MINISTRY OF SCIENCE AND TECHNOLOGY
POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

o0o

# ASSIGNMENT REPORT 2

# PYTHON PROGRAMMING LANGUAGE

| | |
|---|---|
| **Instructor:** | **Kim Ngoc Bach** |
| **Student:** | **Chu Tuyet Nhi** |
| **Student ID:** | **B23DCCE075** |
| **Class:** | **D23CQCEO6-B** |
| **Academic Year:** | **2023 - 2028** |
| **Training System:** | **Full-time University** |

**Hanoi, 2025**

# LECTURER'S COMMENTS

.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................
.........................................................................................................................

**Score:** **( In words:** )

Hanoi, date month year 20...

**Lecturer**

# Contents

# List of Figures

# List of Code Listings

# Introduction

This report fulfills the requirements of Project 2 for the Python Programming Language course, focusing on the image classification problem on the CIFAR-10 dataset. The objective is to build, train, and evaluate a Convolutional Neural Network (CNN) model using the PyTorch library.

The CIFAR-10 dataset consists of 60,000 32x32 pixel color images, divided into 10 classes. The implementation uses Python, PyTorch, torchvision, Matplotlib, Seaborn, NumPy, and Scikit-learn.

The report content includes:

**Chapter 1. Data Preparation**: Declaring libraries, parameters, loading and preprocessing the CIFAR-10 data, including transforms, splitting the dataset, and creating DataLoaders.

**Chapter 2. Model Building**: Presenting the CNN architecture, including the model class definition, convolutional blocks, classifier, and the `forward` method.

**Chapter 3. Model Training**: Describing the training setup (model, loss function, optimizer), learning rate adjustment, early stopping, training and evaluation functions per epoch, and the main training loop.

**Chapter 4. Experiments and Evaluation**: Analyzing results on the test set, including overall accuracy, learning curves, confusion matrix, class-wise accuracy, and general remarks.

# Chapter 1

# Data Preparation

This section describes the data preparation process, including declaring necessary libraries, setting parameters, and the steps for loading and preprocessing the CIFAR-10 dataset.

## 1.1 Library and Parameter Declaration

First, I import the necessary libraries for the project.

```python
# Building and training deep learning models
import torch
import torch.nn as nn
import torch.optim as optim
# Handling and transforming image data
import torchvision
import torchvision.transforms as transforms
# Plotting graphs and visualizing results
import matplotlib.pyplot as plt
import seaborn as sns
# Numerical computation
import numpy as np
# Evaluating classification models
from sklearn.metrics import confusion_matrix
```

Code Listing 1.1: Declaration of necessary libraries

Next, I define important parameters that will be used throughout the training and data processing:

```python
BATCH, LR, EPOCHS = 64, 1e-3, 50 # Changed EPOCHS to 50 to match text
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
CLASSES = ('airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

Code Listing 1.2: Definition of main parameters

1. `BATCH`: Data batch size for each weight update, here I chose 64.
2. `LR`: Initial learning rate for the Adam optimization algorithm, set to $1 \times 10^{-3}$.
3. `EPOCHS`: Maximum number of training cycles, I set it to 50, as this is a common value used for datasets like the one in the assignment. However, the training process may stop earlier due to the Early Stopping mechanism.
4. `DEVICE`: Computation device (`cuda` if GPU is available, otherwise `cpu`). This helps leverage the parallel computation capabilities of a GPU if available.
5. `CLASSES`: List of class names in the CIFAR-10 dataset.

## 1.2 Data Loading and Preprocessing

I built the `load_data()` function to perform data loading and preprocessing.
**Main steps in the `load_data()` function:**

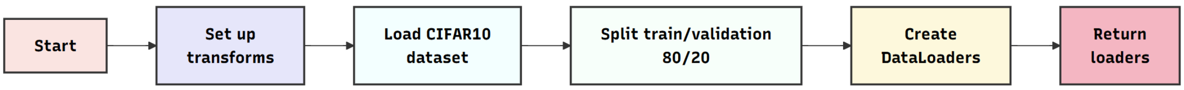


Figure 1.1: Flowchart of the load_data() function

### 1.2.1 Defining Transformations (Transforms):

