BỘ THÔNG TIN VÀ TRUYỀN THÔNG

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÀI BÁO CÁO

MÔN: LẬP TRÌNH PYTHON

Giảng viên: Kim Ngọc Bách

Tên sinh viên: Nguyễn Minh Tuấn Kiệt + Arijunargal Tergel

Mã sinh viên: B23DCCE060

Lớp: D23CQCE06 - B

Môn: Lập trình Python

Hà Nội, Tháng 6/2025

1 Mã Nguồn Đầy Đủ

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix
from multiprocessing import freeze_support # Import freeze_support

# --- 1. Build MLP Model --- (Moved class definitions outside the main block)
class MLP(nn.Module):
    def __init__(self, input_size, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, 512) # Layer 1
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(512, 256) # Layer 2
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(256, num_classes)# Layer 3 (Output)

    def forward(self, x):
        # input_size_mlp needs to be accessible here if not passed.
        # It's better to define it where the model is instantiated or pass it.
        # For now, assuming it's globally defined or passed to __init__ and
            stored.
        # Let's ensure input_size_mlp is defined before MLP is called.
        x = x.view(-1, 32 * 32 * 3) # Flatten the image
        x = self.relu1(self.fc1(x))
        x = self.relu2(self.fc2(x))
        x = self.fc3(x)
        return x

# --- 2. Build CNN Model --- (Moved class definitions outside the main block)
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        # Layer 1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
            padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 2
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
            padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
47
48          # Layer 3
49          self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
                padding=1)
50          self.relu3 = nn.ReLU()
51          self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
52
53          # Fully connected layer
54          self.fc = nn.Linear(128 * 4 * 4, num_classes)
55
56      def forward(self, x):
57          x = self.pool1(self.relu1(self.conv1(x)))
58          x = self.pool2(self.relu2(self.conv2(x)))
59          x = self.pool3(self.relu3(self.conv3(x)))
60          x = x.view(-1, 128 * 4 * 4) # Flatten for FC layer
61          x = self.fc(x)
62          return x
63
64  # --- 3. Training Function --- (Moved function definitions outside the main
        block)
65  def train_model(model, train_loader, val_loader, criterion, optimizer,
        num_epochs=20, device='cpu'):
66      train_losses = []
67      val_losses = []
68      train_accuracies = []
69      val_accuracies = []
70
71      for epoch in range(num_epochs):
72          model.train()
73          running_loss = 0.0
74          correct_train = 0
75          total_train = 0
76
77          for i, (inputs, labels) in enumerate(train_loader):
78              inputs, labels = inputs.to(device), labels.to(device)
79
80              optimizer.zero_grad()
81              outputs = model(inputs)
82              loss = criterion(outputs, labels)
83              loss.backward()
84              optimizer.step()
85
86              running_loss += loss.item() * inputs.size(0)
87              _, predicted = torch.max(outputs.data, 1)
88              total_train += labels.size(0)
89              correct_train += (predicted == labels).sum().item()
90
91          epoch_train_loss = running_loss / len(train_loader.dataset)
92          epoch_train_acc = correct_train / total_train
93          train_losses.append(epoch_train_loss)
94          train_accuracies.append(epoch_train_acc)
```

```
95
96          model.eval()
97          running_val_loss = 0.0
98          correct_val = 0
99          total_val = 0
100         with torch.no_grad():
101             for inputs, labels in val_loader:
102                 inputs, labels = inputs.to(device), labels.to(device)
103                 outputs = model(inputs)
104                 loss = criterion(outputs, labels)
105                 running_val_loss += loss.item() * inputs.size(0)
106                 _, predicted = torch.max(outputs.data, 1)
107                 total_val += labels.size(0)
108                 correct_val += (predicted == labels).sum().item()
109
110         epoch_val_loss = running_val_loss / len(val_loader.dataset)
111         epoch_val_acc = correct_val / total_val
112         val_losses.append(epoch_val_loss)
113         val_accuracies.append(epoch_val_acc)
114
115         print(f"Epoch [{epoch+1}/{num_epochs}], "
116               f"Train Loss: {epoch_train_loss:.4f}, Train Acc:
                      {epoch_train_acc:.4f}, "
117               f"Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f}")
118
119     return train_losses, val_losses, train_accuracies, val_accuracies
120
121 # --- 4. Evaluation Function & Plotting --- (Moved function definitions
      outside)
122 def plot_learning_curves(train_losses, val_losses, train_accuracies,
      val_accuracies, model_name):
123     epochs_range = range(1, len(train_losses) + 1)
124     plt.figure(figsize=(12, 4))
125     plt.subplot(1, 2, 1)
126     plt.plot(epochs_range, train_losses, label='Training Loss')
127     plt.plot(epochs_range, val_losses, label='Validation Loss')
128     plt.xlabel('Epochs')
129     plt.ylabel('Loss')
130     plt.title(f'{model_name} - Loss Curves')
131     plt.legend(); plt.grid(True)
132     plt.subplot(1, 2, 2)
133     plt.plot(epochs_range, train_accuracies, label='Training Accuracy')
134     plt.plot(epochs_range, val_accuracies, label='Validation Accuracy')
135     plt.xlabel('Epochs')
136     plt.ylabel('Accuracy')
137     plt.title(f'{model_name} - Accuracy Curves')
138     plt.legend(); plt.grid(True)
139     plt.suptitle(f'Learning Curves for {model_name}', fontsize=16)
140     plt.tight_layout(rect=[0, 0.03, 1, 0.95])
141     plt.show()
142
```

