BỘ THÔNG TIN VÀ TRUYỀN THÔNG
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN I

BÀI BÁO CÁO

BÀI TẬP LỚN: PYTHON

Giảng viên: KIM NGỌC BÁCH

Nhóm sinh viên thực hiện: Nguyễn Việt Anh - B23DCCE009

Lê Huy Hoàng - B23DCVT171

Hà Nội, Tháng 5/2025

Mục lục

1 Lời mở đầu **3**

2 Chuẩn bị dữ liệu và Xây dựng mô hình **3**

2.1 Tổng quan về mã code và giải thích ý tưởng chung 3

2.2 Thiết lập môi trường và thư viện 4

2.3 Chuẩn bị dữ liệu CIFAR-10 5

2.3.1 Tải và tiền xử lý dữ liệu 5

2.3.2 Phân chia dữ liệu và tạo DataLoaders 6

2.4 Xây dựng mô hình Mạng Perceptron Đa Lớp (MLP) 7

2.4.1 Định nghĩa kiến trúc mô hình 7

2.4.2 Vai trò của Dropout 8

3 Huấn luyện và Đánh giá mô hình **8**

3.1 Hàm huấn luyện mô hình (train_model) 8

3.1.1 Mục đích 8

3.1.2 Tham số đầu vào 8

3.1.3 Cơ chế hoạt động 8

3.1.4 Tối ưu hóa và Regularization 12

3.1.5 Kỹ thuật Early Stopping 12

3.1.6 Kết quả trả về 12

3.2 Hàm đánh giá mô hình (evaluate_model) 12

3.2.1 Mục đích 12

3.2.2 Tham số đầu vào 12

3.2.3 Cơ chế hoạt động 12

3.2.4 Kết quả trả về 14

4 Trực quan hóa kết quả **14**

4.1 Hàm vẽ biểu đồ Learning Curves (plot_learning_curves) 14

4.1.1 Mục đích 14

4.1.2 Tham số đầu vào 14

4.1.3 Cơ chế hoạt động 14

4.2 Hàm vẽ Confusion Matrix (plot_confusion_matrix) 15

4.2.1 Mục đích 15

4.2.2 Tham số đầu vào 15

4.2.3 Cơ chế hoạt động 16

5 Thực thi và Kết quả **16**

5.1 Khởi tạo và huấn luyện mô hình MLP 16

5.2 Đánh giá mô hình trên các tập dữ liệu 17

5.3 Trực quan hóa Learning Curves 17
5.4 Trực quan hóa Confusion Matrices 18
5.5 Phân tích kết quả 21

6 Kết luận **22**

1 Lời mở đầu

Thị giác máy tính là một lĩnh vực then chốt trong trí tuệ nhân tạo, và phân loại ảnh đóng vai trò là một trong những nhiệm vụ nền tảng. Bộ dữ liệu CIFAR-10, với 60.000 ảnh màu 32x32 pixel thuộc 10 lớp đối tượng, thường được dùng làm tiêu chuẩn để kiểm tra và so sánh hiệu năng của các mô hình học máy.

Bài tập này tập trung vào việc thiết kế, huấn luyện và đánh giá một Mạng Perceptron Đa Lớp (MLP) để phân loại ảnh trên bộ dữ liệu CIFAR-10. Toàn bộ quá trình được thực hiện bằng ngôn ngữ lập trình Python và thư viện học sâu PyTorch. Các giai đoạn chính bao gồm: chuẩn bị dữ liệu (kết hợp kỹ thuật tăng cường dữ liệu - Data Augmentation), xây dựng kiến trúc mạng (sử dụng Dropout để hạn chế hiện tượng overfitting), huấn luyện mô hình (áp dụng Weight Decay và Early Stopping để tối ưu hóa hiệu suất và thời gian huấn luyện), đánh giá hiệu năng, và trực quan hóa đường cong học (learning curves) cũng như ma trận nhầm lẫn (Confusion Matrix) để có cái nhìn sâu sắc về khả năng của mô hình.

Mục tiêu của bài tập là xây dựng một quy trình hoàn chỉnh cho bài toán phân loại ảnh, đồng thời ứng dụng các kỹ thuật phổ biến nhằm nâng cao độ chính xác và khả năng tổng quát hóa của mô hình học sâu.

2 Chuẩn bị dữ liệu và Xây dựng mô hình

2.1 Tổng quan về mã code và giải thích ý tưởng chung

Đoạn mã này thực hiện một quy trình học máy hoàn chỉnh để phân loại ảnh từ bộ dữ liệu CIFAR-10 bằng Mạng Perceptron Đa Lớp (MLP) với thư viện PyTorch. Các bước chính bao gồm:

