BỘ KHOA HỌC VÀ CÔNG NGHỆ
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

—————o0o—————


# BÁO CÁO BÀI TẬP LỚN 2

# NGÔN NGỮ LẬP TRÌNH PYTHON


| | |
|---|---|
| **Giảng viên hướng dẫn:** | **Kim Ngọc Bách** |
| **Sinh viên:** | **Lê Như Quỳnh** |
| **Mã sinh viên:** | **B23DCCE081** |
| **Lớp:** | **D23CQCE06-B** |
| **Niên khóa:** | **2023 - 2028** |
| **Hệ đào tạo:** | **Đại học chính quy** |


**Hà Nội, 2025**

# NHẬN XÉT CỦA GIẢNG VIÊN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

**Điểm:** **( Bằng chữ: )**

Hà Nội, ngày tháng năm 20...

**Giảng viên**

# Mục lục

# Danh sách hình vẽ

# Danh sách Liệt kê Mã nguồn

# Phần mở đầu

Bài báo cáo này, em sẽ trình bày chi tiết quá trình xây dựng, huấn luyện và đánh giá một mô hình mạng nơ-ron tích chập (**Convolutional Neural Network - CNN**) nhằm giải quyết bài toán phân loại ảnh trên tập dữ liệu **CIFAR-10**. Đây là bộ dữ liệu gồm 60.000 ảnh màu kích thước $32 \times 32$ pixel, được chia đều thành 10 lớp đối tượng khác nhau, rất phổ biến trong các nghiên cứu về thị giác máy tính.

Toàn bộ quá trình được thực hiện bằng ngôn ngữ lập trình Python, trong đó PyTorch là thư viện chính được sử dụng cho việc xây dựng và huấn luyện mô hình. Bên cạnh đó, các thư viện hỗ trợ như torchvision, Matplotlib, Seaborn, NumPy và Scikit-learn cũng được sử dụng để xử lý dữ liệu, trực quan hóa kết quả và đánh giá hiệu năng mô hình một cách toàn diện.

# Chương 1

# Chuẩn bị Dữ liệu

Bước đầu tiên và quan trọng là chuẩn bị dữ liệu. Trong chương này, em sẽ trình bày việc khai báo thư viện, các tham số em sử dụng, cùng các bước tải và tiền xử lý bộ dữ liệu **CIFAR-10**.

## 1.1 Khai báo Thư viện và Tham số

Đầu tiên, em import các thư viện và định nghĩa các tham số cần thiết.

### 1.1.1 Khai báo các thư viện

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader, random_split
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix
```

Liệt kê mã 1.1: Khai báo các thư viện Python

Các thư viện này có vai trò trong việc xây dựng mô hình, xử lý dữ liệu, tối ưu hóa và trực quan hóa kết quả.

### 1.1.2 Khai báo các tham số chính

Cụ thể, `BATCH_SIZE` em đặt là 64, tốc độ học `LR` ban đầu là $1 \times 10^{-3}$, số `EPOCHS` tối đa cho quá trình huấn luyện là 40. `DEVICE` sẽ tự động chọn GPU nếu có, và `CLASS_NAMES` là

danh sách tên các lớp của **CIFAR-10**.

```
BATCH_SIZE = 64
LR = 1e-3
EPOCHS = 40
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
CLASS_NAMES = ('airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck')
```

Liệt kê mã 1.2: Khai báo các tham số chính

## 1.2 Tải và Tiền xử lý Dữ liệu

Em viết hàm `get_data_loaders` để thực hiện việc tải và tiền xử lý dữ liệu.

### 1.2.1 Định nghĩa các phép biến đổi (Transforms)

```
    transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
```

Liệt kê mã 1.3: Định nghĩa các phép biến đổi dữ liệu

Các phép biến đổi này bao gồm lật ảnh ngẫu nhiên theo chiều ngang, cắt ảnh ngẫu nhiên, chuyển ảnh sang Tensor và chuẩn hóa giá trị pixel.

### 1.2.2 Tải bộ dữ liệu CIFAR-10

Em tải bộ dữ liệu CIFAR-10 từ torchvision.datasets và áp dụng các phép biến đổi trên:

```
    full_train = CIFAR10(root='./data', train=True, download=True,
    ↪  transform=transform)
    test_set = CIFAR10(root='./data', train=False, download=True,
    ↪  transform=transform)
```

Liệt kê mã 1.4: Tải bộ dữ liệu CIFAR-10

### 1.2.3 Phân chia tập Huấn luyện và Kiểm định

Em chia tập full_train thành 80% cho huấn luyện (train_set) và 20% cho kiểm định (val_set):

```
train_len = int(0.8 * len(full_train))
val_len = len(full_train) - train_len
train_set, val_set = random_split(full_train, [train_len, val_len])
```

Liệt kê mã 1.5: Phân chia tập huấn luyện và kiểm định

### 1.2.4 Tạo các DataLoaders

Cuối cùng, em tạo các đối tượng DataLoader để cung cấp dữ liệu theo từng lô cho mô hình, với shuffle=True cho train_loader: Hàm `get_data_loaders` sẽ trả về ba `DataLoader` này.

