Lab 5.1: Xây dựng mô hình ANN phân loại dữ liệu phi tuyến

ThS. Lê Thị Thùy Trang

2025-2-16

1. Giới thiệu về bài thực hành

Trong bài thực hành này, sinh viên xây dựng một mô hình ANN đơn giản để phân loại dữ liệu phi tuyến (`nonlinear data`), bài tập này nhằm mục đích mô tả về khả năng hoạt động của một mạng Multilayer Perceptron với dữ liệu phi tuyến.

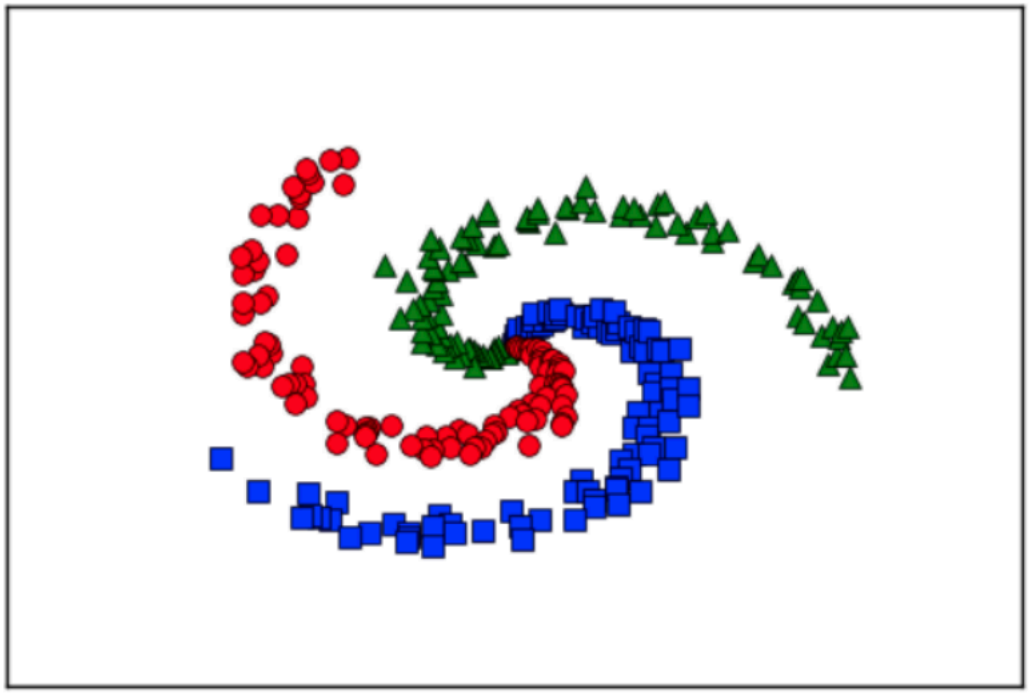

Figure 1: Trực quan hoá trên không gian 2D của bộ dữ liệu phi tuyến

Sinh viên thực hiện thay đổi cấu trúc mạng và hàm kích hoạt sau đó đưa ra nhận xét về ảnh hưởng của các yếu tố này đến hiệu suất của mô hình.

1.1 Hàm kích hoạt

Hàm kích hoạt là các hàm toán học được áp dụng cho đầu ra của các nơ-ron trong MLP, với ý nghĩa chính là giúp MLP đưa ra quyết định có nên kích hoạt nơ-ron hay không. Nó giúp chuyển đổi đầu ra tuyến tính của một lớp thành một dạng phi tuyến cho phép mạng nơ-ron học được các quan hệ phức tạp hơn.

