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:	**Chu Tuyết Nhi**
Mã sinh viên:	**B23DCCE075**
Lớp:	**D23CQCEO6-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

Báo cáo này thực hiện yêu cầu của Bài tập lớn 2 môn Ngôn ngữ lập trình Python, tập trung vào bài toán phân loại hình ảnh trên bộ dữ liệu CIFAR-10. Mục tiêu là xây dựng, huấn luyện và đánh giá một mô hình mạng nơ-ron tích chập (Convolutional Neural Network - CNN) sử dụng thư viện PyTorch.
Bộ dữ liệu CIFAR-10 gồm 60,000 ảnh màu 32x32 pixel, chia thành 10 lớp. Quá trình thực hiện sử dụng Python, PyTorch, torchvision, Matplotlib, Seaborn, NumPy và Scikit-learn.

Nội dung báo cáo bao gồm:

Chương 1. Chuẩn bị Dữ liệu: Khai báo thư viện, tham số, tải và tiền xử lý dữ liệu CIFAR-10, bao gồm transforms, chia tập dữ liệu và tạo DataLoaders.

Chương 2. Xây dựng Mô hình: Trình bày kiến trúc mạng CNN, bao gồm định nghĩa lớp mô hình, các khối tích chập, bộ phân loại và phương thức `forward`.

Chương 3. Huấn luyện Mô hình: Mô tả thiết lập huấn luyện (mô hình, hàm mất mát, optimizer), điều chỉnh tốc độ học, dừng sớm, các hàm huấn luyện, đánh giá mỗi epoch và vòng lặp huấn luyện chính.

Chương 4. Thực nghiệm và Đánh giá: Phân tích kết quả trên tập kiểm thử, bao gồm độ chính xác tổng thể, đồ thị quá trình học, ma trận nhầm lẫn, độ chính xác theo lớp và nhận xét chung.

Chương 1

Chuẩn bị Dữ liệu

Phần này em trình bày về quá trình chuẩn bị dữ liệu, bao gồm việc khai báo các thư viện cần thiết, thiết lập tham số và các bước tải, tiền xử lý bộ dữ liệu CIFAR-10.

1.1 Khai báo Thư viện và Tham số

Trước tiên, em tiến hành import các thư viện cần thiết cho bài.

```
# Xây dựng và huấn luyện mô hình học sâu
import torch
import torch.nn as nn
import torch.optim as optim
# Xử lý và biến đổi dữ liệu ảnh
import torchvision
import torchvision.transforms as transforms
# Vẽ biểu đồ và trực quan hóa kết quả
import matplotlib.pyplot as plt
import seaborn as sns
# Tính toán số học
import numpy as np
# Đánh giá mô hình phân loại
from sklearn.metrics import confusion_matrix
```

Liệt kê mã 1.1: Khai báo các thư viện cần thiết

Tiếp theo, em định nghĩa các tham số quan trọng sẽ được sử dụng trong suốt quá trình huấn luyện và xử lý dữ liệu:

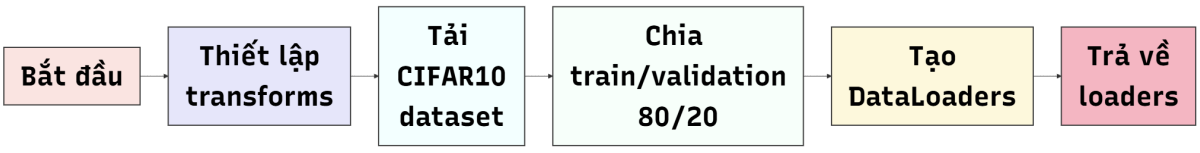
```python
BATCH, LR, EPOCHS = 64, 1e-3, 10
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
CLASSES = ('airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

Liệt kê mã 1.2: Định nghĩa các tham số chính

1. `BATCH`: Kích thước lô dữ liệu cho mỗi lần cập nhật trọng số, ở đây em chọn là 64.
2. `LR`: Tốc độ học (learning rate) ban đầu cho thuật toán tối ưu Adam, được đặt là 1×10^{-3}.
3. `EPOCHS`: Số lượng chu kỳ huấn luyện tối đa, em đặt là 50, vì đây là giá trị phổ biến thường được dùng cho bộ dữ liệu như đề bài. Tuy nhiên, quá trình huấn luyện có thể dừng sớm hơn nhờ cơ chế Early Stopping.
4. `DEVICE`: Thiết bị tính toán (`cuda` nếu có GPU, ngược lại là `cpu`). Việc này giúp tận dụng khả năng tính toán song song của GPU nếu có.
5. `CLASSES`: Danh sách tên các lớp trong bộ dữ liệu CIFAR-10.

1.2 Tải và Tiền xử lý Dữ liệu

Em xây dựng hàm `load_data()` để thực hiện việc tải và tiền xử lý dữ liệu.
Các bước chính trong hàm `load_data()`:



Hình 1.1: Lưu đồ hàm load_data()

1.2.1 Định nghĩa phép biến đổi (Transforms):