```
return (
    DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True),
    DataLoader(val_set, batch_size=BATCH_SIZE),
    DataLoader(test_set, batch_size=BATCH_SIZE)
)
```

Liệt kê mã 1.6: Tạo các DataLoaders

# Chương 2

# Xây dựng Mô hình

Chương này em trình bày về kiến trúc mạng nơ-ron tích chập (CNN) mang tên **ConvNet** mà em đã thiết kế bằng PyTorch để phân loại ảnh CIFAR-10.

## 2.1 Em định nghĩa Lớp Mô hình CNN (ConvNet)

Lớp `ConvNet` của em kế thừa từ `torch.nn.Module`. Toàn bộ kiến trúc được định nghĩa trong một `nn.Sequential` để các lớp được thực thi tuần tự.

```
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # --- Khối Tích chập 1 ---
            nn.Conv2d(3, 32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- Khối Tích chập 2 ---
            nn.Conv2d(32, 64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- Khối Tích chập 3 ---
            nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- Bộ Phân loại (Classifier) ---
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 10)
        )

    def forward(self, x):
        return self.model(x)
```

Liệt kê mã 2.1: Lớp ConvNet

### 2.1.1 Các Khối Tích chập (Convolutional Blocks)

Mô hình của em sử dụng ba khối tích chập để trích xuất đặc trưng:

- **Khối 1:** Gồm `Conv2d(3, 32, kernel_size=3, padding=1)`, `ReLU`, và `MaxPool2d(2)`. Khối này chuyển ảnh đầu vào 3 kênh thành 32 bản đồ đặc trưng, mỗi bản đồ có kích thước giảm một nửa (từ 32×32 xuống 16×16).
- **Khối 2:** Gồm `Conv2d(32, 64, kernel_size=3, padding=1)`, `ReLU`, và `MaxPool2d(2)`. Số kênh tăng lên 64, kích thước bản đồ đặc trưng tiếp tục giảm xuống 8×8.
- **Khối 3:** Gồm `Conv2d(64, 128, kernel_size=3, padding=1)`, `ReLU`, và `MaxPool2d(2)`. Số kênh tăng lên 128, kích thước bản đồ đặc trưng cuối cùng là 4×4.

### 2.1.2 Bộ Phân loại (Classifier)

Sau các khối tích chập, bộ phân loại xử lý các đặc trưng đã học:

- `nn.Flatten()`: Chuyển đổi tensor đặc trưng (128 kênh, 4×4) thành một vector phẳng có 128×4×4=2048 phần tử.
- `nn.Linear(2048, 256)` và `nn.ReLU()`: Lớp kết nối đầy đủ đầu tiên ánh xạ 2048 đặc trưng xuống 256 đặc trưng, sau đó qua hàm kích hoạt ReLU.
- `nn.Dropout(0.5)`: Áp dụng dropout với tỷ lệ 0.5 để giảm overfitting.
- `nn.Linear(256, 10)`: Lớp kết nối đầy đủ cuối cùng tạo ra 10 logits, tương ứng với điểm số cho 10 lớp của CIFAR-10.

### 2.1.3 Phương thức forward

Phương thức `forward(self, x)` định nghĩa cách dữ liệu truyền qua mạng:

```
1 def forward(self, x):
2     # Đưa dữ liệu x qua self.model (nn.Sequential) đã định nghĩa
3     return self.model(x)
```

Liệt kê mã 2.2: Phương thức forward

Khi dữ liệu `x` được đưa vào, nó sẽ tuần tự đi qua các lớp trong `self.model` để cho ra kết quả dự đoán.

# Chương 3

# Huấn luyện Mô hình

Sau khi hoàn tất việc chuẩn bị dữ liệu (Chương 1) và xây dựng mô hình mạng nơ-ron tích chập (Chương 2), trong chương này em sẽ trình bày quá trình huấn luyện mô hình CNN mà em đã thiết kế. Quá trình này bao gồm việc khởi tạo mô hình và các thành phần cần thiết, định nghĩa các hàm phục vụ huấn luyện và đánh giá, thiết lập vòng lặp huấn luyện chính, cũng như các kỹ thuật giúp cải thiện chất lượng huấn luyện như lưu mô hình tốt nhất và dừng sớm (early stopping).

## 3.1 Khởi tạo các Thành phần Huấn luyện

Để bắt đầu quá trình huấn luyện, trước tiên em cần khởi tạo mô hình, chọn hàm mất mát phù hợp và thiết lập bộ tối ưu hóa. Đồng thời, em cũng cần chuẩn bị các DataLoader cho dữ liệu huấn luyện, kiểm định và kiểm thử.

### 3.1.1 Mô hình, Hàm mất mát và Bộ tối ưu hóa

Mô hình được khởi tạo từ lớp `ConvNet` (em đã định nghĩa trong Chương 2) và chuyển lên thiết bị tính toán phù hợp (GPU nếu có, nếu không thì dùng CPU). Em sử dụng `CrossEntropyLoss` làm hàm mất mát vì đây là bài toán phân loại đa lớp. Bộ tối ưu hóa em chọn là Adam với tốc độ học đã khai báo từ trước.