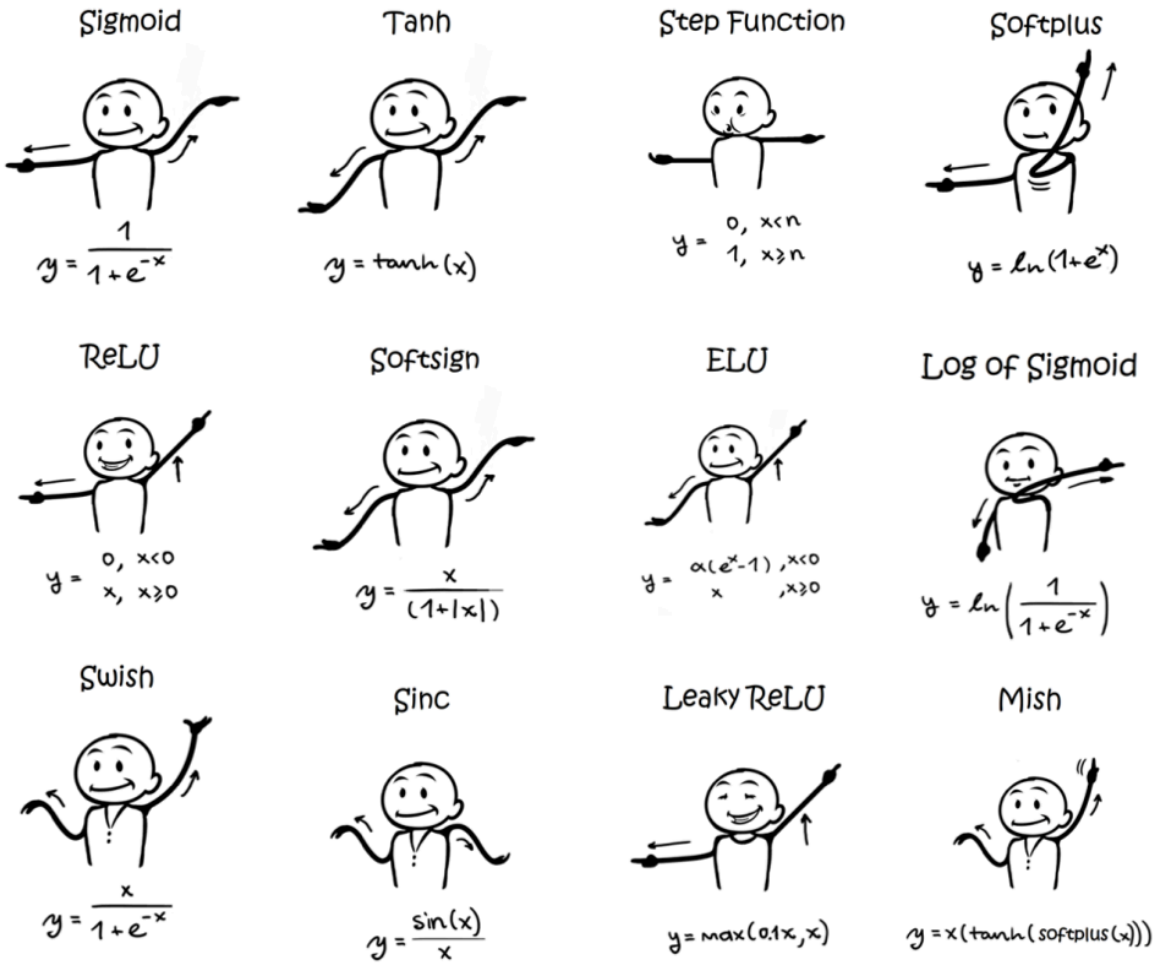

Figure 2: Một số hàm kích hoạt thông dụng. Nguồn: link

2. Import thư viện cần thiết

Để thực hiện bài thực hành này, ta sử dụng thư viện như `numpy`.

```
import numpy as np
```

3. Chuẩn bị dữ liệu

Sinh viên tải tập dữ liệu tại đây.

