POSTS AND TELECOMMUNICATIONS
INSTITUTE OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY I

DEPARTMENT OF PYTHON PROGRAMMING

PYTHON PROGRAMMING REPORT 2

Lecturers of the Department : KIM NGOC BÁCH
Class : D23CQCE04 - B
Full name : TRAN TRUNG HIEU - B23DCCN312
: NGUYEN HUU NIEM - B23DCCE076

Ha Noi – 2025

PREFACE

In today's digital age, computer vision has become an important field, contributing to the development of many practical applications from medicine, manufacturing to self-driving cars and security. At the heart of computer vision is the ability for computers to "understand" and interpret information from digital images or videos. One of the most basic and fundamental problems in this field is image classification. Image classification requires computers to assign a specific label or category to an input image, simulating human recognition capabilities.

With the explosion of Deep Learning, especially the emergence and development of Convolutional Neural Networks (CNNs), the performance of image classification systems has been raised to a new level, far surpassing traditional methods. To understand and compare the effectiveness of neural network architectures, this exercise focuses on implementing and evaluating two main types of models: Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN), on the CIFAR-10 benchmark dataset.

This paper details the data preparation process, architecture design and implementation of MLP and CNN, training and testing process, and analysis and comparison of the results. Through this, the paper aims to clarify the performance differences and outstanding advantages of CNN compared to MLP in image classification. The entire implementation process is based on the PyTorch library, a powerful and flexible tool in deep learning research and development.

We hope that this report will provide a comprehensive and in-depth look at the application of MLPs and CNNs in image classification, while also affirming the importance of choosing the right model architecture given the nature of the data and the problem.

Best regards!

Contents

1 INTRODUCTION

1.1 Overview of the image classification problem

Image classification is one of the most fundamental and important tasks in the field of computer vision. The main goal of image classification is to assign a specific label to an image based on its content. This process typically involves extracting features from the image and using machine learning or deep learning algorithms to learn patterns related to each object class. Image classification applications are diverse and have a significant impact on many industries, including:

- Object recognition: Used in self-driving cars to detect pedestrians, traffic, and road signs.
- Medical image analysis: Supports disease diagnosis from X-ray and MRI images.
- Image search: Finds similar images based on content.
- Quality control: In production to detect product defects.

The development of Deep Learning, especially Convolutional Neural Networks (CNNs), has made significant breakthroughs in the performance of image classification systems.

1.2 Introduction to CIFAR-10 dataset

This exercise uses the CIFAR-10 dataset, a popular and widely used dataset in research and development of computer vision algorithms. CIFAR-10 has the following salient features:

- Size: Includes 60,000 color images (RGB) with a compact size of 32x32 pixels.
- Classify: Divided into 10 different object classes, each with 6,000 images.
- Classes: The 10 classes include: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', and 'truck'.
- Split episode: The dataset is pre-split into 50,000 training images and 10,000 test images.

1.3 Objectives and scope of the report

The main objective of this paper is to perform image classification task on CIFAR-10 dataset by building, training and evaluating two types of neural network architectures:

- Multilayer Feedforward Neural Network (MLP): A basic network with 3 hidden layers.
- Convolutional Neural Network (CNN): An image-specific network with 3 convolutional layers.

The scope of the report will include steps from data preparation, model architecture design, training process, testing and performance evaluation of each model. Finally, we will conduct a detailed comparison and discussion of the results achieved between MLP and CNN to clarify the advantages and disadvantages of each architecture in this problem.

1.4 Tools and libraries used

The entire implementation and analysis process is done in Python programming language. The main libraries used include:

- PyTorch: The main framework for building and training neural network models.
- torchvision: PyTorch's data and computer vision model support library, used to load the CIFAR-10 dataset and apply image transformations.
- torch.nn: The module contains the classes for building neural networks.
- torch.optim: Module containing optimizer algorithms.
- matplotlib.pyplot: To draw graphs (learning curves).
- numpy: Support array calculations.
- sklearn.metrics: To calculate and display confusion matrix and classification report.
- seaborn: To visualize the confusion matrix more beautifully.

2 DATA PREPARATION

Data preparation and preprocessing is a crucial step to ensure effective model learning, enhance generalization ability, and minimize training issues.