```
model = ConvNet().to(DEVICE)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LR, weight_decay=1e-4)
```

Liệt kê mã 3.1: Khởi tạo mô hình, hàm mất mát và optimizer

### 3.1.2 Tải dữ liệu bằng DataLoader

Hàm `get_data_loaders()` được em định nghĩa ở Chương 1 sẽ trả về ba đối tượng `DataLoader` tương ứng với tập huấn luyện, kiểm định và kiểm thử.

```
train_loader, val_loader, test_loader = get_data_loaders()
```

Liệt kê mã 3.2: Tải dữ liệu bằng hàm `get_data_loaders()`

## 3.2 Các Hàm Phục vụ Huấn luyện và Đánh giá

### 3.2.1 Hàm huấn luyện một epoch

Hàm `train_one_epoch()` được dùng để huấn luyện mô hình với toàn bộ dữ liệu trong một epoch. Trong hàm này, em thực hiện lan truyền tiến, tính toán mất mát, lan truyền ngược và cập nhật trọng số. Hàm trả về giá trị mất mát trung bình và độ chính xác trên epoch đó.

```
def train_one_epoch(model, loader, loss_fn, optimizer):
    model.train()
    running_loss, correct, total = 0.0, 0, 0

    for inputs, targets in loader:
        inputs, targets = inputs.to(DEVICE), targets.to(DEVICE)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        correct += (outputs.argmax(1) == targets).sum().item()
        total += targets.size(0)

    avg_loss = running_loss / total
    acc = 100 * correct / total
    return avg_loss, acc
```

Liệt kê mã 3.3: Hàm huấn luyện một epoch

### 3.2.2 Hàm đánh giá mô hình

Hàm `evaluate()` dùng để kiểm tra hiệu suất của mô hình. Hàm này có thể trả về độ chính xác, mất mát, hoặc danh sách nhãn dự đoán và nhãn thực tế nếu cần. Điều này rất tiện khi vẽ ma trận nhầm lẫn hoặc đánh giá cuối cùng trên tập test.

```
def evaluate(model, loader, loss_fn=None, return_preds=False):
    model.eval()
    loss_sum, correct, total = 0.0, 0, 0
    true, pred = [], []

    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(DEVICE), y.to(DEVICE)
            out = model(x)
            preds = out.argmax(dim=1)

            if loss_fn:
                loss_sum += loss_fn(out, y).item() * x.size(0)
                correct += (preds == y).sum().item()
                total += y.size(0)

            if return_preds:
                true.extend(y.cpu().numpy())
                pred.extend(preds.cpu().numpy())

    if loss_fn:
        return loss_sum / total, 100 * correct / total
    return true, pred
```

Liệt kê mã 3.4: Hàm đánh giá mô hình

## 3.3 Vòng lặp Huấn luyện Chính và Lưu Mô hình Tốt nhất

Sau khi có mô hình và dữ liệu sẵn sàng, em tiến hành huấn luyện trong nhiều epoch. Trong mỗi epoch, mô hình sẽ được huấn luyện trên tập huấn luyện và đánh giá trên tập kiểm định. Em ghi lại kết quả vào file log, đồng thời lưu mô hình tốt nhất dựa trên giá trị `val_loss` thấp nhất. Nếu không có cải thiện sau một số epoch liên tiếp, em áp dụng kỹ thuật dừng sớm.

```python
train_loss_vals, val_loss_vals = [], []
train_acc_vals, val_acc_vals = [], []

best_val_loss = float('inf')
patience = 5
counter = 0

with open("training_log.txt", "w") as log_file:
    log_file.write("Epoch\tTrainLoss\tTrainAcc\tValLoss\tValAcc\n")

    for epoch in range(EPOCHS):
        tr_loss, tr_acc = train_one_epoch(model, train_loader,
        ↪  criterion, optimizer)
        val_loss, val_acc = evaluate(model, val_loader, criterion)

        train_loss_vals.append(tr_loss)
        val_loss_vals.append(val_loss)
        train_acc_vals.append(tr_acc)
        val_acc_vals.append(val_acc)

