POST AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

PYTHON PROGRAMMING

PYTHON ASSIGNMENT 2

LECTURER : KIM NGOC BACH
STUDENT NAME : NGUYEN TIEN THANG - B23DCCE088
: NGUYEN ANH TRUONG - B23DCCE094
CLASS : D23CQCE04-B

Ha Noi – 2025

Contents

PREFACE

The field of artificial intelligence, particularly deep learning, has witnessed remarkable advancements in recent years, with image classification standing as a cornerstone application. This project delves into the practical implementation and comparative analysis of two foundational neural network architectures: the Multi-Layer Perceptron (MLP) and the Convolutional Neural Network (CNN).

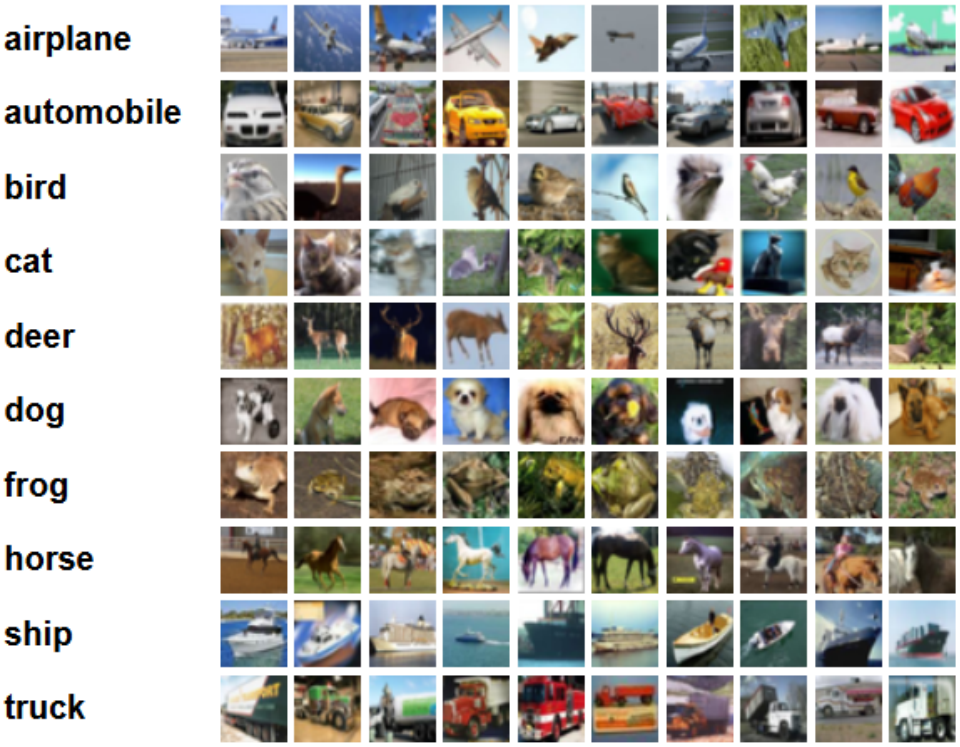
Our exploration is centered on the CIFAR-10 dataset, a widely recognized benchmark that, despite its modest image resolution, presents a rich learning ground for understanding the intricacies of feature extraction and pattern recognition in visual data. Through this assignment, we aimed not only to fulfill academic requirements but also to gain hands-on experience with the PyTorch framework and to empirically observe the striking differences in how these architectures approach and solve image-based challenges.

The journey involved navigating data preprocessing, model design, training intricacies, and rigorous evaluation. The ensuing report documents our methodology, presents the empirical results, and offers a discussion on the performance disparities observed, particularly highlighting the inherent strengths of CNNs in handling spatial data.

We would like to acknowledge the guidance provided by lecturer Kim Ngoc Bach. This project represents a collaborative effort between Nguyen Tien Thang and Nguyen Anh Truong, with both members contributing to the coding, analysis, and report generation. We hope this work provides a clear insight into our learning process and the capabilities of MLP and CNN models in the realm of image classification.

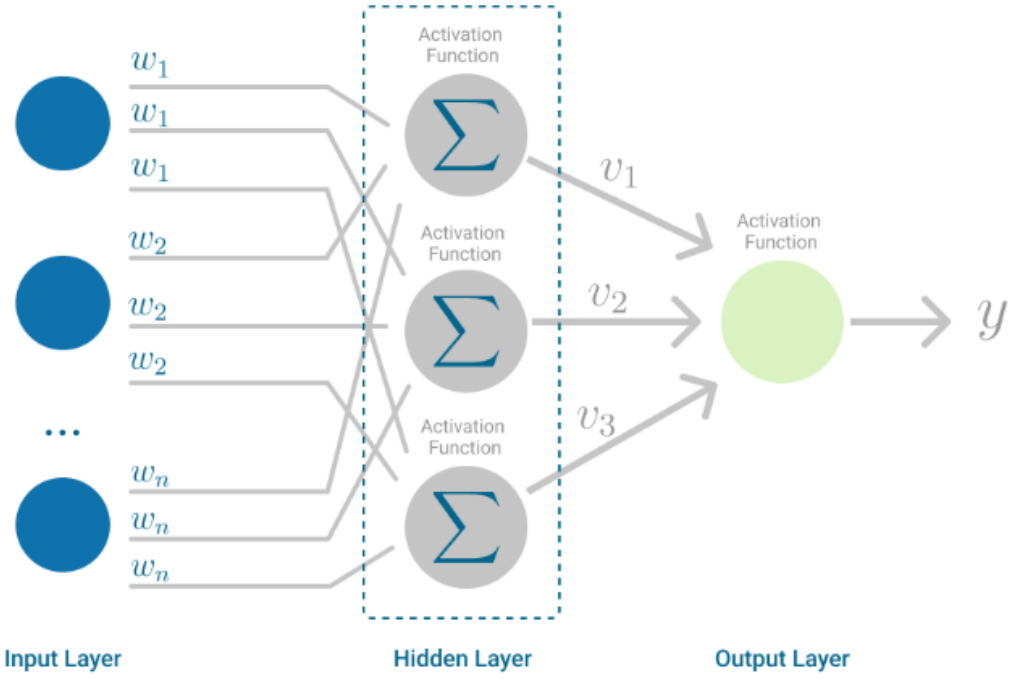
1 INTRODUCTION TO CIFAR-10

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.
Here are the classes in the dataset, as well as 10 random images from each:



The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

2 INTRODUCTION TO MLP (Multi-Layer Perceptron)



A Multi-Layer Perceptron (MLP) is a fundamental type of feedforward artificial neural network. It consists of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. Except for the input nodes, each node is a neuron that uses a non-linear activation function. MLPs are a class of universal approximators, meaning they can approximate any continuous function, which makes them suitable for a variety of learning tasks.
A basic MLP typically includes the following components:

2.1 Input Layer

This layer receives the initial data for the network. For image data, this typically involves flattening the image from its 2D (or 3D if color channels are considered) structure into a 1D vector. Each element of this vector then corresponds to an input node. No computation is performed in the input layer itself; it simply passes the data to the first hidden layer.

2.2 Hidden Layers

These layers are positioned between the input and output layers and are crucial for the MLP's ability to learn complex patterns.

- Each neuron in a hidden layer is fully connected to all neurons in the previous layer. This means each connection has a weight, and each neuron has a bias.
- The weighted sum of inputs to a neuron is passed through an activation function (like ReLU, Sigmoid, or Tanh) to introduce non-linearity, allowing the network to learn more than just linear relationships.

- An MLP can have one or multiple hidden layers. Increasing the number of hidden layers or the number of neurons within them can increase the model's capacity but also the risk of overfitting and computational cost.

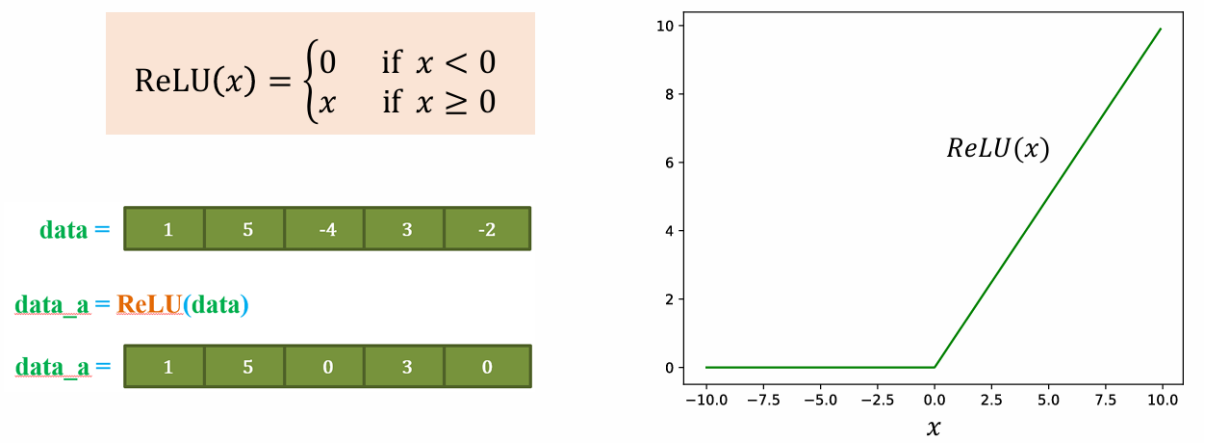
2.3 Output Layer

This is the final layer of the network and produces the model's prediction.

- Like hidden layers, neurons in the output layer are typically fully connected to the neurons in the last hidden layer.
- The choice of activation function for the output layer depends on the task:
  - For regression tasks, a linear activation function is often used.
  - For binary classification, a sigmoid activation function is common.
  - For multi-class classification (like CIFAR-10), a softmax activation function (often implicitly combined with the loss function like CrossEntropyLoss in PyTorch) is used to output probability-like scores for each class.
- The number of neurons in the output layer corresponds to the number of values the model needs to output (e.g., 10 neurons for 10 classes in CIFAR-10).