```
def load_data():
    mean, std = (0.485, 0.456, 0.406), (0.229, 0.224, 0.225)
    # Mean and standard deviation values from 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)
    ])
    #...
```

Code Listing 1.3: Code snippet for transforming datasets

**Train set transformations** `train_tf`: A sequence of transformations applied to the training set. I use data augmentation techniques to enrich the training data and help the model generalize better:

1. `transforms.RandomHorizontalFlip()`: Randomly flips the image horizontally.
2. `transforms.RandomRotation(10)`: Randomly rotates the image by an angle in the range of $\pm 10$ degrees.
3. `transforms.RandomCrop(32, padding=4)`: Randomly crops a 32x32 region of the image after padding 4 pixels to each side.
4. `transforms.ColorJitter(0.2, 0.2, 0.2, 0.1)`: Randomly changes the brightness, contrast, saturation, and hue of the image.
5. `transforms.ToTensor()`: Converts a PIL image or NumPy array to a PyTorch Tensor.
6. `transforms.Normalize(mean, std)`: Normalizes the pixel values of the image by subtracting the mean (`mean`) and dividing by the standard deviation (`std`) for each color channel. I use the common `mean` and `std` values from the ImageNet dataset.

**Test set transformations** `test_tf`: A sequence of transformations for the validation and test sets. For these sets, I only perform `ToTensor` and `Normalize` to ensure consistency and prevent information leakage from augmentation into the evaluation process.

### 1.2.2 Loading the CIFAR-10 Dataset:

`train_full`: Loads the entire CIFAR-10 training set, applying `train_tf`.
`test_set`: Loads the CIFAR-10 test set, applying `test_tf`.

```python
# def load_data(): # Already defined
    #...
    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)
    #...
```

Code Listing 1.4: Code snippet for loading the CIFAR-10 dataset

### 1.2.3 Splitting Training and Validation Sets:

I split the `train_full` set into two parts: `train_set` (80%) and `val_set` (20%) to be used for training and validating the model during the training process.

```python
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)
    #...
```

Code Listing 1.5: Code snippet for Splitting Training and Validation Sets

Notably, after splitting, I reassign the `transform` for `val_set` to `test_tf`. This ensures that the validation set is processed in the same way as the test set, unaffected by the data augmentation operations of the training set, thus allowing for a more objective evaluation of the model's performance on unseen data.

### 1.2.4 Creating DataLoaders:

```python
# def load_data(): # Already defined
    #...
    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)
    #...
```

Code Listing 1.6: Code snippet for Creating DataLoaders

A lambda function `loader` is defined to conveniently create `torch.utils.data.DataLoader`. `DataLoader` helps divide the data into batches, shuffle the data (for the training set), and load data in parallel using multiple `workers`.

The function returns three `DataLoader`s: one for the training set (`shuffle=True`), one for the validation set (`shuffle=False`), and one for the test set (`shuffle=False`).

# Chapter 2

# Building the Model

This chapter presents the CNN (Convolutional Neural Network) architecture that I used to classify images from the CIFAR-10 dataset.

## 2.1 Defining the CNN Model Class

I define the `CNN` class, which inherits from `nn.Module` of PyTorch. This architecture includes custom-designed convolutional blocks and a classifier based on fully connected layers.

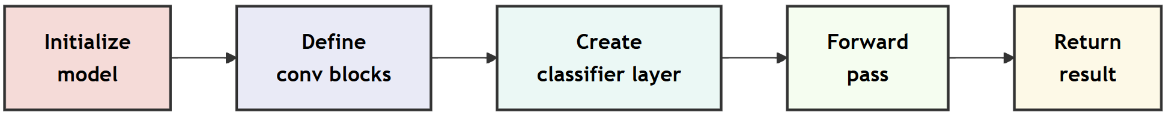


Figure 2.1: CNN model flowchart

### 2.1.1 Model Initialization and Basic Convolutional Block

The `CNN` class is initialized with two main parameters: `num_classes` (number of output classes, default is 10) and `dropout` (dropout rate for layers in the classifier, default is 0.5). Inside the `__init__` constructor, I define a utility function `conv_block(in_c, out_c)`. The purpose of this function is to encapsulate a sequence of operations commonly found in CNNs to create a convolutional block capable of feature extraction and regularization.