1. **Chuẩn bị dữ liệu** : Tải CIFAR-10, áp dụng các phép biến đổi tăng cường dữ liệu (Data Augmentation) cho tập huấn luyện nhằm tăng tính đa dạng và khả năng khái quát hóa. Dữ liệu sau đó được chuẩn hóa và nạp vào các DataLoader cho quá trình huấn luyện (train), kiểm định (validation) và kiểm thử (test).
2. **Xây dựng mô hình MLP**: Định nghĩa một kiến trúc mạng nơ-ron nhân tạo với các lớp kết nối đầy đủ (Linear), hàm kích hoạt ReLU, và các lớp Dropout để giảm overfitting.
3. **Huấn luyện mô hình**: Phát triển một hàm huấn luyện cho phép theo dõi loss và accuracy trên ba tập dữ liệu (train, validation, test) qua từng epoch. Hàm này tích hợp Weight Decay trong bộ tối ưu Adam và kỹ thuật Early Stopping dựa trên hiệu suất của tập validation để chọn ra mô hình tốt nhất.
4. **Đánh giá mô hình**: Xây dựng hàm đánh giá để tính toán loss và accuracy cuối cùng trên các tập dữ liệu, đồng thời thu thập các nhãn dự đoán và nhãn thực tế để vẽ ma trận nhầm lẫn.
5. **Trực quan hóa**: Các hàm được viết để vẽ biểu đồ learning curves (loss và accuracy theo epoch) và ma trận nhầm lẫn, giúp đánh giá trực quan hiệu suất của mô hình.

6. **Thực thi**: Toàn bộ quy trình được thực thi để huấn luyện, đánh giá mô hình MLP, và lưu lại các biểu đồ kết quả.

Mục đích là xây dựng một mô hình phân loại ảnh cơ bản nhưng hiệu quả, áp dụng các phương pháp tốt nhất trong học sâu như tăng cường dữ liệu, regularization (Dropout, Weight Decay), và early stopping.

2.2 Thiết lập môi trường và thư viện

Đầu tiên, chương trình nhập các thư viện cần thiết cho việc xây dựng mô hình, xử lý dữ liệu và trực quan hóa.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Set up devices
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Giải thích:

- `torch`, `torch.nn`, `torch.optim`: Các module cốt lõi của PyTorch, cung cấp cấu trúc tensor, các lớp xây dựng mạng nơ-ron (ví dụ: `nn.Linear`, `nn.ReLU`, `nn.Dropout`, `nn.CrossEntropyLoss`), và các thuật toán tối ưu hóa (ví dụ: `optim.Adam`).
- `torchvision`, `torchvision.transforms`: Cung cấp các bộ dữ liệu chuẩn (như CIFAR-10) và các phép biến đổi ảnh.
- `matplotlib.pyplot`: Thư viện để vẽ đồ thị trong Python.
- `numpy`: Thư viện cho tính toán số học.
- `sklearn.metrics.confusion_matrix`: Hàm từ Scikit-learn để tính toán ma trận nhầm lẫn.
- `seaborn`: Thư viện trực quan hóa dữ liệu dựa trên Matplotlib, giúp tạo các biểu đồ đẹp mắt hơn.
- `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`: Dòng này xác định thiết bị tính toán. Nếu có GPU (CUDA) thì sử dụng 'cuda' để tăng tốc, ngược lại dùng 'cpu'.

2.3 Chuẩn bị dữ liệu CIFAR-10

2.3.1 Tải và tiền xử lý dữ liệu

Bộ dữ liệu CIFAR-10 được tải về và xử lý sơ bộ bằng các phép biến đổi. Kỹ thuật Data Augmentation được áp dụng cho tập huấn luyện.

```
# Prepare data with Data Augmentation
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
   download=True, transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
   download=True, transform=transform_test)
```

Giải thích:

- `transforms.Compose([...])`: Kết hợp nhiều phép biến đổi thành một chuỗi xử lý.
- `transform_train`: Chuỗi biến đổi cho dữ liệu huấn luyện.
- `transforms.RandomHorizontalFlip()`: Lật ảnh ngẫu nhiên theo chiều ngang.
- `transforms.RandomRotation(10)`: Xoay ảnh ngẫu nhiên một góc trong khoảng [-10, 10] độ.
- `transforms.ToTensor()`: Chuyển ảnh thành PyTorch Tensor và chuẩn hóa giá trị pixel về khoảng [0.0, 1.0].
- `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Chuẩn hóa giá trị pixel của tensor về khoảng [-1.0, 1.0] bằng cách trừ đi trung bình (0.5) và chia cho độ lệch chuẩn (0.5) cho mỗi kênh màu.
- `transform_test`: Chuỗi biến đổi cho dữ liệu kiểm thử và kiểm định, không bao gồm Data Augmentation.
- `torchvision.datasets.CIFAR10(...)`: Tải bộ dữ liệu CIFAR-10.

- `root='./data'`: Thư mục lưu trữ.
- `train=True/False`: Chọn tập huấn luyện hoặc kiểm thử.
- `download=True`: Tự động tải nếu chưa có.
- `transform=...`: Áp dụng chuỗi biến đổi.