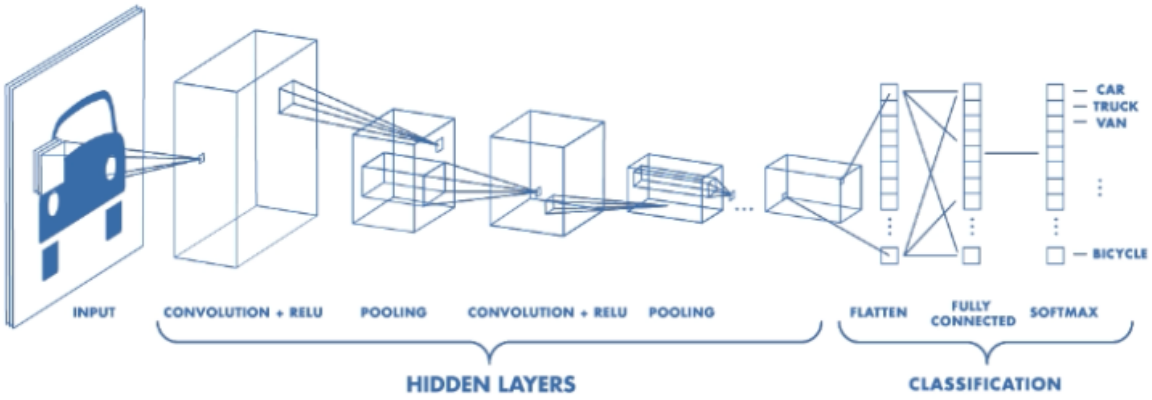
2.4 Activation Functions

As with CNNs, activation functions like ReLU (f(x)=max(0,x)) are vital in MLPs. They introduce non-linearities that enable the network to learn complex mappings from inputs to outputs. Without non-linear activation functions, an MLP, no matter how many layers it has, would behave like a single linear model.



Training and Application: MLPs are trained using an optimization algorithm like gradient descent (or its variants like Adam, SGD) along with backpropagation to adjust the weights and biases based on a defined loss function. While powerful, MLPs process input data as flat vectors, which means they do not inherently take advantage of spatial hierarchies or local correlations present in data like images. This is a key difference and often a limitation when compared to CNNs for image-related tasks. However, MLPs remain crucial for many other types of data and as components within more complex architectures.

3 INTRODUCTION TO CNN (Convolutional Neural Network)

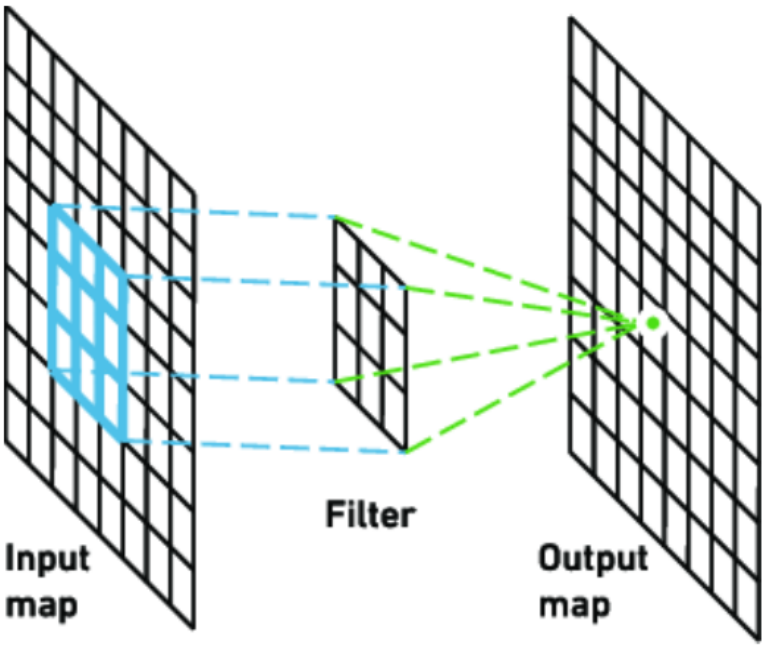


A Convolutional Neural Network (CNN) is a deep learning architecture composed of multiple layers stacked together, often utilizing non-linear activation functions like ReLU to introduce complex pattern recognition capabilities at each node. Each successive layer, after passing through an activation function, tends to generate increasingly abstract information from the preceding layer's output. This hierarchical feature extraction makes CNNs one of the most advanced Deep Learning models, enabling the development of highly accurate intelligent systems.
A basic CNN typically includes the following fundamental layers:

3.1 Convolutional Layer

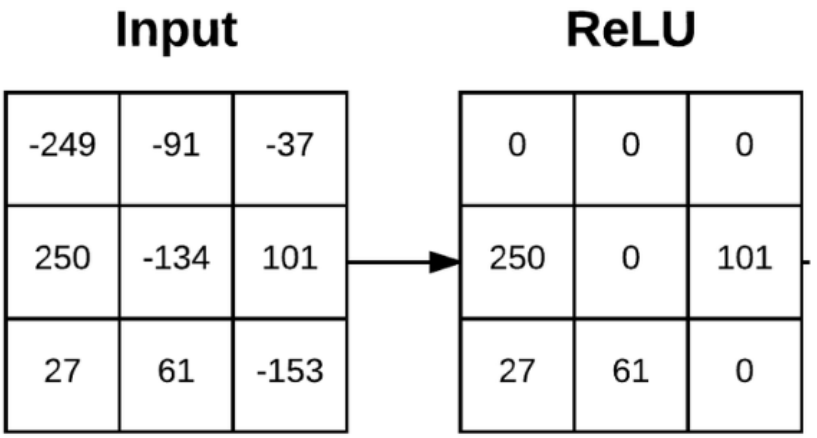
This is the most crucial layer in a CNN, responsible for the primary computational tasks related to feature extraction. Increasing the number of convolutional layers generally leads to improved model performance, though often 3 to 4 well-structured layers can yield desired results. Key elements that define a convolutional layer's operation include: stride, padding, filters (kernels), and the resulting feature maps.
CNNs use filters (also known as kernels) that are applied to regions of the input image. These filters can be visualized as small matrices containing numerical values, which are the learnable parameters of the layer.

- Stride: This defines the step size by which the filter moves across the image, pixel by pixel, from left to right and top to bottom.
- Padding: This refers to adding border pixels (often with a value of 0) around the input image. Padding can help control the spatial dimensions of the output feature map and ensure that pixels at the edges of the image are processed adequately by the filters.
- Feature Map: This is the output generated each time a filter scans across the input. It represents the presence of specific learned features (like edges, textures, etc.) at different spatial locations in the input. The collection of feature maps forms the output of the convolutional layer and serves as input to subsequent layers or pooling operations.



3.2 ReLU Activation Function

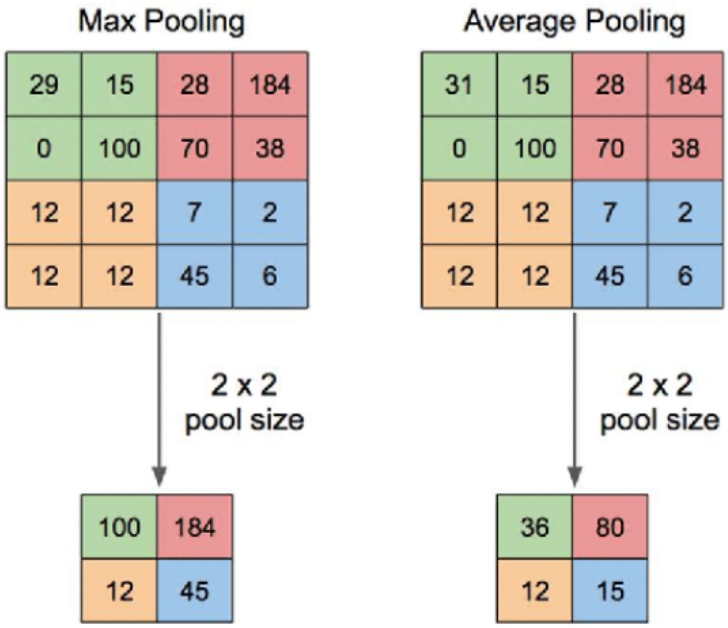
The ReLU (Rectified Linear Unit) layer is a widely used activation function in neural networks. Its mathematical definition is f(x)=max(0,x). This activation function is designed to mimic the way neurons fire proportionally to the input they receive beyond a certain threshold.
While "activation function" is a general term, specific examples include ReLU, Leaky ReLU, Tanh, Sigmoid, Maxout, among others. Currently, ReLU is one of the most popular and commonly used activation functions due to its simplicity and effectiveness.
It is frequently employed in training neural networks because it offers several notable advantages, such as speeding up computation (compared to sigmoid or tanh, its derivative is simpler). When using ReLU, it's important to consider factors like appropriate learning rate selection and monitoring for "dead neurons" (neurons that always output zero and stop learning).
In the context of a CNN, a ReLU layer is typically applied after the feature maps are computed by a convolutional layer, applying the ReLU function element-wise to the values in these feature maps.