2.1 Load and transform data

The CIFAR-10 dataset is downloaded and the necessary transformations are applied to both the training and testing/validation sets.

2.1.1 Data Augmentation

To improve the generalizability of the model and reduce overfitting, Data Augmentation techniques are applied to the training set. These transformations help the model get used to many variations of the same object:

- `transforms.RandomCrop(32, padding=1)`: Randomly crops a 32x32 pixel portion of the image after adding 1 pixel of padding around the image. This helps the model not to be too dependent on the exact location of the object in the image and get used to different parts of the object.
- `transforms.RandomHorizontalFlip()`: Randomly flips the image horizontally. This is a simple but effective transformation that helps the model learn that the object can appear in any horizontal direction.
- `transforms.ToTensor()`: Converts an image from a PIL Image or NumPy array format to a PyTorch Tensor. Tensor is the data format that PyTorch uses to perform calculations on the GPU or CPU.

- `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Normalizes the pixel values of each color channel (RGB) to the range [-1, 1]. This normalization helps the input values of the network have the same scale, thereby speeding up the convergence of the training process and improving the performance of the model.

2.1.2 Data transformation for testing/validation set

For the test and validation sets, the goal is to accurately evaluate the real-world performance of the model on unseen data. Therefore, we only apply the necessary transformations to format and normalize the data, without using data augmentation techniques:

- `transforms.ToTensor()`: Converts an image to a PyTorch Tensor.
- `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Normalizes pixel values to the range [-1, 1].

2.2 Data set division

The downloaded CIFAR-10 dataset has been divided into 50,000 training images and 10,000 testing images. To ensure that the model validation process is objective and not influenced by the training data, we further divide the original training set into a training set and a validation set:

- Training set: Downloaded and applied `transform_train`.
- Test set: Downloaded and applied `transform_test`.
- Validation split: 10% of the data from the `trainset` is used as the validation set (`val_dataset`), and the remaining 90% is the actual training set (`train_dataset`).
- Training set size: 45,000 images.
- Audit set size: 5,000 images.
- Test set size: 10,000 images.

2.3 DataLoader Configuration

To manage and load data in batches during training and testing, we use PyTorch's DataLoader:

- Batch Size: `batch_size` is set to 128. Batch size affects training speed (larger batches are usually faster) and gradient stability (smaller batches usually have noisier gradients).
- Data Shuffle: `shuffle=True` is set to the `trainloader` to ensure each epoch of data is randomly shuffled, which helps the model not learn the order of the data and improves generalization ability.
- Multithreading: `num_workers=2` is used to load data in parallel, which helps speed up data loading and avoids CPU/GPU waiting due to slow data loading.

- `trainloader`: DataLoader for the training set.
- `valloader`: DataLoader for the validation set.
- `testloader`: DataLoader for the test set.

2.4 Object classes in CIFAR-10

CIFAR-10 consists of 10 different object classes. Knowing these classes helps us better understand the problem and analyze the results later, especially in the confusion matrix. These classes are defined as a tuple in the source code:
Classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

3 MODEL ARCHITECTURE

In this section, we will detail the architecture of the two neural network models built: MLP and CNN. Both models are implemented using PyTorch.

3.1 Multi-Layer Perceptron (MLP)

3.1.1 Overview of MLP

MLP, also known as feedforward neural network, is one of the most basic neural network architectures. MLP consists of an input layer, one or more hidden layers, and an output layer. The neurons between the layers are fully connected, and information flows only in one direction from the input layer to the output layer. MLP is suitable for problems where the input data can be represented as a flat vector. However, when processing images, MLP is often not as effective as CNN because it does not take advantage of the spatial structure of pixels.

3.1.2 MLP architecture design

The MLP model is designed to process CIFAR-10 images. The input is a 32x32 pixel (3 channel) color image, which will be flattened into a 1D vector.

- **Flatten class (`nn.Flatten()`):** The first layer in the model is responsible for converting the input tensor from (batch_size, channels, height, width) to (batch_size, channels * height * width). For the CIFAR-10 image, the input size will be $3 \times 32 \times 32 = 3072$ features.
- **Fully Connected (FC) or Linear (`nn.Linear`) classes:** The model uses 3 FC layers:
  - `self.fc1 = nn.Linear(3 * 32 * 32, 512)`: The first hidden layer, receives 3072 features from the Flatten layer and transforms them into 512 features.
  - `self.fc2 = nn.Linear(512, 256)`: Second hidden layer, takes 512 features from the previous layer and transforms them into 256 features.
  - `self.fc3 = nn.Linear(256, num_classes)`: Output layer, takes 256 features and produces `num_classes` (10) outputs, corresponding to the scores (logits) for each class.