2.3.2 Phân chia dữ liệu và tạo DataLoaders

Tập huấn luyện ban đầu được chia thành tập huấn luyện mới và tập kiểm định. Các DataLoader được tạo để cung cấp dữ liệu theo batch.

```
# Split the train set into train and validation
train_size = int(0.8 * len(trainset))
val_size = len(trainset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(trainset, [
    train_size, val_size])

trainloader = torch.utils.data.DataLoader(train_dataset, batch_size
    =100, shuffle=True)
valloader = torch.utils.data.DataLoader(val_dataset, batch_size=100,
    shuffle=False)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,
    shuffle=False)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', '
    horse', 'ship', 'truck')
```

Giải thích:

- `torch.utils.data.random_split(...)`: Chia `trainset` thành `train_dataset` (80%) và `val_dataset` (20%).
- `torch.utils.data.DataLoader(...)`: Tạo trình tải dữ liệu.
- `batch_size=100`: Số mẫu trong mỗi batch.
- `shuffle=True` (cho `trainloader`): Xáo trộn dữ liệu huấn luyện mỗi epoch.
- `shuffle=False` (cho `valloader`, `testloader`): Không xáo trộn khi đánh giá.
- `classes`: Tuple chứa tên 10 lớp của CIFAR-10.

2.4 Xây dựng mô hình Mạng Perceptron Đa Lớp (MLP)

2.4.1 Định nghĩa kiến trúc mô hình

Lớp MLP kế thừa từ `nn.Module` định nghĩa kiến trúc mạng.

```
# Building MLP model with Dropout
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3 * 32 * 32, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x
```

Giải thích:

- `self.flatten = nn.Flatten()`: Làm phẳng tensor đầu vào.
- `self.fc1 = nn.Linear(3 * 32 * 32, 512)`: Lớp kết nối đầy đủ thứ nhất (3072 đầu vào, 512 đầu ra).
- `self.fc2 = nn.Linear(512, 256)`: Lớp kết nối đầy đủ thứ hai (512 đầu vào, 256 đầu ra).
- `self.fc3 = nn.Linear(256, 10)`: Lớp kết nối đầy đủ cuối cùng (lớp output, 256 đầu vào, 10 đầu ra - tương ứng 10 lớp).
- `self.relu = nn.ReLU()`: Hàm kích hoạt ReLU.
- `self.dropout = nn.Dropout(0.5)`: Lớp Dropout với tỷ lệ bỏ qua 0.5.
- `forward(self, x)`: Định nghĩa quá trình truyền thẳng dữ liệu qua mạng, bao gồm làm phẳng, qua các lớp Linear, ReLU và Dropout.

2.4.2 Vai trò của Dropout

Dropout là một kỹ thuật regularization giúp giảm overfitting. Trong quá trình huấn luyện, nó ngẫu nhiên "tắt" (đặt giá trị bằng 0) một tỷ lệ nơ-ron nhất định (0.5 trong trường hợp này). Điều này buộc mạng phải học các đặc trưng mạnh mẽ hơn và không quá phụ thuộc vào một số ít nơ-ron.

3 Huấn luyện và Đánh giá mô hình

3.1 Hàm huấn luyện mô hình (train_model)

Hàm này điều khiển toàn bộ quá trình huấn luyện và đánh giá định kỳ.

3.1.1 Mục đích

- Huấn luyện mô hình MLP trên `trainloader`.
- Đánh giá hiệu suất trên `valloader` và `testloader` sau mỗi epoch.
- Lưu trữ lịch sử loss và accuracy.
- Thực hiện Early Stopping để tránh overfitting và tiết kiệm thời gian.
- Trả về mô hình với trọng số tốt nhất và lịch sử huấn luyện.

3.1.2 Tham số đầu vào

```
def train_model(model, trainloader, valloader, testloader, epochs=20,
    patience=3):
```

- `model`: Mô hình MLP cần huấn luyện.
- `trainloader`, `valloader`, `testloader`: Các DataLoader.
- `epochs=20`: Số epoch huấn luyện tối đa.
- `patience=3`: Số epoch chờ đợi sự cải thiện trên validation loss trước khi dừng sớm.

3.1.3 Cơ chế hoạt động

```
def train_model(model, trainloader, valloader, testloader, epochs=20,
    patience=3):
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay
        =1e-4)

    train_losses, val_losses, test_losses = [], [], []
```

```
    train_accs, val_accs, test_accs = [], [], []

    best_val_loss = float('inf')
    epochs_no_improve = 0
    best_model_state = None

    for epoch in range(epochs):
        # Training
        model.train()
        running_loss, correct, total = 0.0, 0, 0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_losses.append(running_loss / len(trainloader))
        train_accs.append(100 * correct / total)

        # Validation
        model.eval()
        val_loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
            for inputs, labels in valloader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_losses.append(val_loss / len(valloader))
        val_accs.append(100 * correct / total)
```

```
48
49          # Test
50          test_loss, correct, total = 0.0, 0, 0
51          with torch.no_grad():
52              for inputs, labels in testloader:
53                  inputs, labels = inputs.to(device), labels.to(device)
54                  outputs = model(inputs)
55                  loss = criterion(outputs, labels)
56                  test_loss += loss.item()
57                  _, predicted = torch.max(outputs.data, 1)
58                  total += labels.size(0)
59                  correct += (predicted == labels).sum().item()
60
61          test_losses.append(test_loss / len(testloader))
62          test_accs.append(100 * correct / total)
63
64          print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_losses
                [-1]:.4f}, Train Acc: {train_accs[-1]:.2f}%, '
65                f'Val Loss: {val_losses[-1]:.4f}, Val Acc: {val_accs
                    [-1]:.2f}%, '
66                f'Test Loss: {test_losses[-1]:.4f}, Test Acc: {test_accs
                    [-1]:.2f}%')
67
68          # Early Stopping
69          if val_losses[-1] < best_val_loss:
70              best_val_loss = val_losses[-1]
71              epochs_no_improve = 0
72              best_model_state = model.state_dict()
73          else:
74              epochs_no_improve += 1
75
76          if epochs_no_improve >= patience:
77              print(f'Early stopping at epoch {epoch+1} due to no
                    improvement in Val Loss.')
78              model.load_state_dict(best_model_state)
79              break
80
81      return train_losses, val_losses, test_losses, train_accs, val_accs
            , test_accs
```

