MINISTRY OF SCIENCE AND TECHNOLOGY
POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

—————o0o—————

MAJOR ASSIGNMENT REPORT 2

PYTHON PROGRAMMING LANGUAGE

Instructor:	**Kim Ngoc Bach**
Student:	**Le Nhu Quynh**
Student ID:	**B23DCCE081**
Class:	**D23CQCE06-B**
Academic Year:	**2023 - 2028**
Training System:	**Full-time University**

Hanoi, 2025

INSTRUCTOR'S COMMENTS

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Grade: **(In words:)**

Hanoi, date month year 20...

Instructor

Contents

List of Figures

List of Code Listings

Introduction

In this report, I will detail the process of building, training, and evaluating a **Convolutional Neural Network (CNN)** model to solve the image classification problem on the **CIFAR-10** dataset. This dataset consists of 60,000 32×32 pixel color images, equally divided into 10 different object classes, and is very popular in computer vision research.

The entire process is implemented using the Python programming language, with PyTorch as the primary library for building and training the model. Additionally, supporting libraries such as torchvision, Matplotlib, Seaborn, NumPy, and Scikit-learn are also used for data processing, result visualization, and comprehensive model performance evaluation.

Chapter 1

Data Preparation

The first and crucial step is data preparation. In this chapter, I will present the library declarations, the parameters I used, and the steps for downloading and preprocessing the **CIFAR-10** dataset.

1.1 Library and Parameter Declaration

First, I import the necessary libraries and define the required parameters.

1.1.1 Declaring Libraries

```
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
```

Code Listing 1.1: Declaring Python libraries

These libraries play roles in model building, data processing, optimization, and result visualization.

1.1.2 Declaring Main Parameters

Specifically, I set `BATCH_SIZE` to 64, the initial learning rate `LR` to 1×10^{-3}, and the maximum number of `EPOCHS` for the training process to 40. `DEVICE` will automatically

select GPU if available, and `CLASS_NAMES` is the list of class names for **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')
```

Code Listing 1.2: Declaring main parameters

1.2 Loading and Preprocessing Data

I wrote the `get_data_loaders` function to perform data loading and preprocessing.

1.2.1 Defining 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))
    ])
```

Code Listing 1.3: Defining data transformations

These transformations include random horizontal flipping, random cropping, converting images to Tensors, and normalizing pixel values.

1.2.2 Loading the CIFAR-10 Dataset

I load the CIFAR-10 dataset from torchvision.datasets and apply the above transformations:

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

Code Listing 1.4: Loading the CIFAR-10 dataset

1.2.3 Splitting Training and Validation Sets

I split the `full_train` set into 80

```
    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])
```

Code Listing 1.5: Splitting training and validation sets

1.2.4 Creating DataLoaders

Finally, I create DataLoader objects to provide data in batches to the model, with `shuffle=True` for `train_loader`:

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

Code Listing 1.6: Creating DataLoaders

The `get_data_loaders` function will return these three `DataLoaders`.

Chapter 2

Model Building

In this chapter, I present the architecture of the Convolutional Neural Network (CNN) named **ConvNet** that I designed using PyTorch for CIFAR-10 image classification.

2.1 Defining the CNN Model Class (ConvNet)

My `ConvNet` class inherits from `torch.nn.Module`. The entire architecture is defined within an `nn.Sequential` container so that the layers are executed sequentially.

```
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # --- Convolutional Block 1 ---
            nn.Conv2d(3, 32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- Convolutional Block 2 ---
            nn.Conv2d(32, 64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- Convolutional Block 3 ---
            nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # --- 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)
```

Code Listing 2.1: The ConvNet Class

2.1.1 Convolutional Blocks

My model uses three convolutional blocks to extract features:

- **Block 1:** Consists of `Conv2d(3, 32, kernel_size=3, padding=1)`, `ReLU`, and `MaxPool2d(2)`. This block transforms the 3-channel input image into 32 feature maps, each with its size halved (from 32×32 to 16×16).
- **Block 2:** Consists of `Conv2d(32, 64, kernel_size=3, padding=1)`, `ReLU`, and `MaxPool2d(2)`. The number of channels increases to 64, and the feature map size further reduces to 8×8.
- **Block 3:** Consists of `Conv2d(64, 128, kernel_size=3, padding=1)`, `ReLU`, and `MaxPool2d(2)`. The number of channels increases to 128, and the final feature map size is 4×4.