```python
class CNN(nn.Module):
    def __init__(self, num_classes=10, dropout=0.5):
        super().__init__()
        # Define a basic convolutional block (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)
        )
        # ... (Other parts will be presented in subsequent sections)
```

Code Listing 2.1: Definition of the `CNN` class and the `conv_block` utility function

The `conv_block(in_c, out_c)` block consists of:

1. Two `Conv2d` layers (3×3, padding=1), each followed by `BatchNorm2d` and `ReLU(inplace=True)` to extract features, stabilize gradients, increase non-linearity, and improve training efficiency. Padding preserves spatial dimensions, while `ReLU` helps prevent the vanishing gradient problem.

2. A `MaxPool2d(2)` layer to halve the spatial dimensions, reduce computation, and increase invariance to small translations.

3. A `Dropout2d(0.25)` layer to randomly drop 25% of channels during training to combat overfitting.

*Result*: Features are learned better, the number of channels can change from `in_c` to `out_c`, dimensions are reduced, and the model is regularized more effectively.

### 2.1.2 Main Convolutional Blocks of the Network

Next, I use the `conv_block` function to build three main convolutional blocks: `self.conv1`, `self.conv2`, and `self.conv3`.

```python
class CNN(nn.Module):
    #... (__init__ from previous part)
    def __init__(self, num_classes=10, dropout=0.5): # Duplicating for context
        super().__init__()
        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)
        ) # End of conv_block def
        self.conv1 = conv_block(3, 32)
        self.conv2 = conv_block(32, 64)
        self.conv3 = conv_block(64, 128)
    # ... (classifier and forward method to follow)
```

Code Listing 2.2: Initialization of main convolutional blocks in `CNN`

*Purpose* of stacking these three blocks is to enable the model to learn increasingly complex and abstract features while progressively reducing spatial dimensions:

1. `self.conv1 = conv_block(3, 32)`: Converts the 3-channel (RGB) input image into 32 feature channels. *Result*: Spatial dimensions are reduced from $32 \times 32$ to $16 \times 16$.

2. `self.conv2 = conv_block(32, 64)`: Increases the number of feature channels from 32 to 64. *Result*: Spatial dimensions are reduced to $8 \times 8$.

3. `self.conv3 = conv_block(64, 128)`: Increases the number of feature channels from 64 to 128. *Result*: Spatial dimensions are reduced to $4 \times 4$.

*Reason* for this architecture is that it follows the common principle of modern CNNs: increasing depth (number of channels) while reducing spatial resolution. *Final result* of the convolutional part is 128 feature maps, each of size $4 \times 4$.

### 2.1.3 Classifier

After feature extraction, `self.classifier` has the *purpose* of mapping these features to the space of output classes for classification.

```python
class CNN(nn.Module):
    def __init__(self, num_classes=10, dropout=0.5): # Assuming previous parts of __init__ are here
        super().__init__()
        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)
        )
        self.conv1 = conv_block(3, 32)
        self.conv2 = conv_block(32, 64)
        self.conv3 = conv_block(64, 128)

        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)
        )
    # ... (forward method to follow)
```

Code Listing 2.3: Initialization of the classifier in `CNN`

`self.classifier` consists of:

1. `nn.Flatten()` to convert the multi-dimensional feature tensor ($128 \times 4 \times 4$) into a flat vector of 2048 dimensions, preparing it for fully connected layers.

2. Two fully connected layers reducing dimensions from 2048 to 512, then to 256, each accompanied by `BatchNorm1d`, `ReLU(inplace=True)`, and `Dropout` to increase non-linearity, stability, and prevent overfitting.

3. The final layer `nn.Linear(256, num_classes)` generates logits for the classes to be classified.

*Result*: The classifier transforms features into accurate and efficient class predictions.

### 2.1.4 The `forward` method

The `forward(self, x)` method defines the data flow through the network.