3.3 Pooling Layer

The Pooling layer, also known as a subsampling or downsampling layer, serves to simplify the information (feature maps) output by convolutional layers. Its primary purpose is to reduce the dimensionality (spatial size like height and width) of the feature maps, which in turn reduces the number of parameters and computations in subsequent layers. This also helps in making the learned features somewhat invariant to small translations in the input image. Pooling layers are typically used immediately after a convolutional layer (and its activation function).
There are two common types of pooling layers:

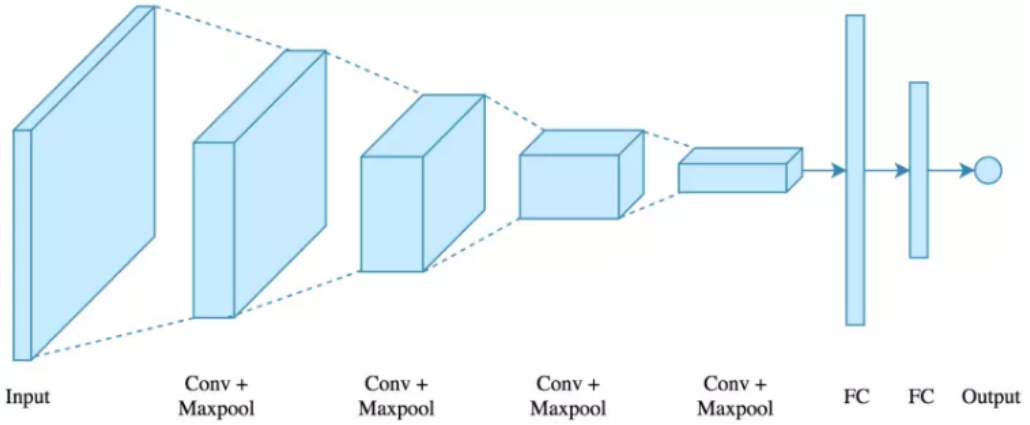
- Max Pooling: Selects the maximum value from each patch of the feature map.
- Average Pooling: Calculates the average value from each patch of the feature map.



The purpose of pooling is straightforward: it reduces the number of parameters (by reducing feature map dimensions) that need to be processed, thereby decreasing computation time and helping to prevent overfitting.
The most commonly encountered type of pooling is Max Pooling, which takes the largest value from within a defined pooling window. Pooling operates somewhat similarly to convolution: it has a window (often called a "pooling window" or kernel) that slides across the input feature map. From the values within this window at each position, a single value is selected (for max pooling, this is the maximum value).
For example, as illustrated, if we choose a pooling window of size 2x2 and a stride of 2, this ensures that the windows do not overlap (or have minimal overlap depending on stride and kernel size relative to each other) when applying either max pooling or average pooling. This 2x2 pooling with a stride of 2 is a very common configuration that effectively halves the height and width of the input feature map.

3.4 Fully Connected Layer

This layer has the task of producing results after the convolutional layer and pooling layer have received the transmitted image. At this point, the result we obtain is that the model has read the image's information, and to link them together as well as to produce multiple outputs, we use a fully connected layer.
Additionally, if a fully connected layer receives image data, it will turn it into an item that has not been quality-classified. This is quite similar to a ballot where it will be evaluated to select the image with the highest quality.
Similar to a neural network, each hidden layer is called fully connected. Usually, after the fully connected layer, there are 2 fully connected layers: 1 layer to gather the features we have found, converting data from 3D or 2D to 1D, meaning only 1 vector remains. The other layer is the output layer; the number of neurons in this layer depends on the number of outputs we want to find.
Example of a standard CNN architecture:



Based on the idea of a standard CNN, along with data contributions from large-scale image recognition challenges like ImageNet or ILSVRC, many CNN models with ever-improving accuracy and more CNN variations have emerged. Some notable CNN models and variations have achieved high results in image classification tasks.
To evaluate a model, first, there needs to be a standard dataset and a metric for accuracy to ensure fairness. In image classification tasks, ImageNet is considered one of the most reliable and intuitive datasets for evaluating recognition models through annual competitions.

4 SOLUTION

4.1 Import libraries and set up devices

Import Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split

import torchvision
import torchvision.transforms as transforms

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

Setting Up Device (CPU or GPU)

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

```
Using device : cuda
```

The first step in the notebook was to establish the working environment. This involved importing essential Python libraries:

- `torch` and its submodules `torch.nn` (for neural network layers and functions) and `torch.optim` (for optimization algorithms).
- `torch.utils.data.DataLoader` and `torch.utils.data.random_split` for efficient data handling and dataset splitting.
- `torchvision` and `torchvision.transforms` for accessing the CIFAR-10 dataset and applying image transformations.
- Standard data science and plotting libraries: `numpy` for numerical operations, `matplotlib.pyplot` and `seaborn` for visualizations, and `sklearn.metrics.confusion_matrix` for evaluation.

To leverage hardware acceleration, the code determined the available computation device:

- `torch.device("cuda" if torch.cuda.is_available() else "cpu")` was used to select a CUDA-enabled GPU if present, otherwise defaulting to the CPU.
- The notebook output confirmed that a "cuda" device was utilized.

4.2 Data Loading and Preprocessing

The CIFAR-10 dataset was prepared as follows:

Image Transformations

```
# Transformations for the CIFAR-10 dataset
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

- A `transform_train` pipeline was defined for the training data. This included `transforms.RandomCrop(32, padding=4)` for spatial augmentation `transforms.RandomHorizontalFlip()` for further augmentation, `transforms.ToTensor()` to convert images to PyTorch tensors, and `transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))` using the standard CIFAR-10 mean and standard deviation for each channel
- A `transform_test` pipeline was defined for the test data, including only `transforms.ToTensor()` and the same normalization.

Dataset Loading

```
# Load CIFAR-10 training dataset
train_dataset = torchvision.datasets.CIFAR10(root='./data',
                                             train=True,
                                             download=True,
                                             transform=transform_train)

# Load CIFAR-10 test dataset
test_dataset = torchvision.datasets.CIFAR10(root='./data',
                                            train=False,
                                            download=True,
                                            transform=transform_test)

# Define classes for CIFAR-10
classes = ('airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

- The CIFAR-10 training dataset (50,000 images) was loaded using `torchvision.datasets.CIFAR10`, specifying `train=True`, `download=True`, and applying `transform_train`.
- The CIFAR-10 test dataset (10,000 images) was loaded similarly, with `train=False` and applying `transform_test`.
- A tuple `classes` was defined to store the names of the 10 CIFAR-10 categories.

Train/Validation Split

Splitting Training Data into Training and Validation Sets

```
[ ]  validation_split = 0.2
     num_train_samples = len(train_dataset)
     num_val_samples = int(validation_split * num_train_samples)
     num_train_samples = num_train_samples - num_val_samples

     train_dataset, val_dataset = random_split(train_dataset, [num_train_samples, num_val_samples])

     print(f"Number of training samples : {len(train_dataset)}")
     print(f"Number of validation samples : {len(val_dataset)}")
     print(f"Number of test samples : {len(test_dataset)}")
```

```
Number of training samples : 40000
Number of validation samples : 10000
Number of test samples : 10000
```

- A `validation_split` ratio of 0.2 (20%) was set.
- The number of samples for the validation set (`num_val_samples`) and the new training set (`num_train_samples`) were calculated based on the original training dataset size.
- The `random_split` function was then used to divide the original `train_dataset` (with augmentations) into the final `train_dataset` (40,000 images) and `val_dataset` (10,000 images). The notebook output confirmed these counts.

DataLoaders Creation

Creating DataLoaders

```
[ ]  batch_size = 64

     train_loader = DataLoader(dataset=train_dataset,
                               batch_size=batch_size,
                               shuffle=True,
                               num_workers=2)

     val_loader = DataLoader(dataset=val_dataset,
                             batch_size=batch_size,
                             shuffle=False,
                             num_workers=2)

     test_loader = DataLoader(dataset=test_dataset,
                              batch_size=batch_size,
                              shuffle=False,
                              num_workers=2)
```

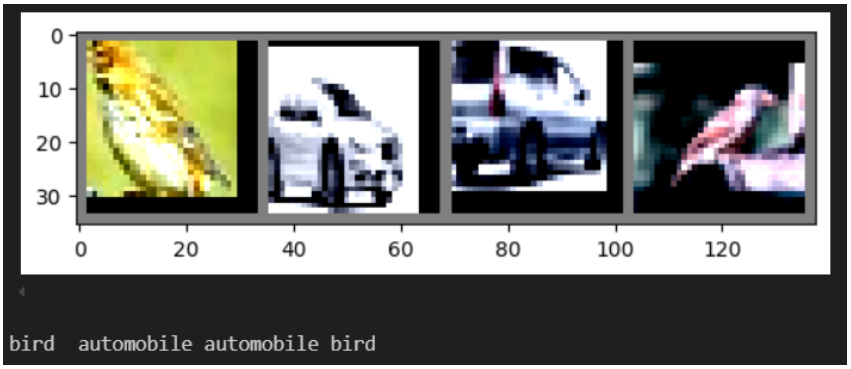
- A `batch_size` of 64 was defined.
- DataLoader instances (`train_loader`, `val_loader`, `test_loader`) were created for each dataset split.
- `shuffle=True` was set for `train_loader` to ensure data randomization during training epochs, while `shuffle=False` was used for `val_loader` and `test_loader`.
- `num_workers=2` was specified for all loaders to enable parallel data fetching.

Sample Visualization

Visualize Some Training Images

```
def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0))) # Convert from CxHxW to HxWxC for displaying
    plt.show()

# Get some random training images
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Show images
imshow(torchvision.utils.make_grid(images[:4]))

# Print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
```

- A function `imshow` was defined to display sample images from the `train_loader` using `matplotlib.pyplot` and `torchvision.utils.make_grid`, demonstrating the loaded data and their labels. The output cell showed a grid of 4 sample images with their corresponding class names.



4.3 Multi-Layer Perceptron (MLP) Implementation

Model Architecture Definition

Defining the MLP Model

```
[ ]  class MLP(nn.Module):
         def __init__(self, input_size, hidden_size1, hidden_size2, num_classes = 10):
             super(MLP, self).__init__()
             self.flatten = nn.Flatten()
             self.fc1 = nn.Linear(input_size, hidden_size1)
             self.relu1 = nn.ReLU()
             self.fc2 = nn.Linear(hidden_size1, hidden_size2)
             self.relu2 = nn.ReLU()
             self.fc3 = nn.Linear(hidden_size2, num_classes)

         def forward(self, x):
             x = self.flatten(x)
             x = self.fc1(x)
             x = self.relu1(x)
             x = self.fc2(x)
             x = self.relu2(x)
             x = self.fc3(x)
             return x
```

- An `MLP` class was defined, inheriting from `nn.Module`.
- The constructor `__init__` defined the layers:
  - `nn.Flatten()` to convert 3D image tensors into 1D vectors.
  - `self.fc1 = nn.Linear(input_size, hidden_size1)`: A fully connected layer from the flattened input size to 512 units.
  - `self.relu1 = nn.ReLU()`: A ReLU activation function.
  - `self.fc2 = nn.Linear(hidden_size1, hidden_size2)`: A second fully connected layer from 512 to 256 units.
  - `self.relu2 = nn.ReLU()`: Another ReLU activation.
  - `self.fc3 = nn.Linear(hidden_size2, num_classes)`: The output layer mapping 256 units to the 10 classes.
- The `forward` method detailed the sequential flow of data through these layers.