- **ReLU activation function (`nn.ReLU()`):** Applied after the `fc1` and `fc2` layers. The ReLU (Rectified Linear Unit) function introduces nonlinearity into the model, helping the network learn more complex relationships in the data. Without the nonlinear function, MLPs can only learn linear relationships.
- **Dropout (`nn.Dropout(p=0.3)`):** Two Dropout layers are added after each ReLU activation function (`self.relu1` and `self.relu2`). Dropout is an effective regularization technique to combat overfitting. During training, a random percentage (p=0.3, or 30%) of neurons are "disconnected" (temporarily not involved in the computation) in each data pass. This forces the network to learn more robust representations that are less dependent on any particular neuron.

3.1.3 Forward Pass Function of MLP

The forward function defines the flow of data through the network, from input to output:

```
def forward(self, x):
    x = self.flatten(x)# Flatten image
    x = self.fc1(x)    # Linear Layer 1
    x = self.relu1(x)  # ReLU 1 activation function
    x = self.dropout1(x) # Apply Dropout

    x = self.fc2(x)     # Linear Layer 2
    x = self.relu2(x)   # ReLU 2 activation function
    x = self.dropout2(x) # Apply Dropout

    x = self.fc3(x)     # Output layer

    return x
```

Figure 1: MLP Forward Pass Function

3.2 Convolutional Neural Network (CNN)

3.2.1 Overview of CNN

CNNs are a type of neural network that is particularly effective at processing grid-structured data such as images. Unlike MLPs, CNNs take advantage of the spatial structure of images through convolutional layers. These layers are capable of automatically learning filters to extract important features from the data, from low-level features (edges, corners) to high-level features (parts of objects).

3.2.2 CNN architecture design

The CNN model is designed with 3 consecutive convolutional blocks, followed by a Fully Connected layer.

- **Convolutional and Pooling layers (`self.conv_layers`):**
  - **Convolution block 1:**
    - `nn.Conv2d(3, 32, kernel_size=3, padding=1)`: Receive RGB image (3 channels), apply 32 3x3 filters. `padding=1` helps keep the spatial size after convolution. Output: 32 channels, spatial size 32x32.

    - `nn.ReLU()`: Nonlinear activation function.
    - `nn.MaxPool2d(kernel_size=2, stride=2)`: Perform Max Pooling with a 2x2 window and stride 2, reducing the spatial size by half. Output: 32 channels, 16x16.
  - **Convolution block 2:**
    - `nn.Conv2d(32, 64, kernel_size=3, padding=1)`: Receive 32 channels from previous layer, apply 64 3x3 filters. Output: 64 channels, 16x16.
    - `nn.ReLU()`: Nonlinear activation function.
    - `nn.MaxPool2d(kernel_size=2, stride=2)`: Reduce the space size. Output: 64 channels, 8x8.
  - **Convolution block 3:**
    - `nn.Conv2d(64, 128, kernel_size=3, padding=1)`: Receive 64 channels from previous layer, apply 128 3x3 filters. Output: 128 channels, 8x8.
    - `nn.ReLU()`: Nonlinear activation function.
    - `nn.MaxPool2d(kernel_size=2, stride=2)`: Reduce the final space size. Output: 128 channels, 4x4.

- **Calculate the input size for the FC layer:** After 3 layers of Max Pooling, the spatial size of the image has been reduced from 32x32 to 4x4. So the input size for the final Fully Connected layer will be $128 \times 4 \times 4 = 2048$ features. `self.fc_input_dim = 128 * 4 * 4`.
- **Dropout (`self.dropout_fc = nn.Dropout(p=0.3)`):** A Dropout layer with a scale of 0.3 is added before the final Fully Connected layer. This helps prevent overfitting by randomly ignoring some of the features learned from the convolutional layers, forcing the model to rely on many different features to make predictions.
- **Final Fully Connected (FC) class (`self.fc`):**
  - `self.fc = nn.Linear(self.fc_input_dim, num_classes)`: This layer takes a flattened vector (2048 features) and produces 10 outputs (logits), corresponding to the 10 object classes of CIFAR-10.