Dữ liệu cho bài toán phân loại được lưu dưới dạng `.npy` (Python NumPy Array). Ta sử dụng `np.load` để tải dữ liệu từ file này, thu được một dictionary với hai phần: X là ma trận đặc trưng có kích thước $(300, 2)$. Tức là tập dữ liệu có 300 mẫu, mỗi mẫu có 2 đặc trưng. y là vector chứa 300 nhãn tương ứng.

```
data_path = 'data/nonLinear_data.npy'
data = np.load(data_path, allow_pickle=True).item()
```

```
X, y = data['X'], data['labels']
print (X.shape, y.shape )
```

Đối với một tác vụ học từ dữ liệu như ANN, ta cần sử dụng dữ liệu để huấn luyện và một phần dữ liệu để kiểm tra. Vì vậy, ta sẽ thực hiện chia dữ liệu thành 80% huấn luyện và 20% kiểm tra. Đoạn code dưới đây thực hiện công việc chia dữ liệu

```
# Split dataset into training and testing sets (80% train, 20% test)
split_ratio = 0.8
random_state = 42
np.random.seed(random_state)
indices = np.random.permutation(len(X))
split_index = int(len(X) * split_ratio)
train_indices, test_indices = indices[:split_index], indices[split_index:]

X_train, X_test = X[train_indices], X[test_indices]
y_train, y_test = y[train_indices], y[test_indices]
```

Quan sát 5 dòng đầu tiên của tập dữ liệu huấn luyện và tập dữ liệu kiểm tra, có thể thấy tập dữ liệu gồm 2 đặc trưng bao gồm 3 lớp, được gán nhãn lần lượt là $0, 1, 2$.

```
First 5 rows of X_train:
[[ 0.02995783 -0.00456091]
 [-0.59477177 -0.30114945]
 [ 0.04739604  0.52310977]
 [ 0.04447445  0.07928736]
 [ 0.13882627 -0.30304847]]
First 5 rows of y_train:
[2 2 1 0 2]
First 5 rows of X_test:
[[-0.73735891 -0.00467537]
 [ 0.00942238 -0.00363995]
 [ 0.17036411  0.5921412 ]
 [ 0.4165839  -0.12293116]
 [ 0.10181868 -0.13827416]]
First 5 rows of y_test:
[2 2 1 0 2]
```

Figure 3: Một số dòng của tập dữ liệu huấn luyện và kiểm thử

4. Xây dựng mạng MLP

Để giải quyết bài toán phân loại phi tuyến với dữ liệu có 2 đặc trưng và 3 lớp, bước đầu chúng ta sẽ xây dựng một mạng MLP gồm 1 lớp ẩn, với hàm kích hoạt `ReLU` sau mỗi lớp. Cấu trúc này giúp mô hình học các đặc trưng phi tuyến trong dữ liệu.

Để xây dựng mạng nơ-ron nhân tạo này ta sử dụng Class `NeuralNetwork`. Class này cung cấp các phương thức để khởi tạo mô hình, thực hiện lan truyền thuận (forward propagation), lan truyền ngược (backpropagation), huấn luyện mô hình và dự đoán kết quả.