        log_line = f"{epoch+1}\t{tr_loss:.4f}\t{tr_acc:.2f}\t{val_loss:
        ↪  .4f}\t{val_acc:.2f}\n"
        print(log_line.strip())
        log_file.write(log_line)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            counter = 0
            torch.save(model.state_dict(), "best_model.pt")
        else:
            counter += 1
            if counter >= patience:
                log_file.write("Early stopping triggered.\n")
                print("Early stopping triggered.")
                break
```

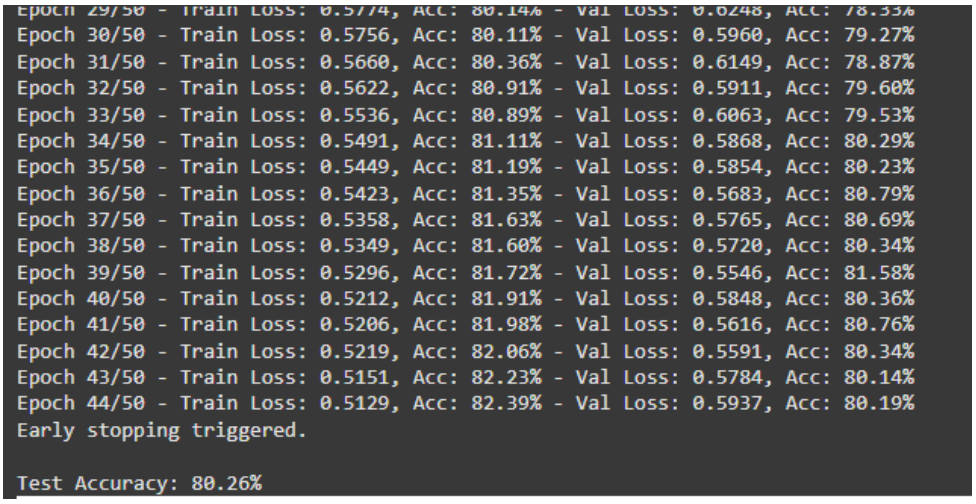
Liệt kê mã 3.5: Vòng lặp huấn luyện chính

### Lý do chọn số epoch là 40 và minh họa cơ chế Early Stopping

Trong quá trình huấn luyện, em thiết lập số vòng lặp tối đa là `EPOCHS = 40`. Con số này không phải tùy ý chọn mà dựa trên:

- Kết quả thử nghiệm sơ bộ với CIFAR-10 cho thấy mô hình bắt đầu hội tụ từ khoảng **epoch 30** trở đi
- Các nghiên cứu phổ biến cũng thường huấn luyện CNN đơn giản trên CIFAR-10 trong khoảng **30–50 epoch** để đạt độ chính xác ổn định,
- Đồng thời, em áp dụng kỹ thuật **early stopping** với `patience = 5`, tức là nếu *validation loss* không được cải thiện trong 5 epoch liên tiếp, quá trình sẽ tự động dừng sớm.

Để kiểm chứng tính hiệu quả của **early stopping**, em thử tăng `EPOCHS = 50` và vẫn giữ nguyên mọi thiết lập khác. Kết quả cho thấy mô hình đã **tự động dừng sớm tại epoch 44**, khi không còn cải thiện về validation loss trong 5 epoch liên tiếp.




Hình 3.1: Kết quả log khi huấn luyện với 50 epoch – hệ thống tự động dừng sớm tại epoch 44.

Việc thử nghiệm này cho thấy: việc chọn `EPOCHS = 40` ban đầu là hợp lý, và cơ chế **early stopping** hoạt động hiệu quả để tránh overfitting, tiết kiệm tài nguyên huấn luyện mà vẫn đảm bảo chọn được mô hình tối ưu.

## 3.4 Tải lại Mô hình Tốt nhất và Đánh giá Trên Tập Test

Sau khi quá trình huấn luyện hoàn tất, em tải lại mô hình có hiệu suất tốt nhất và đánh giá nó trên tập kiểm thử để xác định độ chính xác cuối cùng.

```
model.load_state_dict(torch.load("best_model.pt"))
print("Đã tải lại trọng số từ best_model.pt để đánh giá cuối cùng.")

y_true, y_pred = evaluate(model, test_loader, return_preds=True)
test_acc = 100 * np.mean((np.array(y_true) ==
 ↪  np.array(y_pred)).astype(np.float32))
print(f"\nTest Accuracy: {test_acc:.2f}%")

with open("training_log.txt", "a") as log_file:
    log_file.write(f"\nTest Accuracy: {test_acc:.2f}%\n")
```

Liệt kê mã 3.6: Tải mô hình tốt nhất và đánh giá trên tập test

## 3.5 Trực quan hóa Kết quả Huấn luyện

Cuối cùng, em vẽ ma trận nhầm lẫn và đường cong học tập để trực quan hóa quá trình mô hình học được từ dữ liệu.