```python
class CNN(nn.Module):
    def __init__(self, num_classes=10, dropout=0.5):
        super().__init__()
        # Define a basic convolutional block (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)
        )
        self.conv1 = conv_block(3, 32)
        self.conv2 = conv_block(32, 64)
        self.conv3 = conv_block(64, 128)
        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)
        )

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

Code Listing 2.4: Definition of the `forward` method in `CNN`

The data `x` is passed sequentially through the `self.conv1`, `self.conv2`, `self.conv3` blocks, and then into `self.classifier` to generate logits. *Result* is the returned tensor containing scores for each class.

In summary, this CNN architecture is built by me based on a combination of `conv_blocks` (including `Conv2d`, `BatchNorm2d`, `ReLU`, `MaxPool2d`, `Dropout2d`) and an MLP classifier (comprising `Linear`, `BatchNorm1d`, `ReLU`, `Dropout` layers). The *main purpose* of this design is to create a model capable of effectively extracting hierarchical features from CIFAR-10 images and performing accurate classification. The *reason* for using techniques like BatchNorm and Dropout throughout the model is to improve the stability of the training process and increase generalization ability, thereby minimizing overfitting.

# Chapter 3

# Training the Model

After preparing the data and building the CNN architecture, this chapter will detail the process of setting up and performing model training.

## 3.1 Training Setup

Before starting the training loop, I need to initialize important components such as the model, loss function, optimizer, learning rate scheduler, and early stopping mechanism.

### 3.1.1 Initializing Model, Loss Function, and Optimizer

First, I initialize the `CNN` model defined in the previous chapter and move the model to the computation device (`DEVICE`). Next, I choose the `CrossEntropyLoss` function because it is the standard loss function for multi-class classification problems. Finally, I use the `Adam` optimizer with the defined learning rate (`LR`) and a small `weight_decay` value ($1 \times 10^{-4}$) to help with regularization and minimize 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)
```

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

### 3.1.2 Learning Rate Scheduler

To improve the convergence process, I use `StepLR` from `torch.optim.lr_scheduler`. This scheduler will reduce the learning rate by a factor of `gamma` (here, 0.5) after a certain `step_size` of epochs (here, 10). This helps the model fine-tune better as it approaches the optimal point.

```python
# ... (following the optimizer initialization)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)
```

Code Listing 3.2: Initializing the learning rate scheduler

### 3.1.3 Early Stopping Mechanism

I define the `EarlyStopping` class to monitor the loss value on the validation set (`val_loss`). If `val_loss` does not improve (decrease by at least `min_delta`) for a certain number of `patience` consecutive epochs, the training process will stop. This helps avoid wasting training time and prevents the model from overfitting excessively.

```python
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

# ... (in main)
early_stop = EarlyStopping()
```

Code Listing 3.3: Definition and initialization of the Early Stopping mechanism

`patience=7`: The training process will stop if there is no improvement after 7 epochs.

`min_delta=1e-3`: The minimum improvement considered significant.

## 3.2 Training and Evaluation Functions per Epoch

I build two main functions: `train_epoch` to perform one training epoch and `run` to evaluate the model on the validation or test set.

### 3.2.1 The `train_epoch` Function

This function performs a full training pass over the entire data of the `loader` (in this case, `train_loader`).