```
def load_data():
    mean, std = (0.485, 0.456, 0.406), (0.229, 0.224, 0.225)
    # Giá trị trung bình và độ lệch chuẩn của ImageNet
    train_tf = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.RandomCrop(32, padding=4),
        transforms.ColorJitter(0.2, 0.2, 0.2, 0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])
    test_tf = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])
    #...
```

Liệt kê mã 1.3: Đoạn code biến đổi cho các tập

Biến đổi tập Train `train_tf`: Chuỗi các phép biến đổi áp dụng cho tập huấn luyện. Em sử dụng các kỹ thuật tăng cường dữ liệu để làm phong phú thêm dữ liệu huấn luyện và giúp mô hình tổng quát hóa tốt hơn:

1. `transforms.RandomHorizontalFlip()`: Lật ảnh ngẫu nhiên theo chiều ngang.
2. `transforms.RandomRotation(10)`: Xoay ảnh ngẫu nhiên một góc trong khoảng ± 10 độ.
3. `transforms.RandomCrop(32, padding=4)`: Cắt ngẫu nhiên một vùng ảnh kích thước 32x32 sau khi đệm (padding) 4 pixel vào mỗi cạnh.
4. `transforms.ColorJitter(0.2, 0.2, 0.2, 0.1)`: Thay đổi ngẫu nhiên độ sáng, độ tương phản, độ bão hòa và sắc độ của ảnh.
5. `transforms.ToTensor()`: Chuyển ảnh PIL hoặc NumPy array sang dạng Tensor của PyTorch.
6. `transforms.Normalize(mean, std)`: Chuẩn hóa giá trị pixel của ảnh bằng cách trừ đi trung bình (`mean`) và chia cho độ lệch chuẩn (`std`) theo từng kênh màu. Em sử dụng giá trị `mean` và `std` phổ biến từ bộ ImageNet.

Biến đổi tập Test `test_tf`: Chuỗi các phép biến đổi cho tập kiểm định (validation) và tập kiểm thử (test). Đối với các tập này, em chỉ thực hiện `ToTensor` và `Normalize` để đảm bảo tính nhất quán và không làm rò rỉ thông tin từ augmentation vào quá trình đánh giá.

1.2.2 Tải bộ dữ liệu CIFAR-10:

`train_full`: Tải toàn bộ tập huấn luyện của CIFAR-10, áp dụng `train_tf`.
`test_set`: Tải tập kiểm thử của CIFAR-10, áp dụng `test_tf`.

```
def load_data():
    #...
    train_full = torchvision.datasets.CIFAR10(root='./data', train=True,
    ↪  download=True, transform=train_tf)
    test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
    ↪  download=True, transform=test_tf)
    #...
```

Liệt kê mã 1.4: Đoạn code tải bộ dữ liệu CIFAR-10

1.2.3 Phân chia tập Huấn luyện và Kiểm định:

Em chia tập `train_full` thành hai phần: `train_set` (80%) và `val_set` (20%) để sử dụng cho việc huấn luyện và kiểm định mô hình trong quá trình training.

```
def load_data():
    #...
    train_size = int(0.8 * len(train_full))
    train_set, val_set = torch.utils.data.random_split(train_full,
    ↪  [train_size, len(train_full) - train_size])
    val_set.dataset = torchvision.datasets.CIFAR10(root='./data',
    ↪  train=True, download=False, transform=test_tf)
    #...
```

Liệt kê mã 1.5: Đoạn code Phân chia tập Huấn luyện và Kiểm định

Đặc biệt, sau khi chia, em gán lại `transform` cho `val_set` bằng `test_tf`. Điều này đảm bảo rằng tập kiểm định được xử lý giống như tập kiểm thử, không bị ảnh hưởng bởi các phép tăng cường dữ liệu của tập huấn luyện, giúp đánh giá khách quan hơn hiệu suất của mô hình trên dữ liệu chưa từng thấy.

1.2.4 Tạo DataLoaders:

```
def load_data():
    #...
    loader = lambda ds, shuffle: torch.utils.data.DataLoader(ds,
    ↪  batch_size=BATCH, shuffle=shuffle, num_workers=2)
    return loader(train_set, True), loader(val_set, False),
    ↪  loader(test_set, False)
    #...
```

Liệt kê mã 1.6: Đoạn code Tạo DataLoaders

Một hàm `loader` lambda được định nghĩa để tạo `torch.utils.data.DataLoader` một cách tiện lợi.

`DataLoader` giúp chia dữ liệu thành các lô (batches), xáo trộn dữ liệu (shuffling) cho tập huấn luyện, và tải dữ liệu song song bằng nhiều `workers`.

Hàm trả về ba `DataLoader`: một cho tập huấn luyện (`shuffle=True`), một cho tập kiểm định (`shuffle=False`), và một cho tập kiểm thử (`shuffle=False`).

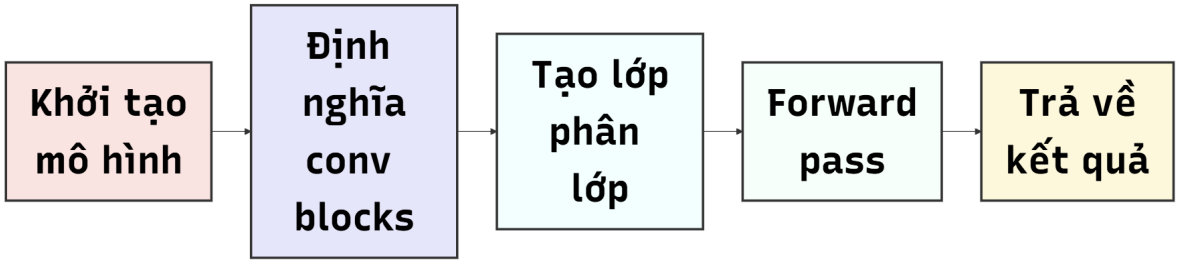
Chương 2

Xây dựng Mô hình

Chương này trình bày kiến trúc mạng CNN (Convolutional Neural Network) được em sử dụng để phân loại ảnh từ bộ dữ liệu CIFAR-10.

2.1 Định nghĩa Lớp Mô hình CNN

Em định nghĩa lớp `CNN` kế thừa từ `nn.Module` của PyTorch. Kiến trúc này bao gồm các khối tích chập được thiết kế tùy chỉnh và một bộ phân loại dựa trên các lớp kết nối đầy đủ.



Hình 2.1: Lưu đồ mô hình CNN

2.1.1 Khởi tạo Mô hình và Khối Tích chập Cơ bản

Lớp `CNN` được khởi tạo với hai tham số chính: `num_classes` (số lớp đầu ra, mặc định là 10) và `dropout` (tỷ lệ dropout cho các lớp trong bộ phân loại, mặc định là 0.5). Bên trong hàm khởi tạo `__init__`, em định nghĩa một hàm tiện ích là `conv_block(in_c, out_c)`. Mục đích của hàm này là đóng gói một chuỗi các thao tác lặp lại thường thấy trong các mạng CNN để tạo ra một khối tích chập có khả năng trích xuất đặc trưng và điều chuẩn.

```python
class CNN(nn.Module):
    def __init__(self, num_classes=10, dropout=0.5):
        super().__init__()
        # Định nghĩa một khối tích chập cơ bản (conv_block)
        def conv_block(in_c, out_c): return nn.Sequential(
            nn.Conv2d(in_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c), nn.ReLU(inplace=True),
            nn.Conv2d(out_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c), nn.ReLU(inplace=True),
            nn.MaxPool2d(2), nn.Dropout2d(0.25)
        )
        # ... (Các phần khác sẽ được trình bày ở các mục sau)
```

Liệt kê mã 2.1: Định nghĩa lớp `CNN` và hàm tiện ích `conv_block`

Khối `conv_block(in_c, out_c)` gồm:

1. Hai lớp `Conv2d` (3×3, padding=1), mỗi lớp kèm `BatchNorm2d` và `ReLU(inplace=True)` để trích xuất đặc trưng, ổn định gradient, tăng tính phi tuyến và hiệu quả huấn luyện. Padding giữ nguyên kích thước không gian, còn `ReLU` giúp tránh hiện tượng mất gradient.

2. Một lớp `MaxPool2d(2)` để giảm nửa kích thước không gian, giảm tính toán và tăng khả năng bất biến với dịch chuyển nhỏ.

3. Một lớp `Dropout2d(0.25)` để bỏ ngẫu nhiên 25% kênh trong quá trình huấn luyện nhằm chống overfitting.

Kết quả: Đặc trưng được học tốt hơn, số kênh có thể thay đổi từ `in_c` sang `out_c`, kích thước giảm và mô hình được điều chuẩn hiệu quả hơn.

2.1.2 Các Khối Tích chập Chính của Mạng

Tiếp theo, em sử dụng hàm `conv_block` để xây dựng ba khối tích chập chính: `self.conv1`, `self.conv2`, và `self.conv3`.

```python
class CNN(nn.Module):
    #...
        self.conv1 = conv_block(3, 32)
        self.conv2 = conv_block(32, 64)
        self.conv3 = conv_block(64, 128)
    # ...
```

Liệt kê mã 2.2: Khởi tạo các khối tích chập chính trong `CNN`

Mục đích của việc xếp chồng ba khối này là để mô hình học các đặc trưng ngày càng phức tạp và trừu tượng hơn, đồng thời giảm dần kích thước không gian:

1. `self.conv1 = conv_block(3, 32)`: Chuyển đổi ảnh đầu vào 3 kênh (RGB) thành 32 kênh đặc trưng. *Kết quả*: Kích thước không gian giảm từ 32×32 xuống 16×16.
2. `self.conv2 = conv_block(32, 64)`: Tăng số kênh đặc trưng từ 32 lên 64. *Kết quả*: Kích thước không gian giảm xuống 8×8.
3. `self.conv3 = conv_block(64, 128)`: Tăng số kênh đặc trưng từ 64 lên 128. *Kết quả*: Kích thước không gian giảm xuống 4×4.

Lý do cho kiến trúc này là nó tuân theo nguyên tắc phổ biến của các mạng CNN hiện đại: tăng độ sâu (số kênh) trong khi giảm độ phân giải không gian. *Kết quả cuối cùng* của phần tích chập là 128 bản đồ đặc trưng, mỗi bản đồ có kích thước 4×4.

2.1.3 Bộ Phân loại (Classifier)

Sau khi trích xuất đặc trưng, `self.classifier` có *mục đích* ánh xạ các đặc trưng này sang không gian của các lớp đầu ra để thực hiện việc phân loại.