```
plot_conf_matrix(y_true, y_pred)
plot_curves(train_loss_vals, val_loss_vals, train_acc_vals,
 ↪  val_acc_vals)
```

Liệt kê mã 3.7: Gọi các hàm vẽ trực quan

Với mô hình đã được huấn luyện kỹ lưỡng, chương tiếp theo sẽ trình bày các kết quả thu được và đánh giá hiệu suất một cách chi tiết trên tập kiểm thử CIFAR-10.

# Chương 4

# Thực nghiệm và Đánh giá

Sau khi hoàn thành quá trình xây dựng và huấn luyện mô hình CNN trên tập dữ liệu CIFAR-10, ở chương này em sẽ trình bày chi tiết quá trình đánh giá mô hình, bao gồm cả cách kiểm tra hiệu suất trên tập kiểm thử và phân tích các kết quả đạt được. Mục tiêu là đánh giá khả năng tổng quát hóa của mô hình và xác định những điểm mạnh, điểm yếu khi áp dụng trên dữ liệu thực tế.

## 4.1 Quy trình đánh giá mô hình

Sau khi hoàn tất quá trình huấn luyện mô hình, em tiến hành đánh giá hiệu quả của mô hình trên tập dữ liệu kiểm thử (test set). Quy trình đánh giá gồm ba bước chính: tải lại mô hình có hiệu suất tốt nhất đã lưu, thực hiện dự đoán trên tập kiểm thử và tính toán độ chính xác tổng thể. Chi tiết từng bước được trình bày dưới đây.

### Bước 1: Tải lại mô hình tốt nhất

Trong quá trình huấn luyện, em đã lưu lại mô hình có kết quả tốt nhất trên tập validation bằng cách sử dụng hàm `torch.save()`. Để đảm bảo việc đánh giá được thực hiện trên mô hình tối ưu nhất, em tải lại trọng số từ tệp `best_model.pt` như sau:

```
1 model.load_state_dict(torch.load("best_model.pt"))
```

Liệt kê mã 4.1: Tải lại mô hình có trọng số tốt nhất.

### Bước 2: Dự đoán trên tập kiểm thử

Sau khi đã nạp mô hình tối ưu, em sử dụng nó để dự đoán nhãn cho các ảnh trong tập kiểm thử. Em gọi hàm `evaluate()` và đặt tham số `return_preds=True` để lấy về cả nhãn thực tế và nhãn mà mô hình dự đoán:

```
y_true, y_pred = evaluate(model, test_loader, loss_fn=None,
↪  return_preds=True)
```

Liệt kê mã 4.2: Dự đoán nhãn của dữ liệu kiểm thử.

Kết quả trả về gồm:

- `y_true`: danh sách các nhãn thật trong tập test.
- `y_pred`: danh sách các nhãn mà mô hình dự đoán.

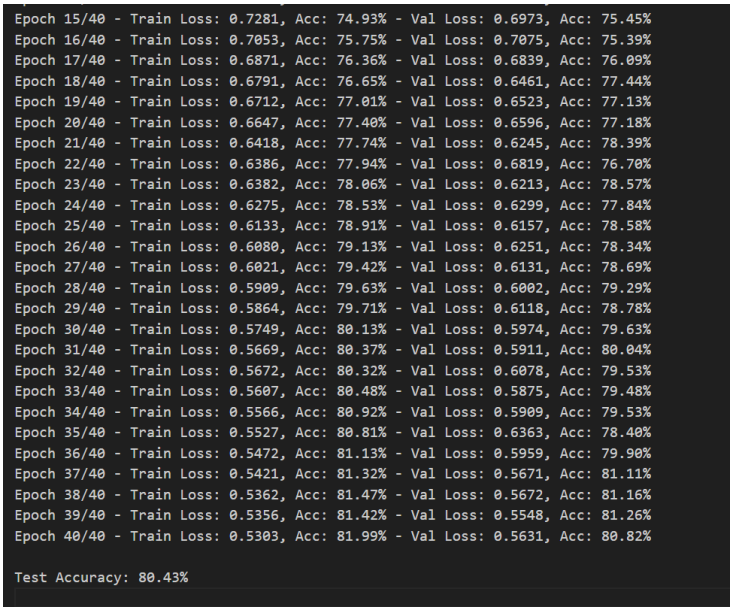
## Bước 3: Tính toán độ chính xác tổng thể

Cuối cùng, em tính toán độ chính xác của mô hình trên tập kiểm thử bằng cách so sánh số lượng dự đoán đúng với tổng số mẫu, và nhân với 100 để tính ra phần trăm:

```
test_acc = 100 * np.mean((np.array(y_true) ==
↪  np.array(y_pred)).astype(np.float32))
```

Liệt kê mã 4.3: Tính toán độ chính xác tổng thể trên tập kiểm thử.

## Kết quả sau khi thực hiện




Hình 4.1: Một phần kết quả sau khi thực hiện chương trình

## Đánh giá độ chính xác và khả năng tổng quát hóa

Giá trị độ chính xác (`test_acc`) là chỉ số quan trọng để đánh giá khả năng tổng quát hóa của mô hình — tức là khả năng áp dụng kiến thức đã học được từ tập huấn luyện sang các dữ liệu mới chưa từng thấy trước đó. Trong thực nghiệm này, mô hình của em đạt được độ chính xác kiểm thử là **80.43%**.

Để đánh giá sâu hơn, em đã so sánh độ chính xác trên tập huấn luyện và tập validation trong suốt quá trình học. Kết quả từ tệp log cho thấy rằng độ chính xác trên tập huấn luyện tăng đều từ 36.99% lên đến 81.99% sau 40 epoch, trong khi độ chính xác validation cũng tăng ổn định từ 48.34% lên đến **81.26%** tại epoch 39 — đây cũng là thời điểm mà mô hình đạt kết quả tốt nhất và được lưu lại.

Điều này cho thấy mô hình học được đặc trưng chung của dữ liệu mà không bị rơi vào tình trạng "học thuộc lòng"(overfitting). Cụ thể:

- Độ chính xác kiểm thử (**80.43%**) gần với độ chính xác validation tại epoch tốt nhất (**81.26%**), điều này chứng minh rằng mô hình không bị mất khả năng tổng quát hóa khi chuyển từ validation sang test.

- Độ chênh lệch giữa accuracy của training (**81.99%**) và validation (**81.26%**) là rất nhỏ ($<1\%$), cho thấy quá trình huấn luyện diễn ra cân bằng, không có dấu hiệu overfitting nghiêm trọng.

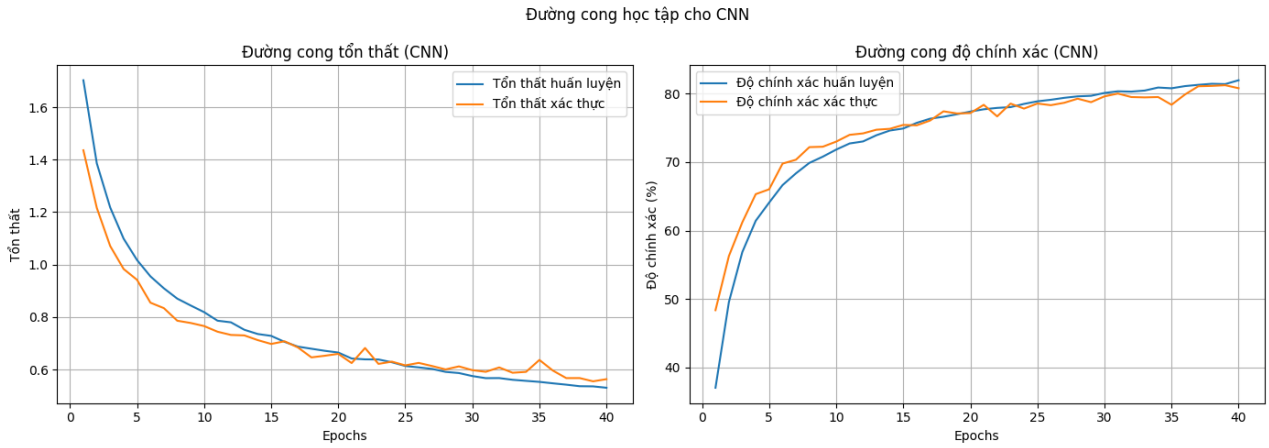
## 4.2 Đồ thị quá trình học (Learning Curves)

### Hàm vẽ đồ thị

```python
def plot_curves(train_loss, val_loss, train_acc, val_acc):
    epochs = range(1, len(train_loss) + 1)
    plt.figure(figsize=(14, 5))

    # Đồ thị Loss
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_loss, label='Train Loss')
    plt.plot(epochs, val_loss, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss Curve')
    plt.legend()
    plt.grid(True)