3.2.3 Forward Pass Function of CNN

The forward function defines the data flow through the CNN network:

```
def forward(self, x):
    x = self.conv_layers(x) # Pass through convolutional and pooling layers
    x = x.view(-1, self.fc_input_dim) # Flatten
    x = self.dropout_fc(x) # Apply Dropout before FC layer
    x = self.fc(x) # Pass through Fully Connected layer
    return x
```

Figure 2: CNN Forward Pass Function

4 TRAINING AND TESTING PROCESS

To ensure fairness and objectivity in evaluating the performance of the two models, the training and testing processes are set up according to a common configuration and use pre-defined functions.

4.1 General configuration for training

- **Device:** The model is trained on GPU (cuda) if available to speed up the computation, otherwise it automatically switches to CPU. During code execution, the device used is cuda.
- **Number of Epochs (`num_epochs`):** Both models will be trained for 20 epochs. This number of epochs is an important hyperparameter that affects the training time and convergence ability of the model.
- **Learning rate (`learning_rate`):** The learning rate is set to 0.001. This is a common value that controls how quickly the model updates its weights during optimization.
- **Loss function (Criterion):** `nn.CrossEntropyLoss()` is used as the loss function. This is the standard choice for multiclass classification problems in PyTorch, combining the softmax function and negative log-likelihood loss.
- **Optimizer:** `optim.Adam()` is used to update the model weights. Adam is a popular adaptive optimization algorithm that typically provides good performance and converges quickly.

4.2 Training function (`train_model`)

The `train_model` function is designed to manage the entire process of training and validating a model over epochs:

- **Transfer model to device:** `model.to(device)` ensures that the model and data are processed on the same device (GPU or CPU).
- **Historical Archives:** The lists `train_losses`, `val_losses`, `train_accuracies`, and `val_accuracies` are initialized to record the loss and accuracy of both the training and validation sets over each epoch. This is necessary to plot the learning curve.
- **Epoch Loop:**
  - **Training mode:** `model.train()` is called at the beginning of each epoch to put the model into training mode. In this mode, the Dropout layers will run and the model parameters will be updated.
  - **Batch Training:** For each batch of data from `train_loader`:
    - Data (`images`, `labels`) are transferred to the device.
    - Perform a straight pass (`model(images)`) to get the model output.
    - Loss calculation (`criterion(outputs, labels)`).
    - Delete the old gradient (`optimizer.zero_grad()`).
    - Backpropagate (`loss.backward()`) to calculate the gradient.
    - Update weights (`optimizer.step()`).
    - Calculate the running loss and training set accuracy for the current epoch.

- **Evaluation mode:** `model.eval()` is called to switch the model to evaluation mode. In this mode, Dropout layers will be disabled and parameters will not be updated.
- **Rating on the review set:** For each batch of data from `val_loader`, and use `with torch.no_grad():` to not calculate the gradient:
  - Data is transferred to the device.
  - Perform straight transmission.
  - Compute loss and precision for the validation set of the current epoch.
- **Print Epoch results:** Print out the loss and accuracy on both the training and validation sets of the current epoch to track progress.

- **Complete training:** After all epochs are completed, the message "Training complete." is printed.
- The function returns the collected loss and accuracy lists.

4.3 Test function (`test_model`)

The `test_model` function is used to evaluate the final performance of the model on the test set:

- **Transfer model to device and evaluation mode:** Similar to the validation process, the model is uploaded to the device and put into `model.eval()` mode with `torch.no_grad()`.
- **Collect predictions:** The function iterates through the `test_loader` to collect all actual labels (`all_labels`) and predicted labels (`all_predicted`) from the model. This is needed to generate the confusion matrix and classification report.
- **Calculate accuracy:** The overall accuracy of the model on the test set is calculated and printed to the console.
- **Return result:** The function returns `all_labels`, `all_predicted` and the final accuracy on the test set.