```python
def train_epoch(model, loader, loss_fn, optim):
    model.train() # Set the model to training mode
    total_loss, correct, total = 0, 0, 0
    for x, y in loader: # Iterate over each batch of data
        x, y = x.to(DEVICE), y.to(DEVICE) # Move data to DEVICE
        optim.zero_grad() # Clear old gradients
        out = model(x) # Forward pass
        loss = loss_fn(out, y) # Calculate loss
        loss.backward() # Backward pass to calculate gradients
        optim.step() # Update model weights
        total_loss += loss.item() # Accumulate loss
        correct += (out.argmax(1) == y).sum().item() # Count correct predictions
        total += y.size(0) # Accumulate total samples
    return total_loss / len(loader), 100 * correct / total # Return average loss and accuracy
```

Code Listing 3.4: Function to train one epoch

**Main steps in `train_epoch`:**

1. `model.train()`: Switches the model to training mode, activating layers like Dropout, BatchNorm appropriately.
2. For each batch of data (`x, y`):
   - Move data to `DEVICE`.
   - `optim.zero_grad()`: Reset gradients of model parameters.
   - `out = model(x)`: Pass data through the model to get predictions.
   - `loss = loss_fn(out, y)`: Calculate the loss function.
   - `loss.backward()`: Calculate gradients based on the loss.
   - `optim.step()`: Update model weights based on gradients.
   - Calculate and accumulate `total_loss`, `correct` (number of correct predictions), and `total` (total number of samples).
3. Return the average loss and accuracy on the training set for that epoch.

### 3.2.2 The `run` Function

The `run` function is flexibly designed to evaluate the model on a `loader` (e.g., `val_loader`, `test_loader`). It can return loss and accuracy, or just predictions and true labels.

```python
def run(model, loader, loss_fn=None, return_preds=False):
    model.eval() # Set the model to evaluation mode
    total_loss, correct, total = 0, 0, 0
    y_true, y_pred = [], []
    with torch.no_grad(): # Do not calculate gradients during evaluation
        for x, y in loader:
            x, y = x.to(DEVICE), y.to(DEVICE)
            out = model(x)
            preds = out.argmax(1) # Get the index of the class with the highest probability
            if loss_fn: # If loss_fn is provided, calculate loss and accuracy
                total_loss += loss_fn(out, y).item()
                correct += (preds == y).sum().item()
                total += y.size(0)
            if return_preds: # If predictions need to be returned
                y_true.extend(y.cpu().numpy())
                y_pred.extend(preds.cpu().numpy())
    # Return values based on input parameters
    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
```

Code Listing 3.5: Function to evaluate the model or get predictions

**Key points in the `run` function:**

1. `model.eval()`: Switches the model to evaluation mode. Dropout layers will be disabled, BatchNorm will use learned mean and variance values.

2. `with torch.no_grad()`: Disables gradient calculation, saving memory and speeding up processing as it's not needed for the backward pass.

3. For each batch of data, the model makes predictions `preds`.

4. If `loss_fn` is provided, the function calculates and returns the average loss and accuracy. This is the case I use for evaluating on the validation set (`val_loader`) after each epoch.

5. If `return_preds` is `True`, the function returns lists of true labels (`y_true`) and predicted labels (`y_pred`). I use this option when evaluating on the test set (`test_loader`) to plot the confusion matrix.

## 3.3 Main Training Loop

This is where the actual training process takes place over multiple epochs.

```python
if __name__ == '__main__': # Continued from previous initializations
    # ...
    train_loss, val_loss, train_acc, val_acc = [], [], [], []
    best_val_acc = 0
    early_stop = EarlyStopping() # Instantiation of early_stop as per original structure
    for epoch in range(EPOCHS):
        # Train the model on the train set
        tr_loss, tr_acc = train_epoch(model, train_loader, loss_fn, optimizer)
        # Evaluate the model on the validation set
        va_loss, va_acc = run(model, val_loader, loss_fn)
        # Update learning rate
        scheduler.step()
        # Save the best model based on validation accuracy
        if va_acc > best_val_acc:
            best_val_acc = va_acc
            torch.save(model.state_dict(), 'best_model.pth') # Save weights
        # Save metrics for plotting later
        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} loss / {tr_acc:.2f}% acc - Val: {va_loss:.4f} loss / {va_acc:.2f}% acc") # slightly modified print for clarity
        # Check early stopping condition
        early_stop(va_loss)
        if early_stop.early_stop:
            print(f"\nEarly stopping at epoch {epoch+1}")
            break
```

Code Listing 3.6: Main training loop

**In each epoch:**

1. Call `train_epoch` to train the model on `train_loader`, receiving train loss and accuracy.

2. Call `run` to evaluate the model on `val_loader`, receiving validation loss and accuracy.

3. `scheduler.step()`: Update the learning rate if necessary.

4. If the validation accuracy (`va_acc`) of the current epoch is higher than the saved `best_val_acc`, then update `best_val_acc` and save the current model's weights to the file `'best_model.pth'`.

5. Save the loss and accuracy values for both training and validation sets for later visualization.

6. Print information for the current epoch.

7. Call `early_stop(va_loss)` to check the early stopping condition. If the condition is met, the loop will terminate.

At the end of this chapter, I have a trained model, and the best weights have been saved based on performance on the validation set.

# Chapter 4

# Experiments and Evaluation

After completing the model training process in the previous chapter, in this chapter, I will load the model with the best saved weights and evaluate its performance on the test set. The experimental results obtained will be presented and analyzed in detail.

## 4.1 Evaluation Process

To evaluate the model objectively, I perform the following steps:

1. **Load the best model:** I load the weights of the model that achieved the best results on the validation set during training. These weights are saved in the file `best_model.pth`.