1. **Khởi tạo**:

- Chuyển mô hình lên `device` (GPU nếu có, CPU nếu không).
- Định nghĩa hàm mất mát `nn.CrossEntropyLoss()`.
- Khởi tạo thuật toán tối ưu `optim.Adam` với tốc độ học `lr=0.0001` và `weight_decay=1e-4` (L2 regularization).
- Khởi tạo các danh sách để lưu trữ loss/accuracy và các biến cho Early Stopping.

2. **Vòng lặp Epoch**: Lặp qua số epochs được chỉ định.
   - **Huấn luyện** (`model.train()`):
     - Đặt mô hình ở chế độ huấn luyện.
     - Duyệt qua từng batch trong `trainloader`.
     - Chuyển dữ liệu batch lên `device`.
     - `optimizer.zero_grad()`: Xóa gradient của các tham số.
     - `outputs = model(inputs)`: Thực hiện forward pass.
     - `loss = criterion(outputs, labels)`: Tính loss.
     - `loss.backward()`: Thực hiện backward pass để tính gradient.
     - `optimizer.step()`: Cập nhật trọng số.
     - Tính toán và lưu trữ loss, accuracy cho tập huấn luyện.
   - **Kiểm định** (`model.eval()`):
     - Đặt mô hình ở chế độ đánh giá (tắt Dropout).
     - `with torch.no_grad()`: Tắt việc tính gradient.
     - Duyệt qua `valloader`, tính loss và accuracy mà không cập nhật gradient.
     - Lưu trữ loss, accuracy cho tập kiểm định.
   - **Kiểm thử** (`model.eval()`): Tương tự như kiểm định, nhưng thực hiện trên `testloader`.
   - **In kết quả**: Hiển thị loss và accuracy của ba tập (train, validation, test) sau mỗi epoch.
   - **Early Stopping**:
     - Nếu validation loss của epoch hiện tại nhỏ hơn `best_val_loss` (loss tốt nhất từng ghi nhận), cập nhật `best_val_loss`, reset `epochs_no_improve`, và lưu lại trọng số của mô hình hiện tại.
     - Nếu không, tăng `epochs_no_improve`.

- Nếu `epochs_no_improve` đạt đến `patience`, dừng huấn luyện sớm và tải lại trọng số của mô hình tốt nhất.

3.1.4 Tối ưu hóa và Regularization

- **Optimizer**: `optim.Adam` được sử dụng với `lr=0.0001` và `weight_decay=1e-4` (L2 regularization) để giúp mô hình hội tụ ổn định và giảm overfitting.
- **Dropout**: Được tích hợp trong kiến trúc mô hình MLP.

3.1.5 Kỹ thuật Early Stopping

Theo dõi hiệu suất trên tập validation. Nếu validation loss không giảm sau một số epoch (`patience`), quá trình huấn luyện dừng lại và mô hình với validation loss tốt nhất được chọn.

3.1.6 Kết quả trả về

Hàm trả về 6 list chứa lịch sử loss và accuracy của ba tập dữ liệu (train, validation, test) qua các epoch.

3.2 Hàm đánh giá mô hình (evaluate_model)

Đánh giá hiệu suất cuối cùng của mô hình và thu thập dự đoán.

3.2.1 Mục đích

- Đánh giá mô hình trên một DataLoader cụ thể.
- Tính loss trung bình và accuracy tổng thể.
- Thu thập danh sách nhãn thực tế (`y_true`) và nhãn dự đoán (`y_pred`).

3.2.2 Tham số đầu vào

```
def evaluate_model(model, dataloader, set_name="Test"):
```

- `model`: Mô hình đã huấn luyện.
- `dataloader`: DataLoader của tập dữ liệu cần đánh giá.
- `set_name="Test"`: Tên tập dữ liệu (mặc định là "Test").