2.1.2 Classifier

After the convolutional blocks, the classifier processes the learned features:

- `nn.Flatten()`: Converts the feature tensor (128 channels, 4×4) into a flat vector with 128×4×4=2048 elements.
- `nn.Linear(2048, 256)` and `nn.ReLU()`: The first fully connected layer maps the 2048 features down to 256 features, followed by a ReLU activation function.
- `nn.Dropout(0.5)`: Applies dropout with a rate of 0.5 to reduce overfitting.
- `nn.Linear(256, 10)`: The final fully connected layer produces 10 logits, corresponding to the scores for the 10 CIFAR-10 classes.

2.1.3 The forward Method

The `forward(self, x)` method defines how data propagates through the network:

```
1 def forward(self, x):
2     # Pass data x through self.model (nn.Sequential) as defined
3     return self.model(x)
```

Code Listing 2.2: The forward method

When data `x` is input, it sequentially passes through the layers in `self.model` to produce the prediction results.

Chapter 3

Model Training

After completing data preparation (Chapter 1) and building the convolutional neural network model (Chapter 2), in this chapter, I will present the training process for the CNN model I designed. This process includes initializing the model and necessary components, defining functions for training and evaluation, setting up the main training loop, as well as techniques to improve training quality such as saving the best model and early stopping.

3.1 Initializing Training Components

To begin the training process, I first need to initialize the model, choose an appropriate loss function, and set up the optimizer. Concurrently, I also need to prepare DataLoaders for the training, validation, and test data.

3.1.1 Model, Loss Function, and Optimizer

The model is initialized from the `ConvNet` class (which I defined in Chapter 2) and moved to the appropriate computing device (GPU if available, otherwise CPU). I use `CrossEntropyLoss` as the loss function because this is a multi-class classification problem. The optimizer I chose is Adam with the previously declared learning rate.

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

Code Listing 3.1: Initializing the model, loss function, and optimizer

3.1.2 Loading Data with DataLoader

The `get_data_loaders()` function, which I defined in Chapter 1, will return three `DataLoader` objects corresponding to the training, validation, and test sets.

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

Code Listing 3.2: Loading data using the `get_data_loaders()` function

3.2 Helper Functions for Training and Evaluation

3.2.1 Function to Train One Epoch

The `train_one_epoch()` function is used to train the model with all the data in one epoch. In this function, I perform forward propagation, calculate the loss, backpropagate, and update the weights. The function returns the average loss and accuracy for that epoch.

```
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
```

Code Listing 3.3: Function to train one epoch

3.2.2 Function to Evaluate the Model

The `evaluate()` function is used to check the model's performance. This function can return accuracy, loss, or a list of predicted and actual labels if needed. This is very convenient for plotting confusion matrices or for final evaluation on the test set.

```python
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
```

Code Listing 3.4: Function to evaluate the model

3.3 Main Training Loop and Saving the Best Model

After the model and data are ready, I proceed to train for multiple epochs. In each epoch, the model will be trained on the training set and evaluated on the validation set. I record the results in a log file, and at the same time, save the best model based on the lowest `val_loss`. If there is no improvement after a certain number of consecutive epochs, I apply the early stopping technique.

```
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
```

Code Listing 3.5: Main training loop

Reason for Choosing 40 Epochs and Illustrating the Early Stopping Mechanism

During the training process, I set the maximum number of iterations to `EPOCHS = 40`. This number was not chosen arbitrarily but based on:

- Preliminary experimental results with CIFAR-10 showed that the model began to converge around **epoch 30** onwards.
- Common research practices also often train simple CNNs on CIFAR-10 for about **30–50 epochs** to achieve stable accuracy.
- Simultaneously, I applied the **early stopping** technique with `patience = 5`, meaning that if the *validation loss* did not improve for 5 consecutive epochs, the process would automatically stop early.

To verify the effectiveness of **early stopping**, I tried increasing `EPOCHS = 50` while keeping all other settings the same. The results showed that the model **automatically stopped early at epoch 44**, as there was no improvement in validation loss for 5 consecutive epochs.