```python
def evaluate_model(model, test_loader, model_name, classes_list, device='cpu'):
    model.eval()
    all_preds = []
    all_labels = []
    test_loss = 0.0
    correct_test = 0
    total_test = 0
    criterion = nn.CrossEntropyLoss()

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs.data, 1)
            total_test += labels.size(0)
            correct_test += (predicted == labels).sum().item()
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    avg_test_loss = test_loss / len(test_loader.dataset)
    test_accuracy = correct_test / total_test
    print(f"\n--- {model_name} Test Results ---")
    print(f"Test Loss: {avg_test_loss:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f} ({correct_test}/{total_test})")

    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=classes_list, yticklabels=classes_list)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix for {model_name}')
    plt.show()
    return test_accuracy, avg_test_loss, cm


# This is the main guard
if __name__ == '__main__':
    freeze_support() # Add this line for Windows compatibility with
        multiprocessing

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # --- Load and Preprocess CIFAR-10 ---
    print("\n--- Loading and Preprocessing CIFAR-10 ---")
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

```
    ])
    train_set_full = torchvision.datasets.CIFAR10(root='./data', train=True,
        download=True, transform=transform)
    test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
        download=True, transform=transform)

    train_size = int(0.8 * len(train_set_full))
    val_size = len(train_set_full) - train_size
    train_set, val_set = random_split(train_set_full, [train_size, val_size])

    batch_size = 64
    # Set num_workers=0 if issues persist, but try with 2 first after the fix
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True,
        num_workers=2)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False,
        num_workers=2)
    test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False,
        num_workers=2)

    classes = ('plane', 'car', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck')
    num_classes = len(classes)
    input_size_mlp = 32 * 32 * 3

    print(f"Training set size: {len(train_set)}")
    print(f"Validation set size: {len(val_set)}")
    print(f"Test set size: {len(test_set)}")
    print(f"Number of classes: {num_classes}")

    # --- Hyperparameters ---
    learning_rate = 0.001
    num_epochs = 25

    # --- 6. Train and Evaluate MLP ---
    print("\n\n--- Training and Evaluating MLP ---")
    mlp_model = MLP(input_size=input_size_mlp,
        num_classes=num_classes).to(device)
    criterion_mlp = nn.CrossEntropyLoss()
    optimizer_mlp = optim.Adam(mlp_model.parameters(), lr=learning_rate)

    mlp_train_losses, mlp_val_losses, mlp_train_accuracies, mlp_val_accuracies
        = train_model(
        mlp_model, train_loader, val_loader, criterion_mlp, optimizer_mlp,
            num_epochs=num_epochs, device=device
    )

    plot_learning_curves(mlp_train_losses, mlp_val_losses,
        mlp_train_accuracies, mlp_val_accuracies, "MLP")
    mlp_test_acc, mlp_test_loss, mlp_cm = evaluate_model(mlp_model,
        test_loader, "MLP", classes, device=device)