```
class CNN(nn.Module):
    # ...

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 512), nn.BatchNorm1d(512),
            nn.ReLU(inplace=True), nn.Dropout(dropout),
            nn.Linear(512, 256), nn.BatchNorm1d(256),
            nn.ReLU(inplace=True), nn.Dropout(dropout),
            nn.Linear(256, num_classes)
        )
    # ...
```

Liệt kê mã 2.3: Khởi tạo bộ phân loại trong `CNN`

`self.classifier` gồm:

1. `nn.Flatten()` để chuyển tensor đặc trưng đa chiều ($128 \times 4 \times 4$) thành vector phẳng 2048 chiều, chuẩn bị cho các lớp kết nối đầy đủ.
2. Hai lớp fully connected giảm kích thước từ 2048 xuống 512 rồi 256, mỗi lớp đi kèm `BatchNorm1d`, `ReLU(inplace=True)` và `Dropout` nhằm tăng tính phi tuyến, ổn định và tránh overfitting.
3. Lớp cuối cùng `nn.Linear(256, num_classes)` tạo logits cho các lớp cần phân loại.

Kết quả: Bộ phân loại chuyển đặc trưng thành dự đoán lớp chính xác và hiệu quả.

2.1.4 Phương thức `forward`

Phương thức `forward(self, x)` xác định luồng dữ liệu đi qua mạng.

```
class CNN(nn.Module):
#     # ... (__init__ đã được định nghĩa đầy đủ ở các mục trên)

    def forward(self, x):
        return self.classifier(self.conv3(self.conv2(self.conv1(x))))
```

Liệt kê mã 2.4: Định nghĩa phương thức `forward` trong `CNN`

Dữ liệu `x` được truyền tuần tự qua các khối `self.conv1`, `self.conv2`, `self.conv3`, rồi vào `self.classifier` để tạo ra logits. *Kết quả* là trả về tensor chứa điểm số cho từng lớp.

Tóm lại, kiến trúc CNN này được em xây dựng dựa trên sự kết hợp của các khối `conv_block` (bao gồm `Conv2d`, `BatchNorm2d`, `ReLU`, `MaxPool2d`, `Dropout2d`) và một bộ phân loại MLP (gồm các lớp `Linear`, `BatchNorm1d`, `ReLU`, `Dropout`). *Mục đích chính* của thiết kế này là tạo ra một mô hình có khả năng trích xuất đặc trưng phân cấp hiệu quả từ ảnh CIFAR-10 và thực hiện phân loại chính xác. *Lý do* sử dụng các kỹ thuật như BatchNorm và Dropout xuyên suốt mô hình là để cải thiện sự ổn định của quá trình huấn luyện và tăng khả năng tổng quát hóa, từ đó giảm thiểu overfitting.

Chương 3

Huấn luyện Mô hình

Sau khi đã chuẩn bị dữ liệu và xây dựng kiến trúc CNN, chương này em sẽ trình bày chi tiết về quá trình thiết lập và thực hiện huấn luyện mô hình.

3.1 Thiết lập Huấn luyện

Trước khi bắt đầu vòng lặp huấn luyện, em cần khởi tạo các thành phần quan trọng như mô hình, hàm mất mát, bộ tối ưu hóa, bộ điều chỉnh tốc độ học và cơ chế dừng sớm.

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

Đầu tiên, em khởi tạo mô hình `CNN` đã định nghĩa ở chương trước và chuyển mô hình lên thiết bị tính toán (`DEVICE`). Kế tiếp, em chọn hàm mất mát `CrossEntropyLoss` vì đây là hàm mất mát tiêu chuẩn cho bài toán phân loại đa lớp. Cuối cùng, em sử dụng bộ tối ưu hóa `Adam` với tốc độ học (`LR`) đã định nghĩa và một giá trị `weight_decay` nhỏ (1×10^{-4}) để giúp điều chuẩn, giảm thiểu overfitting.

```
if __name__ == '__main__':
train_loader, val_loader, test_loader = load_data()
model = CNN().to(DEVICE)
loss_fn = 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 bộ tối ưu hóa

3.1.2 Bộ điều chỉnh Tốc độ học (Learning Rate Scheduler)

Để cải thiện quá trình hội tụ, em sử dụng `StepLR` từ `torch.optim.lr_scheduler`. Bộ điều chỉnh này sẽ giảm tốc độ học đi một hệ số `gamma` (ở đây là 0.5) sau một số `step_size` epochs nhất định (ở đây là 10). Việc này giúp mô hình tinh chỉnh tốt hơn khi gần tới điểm tối ưu.

```
# ... (tiếp theo phần khởi tạo optimizer)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10,
↪  gamma=0.5)
```

Liệt kê mã 3.2: Khởi tạo bộ điều chỉnh tốc độ học

3.1.3 Cơ chế Dừng sớm (Early Stopping)

Em định nghĩa lớp `EarlyStopping` để theo dõi giá trị mất mát trên tập kiểm định (`val_loss`). Nếu `val_loss` không cải thiện (giảm đi ít nhất `min_delta`) trong một số lượng `patience` epochs liên tiếp, quá trình huấn luyện sẽ dừng lại. Điều này giúp tránh lãng phí thời gian huấn luyện và ngăn mô hình bị overfitting quá mức.

```
class EarlyStopping:
    def __init__(self, patience=7, min_delta=1e-3):
        self.patience, self.min_delta = patience, min_delta
        self.counter, self.best_loss, self.early_stop = 0, float('inf'),
        ↪  False
    def __call__(self, val_loss):
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss, self.counter = val_loss, 0
        else:
            self.counter += 1
            if self.counter >= self.patience: self.early_stop = True

# ... (trong khối main)
early_stop = EarlyStopping()
```

Liệt kê mã 3.3: Định nghĩa và khởi tạo cơ chế Dừng sớm

`patience=7`: Quá trình huấn luyện sẽ dừng nếu không có cải thiện sau 7 epochs.