Model Instantiation and Training Setup

Instantiate the Model, Define Loss Function and Optimizer

```
[ ]  input_size_mlp = 3 * 32 * 32
     hidden_size1_mlp = 512
     hidden_size2_mlp = 256
     num_classes = 10
     learning_rate_mlp = 0.001
     num_epochs_mlp = 20

     # Create the MLP model instance
     mlp_model = MLP(input_size_mlp, hidden_size1_mlp, hidden_size2_mlp, num_classes).to(device)

     # Loss function
     criterion = nn.CrossEntropyLoss()

     # Optimizer
     optimizer_mlp = optim.Adam(mlp_model.parameters(), lr=learning_rate_mlp)
```

- Hyperparameters were defined: `input_size_mlp = 3 * 32 * 32` (3072), `hidden_size1_mlp = 512`, `hidden_size2_mlp = 256`, `num_classes = 10`, `learning_rate_mlp = 0.001`, and `num_epochs_mlp = 20`.
- An instance of the MLP model (`mlp_model`) was created with these parameters and moved to the configured device using `.to(device)`.
- The loss function was set to `criterion = nn.CrossEntropyLoss()`.
- The optimizer was configured as `optimizer_mlp = optim.Adam(mlp_model.parameters(), lr=learning_rate_mlp)`.

MLP Training and Validation Loop

Training and Validation Loop for MLP

```
# Lists to store metrics for MLP
mlp_train_losses = []
mlp_train_accuracies = []
mlp_val_losses = []
mlp_val_accuracies = []

print("Starting MLP Training...")
for epoch in range(num_epochs_mlp):
```

- Lists (`mlp_train_losses`, `mlp_train_accuracies`, etc.) were initialized to store metrics for each epoch.
- A `for` loop iterated `num_epochs_mlp times.`

- **Training Phase (per epoch)**:

```
# --- Training ---
mlp_model.train()
running_train_loss = 0.0
correct_train = 0
total_train = 0

for i, (images, labels) in enumerate(train_loader):
    images = images.to(device)
    labels = labels.to(device)

    # Forward pass
    outputs = mlp_model(images)
    loss = criterion(outputs, labels)

    # Backward pass and optimization
    optimizer_mlp.zero_grad()
    loss.backward()
    optimizer_mlp.step()

    # Accumulate loss
    running_train_loss += loss.item() * images.size(0)

    # Calculate accuracy
    _, predicted = torch.max(outputs.data, 1)
    total_train += labels.size(0)
    correct_train += (predicted == labels).sum().item()

epoch_train_loss = running_train_loss / len(train_dataset)
epoch_train_accuracy = 100 * correct_train / total_train
mlp_train_losses.append(epoch_train_loss)
mlp_train_accuracies.append(epoch_train_accuracy)
```

  - `mlp_model.train()` set the model to training mode.
  - Variables for running loss and correct predictions were initialized.
  - The code iterated through `train_loader` (providing images and labels). Data was moved to `device`.
  - The optimizer's gradients were zeroed (`optimizer_mlp.zero_grad()`).
  - A forward pass was made (`outputs = mlp_model(images)`).
  - Loss was calculated (`loss = criterion(outputs, labels)`).
  - Gradients were computed via backpropagation (`loss.backward()`).
  - Model parameters were updated (`optimizer_mlp.step()`).
  - Batch loss and accuracy were accumulated.

- **Validation Phase (per epoch)**:

```
# --- Validation ---
mlp_model.eval()
running_val_loss = 0.0
correct_val = 0
total_val = 0
with torch.no_grad():
    for images, labels in val_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = mlp_model(images)
        loss = criterion(outputs, labels)
        running_val_loss += loss.item() * images.size(0)

        _, predicted = torch.max(outputs.data, 1)
        total_val += labels.size(0)
        correct_val += (predicted == labels).sum().item()

epoch_val_loss = running_val_loss / len(val_dataset)
epoch_val_accuracy = 100 * correct_val / total_val
mlp_val_losses.append(epoch_val_loss)
mlp_val_accuracies.append(epoch_val_accuracy)
```

  - `mlp_model.eval()` set the model to evaluation mode.
  - Within `with torch.no_grad()`: to disable gradient computations, the code iterated through `val_loader`.
  - In the loop:
    - Move data to the device.
    - Run forward pass, compute and accumulate loss.
    - Run forward pass, count correct ones.
  - After the loop:
    - Compute average validation loss and accuracy.
    - Store results for later analysis or plotting.

- Epoch-wise average training and validation loss/accuracy were computed, appended to their respective lists, and printed. The notebook output shows this printout for each of the 20 epochs.
  - After each epoch, average training and validation loss and accuracy were computed.
  - These values were added to their respective tracking lists.
  - A formatted print statement showed progress at each epoch.
  - The final output confirms the end of training after 20 epochs.

```
    print(f"MLP Epoch [{epoch+1}/{num_epochs_mlp}], "
          f"Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_accuracy:.2f}%, "
          f"Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_accuracy:.2f}%")

print("MLP Training Finished.")
```

```
Starting MLP Training...
MLP Epoch [1/20], Train Loss: 1.8415, Train Acc: 33.23%, Val Loss: 1.7602, Val Acc: 35.94%
MLP Epoch [2/20], Train Loss: 1.6855, Train Acc: 39.22%, Val Loss: 1.6468, Val Acc: 40.59%
MLP Epoch [3/20], Train Loss: 1.6181, Train Acc: 41.73%, Val Loss: 1.5829, Val Acc: 43.09%
MLP Epoch [4/20], Train Loss: 1.5837, Train Acc: 42.89%, Val Loss: 1.5656, Val Acc: 44.22%
MLP Epoch [5/20], Train Loss: 1.5493, Train Acc: 44.26%, Val Loss: 1.5622, Val Acc: 44.02%
MLP Epoch [6/20], Train Loss: 1.5266, Train Acc: 45.08%, Val Loss: 1.5572, Val Acc: 44.76%
MLP Epoch [7/20], Train Loss: 1.5089, Train Acc: 45.67%, Val Loss: 1.5202, Val Acc: 45.44%
MLP Epoch [8/20], Train Loss: 1.4944, Train Acc: 46.25%, Val Loss: 1.5023, Val Acc: 46.28%
MLP Epoch [9/20], Train Loss: 1.4736, Train Acc: 47.03%, Val Loss: 1.5017, Val Acc: 45.85%
MLP Epoch [10/20], Train Loss: 1.4645, Train Acc: 47.42%, Val Loss: 1.5025, Val Acc: 46.29%
MLP Epoch [11/20], Train Loss: 1.4540, Train Acc: 47.94%, Val Loss: 1.4703, Val Acc: 47.14%
MLP Epoch [12/20], Train Loss: 1.4430, Train Acc: 48.60%, Val Loss: 1.4812, Val Acc: 46.69%
MLP Epoch [13/20], Train Loss: 1.4337, Train Acc: 48.57%, Val Loss: 1.4640, Val Acc: 48.06%
MLP Epoch [14/20], Train Loss: 1.4275, Train Acc: 48.67%, Val Loss: 1.4451, Val Acc: 48.53%
MLP Epoch [15/20], Train Loss: 1.4166, Train Acc: 49.00%, Val Loss: 1.4375, Val Acc: 48.58%
MLP Epoch [16/20], Train Loss: 1.4090, Train Acc: 49.42%, Val Loss: 1.4390, Val Acc: 48.44%
MLP Epoch [17/20], Train Loss: 1.4112, Train Acc: 49.47%, Val Loss: 1.4554, Val Acc: 48.29%
MLP Epoch [18/20], Train Loss: 1.4077, Train Acc: 49.58%, Val Loss: 1.4362, Val Acc: 49.12%
MLP Epoch [19/20], Train Loss: 1.3950, Train Acc: 50.00%, Val Loss: 1.4374, Val Acc: 48.29%
MLP Epoch [20/20], Train Loss: 1.3981, Train Acc: 50.25%, Val Loss: 1.4235, Val Acc: 49.23%
MLP Training Finished.
```

4.4 Convolutional Neural Network (CNN) Implementation

Model Architecture Definition

- A `CNN` class was defined, inheriting from `nn.Module`.

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

- The constructor `__init__` defined the layers:

```
def __init__(self, num_classes=10):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.flattened_size = 64 * 4 * 4

    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(self.flattened_size, 128)
    self.relu4 = nn.ReLU()
    self.fc2 = nn.Linear(128, num_classes)
```

  - Convolutional Block 1: `nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)`, followed by `nn.ReLU()` and `nn.MaxPool2d(kernel_size=2, stride=2)`.
  - Convolutional Block 2: `nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)`, `nn.ReLU()`, `nn.MaxPool2d(kernel_size=2, stride=2)`. (A code snippet for conv2 and pool2 is shown in image : `self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)`, `self.pool2 = nn.MaxPool2d()`)
  - Convolutional Block 3: `nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)`, `nn.ReLU()`, `nn.MaxPool2d(kernel_size=2, stride=2)`.
  - `self.flattened_size` was calculated as `64 * 4 * 4` based on the output dimensions after three pooling layers.
  - `nn.Flatten()` layer.
  - `self.fc1 = nn.Linear(self.flattened_size, 128)` followed by `nn.ReLU()`.
  - `self.fc2 = nn.Linear(128, num_classes)` as the output layer.