```

```
    # --- 7. Train and Evaluate CNN ---
    print("\n\n--- Training and Evaluating CNN ---")
    cnn_model = CNN(num_classes=num_classes).to(device)
    criterion_cnn = nn.CrossEntropyLoss()
    optimizer_cnn = optim.Adam(cnn_model.parameters(), lr=learning_rate)

    cnn_train_losses, cnn_val_losses, cnn_train_accuracies, cnn_val_accuracies
        = train_model(
        cnn_model, train_loader, val_loader, criterion_cnn, optimizer_cnn,
            num_epochs=num_epochs, device=device
    )

    plot_learning_curves(cnn_train_losses, cnn_val_losses,
        cnn_train_accuracies, cnn_val_accuracies, "CNN")
    cnn_test_acc, cnn_test_loss, cnn_cm = evaluate_model(cnn_model,
        test_loader, "CNN", classes, device=device)

    # --- 8. Compare Results ---
    print("\n\n--- Comparison of MLP and CNN ---")
    print(f"MLP Test Accuracy: {mlp_test_acc:.4f}, MLP Test Loss:
        {mlp_test_loss:.4f}")
    print(f"CNN Test Accuracy: {cnn_test_acc:.4f}, CNN Test Loss:
        {cnn_test_loss:.4f}")
```

2 Phân loại ảnh CIFAR-10 sử dụng MLP và CNN với PyTorch

Nội dung này trình bày chi tiết về mã Python thực hiện nhiệm vụ phân loại hình ảnh bằng cách sử dụng bộ dữ liệu CIFAR-10 với hai kiến trúc mạng nơ-ron khác nhau: Multi-Layer Perceptron (MLP) và Convolutional Neural Network (CNN), sử dụng thư viện PyTorch.

2.1 Các thư viện cần cài đặt

Để chạy mã này, bạn cần cài đặt các thư viện Python sau:

- **PyTorch**: Thư viện học sâu mã nguồn mở chính.

  ```
  pip install torch torchvision torchaudio
  ```

  (Truy cập trang web chính thức của PyTorch để có lệnh cài đặt phù hợp nhất với hệ điều hành và cấu hình CUDA của bạn nếu bạn có GPU).

- **Matplotlib**: Thư viện để vẽ biểu đồ.

```
pip install matplotlib
```

- **NumPy**: Thư viện cho tính toán khoa học, đặc biệt là xử lý mảng.

  ```
  pip install numpy
  ```

- **Seaborn**: Thư viện trực quan hóa dữ liệu dựa trên Matplotlib, dùng để vẽ confusion matrix đẹp hơn.

  ```
  pip install seaborn
  ```

- **Scikit-learn**: Thư viện học máy, ở đây dùng để tính toán confusion matrix.

  ```
  pip install scikit-learn
  ```

2.2 Giới thiệu chi tiết về bộ dữ liệu CIFAR-10

CIFAR-10 là một bộ dữ liệu hình ảnh được sử dụng rộng rãi trong nghiên cứu học máy và thị giác máy tính.

- **Nội dung**: Bộ dữ liệu này bao gồm 60.000 hình ảnh màu nhỏ.
- **Kích thước ảnh**: Mỗi hình ảnh có kích thước 32x32 pixel.
- **Số lớp (Classes)**: Có 10 lớp đối tượng, và mỗi lớp có 6.000 hình ảnh. Các lớp này là:
  1. `plane` (máy bay)
  2. `car` (ô tô)
  3. `bird` (chim)
  4. `cat` (mèo)
  5. `deer` (nai)
  6. `dog` (chó)
  7. `frog` (ếch)
  8. `horse` (ngựa)
  9. `ship` (tàu thủy)
  10. `truck` (xe tải)
- **Phân chia dữ liệu**:
  - **Tập huấn luyện (Training set)**: 50.000 hình ảnh (5.000 hình ảnh cho mỗi lớp).

- **Tập kiểm thử (Test set)**: 10.000 hình ảnh (1.000 hình ảnh cho mỗi lớp).

- **Đặc điểm**: Các lớp hoàn toàn loại trừ lẫn nhau. Hình ảnh trong CIFAR-10 có độ phân giải thấp và chứa các đối tượng ở nhiều góc độ, kích thước và điều kiện ánh sáng khác nhau.

2.3 Giới thiệu chi tiết về mô hình MLP (Multi-Layer Perceptron)

MLP là một loại mạng nơ-ron nhân tạo truyền thẳng cơ bản, bao gồm lớp đầu vào, một hoặc nhiều lớp ẩn, và một lớp đầu ra.

- **Kiến trúc MLP trong mã (3 lớp chính tính từ lớp ẩn đầu tiên)**:
  - **Lớp đầu vào**: Hình ảnh CIFAR-10 (3x32x32) được "làm phẳng" thành vector một chiều kích thước $32 \times 32 \times 3 = 3072$.
  - **Lớp ẩn 1**: Gồm một lớp kết nối đầy đủ `nn.Linear(3072, 512)` theo sau bởi hàm kích hoạt `nn.ReLU()`. Hàm ReLU được định nghĩa là $ReLU(x) = \max(0, x)$.
  - **Lớp ẩn 2**: Gồm một lớp kết nối đầy đủ `nn.Linear(512, 256)` theo sau bởi hàm kích hoạt `nn.ReLU()`.
  - **Lớp đầu ra**: Một lớp kết nối đầy đủ `nn.Linear(256, num_classes)` (với `num_classes` là 10) để tạo ra điểm số cho mỗi lớp.

