรับเลเยอร์สุดท้าย
    AL = caches[f"A{L}"]
    dZL = AL - Y  # ใช้ AL แทนที่ caches[f"A{L}"]

    # คำนวณ gradients สำหรับเลเยอร์สุดท้าย
    grads[f"dweights{L}"] = np.dot(caches[f"A{L-1}"].T, dZL) / m
    grads[f"dbiases{L}"] = np.sum(dZL, axis=0, keepdims=True) / m

    # สำหรับเลเยอร์อื่นๆ
    for l in reversed(range(1, L)):
        dA_prev = np.dot(dZL, parameters[f"weights{l+1}"].T)
        dZ = dA_prev * Sigmoid_Derivative(caches[f"A{l}"])
        grads[f"dweights{l}"] = np.dot(caches[f"A{l-1}"].T, dZ) / m if l > 1 else np.dot(X.T, dZ) / m
        grads[f"dbiases{l}"] = np.sum(dZ, axis=0, keepdims=True) / m
        dZL = dZ

    return grads  # คืนค่าเกรดทั้งหมดที่คำนวณได้

# ฟังก์ชันอัปเดตพารามิเตอร์
def Update_Parameters(parameters, grads, velocity, learning_rate, momentum_rate):
    L = len(parameters) // 2  # จำนวนชั้นในโมเดล (ไม่นับ input layer)

    for l in range(1, L + 1):  # วนลูปผ่านทุกชั้น
        # ตรวจสอบขนาดของ grads กับ velocity เพื่อป้องกันข้อผิดพลาด
        assert grads[f"dweights{l}"].shape == velocity[f"dweights{l}"].shape, "Shape mismatch in dweights"
        assert grads[f"dbiases{l}"].shape == velocity[f"dbiases{l}"].shape, "Shape mismatch in dbiases"

        # คำนวณค่า velocity สำหรับ weights โดยใช้ Momentum
        velocity[f"dweights{l}"] = momentum_rate * velocity[f"dweights{l}"] + (1 - momentum_rate) * grads[f"dweights{l}"]
        # คำนวณค่า velocity สำหรับ biases โดยใช้ Momentum
        velocity[f"dbiases{l}"] = momentum_rate * velocity[f"dbiases{l}"] + (1 - momentum_rate) * grads[f"dbiases{l}"]

        # อัปเดตค่า weights ด้วยค่า velocity
        parameters[f"weights{l}"] -= learning_rate * velocity[f"dweights{l}"]
        # อัปเดตค่า biases ด้วยค่า velocity
        parameters[f"biases{l}"] -= learning_rate * velocity[f"dbiases{l}"]

    return parameters, velocity  # คืนค่า parameters และ velocity ที่อัปเดตแล้ว

```

```python
# ฟังก์ชันสำหรับการฝึกฝน Multi-Layer Perceptron (MLP)
def Train_MLP(X, Y, hidden_layers, epochs, learning_rate, momentum_rate, X_test, Y_test):
    input_dim = X.shape[1]  # จำนวนฟีเจอร์ในข้อมูลอินพุต
    output_dim = Y.shape[1]  # จำนวนฟีเจอร์ในข้อมูลเอาท์พุต
    parameters = Initialize_Parameters(input_dim, hidden_layers, output_dim)  # สุ่มค่าพารามิเตอร์
    velocity = Initialize_Velocity(parameters)  # สุ่มค่าเทอมความเร็ว
    loss_per_epoch = []  # เก็บค่า loss ต่อ epoch
    percentage_loss_per_epoch = []  # เก็บค่าเปอร์เซ็นต์ loss ต่อ epoch

    for epoch in range(epochs):
        AL, caches = Forward_Propagation(X, parameters, hidden_layers)  # ทำ Forward Propagation
        loss = MSE_Loss(Y, AL)  # คำนวณค่า loss
        percentage_loss = Percentage_Loss(Y, AL)  # คำนวณค่าเปอร์เซ็นต์ loss
        loss_per_epoch.append(loss)  # เก็บค่า loss ต่อ epoch
        percentage_loss_per_epoch.append(percentage_loss)  # เก็บค่าเปอร์เซ็นต์ loss ต่อ epoch
        grads = Backward_Propagation(X, Y, parameters, caches, hidden_layers)  # ทำ Backward Propagation
        parameters, velocity = Update_Parameters(parameters, grads, velocity, learning_rate, momentum_rate)  # อัปเดตพารามิเตอร์

        if epoch % 100 == 0 or epoch == epochs - 1:  # แสดงผลลัพธ์ทุก ๆ 100 epochs หรือเมื่อถึง epoch สุดท้าย
            print(f"Epoch {epoch+1}/{epochs} - Loss: {loss} - Percentage Loss: {percentage_loss}%")

    # ทำนายค่าและคำนวณ loss สำหรับชุดทดสอบ
    Y_pred_test, _ = Forward_Propagation(X_test, parameters, hidden_layers)  # ทำการทำนายด้วยโมเดล
    test_loss = MSE_Loss(Y_test, Y_pred_test)  # คำนวณ loss สำหรับชุดทดสอบ
    test_percentage_loss = Percentage_Loss(Y_test, Y_pred_test)  # คำนวณเปอร์เซ็นต์ loss สำหรับชุดทดสอบ
    print(f"Test Loss: {test_loss} - Test Percentage Loss: {test_percentage_loss}%")

    return loss_per_epoch, percentage_loss_per_epoch, parameters, AL  # คืนค่า AL สำหรับการพล็อตกราฟ