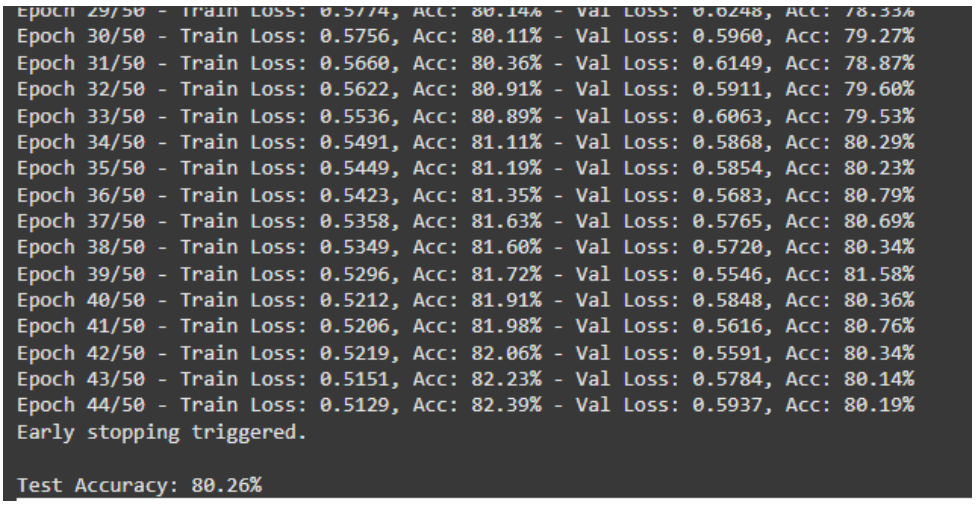

Figure 3.1: Log results when training with 50 epochs – the system automatically stopped early at epoch 44.

This experiment shows that: choosing `EPOCHS = 40` initially was reasonable, and the **early stopping** mechanism works effectively to prevent overfitting, save training resources, while still ensuring the selection of an optimal model.

3.4 Reloading the Best Model and Evaluating on the Test Set

After the training process is complete, I reload the model with the best performance and evaluate it on the test set to determine the final accuracy.

```
model.load_state_dict(torch.load("best_model.pt"))
print("Đã tải lại trọng số từ best_model.pt để đánh giá cuối cùng.") # Kept original Vietnamese string in code

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")
```

Code Listing 3.6: Loading the best model and evaluating on the test set

3.5 Visualizing Training Results

Finally, I plot the confusion matrix and learning curves to visualize how the model learned from the data.

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

Code Listing 3.7: Calling visualization functions

With the model thoroughly trained, the next chapter will present the obtained results and evaluate its performance in detail on the CIFAR-10 test set.

Chapter 4

Experiments and Evaluation

After completing the process of building and training the CNN model on the CIFAR-10 dataset, in this chapter, I will present in detail the model evaluation process, including how to check performance on the test set and analyze the achieved results. The goal is to assess the model's generalization ability and identify its strengths and weaknesses when applied to real-world data.

4.1 Model Evaluation Process

After completing the model training process, I proceed to evaluate the model's effectiveness on the test set. The evaluation process consists of three main steps: reloading the best-performing saved model, making predictions on the test set, and calculating the overall accuracy. The details of each step are presented below.

Step 1: Reloading the Best Model

During the training process, I saved the model with the best results on the validation set using the `torch.save()` function. To ensure that the evaluation is performed on the most optimal model, I reload the weights from the `best_model.pt` file as follows:

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

Code Listing 4.1: Reloading the model with the best weights.

Step 2: Predicting on the Test Set

After loading the optimal model, I use it to predict labels for the images in the test set. I call the `evaluate()` function and set the parameter `return_preds=True` to retrieve both the actual labels and the labels predicted by the model:

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

Code Listing 4.2: Predicting labels for the test data.

The returned results include:

- `y_true`: a list of the true labels in the test set.
- `y_pred`: a list of the labels predicted by the model.

Step 3: Calculating Overall Accuracy

Finally, I calculate the model's accuracy on the test set by comparing the number of correct predictions with the total number of samples, and multiplying by 100 to get the percentage:

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

Code Listing 4.3: Calculating overall accuracy on the test set.