- The `forward` method defined the sequential data flow through these blocks and layers.

```
def forward(self, x):
    # Layer 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Layer 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Layer 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)

    # Flatten and pass to fully connected layers
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu4(x)
    x = self.fc2(x)
    return x
```

Model Instantiation and Training Setup

Instantiate the CNN Model, Define Loss Function and Optimizer

```
# Hyperparameters for the CNN model and training
learning_rate_cnn = 0.001
num_epochs_cnn = 25

# Create the CNN model instance
cnn_model = CNN(num_classes=num_classes).to(device)

# Optimizer
optimizer_cnn = optim.Adam(cnn_model.parameters(), lr=learning_rate_cnn)
```

- Hyperparameters: `learning_rate_cnn = 0.001` and `num_epochs_cnn = 25`.
- An instance `cnn_model` was created and moved to `device`.
- `optimizer_cnn = optim.Adam(cnn_model.parameters(), lr=learning_rate_cnn)` was configured.

CNN Training and Validation Loop

Training and Validation Loop for CNN

```
[ ]  # Lists to store metrics for CNN
     cnn_train_losses = []
     cnn_train_accuracies = []
     cnn_val_losses = []
     cnn_val_accuracies = []

     print("\nStarting CNN Training...")
     for epoch in range(num_epochs_cnn):
         # --- Training ---
         cnn_model.train()
         running_train_loss = 0.0
         correct_train = 0
         total_train = 0

         for i, (images, labels) in enumerate(train_loader):
             images = images.to(device)
             labels = labels.to(device)

             # Forward pass
             outputs = cnn_model(images)
             loss = criterion(outputs, labels)

             # Backward pass and optimization
             optimizer_cnn.zero_grad()
             loss.backward()
             optimizer_cnn.step()
```

```
        running_train_loss += loss.item() * images.size(0)
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    epoch_train_loss = running_train_loss / len(train_dataset)
    epoch_train_accuracy = 100 * correct_train / total_train
    cnn_train_losses.append(epoch_train_loss)
    cnn_train_accuracies.append(epoch_train_accuracy)

    # --- Validation ---
    cnn_model.eval()
    running_val_loss = 0.0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = cnn_model(images)
            loss = criterion(outputs, labels)
            running_val_loss += loss.item() * images.size(0)

            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    epoch_val_loss = running_val_loss / len(val_dataset)
    epoch_val_accuracy = 100 * correct_val / total_val
    cnn_val_losses.append(epoch_val_loss)
    cnn_val_accuracies.append(epoch_val_accuracy)

    print(f"CNN Epoch [{epoch+1}/{num_epochs_cnn}], "
          f"Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_accuracy:.2f}%, "
          f"Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_accuracy:.2f}%")

print("CNN Training Finished.")
```

- This loop was structurally identical to the MLP's training/validation loop, using the CNN-specific model (`cnn_model`), optimizer (`optimizer_cnn`), and iterating for `num_epochs_cnn`. Metrics were stored in cnn_train_losses, etc. The notebook output shows this printout for each of the 25 epochs.

```
Starting CNN Training...
CNN Epoch [1/25], Train Loss: 1.6411, Train Acc: 39.81%, Val Loss: 1.4309, Val Acc: 47.50%
CNN Epoch [2/25], Train Loss: 1.3020, Train Acc: 53.16%, Val Loss: 1.2171, Val Acc: 57.08%
CNN Epoch [3/25], Train Loss: 1.1559, Train Acc: 58.72%, Val Loss: 1.1135, Val Acc: 59.82%
CNN Epoch [4/25], Train Loss: 1.0583, Train Acc: 62.62%, Val Loss: 1.0195, Val Acc: 63.49%
CNN Epoch [5/25], Train Loss: 0.9887, Train Acc: 65.16%, Val Loss: 0.9653, Val Acc: 65.80%
CNN Epoch [6/25], Train Loss: 0.9312, Train Acc: 67.09%, Val Loss: 0.9177, Val Acc: 68.00%
CNN Epoch [7/25], Train Loss: 0.8916, Train Acc: 68.59%, Val Loss: 0.8887, Val Acc: 68.91%
CNN Epoch [8/25], Train Loss: 0.8576, Train Acc: 69.63%, Val Loss: 0.8593, Val Acc: 69.42%
CNN Epoch [9/25], Train Loss: 0.8250, Train Acc: 71.01%, Val Loss: 0.8646, Val Acc: 70.03%
CNN Epoch [10/25], Train Loss: 0.8031, Train Acc: 71.79%, Val Loss: 0.8080, Val Acc: 71.99%
CNN Epoch [11/25], Train Loss: 0.7851, Train Acc: 72.45%, Val Loss: 0.7993, Val Acc: 71.92%
CNN Epoch [12/25], Train Loss: 0.7615, Train Acc: 73.10%, Val Loss: 0.8120, Val Acc: 72.20%
CNN Epoch [13/25], Train Loss: 0.7487, Train Acc: 73.84%, Val Loss: 0.8085, Val Acc: 71.72%
CNN Epoch [14/25], Train Loss: 0.7335, Train Acc: 74.22%, Val Loss: 0.7700, Val Acc: 73.38%
CNN Epoch [15/25], Train Loss: 0.7204, Train Acc: 74.81%, Val Loss: 0.7425, Val Acc: 74.20%
CNN Epoch [16/25], Train Loss: 0.7080, Train Acc: 75.00%, Val Loss: 0.8004, Val Acc: 71.58%
CNN Epoch [17/25], Train Loss: 0.6980, Train Acc: 75.50%, Val Loss: 0.7606, Val Acc: 73.85%
CNN Epoch [18/25], Train Loss: 0.6797, Train Acc: 76.21%, Val Loss: 0.7698, Val Acc: 73.50%
CNN Epoch [19/25], Train Loss: 0.6798, Train Acc: 76.13%, Val Loss: 0.7481, Val Acc: 74.24%
CNN Epoch [20/25], Train Loss: 0.6632, Train Acc: 76.63%, Val Loss: 0.7203, Val Acc: 74.88%
CNN Epoch [21/25], Train Loss: 0.6558, Train Acc: 76.89%, Val Loss: 0.7280, Val Acc: 74.62%
CNN Epoch [22/25], Train Loss: 0.6516, Train Acc: 77.06%, Val Loss: 0.7314, Val Acc: 74.59%
CNN Epoch [23/25], Train Loss: 0.6406, Train Acc: 77.42%, Val Loss: 0.7495, Val Acc: 74.63%
CNN Epoch [24/25], Train Loss: 0.6345, Train Acc: 77.54%, Val Loss: 0.7348, Val Acc: 74.69%
CNN Epoch [25/25], Train Loss: 0.6299, Train Acc: 77.81%, Val Loss: 0.7245, Val Acc: 75.31%
CNN Training Finished.
```

4.5 Model Evaluation Procedures

Test Function Definition (test_model)

```python
def test_model(model, test_loader, criterion, device):
    model.eval()
    test_loss = 0.0
    correct_test = 0
    total_test = 0
    all_predicted = []
    all_labels = []

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item() * images.size(0)

            _, predicted = torch.max(outputs.data, 1)
            total_test += labels.size(0)
            correct_test += (predicted == labels).sum().item()

            all_predicted.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    avg_test_loss = test_loss / len(test_loader.dataset)
    test_accuracy = 100 * correct_test / total_test

    print(f'Test Loss: {avg_test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%')
    return avg_test_loss, test_accuracy, all_labels, all_predicted
```

- A function `test_model` was created to evaluate a given model on a data loader (specifically, the `test_loader`).
- It set the model to `model.eval()` and used `with torch.no_grad()`: to disable gradient calculations.
- It iterated through the `test_loader`, performed forward passes, calculated loss, and aggregated total loss and correct predictions.
- A list of all true labels (`all_labels`) and predicted labels (`all_predicted`) from the test set were collected and returned for confusion matrix generation.
- The function returned the average test loss, overall test accuracy, and these label lists.

Model Testing

```python
# Test the MLP model
print("\nTesting MLP Model :")
mlp_test_loss, mlp_test_accuracy, mlp_true_labels, mlp_predicted_labels = test_model(mlp_model, test_loader, criterion, device)

# Test the CNN model
print("\nTesting CNN Model :")
cnn_test_loss, cnn_test_accuracy, cnn_true_labels, cnn_predicted_labels = test_model(cnn_model, test_loader, criterion, device)
```

- The test_model function was called for the trained mlp_model and cnn_model using test_loader, criterion, and device. The resulting test loss and accuracy for both models were printed.

```
Testing MLP Model :
Test Loss: 1.6311, Test Accuracy: 42.77%

Testing CNN Model :
Test Loss: 0.6882, Test Accuracy: 77.01%
```