# ฟังก์ชันสำหรับการทำ K-Fold Cross Validation
def KFold_CrossValidation(X, Y, hidden_layers, epochs, learning_rate, momentum_rate, K=10):
    fold_size = X.shape[0] // K  # ขนาดของแต่ละ fold
    losses = []  # เก็บค่า loss ของแต่ละ fold
    percentage_losses = []  # เก็บค่าเปอร์เซ็นต์ loss ของแต่ละ fold
    fold_scores = []  # เก็บค่า loss ต่อ epoch ของแต่ละ fold

    for k in range(K):
        print(f"Fold {k+1}/{K}")  # แสดงหมายเลข fold ที่กำลังทำการประเมิน
        start, end = k * fold_size, (k + 1) * fold_size  # กำหนดขอบเขตของ fold ปัจจุบัน
        X_train = np.concatenate((X[:start], X[end:]), axis=0)  # ข้อมูลฝึกสอน
        Y_train = np.concatenate((Y[:start], Y[end:]), axis=0)  # ค่าฝึกสอน
        X_valid = X[start:end]  # ข้อมูลตรวจสอบ
        Y_valid = Y[start:end]  # ค่าตรวจสอบ

        loss_per_epoch, percentage_loss_per_epoch, _, AL_train = Train_MLP(X_train, Y_train, hidden_layers, epochs, learning_rate, momentum_rate, X_valid, Y_valid)  # ฝึกสอนโมเดลและเก็บผลลัพธ์
        losses.append(loss_per_epoch[-1])  # เก็บค่า loss สุดท้ายของ fold นี้
        percentage_losses.append(percentage_loss_per_epoch[-1])  # เก็บค่าเปอร์เซ็นต์ loss สุดท้ายของ fold นี้
        fold_scores.append(loss_per_epoch)  # เก็บ loss ต่อ epoch ของ fold นี้

    # พล็อตกราฟ Test MSE Loss เทียบกับ epochs สำหรับแต่ละ fold
    plt.figure(figsize=(10, 5))
    colors = plt.cm.viridis(np.linspace(0, 1, len(fold_scores)))  # สร้างชุดสีสำหรับแต่ละ fold
    for i, epoch_losses in enumerate(fold_scores):
        print(f"Fold {i+1} - Epoch Losses Length: {len(epoch_losses)}")  # แสดงความยาวของการสูญเสียในแต่ละ epoch ของ fold นี้
        plt.plot(range(len(epoch_losses)), epoch_losses, color=colors[i], label=f'Fold {i+1}')  # พล็อตกราฟการสูญเสียในแต่ละ fold
    plt.xlabel('Epoch')  # แกน X
    plt.ylabel('Test MSE Loss')  # แกน Y
    plt.title('Test MSE Loss vs Epochs for Each Fold')  # ชื่อกราฟ
    plt.legend()  # แสดงคำนาน
    plt.show()  # แสดงกราฟ

    # พล็อตกราฟค่า Test MSE Loss สุดท้ายสำหรับแต่ละ fold
    plt.figure(figsize=(10, 5))
    plt.bar(range(1, len(losses) + 1), losses, color='skyblue', edgecolor='black')  # สร้างกราฟแท่ง
    plt.xlabel('Fold')  # แกน X
    plt.ylabel('Test MSE Loss')  # แกน Y
    plt.title('Test MSE Loss for Each Fold')  # ชื่อกราฟ
    plt.xticks(range(1, len(losses) + 1))  # ตั้งค่าตำแหน่งของแท่ง
    plt.show()  # แสดงกราฟ

    return np.mean(losses), np.mean(percentage_losses), AL_train  # คืนค่าเฉลี่ยของ loss และเปอร์เซ็นต์ loss

if __name__ == "__main__":
    input_data, output_data = Read_Flood_data()  # อ่านข้อมูลจากไฟล์
    input_data, mean, std = Normalize(input_data)  # ทำ Normalization ข้อมูล

    hidden_layers = [8, 5]  # กำหนดจำนวน hidden layers
    epochs = 1000  # จำนวน epoch
    learning_rate = 0.01  # Learning rate
    momentum_rate = 0.9  # Momentum rate

    # ทำ K-Fold Cross Validation
    mean_loss, mean_percentage_loss, last_Y_train = KFold_CrossValidation(input_data, output_data, hidden_layers, epochs, learning_rate, momentum_rate)
    print(f"Mean Loss: {mean_loss} - Mean Percentage Loss: {mean_percentage_loss}%")

    # ฝึกสอนโมเดลสุดท้ายและเก็บค่า loss ต่อ epoch สำหรับการพล็อต
    loss_per_epoch, percentage_loss_per_epoch, _, AL_train = Train_MLP(input_data, output_data, hidden_layers, epochs, learning_rate, momentum_rate, input_data, output_data)

    # พล็อตกราฟ Training Data - Final MSE Loss
    plt.figure(figsize=(10, 5))
    plt.plot(output_data, label='Real Data (Train)', color='blue')  # พล็อตค่าจริง
    plt.plot(AL_train, label='Predicted Data (Train)', color='red')  # พล็อตค่าที่ทำนายได้
    plt.legend()  # แสดงคำนาน
    plt.title(f'Training Data - Final MSE Loss: {mean_loss:.7f}, % Loss: {mean_percentage_loss:.2f}%')  # ชื่อกราฟ
    plt.show()  # แสดงกราฟ

    # พล็อตกราฟ Loss ต่อ Epoch
    plt.plot(loss_per_epoch, label="Training Loss")  # พล็อต loss ต่อ epoch
    plt.xlabel("Epoch")  # แกน X
    plt.ylabel("Loss")  # แกน Y
    plt.title("Training Loss over Epochs")  # ชื่อกราฟ
    plt.legend()  # แสดงคำนาน
    plt.show()  # แสดงกราฟ
```