`min_delta=1e-3`: Mức cải thiện tối thiểu được coi là có ý nghĩa.

3.2 Hàm Huấn luyện và Đánh giá trong Epoch

Em xây dựng hai hàm chính: `train_epoch` để thực hiện một epoch huấn luyện và `run` để đánh giá mô hình trên tập kiểm định hoặc tập kiểm thử.

3.2.1 Hàm `train_epoch`

Hàm này thực hiện một lượt huấn luyện đầy đủ trên toàn bộ dữ liệu của `loader` (trong trường hợp này là `train_loader`).

```python
def train_epoch(model, loader, loss_fn, optim):
    model.train() # Đặt mô hình ở chế độ huấn luyện
    total_loss, correct, total = 0, 0, 0
    for x, y in loader: # Lặp qua từng batch dữ liệu
        x, y = x.to(DEVICE), y.to(DEVICE) # Chuyển dữ liệu lên DEVICE
        optim.zero_grad() # Xóa gradient cũ
        out = model(x) # Lan truyền tiến (forward pass)
        loss = loss_fn(out, y) # Tính toán loss
        loss.backward() # Lan truyền ngược để tính gradient
        optim.step() # Cập nhật trọng số mô hình
        total_loss += loss.item() # Cộng dồn loss
        correct += (out.argmax(1) == y).sum().item() # Đếm số dự đoán đúng
        total += y.size(0) # Cộng dồn tổng số mẫu
    return total_loss / len(loader), 100 * correct / total # Trả về loss trung bình và độ chính xác
```

Liệt kê mã 3.4: Hàm huấn luyện một epoch

Các bước chính trong `train_epoch`:

1. `model.train()`: Chuyển mô hình sang chế độ huấn luyện, kích hoạt các lớp như Dropout, BatchNorm theo đúng cách.
2. Với mỗi lô dữ liệu (`x, y`):
   - Chuyển dữ liệu lên `DEVICE`.
   - `optim.zero_grad()`: Reset gradient của các tham số mô hình.
   - `out = model(x)`: Đưa dữ liệu qua mô hình để nhận dự đoán.
   - `loss = loss_fn(out, y)`: Tính toán hàm mất mát.
   - `loss.backward()`: Tính toán gradient dựa trên loss.
   - `optim.step()`: Cập nhật trọng số của mô hình dựa trên gradient.
   - Tính toán và cộng dồn `total_loss`, `correct` (số lượng dự đoán đúng) và `total` (tổng số mẫu).
3. Trả về giá trị loss trung bình và độ chính xác (accuracy) trên tập huấn luyện trong epoch đó.

3.2.2 Hàm `run`

Hàm `run` được thiết kế linh hoạt để đánh giá mô hình trên một `loader` (ví dụ: `val_loader`, `test_loader`). Nó có thể trả về loss và accuracy, hoặc chỉ các dự đoán và nhãn thật.

```python
def run(model, loader, loss_fn=None, return_preds=False):
    model.eval() # Đặt mô hình ở chế độ đánh giá
    total_loss, correct, total = 0, 0, 0
    y_true, y_pred = [], []
    with torch.no_grad(): # Không tính toán gradient trong quá trình đánh giá
        for x, y in loader:
            x, y = x.to(DEVICE), y.to(DEVICE)
            out = model(x)
            preds = out.argmax(1) # Lấy chỉ số của lớp có xác suất cao nhất
            if loss_fn: # Nếu cung cấp loss_fn, tính loss và accuracy
                total_loss += loss_fn(out, y).item()
                correct += (preds == y).sum().item()
                total += y.size(0)
            if return_preds: # Nếu cần trả về dự đoán
                y_true.extend(y.cpu().numpy())
                y_pred.extend(preds.cpu().numpy())
    # Trả về các giá trị tùy theo tham số đầu vào
    if return_preds and not loss_fn: return y_true, y_pred
    if loss_fn and not return_preds:
        return total_loss / len(loader), 100 * correct / total
    if loss_fn and return_preds:
        return total_loss / len(loader), 100 * correct / total, y_true, y_pred
```

Liệt kê mã 3.5: Hàm đánh giá mô hình hoặc lấy dự đoán

Các điểm chính trong hàm `run`:

1. `model.eval()`: Chuyển mô hình sang chế độ đánh giá. Lớp Dropout sẽ bị vô hiệu hóa, BatchNorm sẽ sử dụng các giá trị trung bình và phương sai đã học được.