- **Quá trình xử lý ảnh của MLP**: Hình ảnh được làm phẳng và truyền qua các lớp tuần tự.

- **Hạn chế của MLP với ảnh**: Việc làm phẳng ảnh làm mất thông tin không gian quan trọng, khiến MLP khó khăn trong việc nhận diện các đặc trưng cục bộ một cách hiệu quả.

2.4 Giới thiệu chi tiết về mô hình CNN (Convolutional Neural Network)

CNN là loại mạng nơ-ron chuyên biệt cho xử lý dữ liệu có cấu trúc lưới như hình ảnh.

- **Các thành phần chính**:
  - **Lớp tích chập (Convolutional Layer)**: Áp dụng các bộ lọc (filters/kernels) để phát hiện đặc trưng cục bộ (cạnh, góc). Quan trọng là cơ chế chia sẻ trọng số (weight sharing).
  - **Hàm kích hoạt (Activation Function)**: Thường là ReLU, thêm tính phi tuyến.
  - **Lớp gộp (Pooling Layer)**: Giảm kích thước không gian của bản đồ đặc trưng, giảm số lượng tham số và giúp kiểm soát overfitting (ví dụ: Max Pooling).

- **Lớp kết nối đầy đủ (Fully Connected Layer)**: Thực hiện phân loại cuối cùng dựa trên các đặc trưng đã học.

- **Kiến trúc CNN trong mã (3 lớp tích chập)**:
  - **Đầu vào**: Hình ảnh màu 3x32x32.
  - **Khối 1**:
    - Lớp tích chập `nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)`.
    - Hàm kích hoạt `nn.ReLU()`.
    - Lớp gộp `nn.MaxPool2d(kernel_size=2, stride=2)` (kích thước giảm từ 32x32 xuống 16x16).
  - **Khối 2**:
    - Lớp tích chập `nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)`.
    - Hàm kích hoạt `nn.ReLU()`.
    - Lớp gộp `nn.MaxPool2d(kernel_size=2, stride=2)` (kích thước giảm từ 16x16 xuống 8x8).
  - **Khối 3**:
    - Lớp tích chập `nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)`.
    - Hàm kích hoạt `nn.ReLU()`.
    - Lớp gộp `nn.MaxPool2d(kernel_size=2, stride=2)` (kích thước giảm từ 8x8 xuống 4x4).
  - **Làm phẳng (Flattening)**: Đầu ra từ khối 3 (128 bản đồ đặc trưng kích thước 4x4) được làm phẳng thành vector $128 \times 4 \times 4 = 2048$ chiều.
  - **Lớp kết nối đầy đủ**: `nn.Linear(2048, num_classes)` để phân loại.

- **Ưu điểm của CNN với ảnh**: Phát hiện đặc trưng cục bộ, phân cấp đặc trưng, bất biến với tịnh tiến (một phần), và giảm số lượng tham số so với MLP tương đương.

2.5 Cách code hoạt động chi tiết

- **Khởi tạo và Cài đặt (trong `if __name__ == '__main__':`)**:
  - `freeze_support()`: Hỗ trợ đa xử lý trên Windows.
  - `device`: Chọn GPU (CUDA) nếu có, nếu không thì dùng CPU.
  - **Tải và Tiền xử lý dữ liệu CIFAR-10**:
    - `transforms.Compose([...])`: Chuỗi các phép biến đổi:
      - `transforms.ToTensor()`: Chuyển ảnh PIL/NumPy (0-255) thành Tensor PyTorch (0.0-1.0) và đổi chiều (C, H, W).

· `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Chuẩn hóa tensor về khoảng [-1.0, 1.0].

* `torchvision.datasets.CIFAR10(...)`: Tải bộ dữ liệu.
* `random_split(...)`: Chia tập huấn luyện gốc thành tập huấn luyện mới (80%) và tập kiểm định (20%).
* `DataLoader(...)`: Tạo đối tượng tải dữ liệu theo lô (batch), có thể xáo trộn (`shuffle=True`) và sử dụng đa tiến trình (`num_workers`).

– Các biến khác: `classes`, `num_classes`, `input_size_mlp`, `learning_rate`, `num_epochs`.