```
if __name__ == '__main__':
#...
model.load_state_dict(torch.load('best_model.pth'))
```

Code Listing 4.1: Load best model weights

2. **Perform predictions on the test set:** I use the `run` function (described in detail in Chapter 3, section referring to 3.5) to have the model predict on the entire `test_loader`. This function returns lists of true labels (`y_true`) and predicted labels (`y_pred`).

```
if __name__ == '__main__':
    # ... (model weights loaded)
    # Ensure test_loader is in scope from `train_loader, val_loader,
    ↪   test_loader = load_data()`
    y_true, y_pred = run(model, test_loader, return_preds=True)
```

Code Listing 4.2: Get predictions on the test set

3. **Calculate evaluation metrics:** From `y_true` and `y_pred`, I calculate overall accuracy, the confusion matrix, and class-wise accuracy.

## 4.2 Experimental Results

Below are the detailed results I obtained after the evaluation process.

### 4.2.1 Overall Accuracy

Accuracy is one of the most important metrics for evaluating the performance of a classification model.

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

Code Listing 4.3: Calculate accuracy on the test set

**Results:**

- Best Validation Accuracy: **84.92%**
- Test Accuracy: **85.25%**

**Remarks:** I observe that the test accuracy (85.25%) is slightly higher than the best validation accuracy (84.92%). This is a very positive sign, indicating that the model is not only not overfitted but also has good generalization ability on completely new, unseen data. This level of accuracy is quite good for the CIFAR-10 dataset with a custom CNN model.

### 4.2.2 Learning Curves

The learning curves, including the changes in Loss and Accuracy on the training and validation sets over 50 epochs, are presented in Figure 4.1.

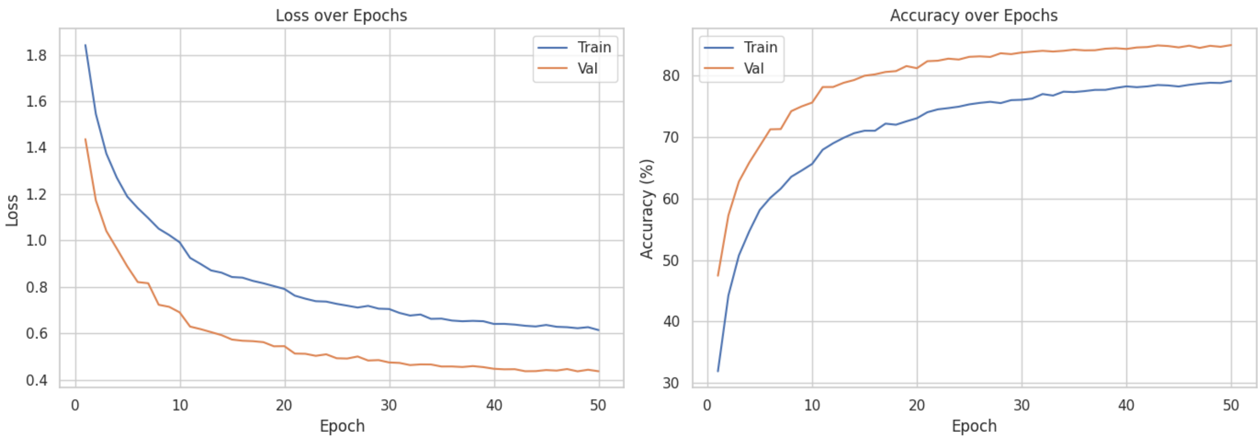


Figure 4.1: Loss function and accuracy graphs over epochs.

**Remarks:**

**Loss Function (Loss):** The loss curves on both the training (Train) and validation (Val) sets show a decreasing trend over epochs. Initially, val loss decreases faster than train loss, but after about 5-7 epochs, train loss starts to decrease more rapidly and remains lower than val loss. Towards the end of the training process, both loss curves show signs of flattening, indicating that the model has converged quite well. Val loss at around 0.436 suggests the model learned useful features.