2. `with torch.no_grad()`: Vô hiệu hóa việc tính toán gradient, giúp tiết kiệm bộ nhớ và tăng tốc độ xử lý vì không cần cho bước backward.

3. Với mỗi lô dữ liệu, mô hình đưa ra dự đoán `preds`.

4. Nếu `loss_fn` được cung cấp, hàm sẽ tính toán và trả về loss trung bình và độ chính xác. Đây là trường hợp em sử dụng để đánh giá trên tập kiểm định (`val_loader`) sau mỗi epoch.

5. Nếu `return_preds` là `True`, hàm sẽ trả về danh sách các nhãn thật (`y_true`) và nhãn dự đoán (`y_pred`). Em sử dụng tùy chọn này khi đánh giá trên tập kiểm thử (`test_loader`) để vẽ ma trận nhầm lẫn.

3.3 Vòng lặp Huấn luyện Chính

Đây là nơi quá trình huấn luyện thực sự diễn ra qua nhiều epochs.

```
# if __name__ == '__main__':
#     # ... (khởi tạo model, loss_fn, optimizer, scheduler, early_stop, loaders)
train_loss, val_loss, train_acc, val_acc = [], [], [], []
best_val_acc = 0

for epoch in range(EPOCHS):
    # Huấn luyện mô hình trên tập train
    tr_loss, tr_acc = train_epoch(model, train_loader, loss_fn, optimizer)
    # Đánh giá mô hình trên tập validation
    va_loss, va_acc = run(model, val_loader, loss_fn)
    # Cập nhật learning rate
    scheduler.step()

    # Lưu lại mô hình tốt nhất dựa trên validation accuracy
    if va_acc > best_val_acc:
        best_val_acc = va_acc
        torch.save(model.state_dict(), 'best_model.pth') # Lưu trọng số

    # Lưu lại các chỉ số để vẽ đồ thị sau này
    train_loss.append(tr_loss); val_loss.append(va_loss)
    train_acc.append(tr_acc); val_acc.append(va_acc)

    print(f"Epoch {epoch+1}/{EPOCHS} - Train: {tr_loss:.4f}/{tr_acc:.2f}% - Val: {va_loss:.4f}/{va_acc:.2f}%")

    # Kiểm tra điều kiện dừng sớm
    early_stop(va_loss)
    if early_stop.early_stop:
        print(f"\nEarly stopping at epoch {epoch+1}")
        break
```

Liệt kê mã 3.6: Vòng lặp huấn luyện chính

Trong mỗi epoch:

1. Gọi `train_epoch` để huấn luyện mô hình trên `train_loader`, nhận về loss và accuracy trên tập train.

2. Gọi `run` để đánh giá mô hình trên `val_loader`, nhận về loss và accuracy trên tập validation.

3. `scheduler.step()`: Cập nhật tốc độ học nếu cần.

4. Nếu độ chính xác trên tập validation (`va_acc`) của epoch hiện tại cao hơn `best_val_acc` đã lưu, thì cập nhật `best_val_acc` và lưu lại trọng số của mô hình hiện tại vào file `'best_model.pth'`.

5. Lưu các giá trị loss và accuracy của cả tập train và validation để tiện cho việc trực quan hóa sau này.

6. In ra thông tin của epoch hiện tại.

7. Gọi `early_stop(va_loss)` để kiểm tra điều kiện dừng sớm. Nếu điều kiện được thỏa mãn, vòng lặp sẽ kết thúc.

Kết thúc chương này, em đã có một mô hình được huấn luyện và trọng số tốt nhất đã được lưu lại dựa trên hiệu suất trên tập kiểm định.

Chương 4

Thực nghiệm và Đánh giá

Sau khi hoàn thành quá trình huấn luyện mô hình ở chương trước, ở chương này, em sẽ tiến hành tải lại mô hình với trọng số tốt nhất đã lưu và đánh giá hiệu suất của nó trên tập dữ liệu kiểm thử (test set). Các kết quả thực nghiệm thu được sẽ được trình bày và phân tích chi tiết.

4.1 Quá trình Đánh giá

Để đánh giá mô hình một cách khách quan, em thực hiện các bước sau:

1. **Tải lại mô hình tốt nhất:** Em tải lại trọng số của mô hình đã cho kết quả tốt nhất trên tập kiểm định trong quá trình huấn luyện. Trọng số này được lưu trong file `best_model.pth`.

```
# if __name__ == '__main__':
#     # model = CNN().to(DEVICE) # Khởi tạo lại kiến trúc mô hình
model.load_state_dict(torch.load('best_model.pth'))
```

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

2. **Thực hiện dự đoán trên tập kiểm thử:** Em sử dụng hàm `run` (đã được mô tả chi tiết ở Chương 3, mục 3.5) để cho mô hình dự đoán trên toàn bộ tập `test_loader`. Hàm này trả về danh sách các nhãn thật (`y_true`) và nhãn dự đoán (`y_pred`).

```
# if __name__ == '__main__':
#     # train_loader, val_loader, test_loader = load_data() # Đảm bảo
    ↪ test_loader
y_true, y_pred = run(model, test_loader, return_preds=True)
```

Liệt kê mã 4.2: Lấy dự đoán trên tập kiểm thử

3. **Tính toán các chỉ số đánh giá:** Từ `y_true` và `y_pred`, em tính toán độ chính xác tổng thể, ma trận nhầm lẫn và độ chính xác theo từng lớp.