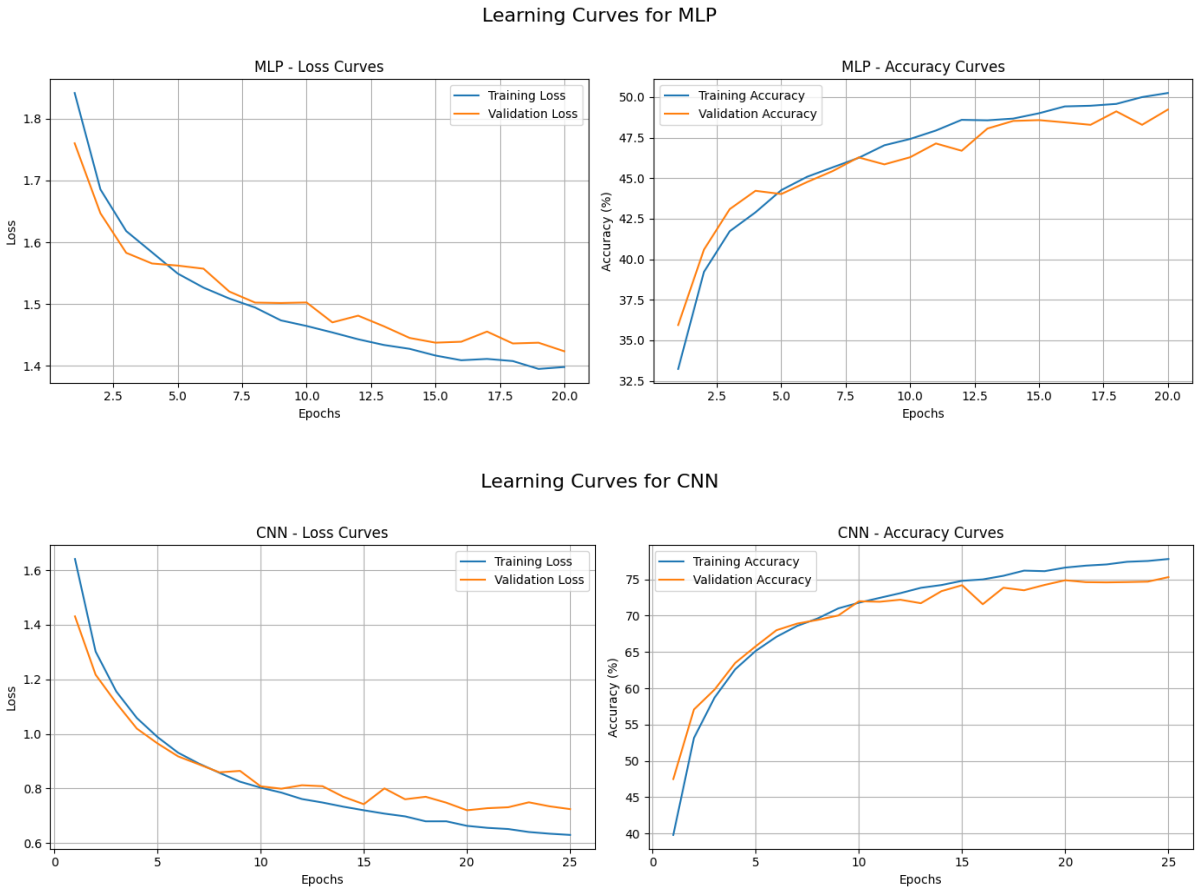
Learning Curve Plotting Function (plot_learning_curves)

```python
def plot_learning_curves(train_losses, val_losses, train_accuracies, val_accuracies, model_name, num_epochs):
    epochs_range = range(1, num_epochs + 1)

    plt.figure(figsize=(14, 5))

    # Plot Loss
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, train_losses, label='Training Loss')
    plt.plot(epochs_range, val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'{model_name} - Loss Curves')
    plt.legend()
    plt.grid(True)

    # Plot Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, train_accuracies, label='Training Accuracy')
    plt.plot(epochs_range, val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title(f'{model_name} - Accuracy Curves')
    plt.legend()
    plt.grid(True)

    plt.suptitle(f'Learning Curves for {model_name}', fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
```

- A function was defined to plot training vs. validation loss and training vs. validation accuracy over epochs using matplotlib.pyplot.
- It took lists of losses, accuracies, a model name string, and the number of epochs as input.

```
# Plot learning curves for MLP
plot_learning_curves(mlp_train_losses, mlp_val_losses,
                     mlp_train_accuracies, mlp_val_accuracies,
                     "MLP", num_epochs_mlp)

# Plot learning curves for CNN
plot_learning_curves(cnn_train_losses, cnn_val_losses,
                     cnn_train_accuracies, cnn_val_accuracies,
                     "CNN", num_epochs_cnn)
```

- This function was called for both the MLP and CNN using their respective stored metric lists.



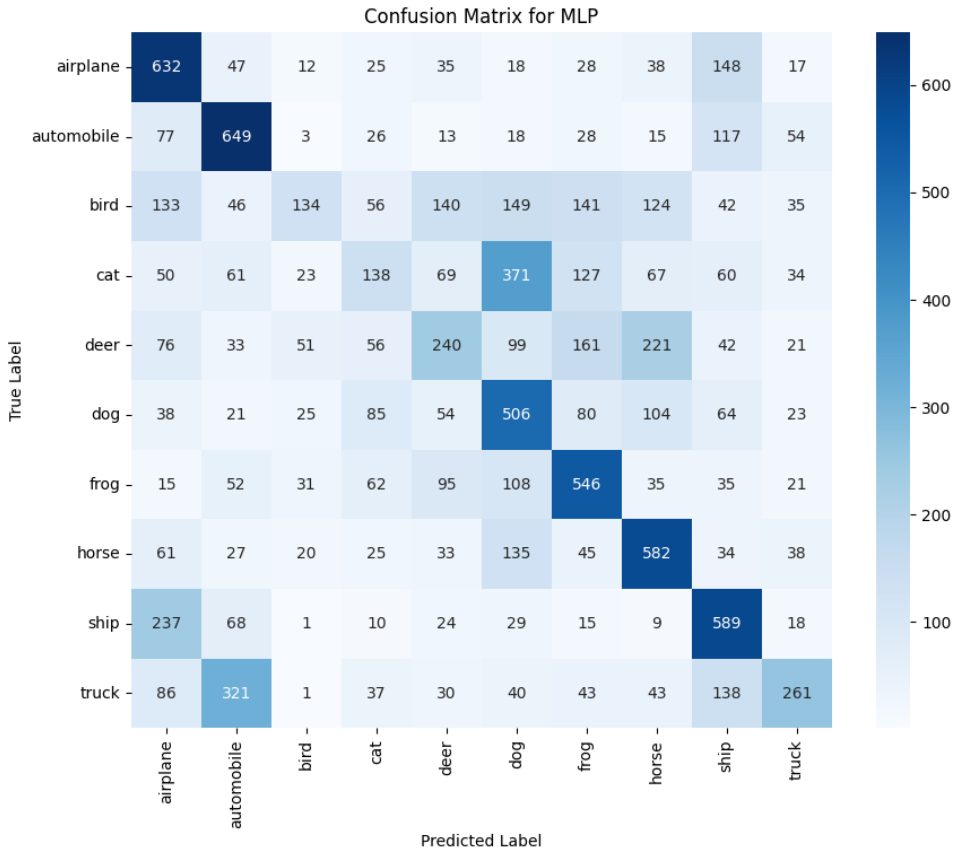
Confusion Matrix Plotting Function (plot_confusion_matrix)

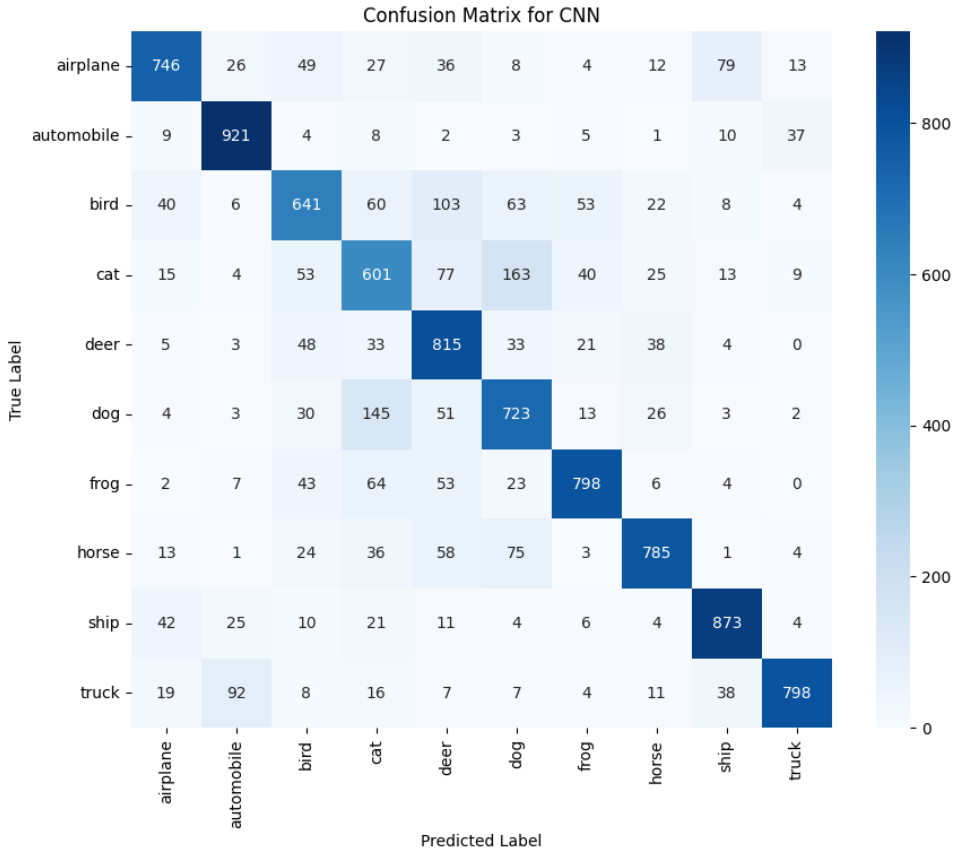
```python
def plot_confusion_matrix(true_labels, predicted_labels, class_names, model_name):
    cm = confusion_matrix(true_labels, predicted_labels)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix for {model_name}')
    plt.show()
```

- A function was defined to compute the confusion matrix using `sklearn.metrics.confusion_matrix` and visualize it as a heatmap using `seaborn.heatmap`.
- It took true labels, predicted labels, class names, and a model name as input.

```python
# Plot confusion matrix for MLP
plot_confusion_matrix(mlp_true_labels, mlp_predicted_labels, classes, "MLP")

# Plot confusion matrix for CNN
plot_confusion_matrix(cnn_true_labels, cnn_predicted_labels, classes, "CNN")
```

- This function was called for both MLP and CNN using the labels collected during the testing phase.