3.2.3 Cơ chế hoạt động

```
def evaluate_model(model, dataloader, set_name="Test"):
    model = model.to(device)
    model.eval()
    y_true, y_pred = [], []
```

```
    loss, correct, total = 0.0, 0, 0
    criterion = nn.CrossEntropyLoss()

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss += criterion(outputs, labels).item()
            _, predicted = torch.max(outputs.data, 1)
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    avg_loss = loss / len(dataloader)
    accuracy = 100 * correct / total
    print(f"{set_name} Loss: {avg_loss:.4f}, {set_name} Accuracy: {
       accuracy:.2f}%")
    return y_true, y_pred, avg_loss, accuracy
```

1. Chuyển mô hình lên `device` và đặt ở chế độ `model.eval()`.
2. Khởi tạo các list `y_true`, `y_pred` và các biến để tính toán `loss`, `correct`, `total`.
3. Sử dụng `nn.CrossEntropyLoss()` làm hàm mất mát.
4. Trong khối `with torch.no_grad()`:
   - Duyệt qua từng batch trong `dataloader`.
   - Thực hiện forward pass để lấy `outputs`.
   - Tính và cộng dồn `loss`.
   - Lấy nhãn `predicted` từ `outputs`.
   - Nối các nhãn thực tế (`labels`) và nhãn dự đoán (`predicted`) vào các list `y_true` và `y_pred` tương ứng.
   - Cập nhật `total` và `correct`.
5. Tính `avg_loss` (loss trung bình) và `accuracy` (độ chính xác).
6. In ra kết quả `loss` và `accuracy` cho `set_name`.

3.2.4 Kết quả trả về

- `y_true`: List các nhãn thực tế.
- `y_pred`: List các nhãn dự đoán.
- `avg_loss`: Loss trung bình.
- `accuracy`: Độ chính xác (%).

4 Trực quan hóa kết quả

Trực quan hóa giúp hiểu rõ hơn quá trình huấn luyện và hiệu suất mô hình.

4.1 Hàm vẽ biểu đồ Learning Curves (plot_learning_curves)

4.1.1 Mục đích

Vẽ biểu đồ thay đổi của loss và accuracy trên các tập train, validation, test qua từng epoch. Giúp đánh giá sự hội tụ và phát hiện overfitting/underfitting.

4.1.2 Tham số đầu vào

```
def plot_learning_curves(train_losses, val_losses, test_losses,
    train_accs, val_accs, test_accs, title):
```

- `train_losses`, `val_losses`, `test_losses`: List loss của các tập qua từng epoch.
- `train_accs`, `val_accs`, `test_accs`: List accuracy của các tập qua từng epoch.
- `title`: Tiêu đề chung cho biểu đồ.

4.1.3 Cơ chế hoạt động

```
def plot_learning_curves(train_losses, val_losses, test_losses,
    train_accs, val_accs, test_accs, title):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss', color='blue')
    plt.plot(val_losses, label='Validation Loss', color='orange')
    plt.plot(test_losses, label='Test Loss', color='green')
    plt.title(f'{title} - Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
```

```
12
13      plt.subplot(1, 2, 2)
14      plt.plot(train_accs, label='Train Accuracy', color='blue')
15      plt.plot(val_accs, label='Validation Accuracy', color='orange')
16      plt.plot(test_accs, label='Test Accuracy', color='green')
17      plt.title(f'{title} - Accuracy')
18      plt.xlabel('Epoch')
19      plt.ylabel('Accuracy (%)')
20      plt.legend()
21
22      plt.tight_layout()
23      plt.savefig('mlp_learning_curves.png')
24      plt.close()
```

Hàm sử dụng thư viện `matplotlib.pyplot` để:

1. Tạo một figure lớn (`plt.figure`) chứa hai subplot (một cho loss, một cho accuracy).
2. **Subplot Loss**: Vẽ các đường cong loss của tập train, validation và test theo số epoch, sử dụng các màu và nhãn khác nhau.
3. **Subplot Accuracy**: Vẽ các đường cong accuracy tương tự.
4. Đặt tiêu đề, nhãn cho các trục và chú thích (`legend`) cho từng biểu đồ.
5. `plt.tight_layout()`: Tự động điều chỉnh khoảng cách giữa các subplot.
6. `plt.savefig('mlp_learning_curves.png')`: Lưu biểu đồ kết quả vào file ảnh.
7. `plt.close()`: Đóng figure.

4.2 Hàm vẽ Confusion Matrix (plot_confusion_matrix)

4.2.1 Mục đích

Vẽ ma trận nhầm lẫn để đánh giá chi tiết hiệu suất phân loại của mô hình, cho thấy số lượng dự đoán đúng và sai cho từng lớp.

4.2.2 Tham số đầu vào

```
1 def plot_confusion_matrix(y_true, y_pred, title, filename):
```

- `y_true`: List các nhãn thực tế.
- `y_pred`: List các nhãn dự đoán.
- `title`: Tiêu đề cho ma trận.

- `filename`: Tên file để lưu ảnh.

4.2.3 Cơ chế hoạt động

```
def plot_confusion_matrix(y_true, y_pred, title, filename):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=
       classes, yticklabels=classes)
    plt.title(f'Confusion Matrix - {title}')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.savefig(filename)
    plt.close()
```

1. `cm = confusion_matrix(y_true, y_pred)`: Sử dụng hàm `confusion_matrix` từ `sklearn.metrics` để tính toán ma trận nhầm lẫn.
2. Tạo một figure mới (`plt.figure`).
3. `sns.heatmap(...)`: Sử dụng hàm `heatmap` của thư viện `seaborn` để vẽ ma trận nhầm lẫn.
   - `annot=True`: Hiển thị giá trị số trong mỗi ô.
   - `fmt='d'`: Định dạng số hiển thị là số nguyên.
   - `cmap='Blues'`: Sử dụng bảng màu "Blues".
   - `xticklabels=classes, yticklabels=classes`: Đặt nhãn cho các trục bằng tên các lớp.
4. Đặt tiêu đề và nhãn cho các trục.
5. `plt.savefig(filename)`: Lưu ma trận nhầm lẫn vào file ảnh.
6. `plt.close()`: Đóng figure.