Results after Execution

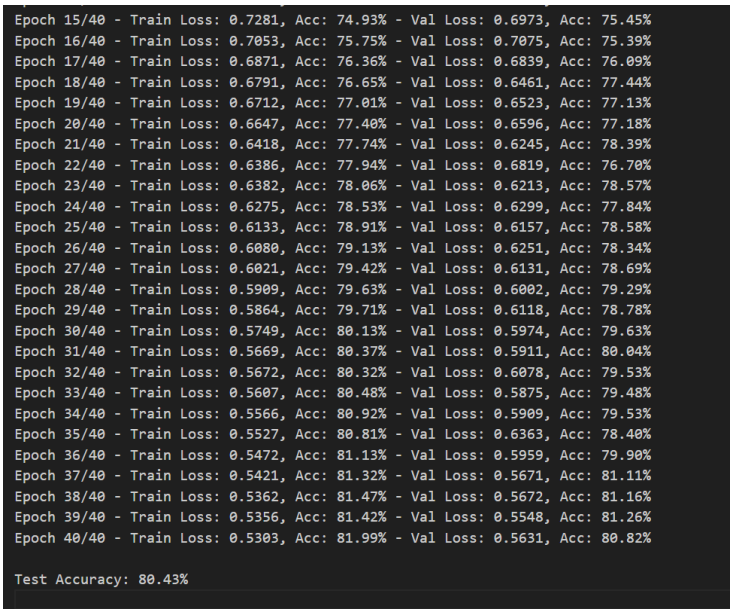

Figure 4.1: Part of the results after program execution

Assessing Accuracy and Generalization Ability

The accuracy value (`test_acc`) is an important indicator for evaluating the model's generalization ability — that is, its ability to apply knowledge learned from the training set to new, previously unseen data. In this experiment, my model achieved a test accuracy of **80.43%**.

To evaluate further, I compared the accuracy on the training and validation sets throughout the learning process. The results from the log file show that the training accuracy increased steadily from 36.99% to 81.99% after 40 epochs, while the validation accuracy also increased steadily from 48.34% to **81.26%** at epoch 39 — this was also when the model achieved its best result and was saved.

This indicates that the model learned general features of the data without falling into "rote learning" (overfitting). Specifically:

- The test accuracy (**80.43%**) is close to the validation accuracy at the best epoch (**81.26%**), which proves that the model did not lose its generalization ability when moving from validation to test.
- The difference between training accuracy (**81.99%**) and validation accuracy (**81.26%**) is very small ($< 1\%$), indicating that the training process was balanced, with no signs of severe overfitting.