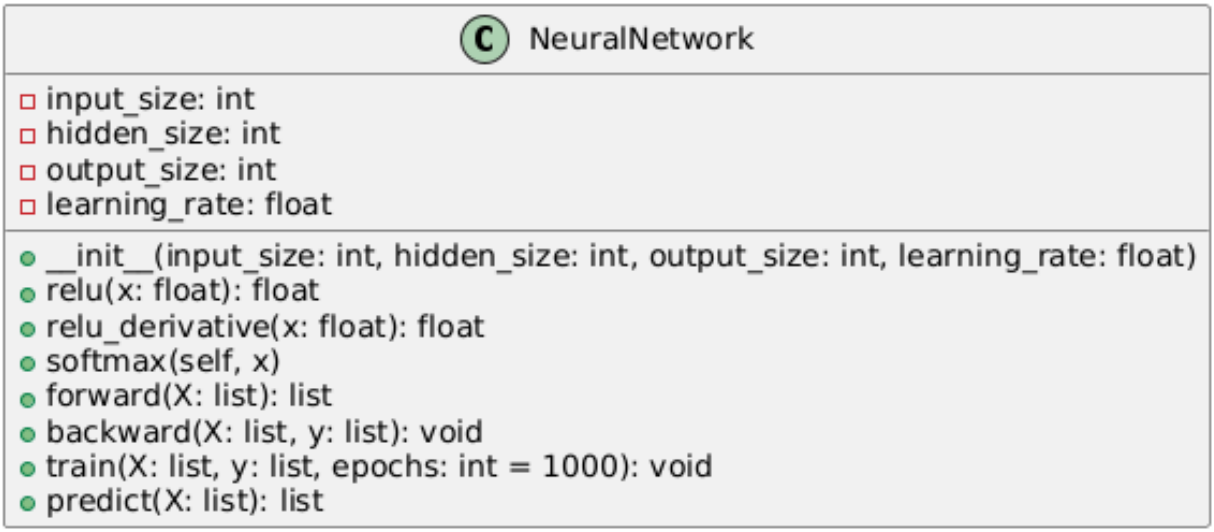

Figure 4: Sơ đồ UML mô phỏng class NeuralNetwork

4.1 Phương thức `__init__`

Phương thức này có nhiệm vụ khởi tạo một mạng nơ-ron nhân tạo đơn giản với một lớp ẩn.

```
def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        """
        Initialize the neural network with random weights and biases.
        :param input_size: Number of input features
        :param hidden_size: Number of neurons in the hidden layer
        :param output_size: Number of output neurons (equal to the number of classes)
        :param learning_rate: Learning rate for weight updates
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.W1 = np.random.randn(self.input_size, self.hidden_size)
        self.b1 = np.zeros((1, self.hidden_size))
        self.W2 = np.random.randn(self.hidden_size, self.output_size)
        self.b2 = np.zeros((1, self.output_size))
```

4.2 Hàm kích hoạt `ReLU`

Biết rằng hàm kích hoạt `ReLU` có công thức như sau:

$$\text{ReLU}(x) = \begin{cases} x, & \text{nếu } x > 0 \\ 0, & \text{nếu } x \leq 0 \end{cases}$$

Q1: Hãy hoàn thành đoạn code hàm kích hoạt `ReLU` dưới đây

```
def relu(self, x):
    """ReLU activation function."""
    # code here
```

Lấy đạo hàm của hàm `ReLU` theo x, ta có:

$$\text{ReLU}'(x) = \begin{cases} 1, & \text{nếu } x > 0 \\ 0, & \text{nếu } x \leq 0 \end{cases}$$

Q2: Hãy hoàn thành đoạn code tính đạo hàm của hàm`ReLU` dưới đây

```
def relu_derivative(self, x):
    """Derivative of ReLU activation function."""
    # code here
```

4.5 Hàm kích hoạt `softmax`

Ta đang thực hiện bài toán phân loại trên tập dữ liệu có 2 đặc trưng và 3 lớp, vì vậy cần đầu ra là xác suất thuộc về mỗi lớp.

Ta sử dụng hàm `softmax` để đảm bảo tổng xác suất của tất cả lớp bằng 1:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Q3: Hãy hoàn thành đoạn code hàm kích hoạt `Softmax` dưới đây

```
def softmax(self, x):
    """Softmax activation function for multi-class classification."""
    # code here
```

4.4 Hàm lan truyền thuận

Hàm này tính toán đầu ra của mô hình dựa trên đầu vào X.

```
    def forward(self, X):
        """
        Perform forward propagation to compute network output.
        :param X: Input data
        :return: Softmax activated output values
        """
        self.z1 = np.dot(X, self.W1) + self.b1  # Compute input to hidden layer
        self.a1 = self.relu(self.z1)  # Use ReLU activation for hidden layer
        self.z2 = np.dot(self.a1, self.W2) + self.b2  # Compute input to output layer
        return self.softmax(self.z2)  # Use softmax activation for output layer
```