5 Thực thi và Kết quả

Mô tả quá trình chạy mã chính và các kết quả thu được.

5.1 Khởi tạo và huấn luyện mô hình MLP

```
# Run training and evaluation
mlp = MLP()
print("Training MLP...")
```

```
mlp_train_losses, mlp_val_losses, mlp_test_losses, mlp_train_accs,
    mlp_val_accs, mlp_test_accs = train_model(mlp, trainloader,
    valloader, testloader, epochs=20, patience=3)
```

Một đối tượng MLP được khởi tạo và hàm `train_model` được gọi để huấn luyện với tối đa 20 epochs và `patience` là 3. Lịch sử loss và accuracy được lưu lại.

5.2 Đánh giá mô hình trên các tập dữ liệu

Sau khi huấn luyện, mô hình được đánh giá trên ba tập dữ liệu.

```
# Evaluate and draw confusion matrix for each episode
print("\nEvaluating MLP on all sets...")
# Training set
mlp_train_y_true, mlp_train_y_pred, mlp_train_loss, mlp_train_acc =
    evaluate_model(mlp, trainloader, "Train")

# Validation set
mlp_val_y_true, mlp_val_y_pred, mlp_val_loss, mlp_val_acc =
    evaluate_model(mlp, valloader, "Validation")

# Test set
mlp_test_y_true, mlp_test_y_pred, mlp_test_loss, mlp_test_acc =
    evaluate_model(mlp, testloader, "Test")

# Print final result
print(f"\nFinal MLP Results:")
print(f"Train Loss: {mlp_train_loss:.4f}, Train Accuracy: {
    mlp_train_acc:.2f}%")
print(f"Validation Loss: {mlp_val_loss:.4f}, Validation Accuracy: {
    mlp_val_acc:.2f}%")
print(f"Test Loss: {mlp_test_loss:.4f}, Test Accuracy: {mlp_test_acc
    :.2f}%")
```

Hàm `evaluate_model` được gọi cho từng DataLoader. Kết quả loss và accuracy cuối cùng cho mỗi tập được in ra.

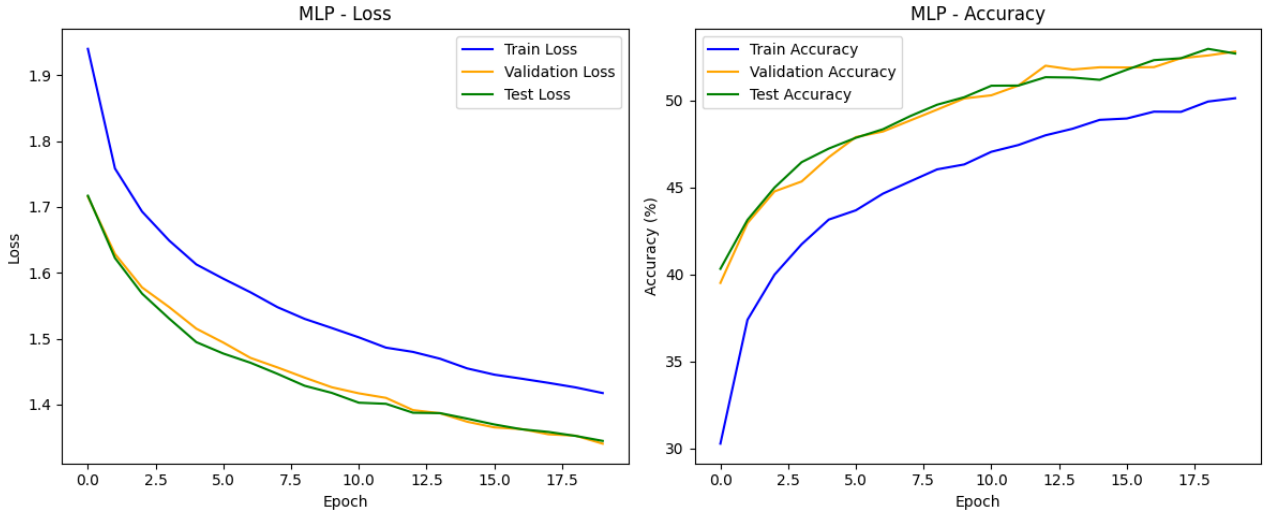
5.3 Trực quan hóa Learning Curves

Biểu đồ learning curves được tạo từ lịch sử loss và accuracy.

```
# Plot learning curves
plot_learning_curves(mlp_train_losses, mlp_val_losses, mlp_test_losses
    ,
```

```
                        mlp_train_accs, mlp_val_accs, mlp_test_accs, 'MLP
                            ')
```

Một file ảnh `mlp_learning_curves.png` được tạo ra, hiển thị các đường cong loss và accuracy.