- **Định nghĩa Mô hình**: Các lớp `MLP(nn.Module)` và `CNN(nn.Module)` được định nghĩa với:
  – Phương thức `__init__()`: Khởi tạo các lớp (layers) của mạng.
  – Phương thức `forward()`: Định nghĩa cách dữ liệu truyền qua mạng.
- **Hàm Huấn luyện (`train_model`)**:
  – **Mục đích**: Huấn luyện mô hình.
  – **Tham số**: model, train_loader, val_loader, criterion (hàm mất mát, ví dụ `nn.CrossEntropyLoss`), optimizer (thuật toán tối ưu, ví dụ `optim.Adam`), num_epochs, device.
  – **Hoạt động**: Lặp qua các epoch. Trong mỗi epoch:
    * **Pha Huấn luyện**: Đặt `model.train()`, lặp qua train_loader, chuyển dữ liệu lên device, xóa gradient cũ (`optimizer.zero_grad()`), thực hiện forward pass, tính loss, thực hiện backward pass (`loss.backward()`), cập nhật trọng số (`optimizer.step()`). Tính toán loss và accuracy trên tập huấn luyện.
    * **Pha Kiểm định**: Đặt `model.eval()`, tắt tính gradient (`with torch.no_grad()`), lặp qua val_loader, tính toán loss và accuracy trên tập kiểm định.
  – **Trả về**: Lịch sử loss và accuracy của training và validation.
- **Hàm Vẽ Biểu đồ Học tập (`plot_learning_curves`)**:
  – **Mục đích**: Trực quan hóa quá trình học.
  – **Hoạt động**: Sử dụng Matplotlib để vẽ biểu đồ loss và accuracy theo epoch cho cả training và validation.
- **Hàm Đánh giá Mô hình (`evaluate_model`)**:
  – **Mục đích**: Đánh giá hiệu suất cuối cùng trên tập test và tạo confusion matrix.
  – **Hoạt động**: Đặt `model.eval()`, tắt gradient, lặp qua test_loader, tính test loss và accuracy. Sử dụng `sklearn.metrics.confusion_matrix` và `seaborn.heatmap` để vẽ confusion matrix.
  – **Trả về**: Test accuracy, test loss, confusion matrix.
- **Thực thi chính**: Khởi tạo, huấn luyện, vẽ biểu đồ, đánh giá cho cả MLP và CNN. Cuối cùng là so sánh kết quả.

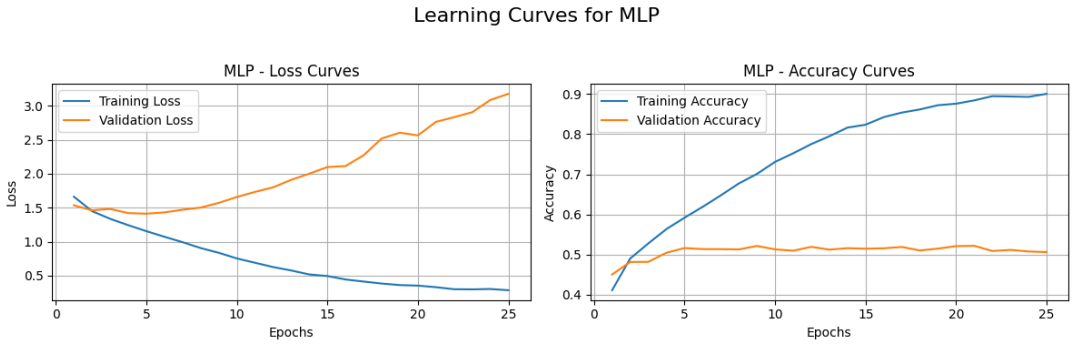
2.6 Kết quả cho ra

Quá trình huấn luyện và đánh giá hai mô hình MLP và CNN đã được thực hiện. Dưới đây là tóm tắt chi tiết các kết quả thu được:

- **Quá trình huấn luyện và đánh giá MLP**:
  - **Tiến trình huấn luyện qua các Epoch**:
    - Epoch 1: Train Loss: 1.6616, Train Acc: 0.4114, Val Loss: 1.5349, Val Acc: 0.4506
    - Epoch 5: Train Loss: 1.1548, Train Acc: 0.5919, Val Loss: 1.4118, Val Acc: 0.5164 (Validation loss thấp nhất)
    - Epoch 10: Train Loss: 0.7533, Train Acc: 0.7309, Val Loss: 1.6575, Val Acc: 0.5131
    - Epoch 15: Train Loss: 0.4946, Train Acc: 0.8234, Val Loss: 2.0980, Val Acc: 0.5148
    - Epoch 20: Train Loss: 0.3534, Train Acc: 0.8757, Val Loss: 2.5638, Val Acc: 0.5212
    - Epoch 25: Train Loss: 0.2856, Train Acc: 0.9002, Val Loss: 3.1752, Val Acc: 0.5064
  - **Kết quả kiểm thử cuối cùng của MLP**:
    - MLP Test Loss: **3.2325**
    - MLP Test Accuracy: **0.5046** (tức là 50.46%, hay 5046/10000 mẫu đúng)
- **Quá trình huấn luyện và đánh giá CNN**:
  - **Tiến trình huấn luyện qua các Epoch**:
    - Epoch 1: Train Loss: 1.4425, Train Acc: 0.4802, Val Loss: 1.1549, Val Acc: 0.5959
    - Epoch 6: Train Loss: 0.6143, Train Acc: 0.7893, Val Loss: 0.7893, Val Acc: 0.7313 (Validation loss gần mức thấp nhất, Val Acc khá cao)
    - Epoch 10: Train Loss: 0.3946, Train Acc: 0.8630, Val Loss: 0.8413, Val Acc: 0.7344 (Validation accuracy cao nhất)
    - Epoch 15: Train Loss: 0.2080, Train Acc: 0.9252, Val Loss: 1.1174, Val Acc: 0.7235
    - Epoch 20: Train Loss: 0.1229, Train Acc: 0.9560, Val Loss: 1.4654, Val Acc: 0.7169
    - Epoch 25: Train Loss: 0.0864, Train Acc: 0.9688, Val Loss: 1.9266, Val Acc: 0.7038
  - **Kết quả kiểm thử cuối cùng của CNN**:
    - CNN Test Loss: **1.9714**
    - CNN Test Accuracy: **0.7137** (tức là 71.37%, hay 7137/10000 mẫu đúng)
- **So sánh tổng hợp trên tập kiểm thử**:
  - MLP Test Accuracy: 0.5046, MLP Test Loss: 3.2325

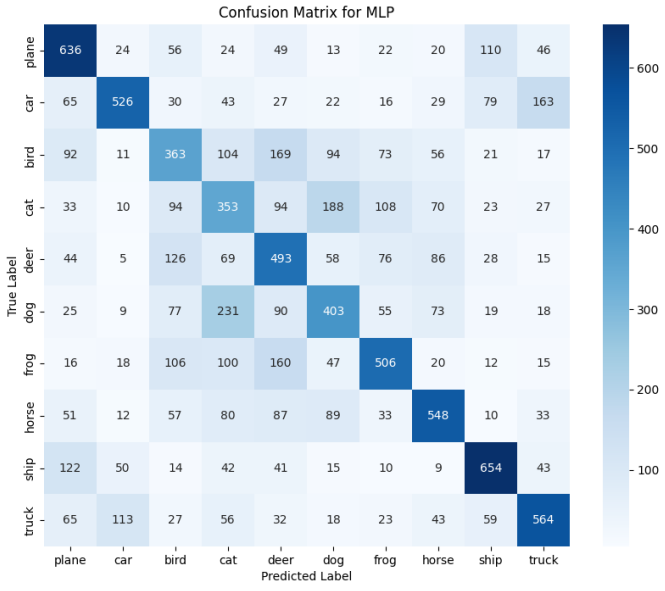
– CNN Test Accuracy: 0.7137, CNN Test Loss: 1.9714

- **Các Biểu đồ Trực quan hóa**:
  - **Learning Curves cho MLP**:
    * *MLP - Loss Curves*: Training Loss giảm đều từ khoảng 1.66 xuống 0.28. Validation Loss ban đầu giảm từ 1.53 xuống mức thấp nhất khoảng 1.41 tại epoch 5, sau đó tăng liên tục lên đến 3.17 ở epoch 25. Điều này cho thấy overfitting bắt đầu từ rất sớm.
    * *MLP - Accuracy Curves*: Training Accuracy tăng từ 0.41 lên 0.90. Validation Accuracy tăng từ 0.45 lên mức cao nhất khoảng 0.52 (epoch 9, 12, 17, 20, 21) và sau đó dao động quanh mức 0.50-0.51. Khoảng cách lớn và ngày càng tăng giữa training và validation accuracy củng cố thêm bằng chứng về overfitting.



Hình 1: Biểu đồ Learning Curves cho mô hình MLP.