4.4 Evaluation support functions (learning graph, confusion matrix)

- **Hàm `plot_learning_curves`:**
  - Get the lists of `train_losses`, `val_losses`, `train_accuracies`, `val_accuracies` and `model_name`.
  - Plot two subplots: one plot showing the training loss and validation loss over epochs, and one plot showing the training accuracy and validation accuracy over epochs.
  - These graphs are useful tools for analyzing the learning process of the model, helping to detect problems such as overfitting or underfitting.
- **`plot_confusion_matrix` function:**

- Get the actual labels (`all_labels`), predicted labels (`all_predicted`), class names (`classes`) and model name (`model_name`).
- Use `sklearn.metrics.confusion_matrix` to calculate the confusion matrix. The confusion matrix is a table that summarizes the performance of a classification algorithm, showing the number of correct and incorrect predictions for each class.
- Use `seaborn.heatmap` to visualize the confusion matrix, making it easy to see which classes are most confused with other classes.
- The values on the main diagonal represent the number of correct predictions.

5 RESULTS AND ANALYSIS

This section presents the training and testing results of both MLP and CNN models, along with detailed analysis.

5.1 MLP model results

5.1.1 Training and testing process

The MLP model is trained for 20 epochs with the configuration described in Section 4.1. Below is the summary training log of each epoch and print test accuracy:

```
--- Training MLP Model ---
Starting training for MLP model for 20 epochs...
Epoch [1/20], Train Loss: 1.7715, Train Acc: 36.62%, Val Loss: 1.6343, Val Acc: 41.54%
Epoch [2/20], Train Loss: 1.6234, Train Acc: 42.57%, Val Loss: 1.5313, Val Acc: 45.84%
Epoch [3/20], Train Loss: 1.5696, Train Acc: 44.48%, Val Loss: 1.5116, Val Acc: 46.44%
Epoch [4/20], Train Loss: 1.5296, Train Acc: 45.88%, Val Loss: 1.4769, Val Acc: 48.24%
Epoch [5/20], Train Loss: 1.5020, Train Acc: 46.76%, Val Loss: 1.4363, Val Acc: 49.30%
Epoch [6/20], Train Loss: 1.4802, Train Acc: 47.74%, Val Loss: 1.4242, Val Acc: 49.78%
Epoch [7/20], Train Loss: 1.4638, Train Acc: 48.06%, Val Loss: 1.4147, Val Acc: 50.06%
Epoch [8/20], Train Loss: 1.4441, Train Acc: 48.78%, Val Loss: 1.4039, Val Acc: 50.70%
Epoch [9/20], Train Loss: 1.4337, Train Acc: 49.04%, Val Loss: 1.3898, Val Acc: 50.66%
Epoch [10/20], Train Loss: 1.4142, Train Acc: 49.91%, Val Loss: 1.3807, Val Acc: 50.86%
Epoch [11/20], Train Loss: 1.4097, Train Acc: 49.86%, Val Loss: 1.3863, Val Acc: 49.78%
Epoch [12/20], Train Loss: 1.3983, Train Acc: 49.91%, Val Loss: 1.3470, Val Acc: 52.28%
Epoch [13/20], Train Loss: 1.3865, Train Acc: 50.75%, Val Loss: 1.3413, Val Acc: 52.72%
Epoch [14/20], Train Loss: 1.3781, Train Acc: 50.86%, Val Loss: 1.3642, Val Acc: 51.10%
Epoch [15/20], Train Loss: 1.3763, Train Acc: 51.06%, Val Loss: 1.3380, Val Acc: 52.74%
Epoch [16/20], Train Loss: 1.3634, Train Acc: 51.22%, Val Loss: 1.3372, Val Acc: 51.96%
Epoch [17/20], Train Loss: 1.3594, Train Acc: 51.84%, Val Loss: 1.3218, Val Acc: 53.36%
Epoch [18/20], Train Loss: 1.3523, Train Acc: 52.07%, Val Loss: 1.3308, Val Acc: 52.48%
Epoch [19/20], Train Loss: 1.3365, Train Acc: 52.42%, Val Loss: 1.3228, Val Acc: 53.30%
Epoch [20/20], Train Loss: 1.3425, Train Acc: 52.11%, Val Loss: 1.2947, Val Acc: 54.72%
Training complete.
MLP model saved to ./mlp_cifar10.pth
Starting testing for MLP model...
Test accuracy for MLP model: 52.99%
```