**Accuracy (Accuracy):** Similarly, accuracy on both sets increases. Val accuracy increases rapidly in the early epochs and reaches about 84.92

**Early Stopping:** The message "Early stopping at epoch 50" (assuming this was an output, the graph shows 50 epochs completed) indicates that the training process ran for the full 50 planned epochs. This means the early stopping condition (no improvement in val_loss for 7 consecutive epochs with min_delta of 0.001) was not triggered before the final epoch, or was just met at the 50th epoch. Looking at the val loss graph, its flatness in the final epochs also emphasizes that the model was close to saturation.

Overall, the graphs show that the training process was stable, the model learned effectively, and there are no clear signs of overfitting.

### 4.2.3 Confusion Matrix

To analyze the model's classification ability for each class in more detail, I created a confusion matrix from the prediction results on the test set.

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

Code Listing 4.4: Calculate confusion matrix

The confusion matrix is visualized in Figure 4.2.

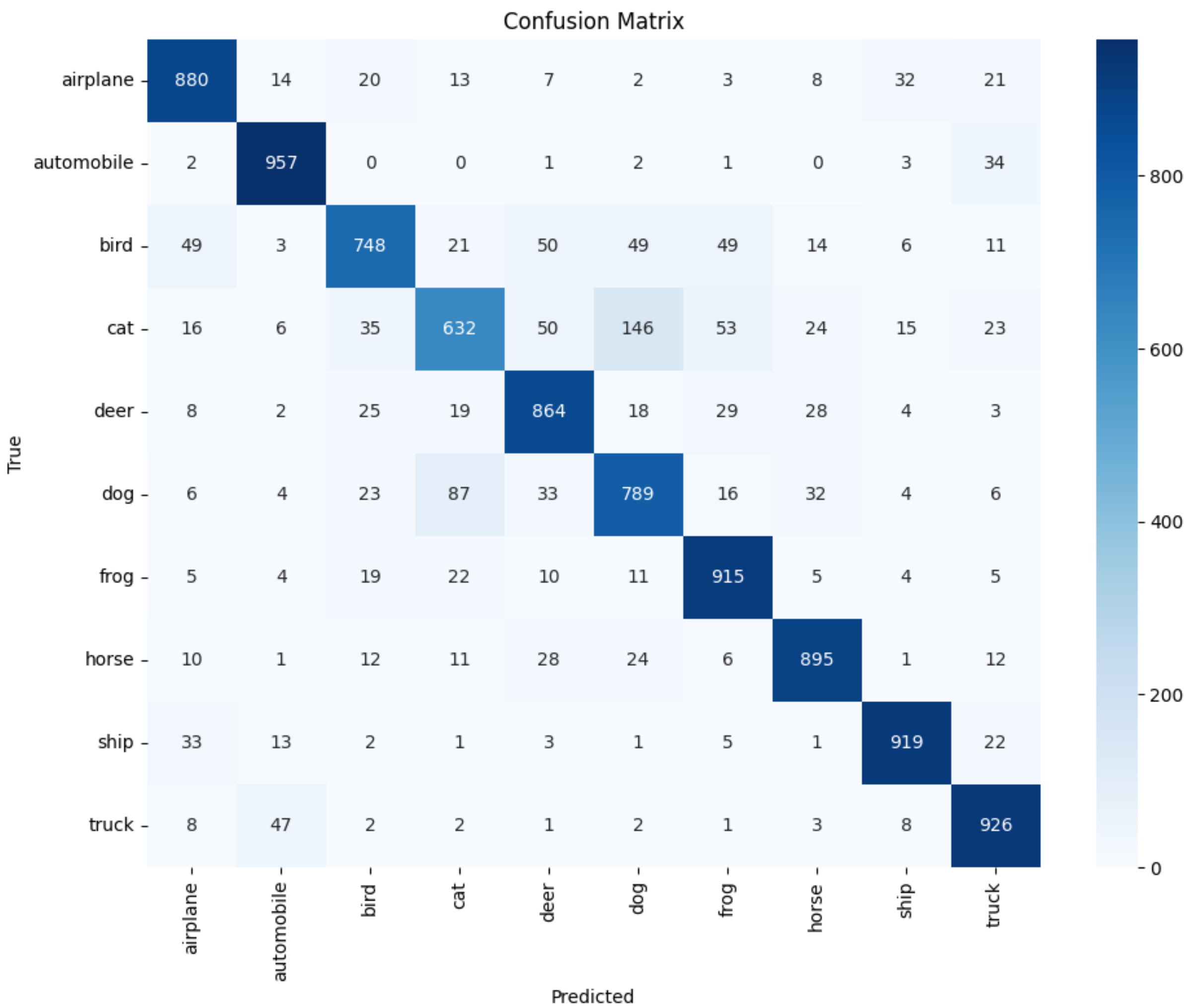


Figure 4.2: Confusion matrix on the test set.

**Remarks:** Observing the confusion matrix, I notice:

The values on the main diagonal of the matrix are relatively high, indicating that the model correctly predicted most samples belonging to each class.

**Well-classified classes:** The model classifies 'automobile' (957/1000 correct samples), 'truck' (926/1000), 'ship' (919/1000), and 'frog' (915/1000) very well.

**Misclassified classes:**

- The 'cat' class seems to be the hardest to classify, with only 632 samples correctly predicted. It is most confused with 'dog' (146 samples), 'deer' (50 samples), and 'frog' (53 samples). Confusion between cat and dog is quite common in image classification tasks.
- The 'bird' class also has a not-so-high accuracy (748 correct samples), being confused with 'deer' (50 samples), 'dog' (49 samples), 'airplane' (49 samples), and 'frog' (49 samples).
- The 'dog' class is often confused with 'cat' (87 samples) and 'deer' (33 samples).

The 'airplane' and 'ship' classes are sometimes confused with each other or with other vehicles, but not to a significant extent.

The confusion matrix provides a detailed insight into the model's strengths and weaknesses for each specific class.

### 4.2.4 Class-wise Accuracy

Based on the confusion matrix, I calculate the specific accuracy for each class as follows:

```
if __name__ == '__main__':
#...
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}%")
```

Code Listing 4.5: Calculate class-wise accuracy

**Detailed results:**

| Object Type | Accuracy |
|---|---|
| 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% |

Table 4.1: Accuracy by object type

These figures reaffirm what I observed from the confusion matrix:

1. The 'automobile' and 'truck' classes have the highest accuracy, above 92%. The 'frog', 'ship', 'horse', 'airplane', 'deer' classes also achieve good accuracy, from 86% upwards.

2. The 'cat' class has the lowest accuracy (63.20%), followed by 'bird' (74.80%) and 'dog' (78.90%). These are the three classes where the model faces the most difficulty in accurate classification, possibly due to the diversity in shape, color, posture, and similarity in features among them in the CIFAR-10 dataset.

## 4.3 General Remarks on Experimental Results

In conclusion, the CNN model that I built and trained achieved an overall accuracy of 85.25% on the CIFAR-10 test set. The training process was stable, and the model demonstrated good generalization ability. Techniques such as data augmentation, batch normalization, dropout, and learning rate scheduling contributed to this result. Although the model performs very well on some classes, there are still challenges with classes that have complex visual features and are easily confused, such as 'cat', 'bird', and 'dog'. These are areas that can be focused on for improvement in future research or experiments.