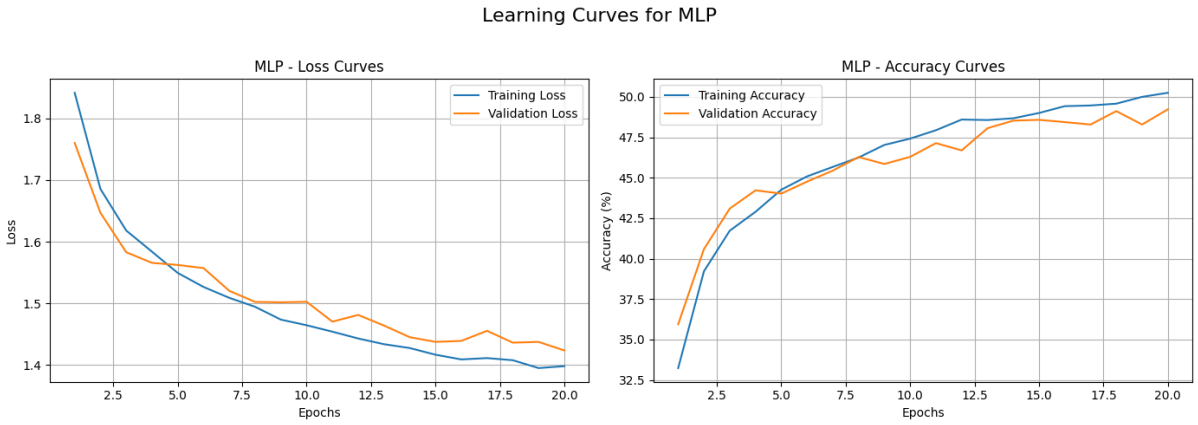
5 DISCUSSION

The empirical results obtained from **training** and **testing** the **Multi-Layer Perceptron (MLP)** and **Convolutional Neural Network (CNN)** on the CIFAR-10 dataset reveal significant differences in their performance and learning dynamics. This discussion will dissect these findings, referencing the learning curves, final test metrics, and the architectural distinctions between the two models.

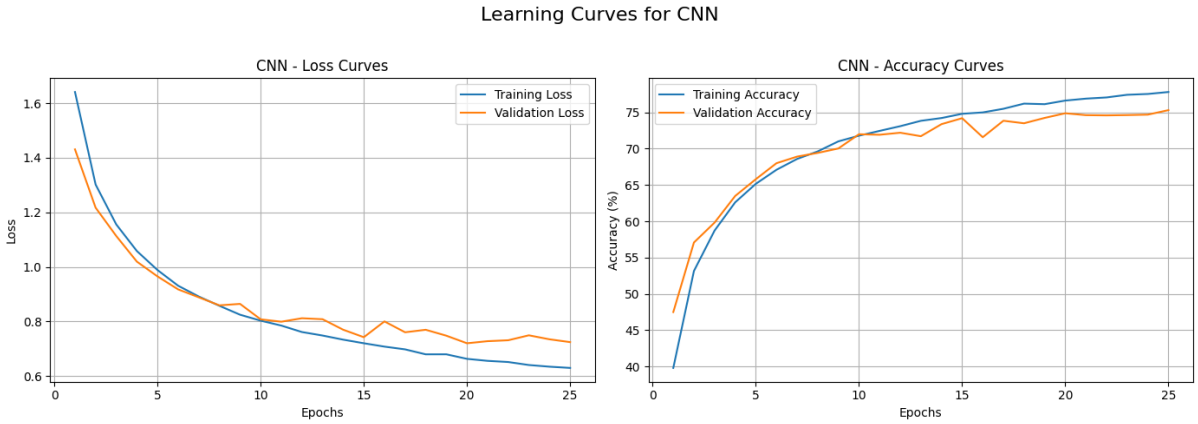
Comparative Performance: The most striking observation is the substantial performance disparity on the unseen test data. The CNN achieved a test accuracy of **77.01%** with a corresponding test loss of **0.6882**. In contrast, the MLP yielded a significantly lower test accuracy of **42.77%** and a higher test loss of **1.6311**. This near **34%** difference in accuracy underscores the CNN's superior capability in classifying the CIFAR-10 images.

Analysis of Learning Curves & Training Dynamics:

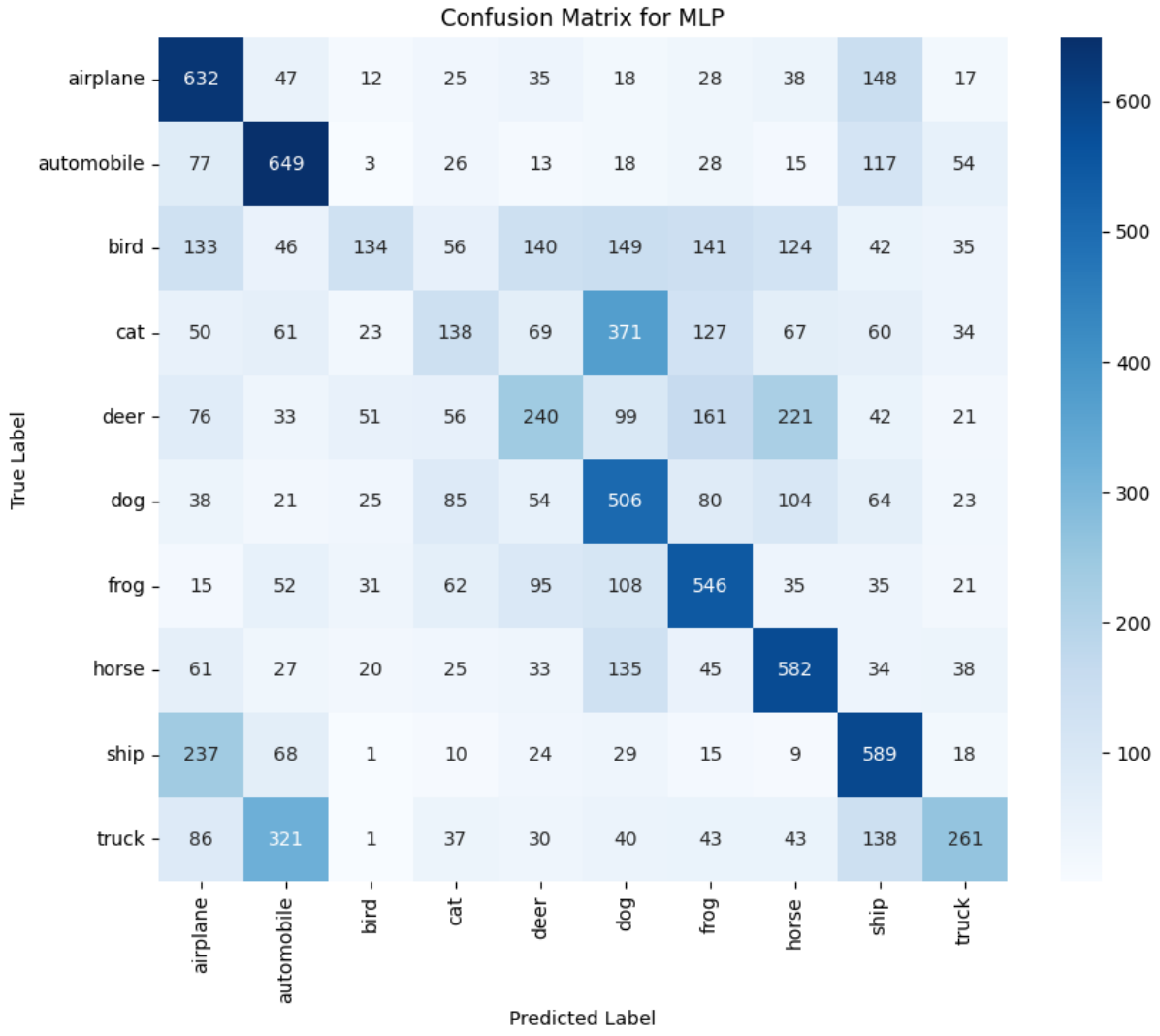
- **MLP Model**:
  - The MLP model, trained for 20 epochs, demonstrated a gradual improvement in both training and validation accuracy. Training accuracy commenced at 33.23% in epoch 1 and concluded at 50.25% in epoch 20. Validation accuracy started at 35.94% and ended at 49.23% in epoch 20.
  - The learning curves would show the training loss steadily decreasing from 1.8415 to 1.3981 and the validation loss also decreasing from 1.7602 to 1.4235 before plateauing.



- **CNN Model:**
  - The CNN model, trained for 25 epochs, exhibited a more robust learning trajectory. Training accuracy began at 39.81% and impressively rose to 77.81%. Similarly, validation accuracy climbed from 47.50% to a peak of 75.31%.
  - The learning curves would illustrate a more significant reduction in training loss (from 1.6411 to 0.6299) and validation loss (from 1.4309 to 0.7245).