    # Đồ thị Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_acc, label='Train Accuracy')
    plt.plot(epochs, val_acc, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy Curve')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.savefig("learning_curves.png", dpi=300)
    plt.show()
```

Liệt kê mã 4.4: Hàm vẽ đồ thị Learning Curves.

## Biểu đồ Learning Curves




Hình 4.2: Đường cong Loss và Accuracy theo Epochs

## Phân tích đường cong học tập (Learning Curves)

Từ biểu đồ `learning_curves` (hình 4.2), em rút ra được các nhận xét chi tiết như sau:

- **Biểu đồ tổn thất (Loss curve)** bên trái cho thấy cả tổn thất huấn luyện và tổn thất xác thực đều giảm liên tục theo số epoch. Tổn thất huấn luyện bắt đầu từ khoảng 1.7 và giảm đều xuống còn xấp xỉ 0.53 sau 40 epoch. Tổn thất validation cũng giảm tương tự từ hơn 1.4 xuống khoảng 0.56. Đường cong mượt và không có dao động đột ngột, chứng tỏ mô hình học ổn định, không bị nhiễu hay mất kiểm soát trong quá trình lan truyền ngược (backpropagation).

- **Biểu đồ độ chính xác (Accuracy curve)** bên phải cho thấy mô hình có tốc độ học nhanh trong giai đoạn đầu (epoch 1–10), với độ chính xác validation tăng mạnh từ khoảng 48% lên hơn 70%. Sau đó, tốc độ cải thiện chậm dần và hội tụ ở mức trên 80% từ khoảng epoch 35 trở đi.

- **Khoảng cách giữa hai đường** (train và validation) trên cả hai biểu đồ là rất nhỏ trong toàn bộ quá trình huấn luyện. Điều này cho thấy mô hình học một cách tổng quát, không xảy ra hiện tượng overfitting nghiêm trọng. Tại epoch 39, mô hình đạt độ chính xác validation cao nhất là **81.26%**, trong khi độ chính xác huấn luyện cùng thời điểm là **81.42%**. Khoảng cách chênh lệch chưa đến 0.2% cho thấy mức độ khớp giữa mô hình và dữ liệu thực tế rất tốt.

- **Tổn thất validation thấp nhất** xuất hiện vào khoảng epoch 39 cũng là thời điểm mô hình được chọn làm mô hình tốt nhất (`best_model.pt`). Điều này cho thấy chiến lược lưu mô hình tốt nhất theo loss đã hoạt động hiệu quả.

- **Không có dấu hiệu overfitting hay underfitting**: Nếu mô hình bị overfitting, ta sẽ thấy đường loss validation bắt đầu tăng trong khi training loss tiếp tục giảm

— điều này không xảy ra ở đây. Ngược lại, nếu mô hình underfitting, cả hai đường accuracy sẽ duy trì ở mức thấp — điều này cũng không xảy ra.

- **Kết luận**: Biểu đồ học tập chứng minh rằng mô hình CNN được huấn luyện hiệu quả, có khả năng tổng quát hóa tốt, và quá trình học diễn ra ổn định. Đây là nền tảng vững chắc để tiếp tục cải tiến mô hình bằng các phương pháp phức tạp hơn nếu cần thiết.

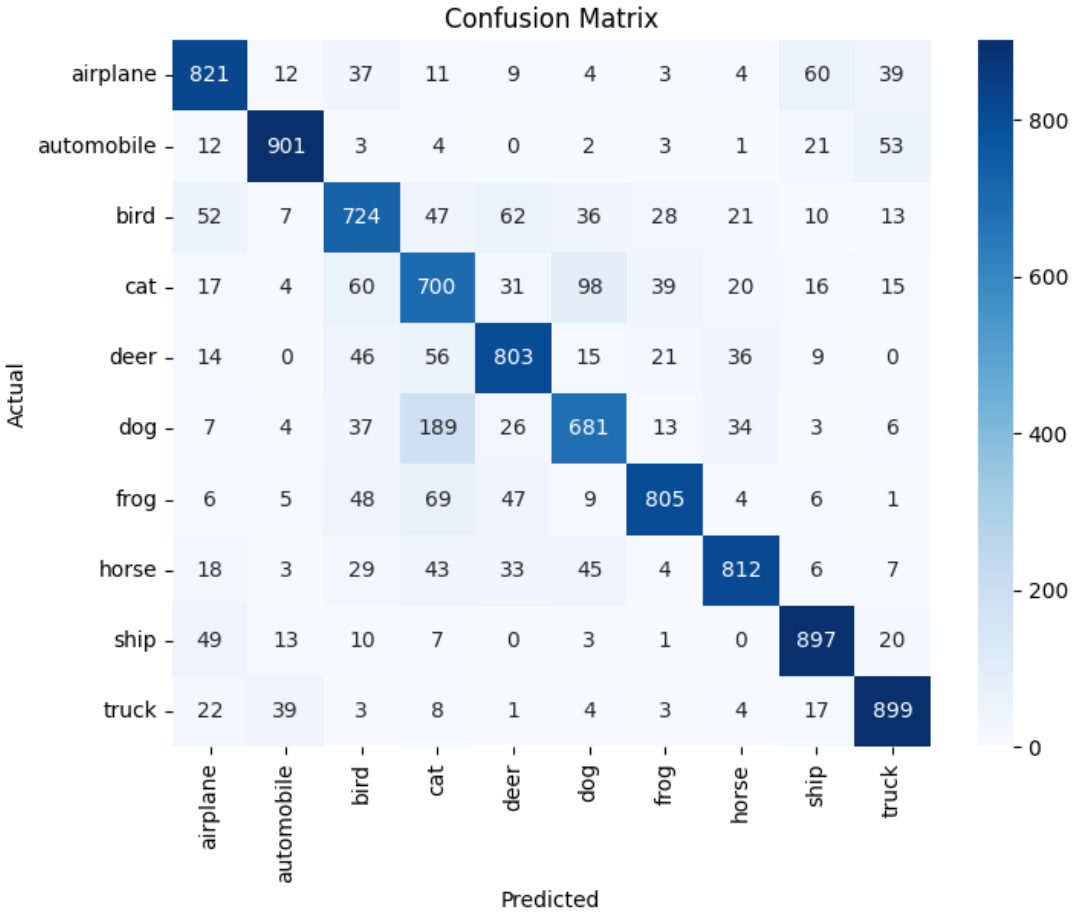
## 4.3 Ma trận nhầm lẫn (Confusion Matrix)

### Hàm vẽ ma trận nhầm lẫn