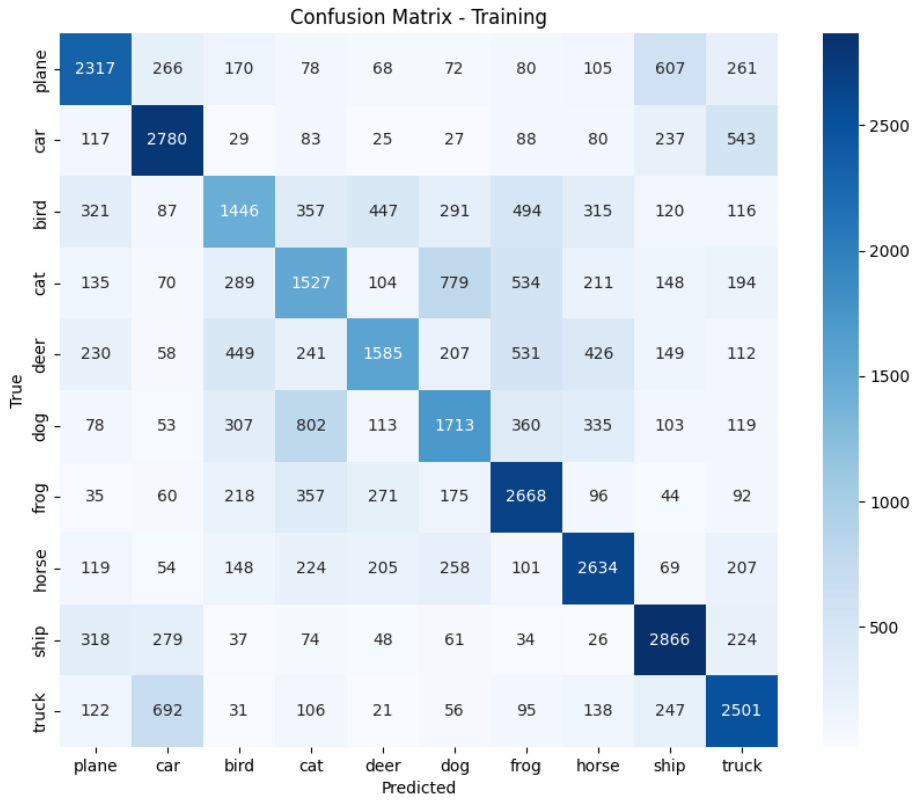
Hình 1: mlp_learning_curves.png

5.4 Trực quan hóa Confusion Matrices

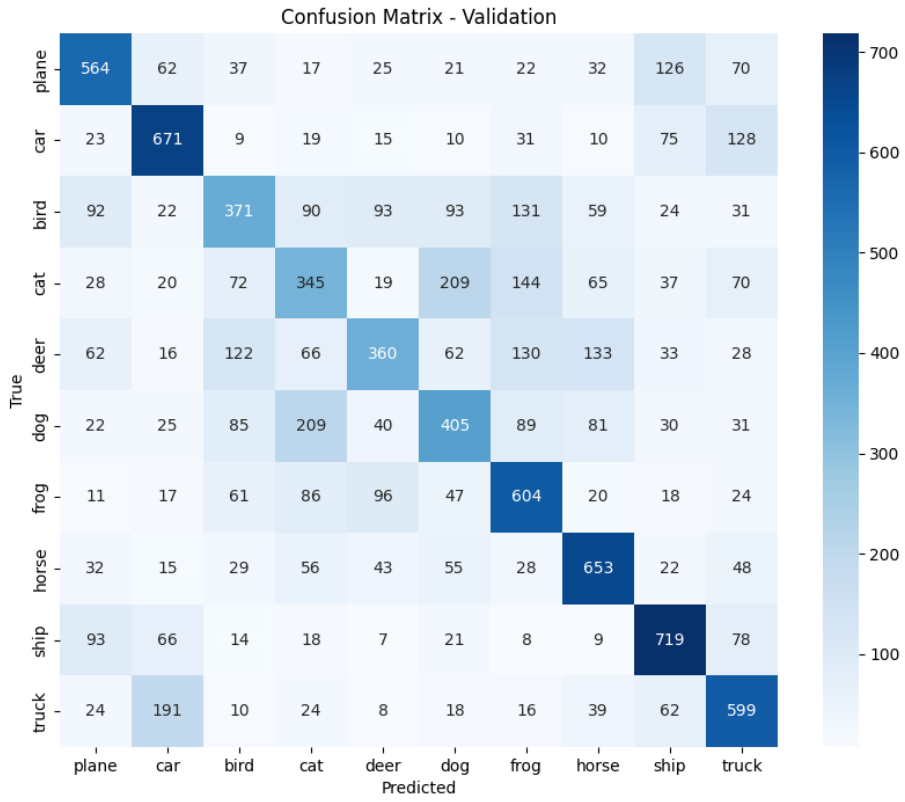
Ma trận nhầm lẫn được vẽ cho từng tập dữ liệu.

```
plot_confusion_matrix(mlp_train_y_true, mlp_train_y_pred, 'Training',
    'mlp_confusion_matrix_train.png')
plot_confusion_matrix(mlp_val_y_true, mlp_val_y_pred, 'Validation', '
    mlp_confusion_matrix_val.png')
plot_confusion_matrix(mlp_test_y_true, mlp_test_y_pred, 'Test', '
    mlp_confusion_matrix_test.png')
```