4.2 Kết quả Thực nghiệm

Dưới đây là các kết quả chi tiết em thu được sau quá trình đánh giá.

4.2.1 Độ chính xác Tổng thể

Độ chính xác là một trong những chỉ số quan trọng nhất để đánh giá hiệu suất của mô hình phân loại.

```
# if __name__ == '__main__':
#     # y_true, y_pred đã có từ bước trước
#     # best_val_acc là biến lưu độ chính xác tốt nhất trên tập validation từ quá trình huấn luyện
test_acc = 100 * np.mean(np.array(y_pred) == np.array(y_true))
# print(f"\nBest Val Acc: {best_val_acc:.2f}%, Test Acc: {test_acc:.2f}%")
```

Liệt kê mã 4.3: Tính toán độ chính xác trên tập test

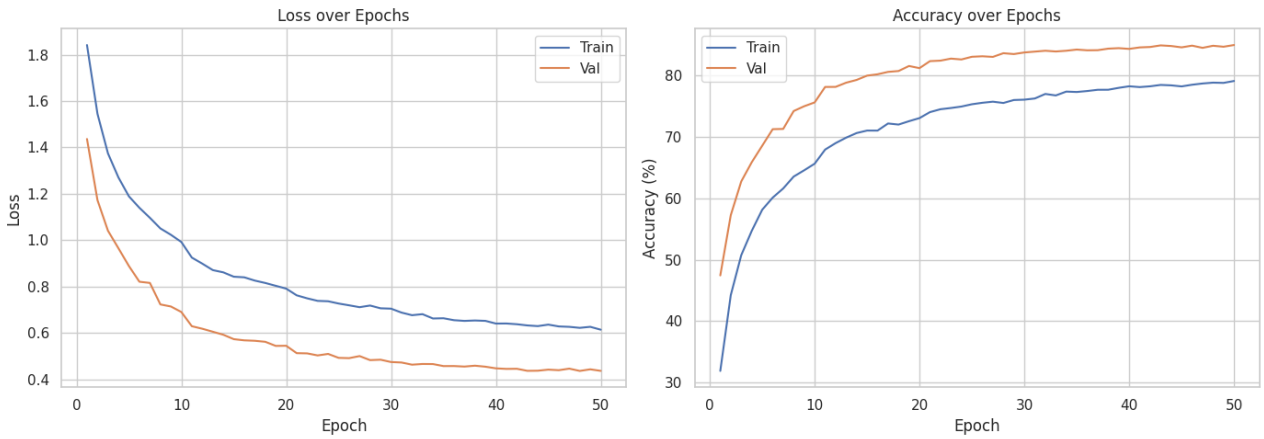
Kết quả:

- Độ chính xác tốt nhất trên tập kiểm định (Best Validation Accuracy): **84.92%**
- Độ chính xác trên tập kiểm thử (Test Accuracy): **85.25%**

Nhận xét: Em thấy rằng độ chính xác trên tập kiểm thử (85.25%) cao hơn một chút so với độ chính xác tốt nhất trên tập kiểm định (84.92%). Điều này là một tín hiệu rất tích cực, cho thấy mô hình không những không bị overfitting mà còn có khả năng tổng quát hóa tốt trên dữ liệu mới hoàn toàn chưa từng thấy. Mức độ chính xác này là khá tốt đối với bộ dữ liệu CIFAR-10 với một mô hình CNN tùy chỉnh.

4.2.2 Đồ thị Quá trình Học (Learning Curves)

Đồ thị quá trình học bao gồm sự thay đổi của hàm mất mát (Loss) và độ chính xác (Accuracy) trên tập huấn luyện và tập kiểm định qua 50 epochs được trình bày ở Hình 4.1.



Hình 4.1: Đồ thị hàm mất mát và độ chính xác qua các epoch.

Nhận xét:

Hàm mất mát (Loss): Đường loss trên cả tập huấn luyện (Train) và tập kiểm định (Val) đều có xu hướng giảm dần qua các epoch. Ban đầu, val loss giảm nhanh hơn train loss, nhưng sau khoảng 5-7 epochs, train loss bắt đầu giảm nhanh hơn và duy trì ở mức thấp hơn val loss. Đến cuối quá trình huấn luyện, cả hai đường loss đều có dấu hiệu đi ngang, cho thấy mô hình đã hội tụ khá tốt. Val loss ở mức khoảng 0.436 cho thấy mô hình học được các đặc trưng hữu ích.

Độ chính xác (Accuracy): Tương tự, độ chính xác trên cả hai tập đều tăng lên. Val accuracy tăng nhanh ở những epoch đầu và đạt đến khoảng 84.92

Early Stopping: Thông báo "Early stopping at epoch 50"cho thấy quá trình huấn luyện đã chạy hết 50 epochs theo kế hoạch. Điều này có nghĩa là điều kiện dừng sớm (không cải thiện val_loss trong 7 epochs liên tiếp với min_delta là 0.001) đã không được kích hoạt trước epoch cuối cùng, hoặc chỉ vừa đủ điều kiện ở epoch thứ 50. Nhìn vào đồ thị val loss, việc nó khá phẳng ở những epoch cuối cũng nhấn mạnh việc mô hình đã gần đạt điểm bão hòa.