```
def plot_conf_matrix(y_true, y_pred, class_names):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names,
                yticklabels=class_names)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Confusion Matrix')
    plt.tight_layout()
    plt.savefig("confusion_matrix.png", dpi=300)
    plt.show()
```

Liệt kê mã 4.5: Hàm vẽ ma trận nhầm lẫn

## Hình ảnh Confusion Matrix




Hình 4.3: Ma trận nhầm lẫn trên tập dữ liệu test

## Phân tích ma trận nhầm lẫn (Confusion Matrix)

Từ biểu đồ `confusion_matrix` (hình 4.3), em phân tích chi tiết hiệu suất phân loại của mô hình trên từng lớp như sau:

- **Các lớp được phân loại tốt nhất** là `automobile` (901/1000), `truck` (899/1000), và `ship` (897/1000). Đây đều là các đối tượng có hình dạng đặc trưng rõ rệt, ít bị che khuất, và dễ nhận diện trong tập dữ liệu CIFAR-10. Mô hình có khả năng học được các đặc trưng thị giác đặc biệt của phương tiện giao thông này, chẳng hạn như hình dáng khối, cạnh thẳng, và màu sắc phổ biến.
- **Các lớp dễ bị nhầm lẫn nhất** là nhóm động vật. Ví dụ điển hình là:
  - Lớp `dog` bị mô hình nhầm thành `cat` tới **189** lần – con số cao nhất trong toàn bộ ma trận nhầm lẫn. Ngược lại, `cat` cũng bị nhầm thành `dog` tới **98** lần. Điều này có thể lý giải bởi đặc điểm hình dạng, màu lông và kích thước của hai loài này khá tương đồng, đặc biệt khi ảnh bị thu nhỏ về kích thước $32 \times 32$ như trong CIFAR-10.

  - Lớp `bird` có tỉ lệ nhầm lẫn khá cao với các lớp như `deer` (62 lần), `cat` (47 lần), và `airplane` (52 lần). Việc nhầm `bird` với `airplane` có thể đến từ các ảnh chim đang bay có hình dáng trải cánh dễ gây nhầm lẫn với máy bay.
  - Lớp `frog` bị nhầm thành `cat` (69 lần) và `bird` (48 lần), cho thấy rằng các động vật nhỏ, có màu sắc sẫm và không rõ biên dạng có xu hướng gây khó khăn cho mô hình trong việc phân biệt.

- **Một số lỗi nhầm lẫn ngẫu nhiên** xuất hiện như việc mô hình dự đoán nhầm `deer` thành `dog` (21 lần), hoặc `horse` thành `dog` (45 lần). Những lỗi này có thể đến từ yếu tố tư thế, nền ảnh hoặc góc chụp gây nhiễu.

**Tổng thể ma trận nhầm lẫn cho thấy** mô hình hoạt động khá tốt với các lớp có đặc trưng hình học và màu sắc rõ ràng (phương tiện giao thông), nhưng còn gặp khó khăn trong việc phân biệt các loài động vật – nhóm có hình dạng mềm, nhiều tư thế, và dễ bị che khuất.

## 4.4 Độ chính xác theo từng lớp (Class-wise Accuracy)

Độ chính xác theo từng lớp được tính theo công thức:

$$\text{Accuracy lớp i} = \frac{\text{số mẫu dự đoán đúng lớp i}}{\text{tổng số mẫu lớp i}} \times 100\%$$

| **Class** | **Correct / Total** | **Accuracy** |
|---|---|---|
| airplane | 821 / 1000 | **82.1%** |
| automobile | 901 / 1000 | **90.1%** |
| bird | 724 / 1000 | **72.4%** |
| cat | 700 / 1000 | **70.0%** |
| deer | 803 / 1000 | **80.3%** |
| dog | 681 / 1000 | **68.1%** |
| frog | 805 / 1000 | **80.5%** |
| horse | 812 / 1000 | **81.2%** |
| ship | 897 / 1000 | **89.7%** |
| truck | 899 / 1000 | **89.9%** |

Bảng 4.1: Accuracy per class

Nhóm các phương tiện như `automobile`, `ship`, `truck` có độ chính xác rất cao, trên 89%, cho thấy mô hình dễ nhận diện chúng nhờ đặc điểm hình ảnh rõ ràng.
Các lớp động vật như `dog`, `cat`, `bird` có độ chính xác thấp hơn, do có nhiều đặc điểm thị giác tương đồng dẫn đến nhầm lẫn.

## 4.5 Đánh giá chung

Mô hình CNN đơn giản em xây dựng đã đạt được độ chính xác 80.43% trên tập test, thể hiện hiệu quả học tập ổn định và khả năng tổng quát tốt. Việc sử dụng các kỹ thuật như dropout, augmentation và early stopping đã giúp mô hình tránh được hiện tượng overfitting, giữ được hiệu suất trên dữ liệu chưa từng thấy. Mặc dù vậy, mô hình vẫn gặp khó khăn trong việc phân biệt các lớp động vật có đặc điểm hình ảnh tương đồng, gây ra một số nhầm lẫn nhất định. Nhìn chung, kết quả thu được cho thấy mô hình ConvNet này là một nền tảng vững chắc, phù hợp để tiếp tục phát triển và nâng cao trong các nghiên cứu tiếp theo trên bộ dữ liệu CIFAR-10.