4.2 Learning Curves

Function to Plot Graphs

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

    # Loss Curve
    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)

    # Accuracy Curve
    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()
```

Code Listing 4.4: Function to plot Learning Curves.

Learning Curves Chart

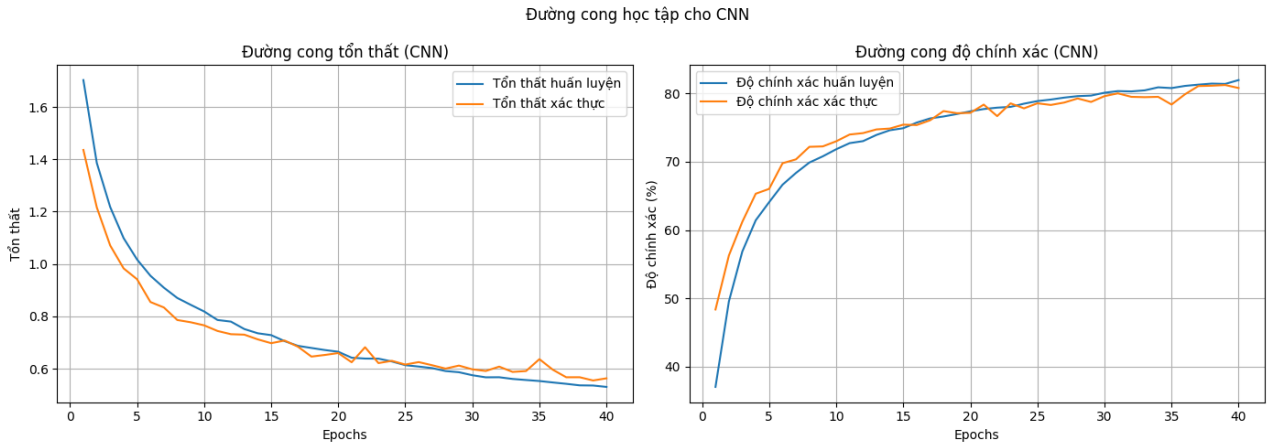

Figure 4.2: Loss and Accuracy Curves by Epochs

Analysis of Learning Curves

From the `learning_curves` chart (figure 4.2), I draw the following detailed observations:

- **The loss curve** on the left shows that both training loss and validation loss continuously decrease with the number of epochs. Training loss starts at around 1.7 and steadily decreases to approximately 0.53 after 40 epochs. Validation loss also decreases similarly from over 1.4 to around 0.56. The curves are smooth and have no abrupt fluctuations, indicating that the model learned stably, without noise or loss of control during backpropagation.

- **The accuracy curve** on the right shows that the model has a fast learning rate in the initial phase (epochs 1–10), with validation accuracy increasing sharply from about 48% to over 70%. After that, the rate of improvement gradually slows down and converges at over 80% from around epoch 35 onwards.

- **The gap between the two curves** (train and validation) on both charts is very small throughout the training process. This indicates that the model learns in a general way, without severe overfitting. At epoch 39, the model achieved the highest validation accuracy of **81.26%**, while the training accuracy at the same time was **81.42%**. The difference of less than 0.2% shows that the fit between the model and the actual data is very good.

- **The lowest validation loss** occurs around epoch 39, which is also when the model was selected as the best model (`best_model.pt`). This shows that the strategy of saving the best model based on loss worked effectively.

- **No signs of overfitting or underfitting**: If the model were overfitting, we would see the validation loss curve start to increase while the training loss continues to

decrease — this does not happen here. Conversely, if the model were underfitting, both accuracy curves would remain at a low level — this also does not happen.

- **Conclusion**: The learning curves demonstrate that the CNN model was trained effectively, has good generalization ability, and the learning process was stable. This is a solid foundation for further improving the model with more complex methods if necessary.

4.3 Confusion Matrix

Function to Plot Confusion Matrix