4.5 Hàm lan truyền ngược

Phương thức `backward` trong lớp `NeuralNetwork` thực hiện lan truyền ngược để cập nhật trọng số và độ lệch bằng cách sử dụng `Gradient Descent`.

```
def backward(self, X, y):
        """
        Perform backpropagation to update weights and biases.
        :param X: Input data
        :param y: True labels (one-hot encoded)
        """
        m = X.shape[0]  # Number of training examples

        # Compute output error
        output = self.forward(X)
        error = output - y
        dW2 = np.dot(self.a1.T, error) / m
        db2 = np.sum(error, axis=0, keepdims=True) / m

        # Compute hidden layer error
        hidden_error = np.dot(error, self.W2.T) * self.relu_derivative(self.z1)
        dW1 = np.dot(X.T, hidden_error) / m
        db1 = np.sum(hidden_error, axis=0, keepdims=True) / m

        # Update weights and biases using gradient descent
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
```

4.6 Hàm huấn luyện

Hàm huấn luyện trong mạng nơ-ron có nhiệm vụ điều chỉnh trọng số (weights) và bias của mô hình để tối ưu hóa dự đoán thông qua quá trình học từ dữ liệu.

Hàm huấn luyện thực hiện các bước sau:

- Nhận dữ liệu đầu vào (X) và nhãn (y);
- Lan truyền thuận;
- Tính toán hàm mất mát;
- Lan truyền ngược;
- Cập nhật trọng số;
- Lặp lại các bước huấn luyện trên toàn bộ tập dữ liệu nhiều lần để mô hình học dần dần trên dữ liệu và cải thiện các tham số.

```
def train(self, X, y, epochs=100):
    """
    Train the neural network using full batch gradient descent.
    :param X: Training input data
    :param y: Training labels (one-hot encoded)
    :param epochs: Number of iterations
    """
```

```
    for epoch in range(epochs):
        # Forward propagation trên toàn bộ tập dữ liệu
        output = self.forward(X)

        # Backward propagation và cập nhật trọng số
        self.backward(X, y)

        # Chỉ in loss sau mỗi 10 epochs
        if epoch % 10 == 0:
            loss = -np.mean(np.sum(y * np.log(output + 1e-8), axis=1))  # Cross-entropy loss
            print(f'Epoch {epoch}, Loss: {loss:.4f}')
```

4.7 Hàm dự đoán

Hàm `predict(self, X)` dùng để dự đoán nhãn lớp sau khi mạng neural đã được huấn luyện.

```
def predict(self, X):
        """
        Make predictions using the trained neural network.
        :param X: Input data
        :return: Predicted class labels
        """
        output = self.forward(X)
        return np.argmax(output, axis=1)
```

4.8 Thực thi chương trình

4.8.1 One-hot Encoding

Nhãn của dữ liệu chúng ta cần phân loại ở dạng số, tuy nhiên các mô hình ANN chỉ chấp nhận nhãn ở dạng nhị phân. Vì vậy, ta áp dụng phương pháp one-hot encoding để chuyển đổi nhãn dạng số thành vector nhị phân có độ dài bằng số lượng lớp.

Mỗi nhãn sẽ được biểu diễn bằng một vector chỉ có một phần tử bằng 1 (ứng với nhãn đó) và các phần tử còn lại là 0.

Ví dụ minh hoạ: Giả sử tập dữ liệu có 3 lớp: $y = [0, 2, 1, 2]$

Áp dụng phương pháp one-hot encoding, ta có kết quả như sau:

$$\begin{bmatrix} [\#\text{ Lớp } 0 \rightarrow [1, 0, 0]] \\ [\#\text{ Lớp } 2 \rightarrow [0, 0, 1]] \\ [\#\text{ Lớp } 1 \rightarrow [0, 1, 0]] \\ [\#\text{ Lớp } 2 \rightarrow [0, 0, 1]] \end{bmatrix}$$

```
# Convert labels to one-hot encoding
y_one_hot = np.eye(len(set(y.flatten())))[y.flatten()]
```