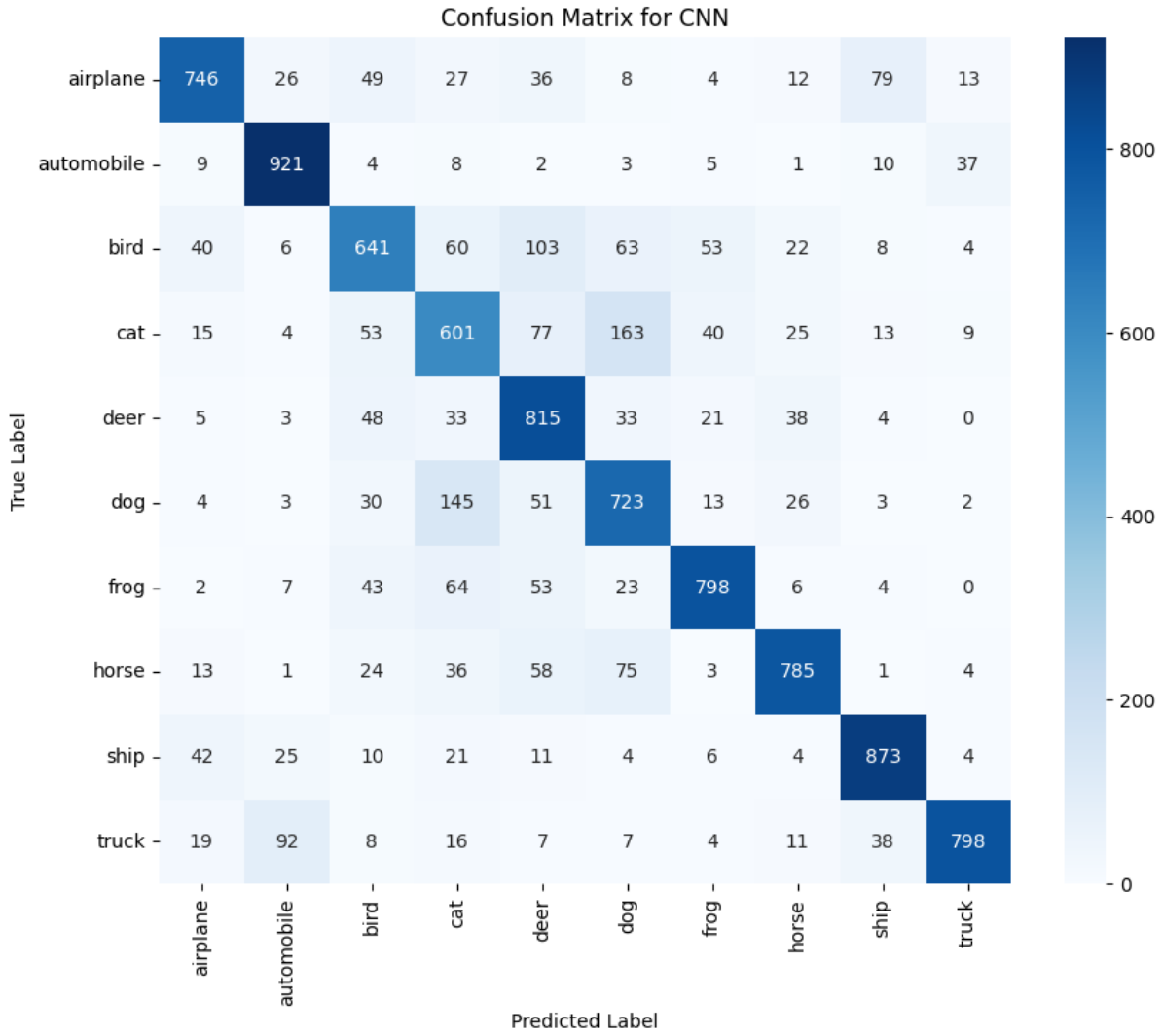
Error Analysis:



- Looking at the confusion matrix of the MLP, we see:
  - **Poor performance:** The MLP misclassifies many samples, especially for classes with complex images.
  - **Typical error patterns:**
    - **Confusion between visually similar classes:** The MLP frequently confuses objects with similar general shapes (e.g., "cat" with "dog," "car" with "truck").
    - **Influence of context/common features:** The model is easily "fooled" by similar colors or background elements rather than the core features of the object (e.g., a "bird" being mistaken for other objects that might appear in a sky background).
    - **Inability to capture fine-grained details:** The MLP struggles to distinguish small but important features that help accurately identify classes.
  - The above errors of the MLP are direct symptoms of this architecture ignoring spatial structure and local relationships in image data:
    - **Loss of spatial information:** By flattening the image into a vector, the MLP loses information about which pixels are adjacent to each other, forming edges, corners, or textures. This information is extremely important for object recognition.

* **Lack of hierarchical feature learning capability:** The MLP attempts to learn direct relationships from all pixels to the output classes. It lacks a mechanism to learn simple features (like edges) and then combine them into more complex features (like parts of an object), a process inherent in how the human visual system and CNN models operate.
* **No translation invariance:** If an object appears in a different location in the image, the MLP has to relearn that feature almost from scratch because the input pixels will be different.

⇒ **In summary** the error analysis of the MLP on CIFAR-10 clearly shows that the lack of spatial receptive mechanisms and hierarchical feature learning makes it unsuitable for complex image classification tasks. This model primarily relies on coarse statistical correlations in the flattened pixel data, leading to high confusion between classes with visually or contextually similar characteristics at a general level, and failing to capture the fine-grained details necessary for accurate classification.



- Looking at the confusion matrix of the CNN, we see:
  - **Superior performance (compared to MLP):** The CNN demonstrates significantly more accurate classification, considerably reducing the number of misclassified samples, even for classes with complex images. However, some errors still persist.

- **Remaining typical error patterns (albeit minimized):**
  - ∗ **Confusion between classes with high visual similarity:** The CNN still encounters some difficulty with certain pairs of classes that have very similar appearances, although the level of confusion is much reduced. The most notable example is between "cat" and "dog."
  - ∗ **Influence (to a lesser extent) from very fine-grained features or diverse instances:** Although the CNN is less "fooled" by general context than the MLP, some image instances with high intra-class variation or extremely subtle distinguishing features can still lead to misclassification.
  - ∗ For example, the "bird" class still shows some confusion with other animals, possibly due to the diversity of species, poses, and backgrounds.
  - ∗ **Challenges with extremely fine-grained details for difficult distinctions:** For cases where the difference between classes lies in very small details or complex relationships, this basic CNN architecture might not fully capture them, leading to some residual errors.
- The remaining errors of the CNN, though far fewer than the MLP's, indicate the limitations of a relatively basic CNN architecture when faced with the complexity and diversity of CIFAR-10. However, the significant improvement of the CNN over the MLP is due to its architecture addressing the weaknesses of the MLP:
  - ∗ **Preservation and utilization of spatial information:** Unlike the MLP which flattens the image, the CNN's convolutional layers process neighboring pixel regions, preserving spatial information. This allows the CNN to learn local features like edges, corners, and textures, which are crucial for object recognition.
  - ∗ **Hierarchical feature learning capability:** The CNN builds increasingly complex features through multiple layers. Early layers learn simple features, and later layers combine them to identify parts of objects and eventually entire objects. This is a capability that the MLP lacks.
  - ∗ **Translation invariance (or local invariance):** Due to the weight-sharing mechanism of convolutional filters and pooling layers, the CNN can recognize a feature even if it appears in slightly different locations in the image, helping the model generalize better than the MLP.

⇒ **In summary**, the error analysis of the CNN on CIFAR-10 shows that, thanks to its spatial receptive mechanisms, hierarchical feature learning capability, and a degree of translation invariance, the CNN is far more suitable for complex image classification tasks than the MLP. The remaining errors of the CNN are mainly concentrated in classes with very high visual similarity or those requiring the distinction of extremely fine-grained details, suggesting that while performance is good, there is still room for further improvement with deeper CNN architectures or more advanced training techniques.

Architectural Impact on Performance: The observed performance differences are fundamentally rooted in the architectural designs of the MLP and CNN and their suitability for image data.

- **MLP Limitations:** The MLP processes images by first flattening the 3D pixel array (3x32x32) into a 1D vector of 3072 features. This operation inherently discards the crucial spatial relationships and local correlations between pixels. Consequently, the MLP cannot easily learn local patterns (like edges, corners, textures) or understand the hierarchical composition of features that define objects in an image. While its fully connected layers provide high model capacity, their lack of spatial inductive bias makes them inefficient and less effective for image tasks, often leading

to a higher number of parameters for comparable receptive fields and increased susceptibility to overfitting if not well-regularized. In this case, its performance suggests it struggled to build meaningful feature representations.

- **CNN Strengths:** CNNs are specifically designed to process grid-like data like images.
  - **Convolutional Layers:** These layers apply learnable filters (e.g., 3x3 kernels as used in this CNN) across the input image, detecting local patterns and creating feature maps that preserve spatial information. The progressive stacking of these layers (three in this implementation) allows the network to learn a hierarchy of features, from simple edges in early layers to more complex object parts in deeper layers.
  - **Parameter Sharing:** Filters are shared across different spatial locations within a feature map, significantly reducing the total number of parameters compared to an MLP and making the learning process more efficient and robust to translations of features.
  - **Pooling Layers:** Max pooling layers (2x2 with stride 2, used after each convolutional block in this CNN) reduce the dimensionality of feature maps, providing a degree of translation invariance and making the learned features more compact and robust.
  - **Data Augmentation:** The use of RandomCrop and RandomHorizontalFlip during the training of the CNN artificially expanded the training dataset, exposing the model to more variations and improving its ability to generalize to unseen images, likely contributing to the smaller gap between training and validation accuracy.

The CNN's architecture is thus inherently better equipped to learn the rich, spatially organized features within the CIFAR-10 images, leading to its significantly higher classification accuracy.

6 CONCLUSION

This project successfully implemented and conducted a comparative evaluation of a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN) for the task of image classification on the CIFAR-10 dataset. The empirical results unequivocally demonstrate the superior efficacy of the CNN architecture in this domain.

The CNN model achieved a notable test accuracy of **77.01%** with a corresponding test loss of **0.6882**. In stark contrast, the MLP model's performance was considerably lower, yielding a test accuracy of **42.77%** and a test loss of **1.6311**. This significant disparity underscores the architectural advantages of CNNs for image-based tasks. The CNN's proficiency stems from its inherent ability to learn and leverage spatial hierarchies of features through its convolutional and pooling layers, which are specifically designed to process grid-like data. Furthermore, the application of data augmentation techniques during training likely contributed to the CNN's enhanced generalization capabilities.

Conversely, the MLP, which processes images as flattened vectors, inherently discards crucial spatial relationships between pixels. This structural limitation impedes its ability to efficiently learn the complex visual patterns necessary for accurate classification in datasets like CIFAR-10, leading to its comparatively modest performance.

In conclusion, this study reaffirms that Convolutional Neural Networks are exceptionally well-suited for computer vision tasks due to their specialized architecture. The choice of model architecture, aligned with the inherent characteristics of the data, is paramount for achieving robust and accurate results in deep learning applications.

I would like to thank my teacher for their guidance throughout this assignment, as well as my friends and classmates for their support and valuable discussions. Their encouragement helped me stay motivated and complete this project successfully. Thank you very much!