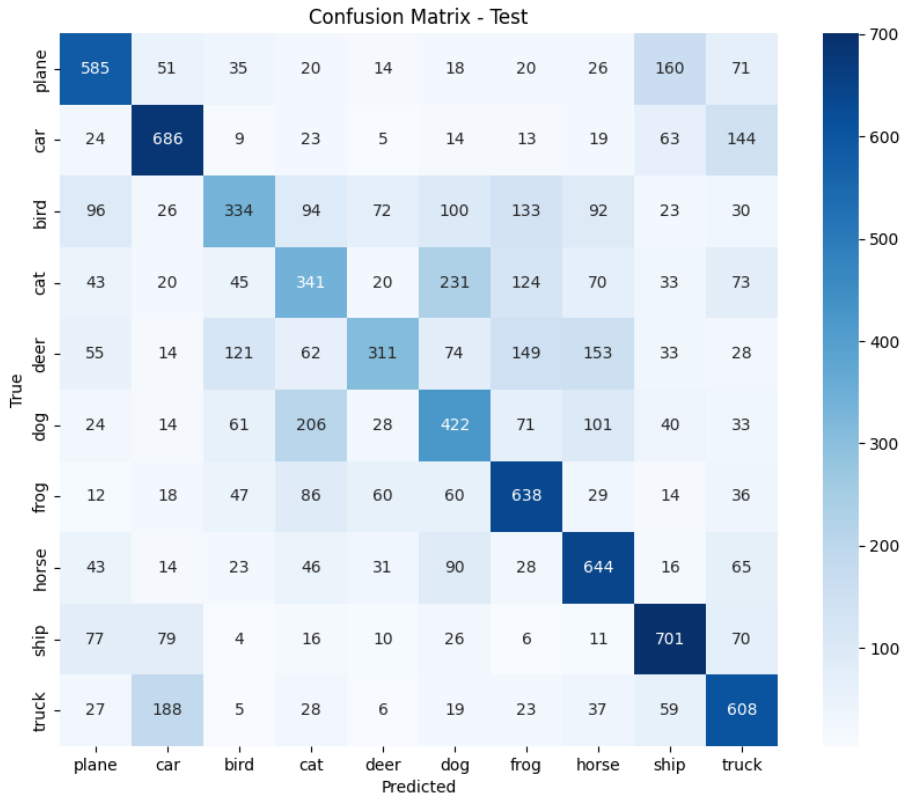
Ba file ảnh ma trận nhầm lẫn (`mlp_confusion_matrix_train.png`, `mlp_confusion_matrix_val.png`, `mlp_confusion_matrix_test.png`) được tạo.



Hình 2: mlp_confusion_matrix_train.png



Hình 3: mlp_confusion_matrix_val.png



Hình 4: mlp_confusion_matrix_test.png

5.5 Phân tích kết quả

Dựa trên các kết quả được in ra và các biểu đồ:

1. **Log huấn luyện**: Theo dõi sự thay đổi của loss và accuracy qua từng epoch để thấy quá trình học và điểm dừng (nếu có Early Stopping).
2. **Kết quả cuối cùng**: So sánh loss và accuracy trên ba tập. Độ chính xác trên tập test là thước đo quan trọng nhất về khả năng tổng quát hóa.
3. **Learning Curves**:
   - Kiểm tra sự hội tụ (loss giảm, accuracy tăng).
   - Phát hiện overfitting: Nếu train accuracy cao nhưng val/test accuracy thấp hơn nhiều (hoặc val loss tăng khi train loss giảm), đó là dấu hiệu overfitting.
   - Đánh giá Early Stopping: Xem liệu việc dừng sớm có phù hợp không.
4. **Confusion Matrices**:
   - Xác định các lớp được phân loại tốt (giá trị lớn trên đường chéo chính).

- Xác định các cặp lớp mà mô hình hay nhầm lẫn (giá trị lớn ngoài đường chéo).

Phân tích này cung cấp cái nhìn toàn diện về hiệu suất của mô hình MLP.

6 Kết luận

Bài tập này đã thực hiện thành công việc xây dựng, huấn luyện và đánh giá một Mạng Perceptron Đa Lớp (MLP) cho bài toán phân loại ảnh trên bộ dữ liệu CIFAR-10 bằng PyTorch. Những điểm nổi bật:

1. **Xử lý dữ liệu**: Áp dụng Data Augmentation hiệu quả.
2. **Kiến trúc MLP**: Xây dựng mô hình MLP với Dropout để chống overfitting.
3. **Huấn luyện nâng cao**: Sử dụng Adam optimizer, Weight Decay và Early Stopping.
4. **Đánh giá toàn diện**: Đánh giá mô hình trên các tập train, validation, và test.
5. **Trực quan hóa**: Tạo learning curves và confusion matrices để phân tích sâu.