4.8.2 Khởi tạo và huấn luyện mô hình ANN

```
nn = NeuralNetwork(input_size=X.shape[1],
                   hidden_size=4,
                   output_size=y_one_hot.shape[1],
                   learning_rate=0.1)

nn.train(X_train,
         y_one_hot[train_indices],
         epochs=100)
```

4.8.3 Dự đoán trên tập thử nghiệm

```
# Make predictions on test data
y_pred = nn.predict(X_test)
print('Predictions on test set:', y_pred)
```

5. Thực hành

Sinh viên đọc hiểu và giải thích code. Thực thi chương trình, nhận xét kết quả.

Chương trình còn một số vấn đề cần tối ưu, hãy thực hiện các bài tập sau để xây dựng chương trình hoàn chỉnh hơn:

Q4: Thay đổi các tham số của mô hình như số lượng nơ-ron trong lớp ẩn, tốc độ học và nhận xét kết quả.

Q5: Tăng số lớp ẩn của mô hình

- Hiện tại mô hình có một lớp ẩn, hãy bổ sung thêm một lớp ẩn nữa để nghiên cứu tác động của độ sâu của mạng tới kết quả huấn luyện.
- Cập nhật hàm `forward(self, X)` để xử lý thêm một lớp
- Điều chỉnh hàm `backward(self, X, y)` để cập nhật trọng số của lớp mới.

Q6: Thay đổi hàm kích hoạt

Thay đổi hàm kích hoạt của lớp ẩn từ ReLU sang một trong các hàm sau, sau đó so sánh hiệu suất huấn luyện. Thử nghiệm một số hàm kích hoạt thay thế sau đây:

- `Sigmoid`
- `Tanh`
- `Leaky ReLU`

Sau khi thay đổi, hãy trả lời các câu hỏi:

- Hàm kích hoạt nào giúp mô hình hội tụ nhanh hơn?
- Hàm nào có vấn đề với vanishing gradient?
- Khi nào nên dùng từng loại?

5.1 Huấn luyện toàn bộ dữ liệu vs huấn luyện theo mini-batch

Trong code mẫu, chương trình được huấn luyện trên toàn bộ tập dữ liệu, khi so sánh với huấn luyện theo mini-batch, huấn luyện toàn bộ dữ liệu thể hiện một số nhược điểm.

Phương pháp	Batch Gradient Descent	Mini-Batch Gradient Descent
Cách thực hiện	Dùng toàn bộ dữ liệu trong mỗi lần cập nhật	Chia dữ liệu thành từng lô nhỏ (mini-batch) để cập nhật
Tốc độ cập nhật trọng số	Cập nhật sau khi quét toàn bộ dữ liệu	Cập nhật sau từng mini-batch
Hiệu suất tính toán	Có thể chậm nếu dữ liệu lớn	Cân bằng giữa tốc độ và độ chính xác
Ổn định của gradient	Ổn định nhưng chậm hội tụ	Hội tụ nhanh hơn nhưng có độ nhiễu
Yêu cầu bộ nhớ	Cao (vì phải tính toán trên toàn bộ dữ liệu)	Giảm tải bộ nhớ do chỉ xử lý từng phần dữ liệu nhỏ
Thích hợp cho	Dữ liệu nhỏ	Dữ liệu lớn, khi không đủ RAM để lưu toàn bộ batch

Q7: Chia dữ liệu thành nhiều `mini-batch` để huấn luyện thay vì huấn luyện toàn bộ tập dữ liệu.

Hãy sửa đổi hàm `train()` để:

- Chia dữ liệu thành mini-batch.
- Duyệt qua từng batch để cập nhật trọng số.
- Tính loss sau mỗi vài epoch để theo dõi quá trình huấn luyện.
- Thử nghiệm các giá trị batch_size khác nhau (ví dụ: $32, 64, 128$)