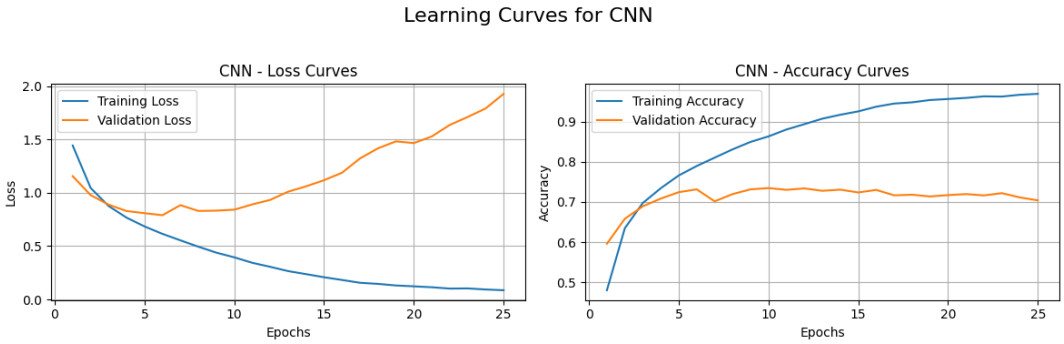
  - **Confusion Matrix cho MLP**: (Phân tích ma trận như đã mô tả ở phiên bản trước, dựa trên hình ảnh).



Hình 2: Ma trận nhầm lẫn (Confusion Matrix) cho mô hình MLP.

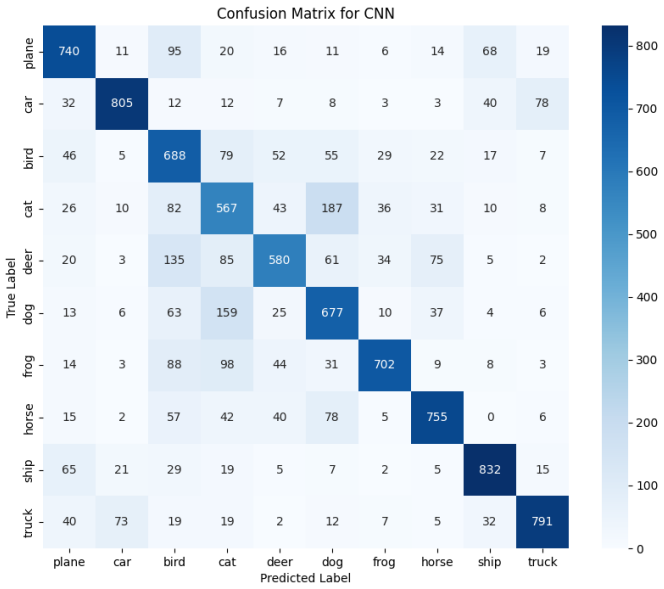
– **Learning Curves cho CNN**:

* *CNN - Loss Curves*: Training Loss giảm mạnh từ 1.44 xuống 0.086. Validation Loss giảm từ 1.15 xuống mức thấp nhất khoảng 0.789 tại epoch 6, sau đó có xu hướng tăng dần lên 1.92 ở epoch 25, cho thấy overfitting cũng xảy ra nhưng bắt đầu muộn hơn so với MLP.
* *CNN - Accuracy Curves*: Training Accuracy tăng nhanh từ 0.48 lên 0.968. Validation Accuracy tăng từ 0.59 lên mức cao nhất khoảng 0.734 tại epoch 10 và sau đó dao động nhẹ, giảm dần về 0.703 ở epoch cuối. Overfitting thể hiện rõ khi training accuracy tiếp tục tăng mạnh trong khi validation accuracy không cải thiện hoặc giảm nhẹ.



Hình 3: Biểu đồ Learning Curves cho mô hình CNN.

– **Confusion Matrix cho CNN (MTRX-CNN.png)**: (Phân tích ma trận như đã mô tả ở phiên bản trước, dựa trên hình ảnh).



Hình 4: Ma trận nhầm lẫn (Confusion Matrix) cho mô hình CNN.