Nhìn chung, các đồ thị cho thấy quá trình huấn luyện diễn ra ổn định, mô hình học được và không có dấu hiệu overfitting rõ rệt.

4.2.3 Ma trận Nhầm lẫn (Confusion Matrix)

Để phân tích chi tiết hơn khả năng phân loại của mô hình đối với từng lớp, em đã tạo ma trận nhầm lẫn từ kết quả dự đoán trên tập kiểm thử.

```python
# if __name__ == '__main__':
#     # y_true, y_pred đã có
cm = confusion_matrix(y_true, y_pred)
```

Liệt kê mã 4.4: Tính toán ma trận nhầm lẫn

Ma trận nhầm lẫn được trực quan hóa trong Hình 4.2.

Hình 4.2: Ma trận nhầm lẫn trên tập kiểm thử.

Nhận xét: Quan sát ma trận nhầm lẫn, em nhận thấy:

Các giá trị trên đường chéo chính của ma trận tương đối cao, cho thấy mô hình đã dự đoán đúng phần lớn các mẫu thuộc về mỗi lớp.

Các lớp phân loại tốt: Mô hình phân loại rất tốt các lớp 'automobile' (957/1000 mẫu đúng), 'truck' (926/1000), 'ship' (919/1000), và 'frog' (915/1000).

Các lớp bị nhầm lẫn:

- Lớp 'cat' (mèo) có vẻ là lớp khó phân loại nhất, với chỉ 632 mẫu được dự đoán đúng. Nó bị nhầm lẫn nhiều nhất với 'dog' (chó - 146 mẫu), 'deer' (hươu - 50 mẫu), và 'frog' (ếch - 53 mẫu). Sự nhầm lẫn giữa mèo và chó là điều khá phổ biến trong các bài toán phân loại ảnh.
- Lớp 'bird' (chim) cũng có độ chính xác không quá cao (748 mẫu đúng), bị nhầm với 'deer' (50 mẫu), 'dog' (49 mẫu), 'airplane' (49 mẫu), và 'frog' (49 mẫu).
- Lớp 'dog' (chó) bị nhầm lẫn nhiều với 'cat' (mèo - 87 mẫu) và 'deer' (hươu - 33 mẫu).

Lớp 'airplane' (máy bay) và 'ship' (tàu thủy) đôi khi bị nhầm lẫn với nhau hoặc với các phương tiện khác, nhưng ở mức độ không quá lớn.

Ma trận nhầm lẫn cung cấp cái nhìn chi tiết về điểm mạnh và điểm yếu của mô hình đối với từng lớp cụ thể.

4.2.4 Độ chính xác theo Từng Lớp (Class-wise Accuracy)

Dựa trên ma trận nhầm lẫn, em tính toán độ chính xác cụ thể cho từng lớp như sau:

```
# if __name__ == '__main__':
# cm và CLASSES đã có
for i, cls_name in enumerate(CLASSES):
     class_accuracy = 100 * cm[i, i] / cm[i, :].sum()
#     print(f"{cls_name:>12}: {class_accuracy:5.2f}%")
```

Liệt kê mã 4.5: Tính toán độ chính xác theo từng lớp

Kết quả chi tiết:

Loại đối tượng	Độ chính xác
airplane	88.00%
automobile	95.70%
bird	74.80%
cat	63.20%
deer	86.40%
dog	78.90%
frog	91.50%
horse	89.50%
ship	91.90%
truck	92.60%

Bảng 4.1: Độ chính xác theo loại đối tượng

Các số liệu này khẳng định lại những gì em đã quan sát từ ma trận nhầm lẫn:

1. Lớp 'automobile' và 'truck' có độ chính xác cao nhất, trên 92%. Các lớp 'frog', 'ship', 'horse', 'airplane', 'deer' cũng đạt độ chính xác tốt, từ 86% trở lên.

2. Lớp 'cat' có độ chính xác thấp nhất (63.20%), tiếp theo là 'bird' (74.80%) và 'dog' (78.90%). Đây là ba lớp mà mô hình gặp nhiều khó khăn nhất trong việc phân loại chính xác, có thể do sự đa dạng về hình dạng, màu sắc, tư thế và sự tương đồng về đặc điểm giữa chúng trong bộ dữ liệu CIFAR-10.

4.3 Nhận xét Chung về Kết quả Thực nghiệm

Tổng kết lại, mô hình CNN mà em xây dựng và huấn luyện đã đạt được độ chính xác tổng thể 85.25% trên tập kiểm thử CIFAR-10. Quá trình huấn luyện diễn ra ổn định, và mô hình cho thấy khả năng tổng quát hóa tốt. Các kỹ thuật như tăng cường dữ liệu, chuẩn hóa theo batch, dropout, và điều chỉnh tốc độ học đã đóng góp vào kết quả này. Mặc dù mô hình hoạt động rất tốt trên một số lớp, nhưng vẫn còn những thách thức đối với các lớp có đặc điểm hình ảnh phức tạp và dễ nhầm lẫn như 'cat', 'bird', và 'dog'. Đây là những điểm có thể được tập trung cải thiện trong các nghiên cứu hoặc thử nghiệm tiếp theo.