```
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()
```

Code Listing 4.5: Function to plot Confusion Matrix

Confusion Matrix Image

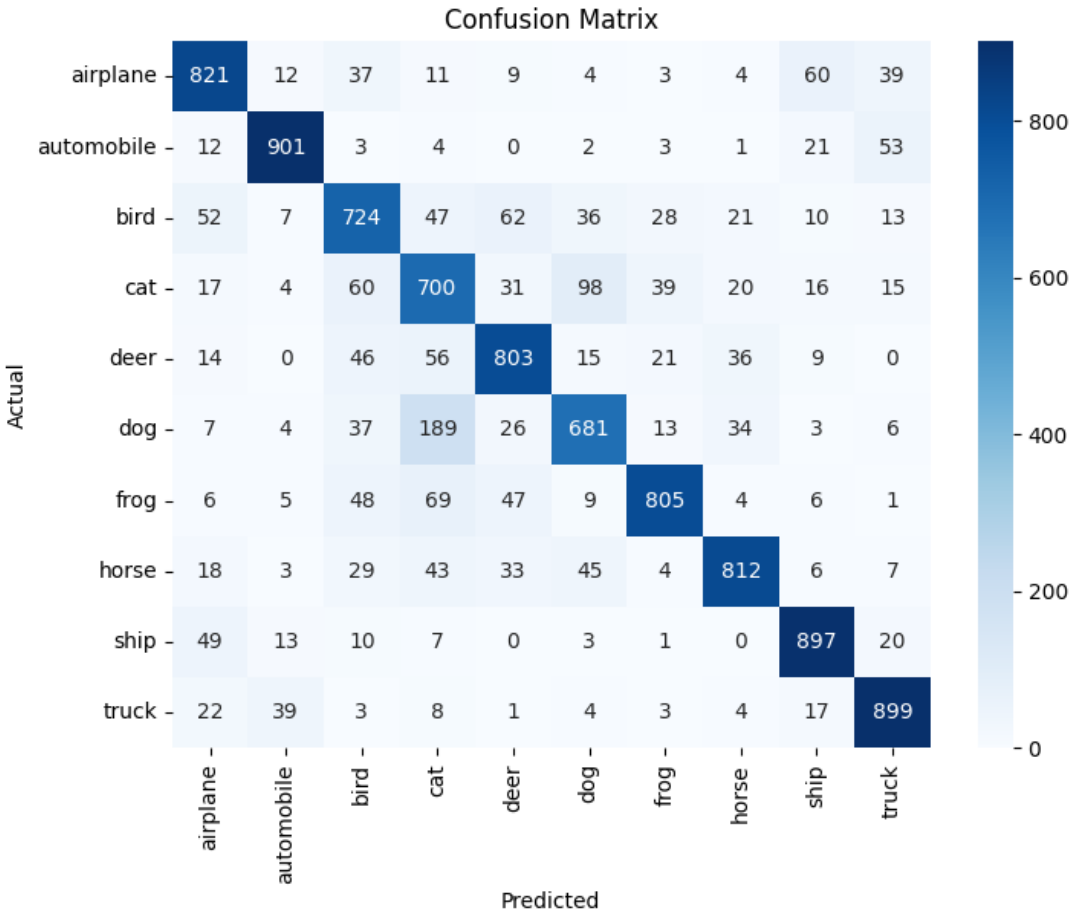

Figure 4.3: Confusion matrix on the test dataset

Analysis of Confusion Matrix

From the `confusion_matrix` chart (figure 4.3), I analyze in detail the classification performance of the model on each class as follows:

- **The best-classified classes** are `automobile` (901/1000), `truck` (899/1000), and `ship` (897/1000). These are all objects with distinct characteristic shapes, are less likely to be obscured, and are easily recognizable in the CIFAR-10 dataset. The model is able to learn the specific visual features of these vehicles, such as their blocky shapes, straight edges, and common colors.
- **The classes most easily confused** are in the animal group. Typical examples are:
  - The `dog` class was misclassified by the model as `cat` **189** times – the highest number in the entire confusion matrix. Conversely, `cat` was also misclassified as `dog` **98** times. This can be explained by the similar shape, fur color, and

size of these two species, especially when images are downscaled to 32×32 as in CIFAR-10.

  - The `bird` class has a fairly high misclassification rate with classes like `deer` (62 times), `cat` (47 times), and `airplane` (52 times). Misclassifying `bird` as `airplane` might stem from images of flying birds with spread wings that can be confused with airplanes.
  - The `frog` class was misclassified as `cat` (69 times) and `bird` (48 times), indicating that small animals with dark colors and indistinct outlines tend to pose difficulties for the model in distinguishing them.

- **Some random misclassifications** appear, such as the model mispredicting `deer` as `dog` (21 times), or `horse` as `dog` (45 times). These errors might be due to factors like posture, image background, or viewing angle causing noise.

Overall, the confusion matrix shows that the model performs quite well with classes that have clear geometric and color features (vehicles), but still faces difficulties in distinguishing between animal species – a group with soft shapes, many postures, and a tendency to be obscured.

4.4 Class-wise Accuracy

Class-wise accuracy is calculated using the formula:

$$\text{Accuracy class i} = \frac{\text{number of correctly predicted samples for class i}}{\text{total number of samples in class i}} \times 100\%$$

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%**

Table 4.1: Accuracy per class

Vehicle groups such as `automobile`, `ship`, `truck` have very high accuracy, above 89%, showing that the model easily recognizes them thanks to clear image features.

Animal classes like `dog`, `cat`, `bird` have lower accuracy, due to many similar visual features leading to confusion.

4.5 Overall Assessment

The simple CNN model I built achieved an accuracy of 80.43% on the test set, demonstrating stable learning effectiveness and good generalization ability. The use of techniques such as dropout, augmentation, and early stopping helped the model avoid overfitting and maintain performance on unseen data. Nevertheless, the model still encounters difficulties in distinguishing animal classes with similar visual characteristics, leading to certain misclassifications. Overall, the obtained results show that this ConvNet model is a solid foundation, suitable for further development and improvement in subsequent research on the CIFAR-10 dataset.