2.7 Thảo luận và So sánh Kết quả

Dựa trên các kết quả kiểm thử, log huấn luyện chi tiết và các biểu đồ trực quan:

- **Hiệu suất tổng thể**: CNN (Test Accuracy: **71.37%**, Test Loss: **1.9714**) rõ ràng vượt trội so với MLP (Test Accuracy: **50.46%**, Test Loss: **3.2325**). Sự chênh lệch gần 21% về độ chính xác trên tập kiểm thử là một minh chứng mạnh mẽ cho thấy kiến trúc CNN phù hợp hơn nhiều cho các tác vụ phân loại hình ảnh như CIFAR-10.
- **Overfitting và Quá trình học**:
  - **MLP**: Mô hình MLP bắt đầu có dấu hiệu overfitting rất sớm. Validation Loss đạt giá trị thấp nhất tại epoch 5 (Val Loss: 1.4118, Val Acc: 0.5164) và sau đó tăng liên tục. Trong khi đó, Training Loss vẫn tiếp tục giảm và Training Accuracy tăng đều. Điều này cho thấy MLP nhanh chóng học thuộc (memorize) dữ liệu huấn luyện nhưng không thể tổng quát hóa tốt cho dữ liệu mới. Đến cuối quá trình huấn luyện (epoch 25), Training Accuracy đạt 0.9002 trong khi Validation Accuracy chỉ còn 0.5064, một khoảng cách rất lớn.
  - **CNN**: Mô hình CNN cũng thể hiện overfitting, nhưng muộn hơn và ít nghiêm trọng hơn. Validation Loss của CNN đạt mức thấp nhất khoảng 0.7893 tại epoch 6 (Val Acc: 0.7313), và Validation Accuracy đạt đỉnh điểm 0.7344 tại epoch 10. Sau đó, Validation Loss bắt đầu tăng dần và Validation Accuracy có xu hướng giảm nhẹ hoặc dao động. Mặc dù Training Accuracy của CNN lên tới 0.9688, Validation Accuracy cuối cùng là 0.7038, cho thấy một mức độ overfitting nhất định, nhưng khả năng tổng quát hóa vẫn tốt hơn nhiều so với MLP.
- **Phân tích Ma trận Nhầm lẫn (Confusion Matrix)**:
  - **MLP (MTRX-MLP.png)**: Như đã phân tích dựa trên hình ảnh, MLP gặp nhiều khó khăn trong việc phân biệt các lớp, đặc biệt là các lớp có đặc điểm hình ảnh tương đồng. Số lượng dự đoán đúng (đường chéo chính) thấp và có nhiều giá trị lớn nằm ngoài đường chéo.
  - **CNN (MTRX-CNN.png)**: CNN cho thấy khả năng phân biệt lớp vượt trội. Các giá trị trên đường chéo chính cao hơn đáng kể. Mặc dù vẫn có một số nhầm lẫn giữa các lớp khó (ví dụ 'cat' và 'dog'), nhưng hiệu suất tổng thể tốt hơn nhiều, phản ánh khả năng của CNN trong việc trích xuất các đặc trưng phân biệt từ hình ảnh.
- **Nguyên nhân của sự khác biệt**:
  - Kiến trúc của **CNN** với các lớp tích chập cho phép nó học các đặc trưng không gian (spatial features) và các mẫu hình ảnh theo một cách phân cấp. Các bộ lọc (kernels) trong lớp tích chập có khả năng phát hiện các cạnh, góc, kết cấu, và các bộ phận phức tạp hơn của đối tượng. Lớp gộp (pooling) giúp giảm độ phức tạp tính toán và tạo ra một mức độ bất biến đối với các thay đổi nhỏ về vị trí. Việc chia sẻ trọng số (weight sharing) trong các lớp tích chập cũng làm giảm đáng kể số lượng tham số cần học so với MLP, giúp mô hình ít bị overfitting hơn và tổng quát hóa tốt hơn.

– **MLP**, khi xử lý hình ảnh đã được làm phẳng thành một vector một chiều, sẽ mất đi toàn bộ thông tin về cấu trúc không gian 2D của ảnh. Nó coi mỗi pixel như một đầu vào độc lập và phải học lại từ đầu mối quan hệ giữa các pixel, điều này rất khó khăn và không hiệu quả đối với dữ liệu hình ảnh có cấu trúc. Đây là lý do chính khiến MLP có hiệu suất thấp hơn và dễ bị overfitting hơn trong bài toán này.

3 Kết luận

Dữ liệu thực nghiệm đã cung cấp bằng chứng rõ ràng về ưu thế của CNN so với MLP trong nhiệm vụ phân loại hình ảnh trên bộ dữ liệu CIFAR-10. Mặc dù cả hai mô hình đều có thể được cải thiện thêm (ví dụ bằng cách thêm các kỹ thuật điều chuẩn như Dropout, Batch Normalization, sử dụng learning rate scheduler, hoặc tinh chỉnh kiến trúc mạng), sự khác biệt cơ bản về kiến trúc đã quyết định hiệu suất vượt trội của CNN.