Figure 3: MLP Training Log

5.1.2 Learning Curves

The learning curve graph for MLP shows how the loss and accuracy change throughout training: **Loss Chart:**

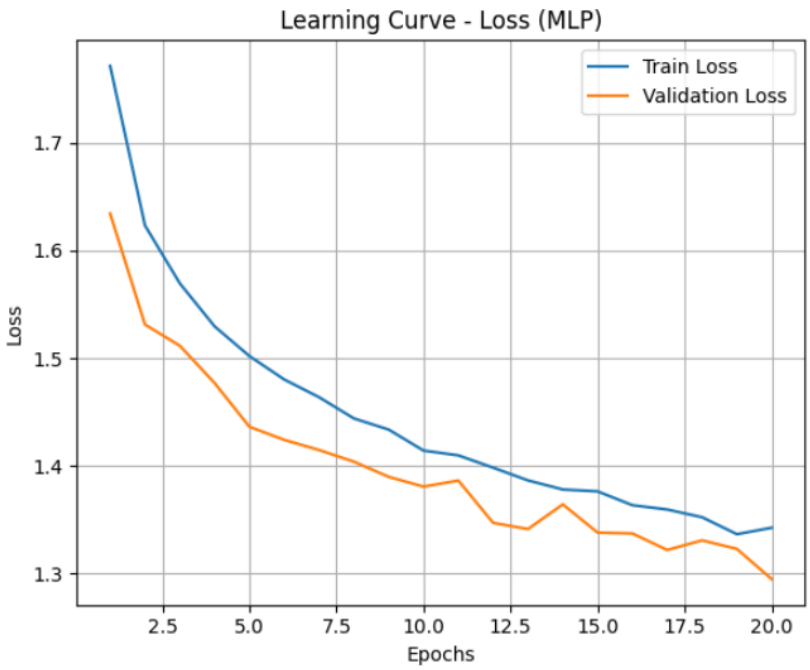

Figure 4: Learning Curve - Loss (MLP)

Comment: The Training Loss decreases over epochs, indicating that the model is learning from the data. However, the Validation Loss tends to decrease slowly and may start to increase slightly in the last epochs, or remain at a large distance from the Training Loss, suggesting overfitting.

Accuracy Chart:

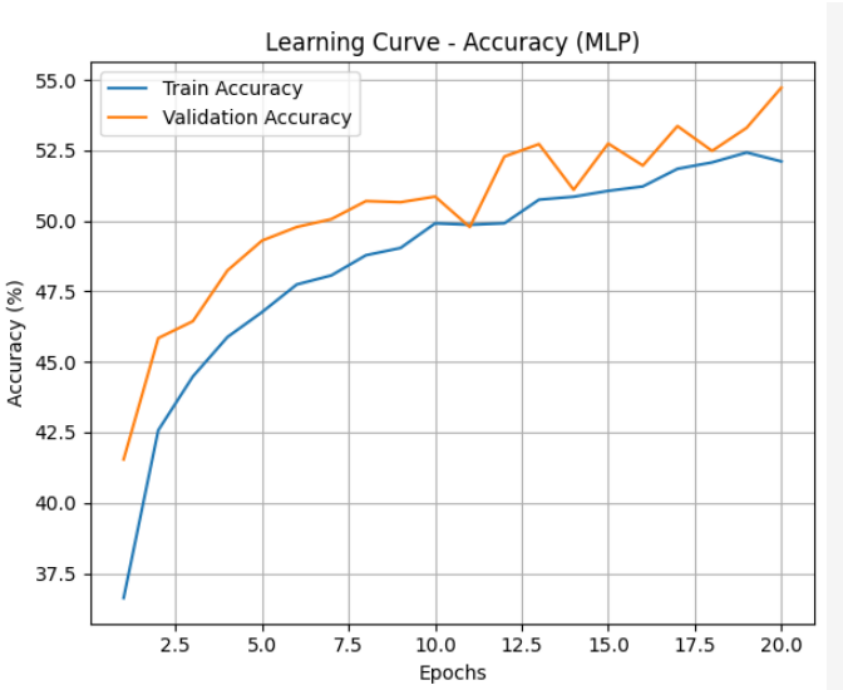

Figure 5: Learning Curve - Accuracy (MLP)

Comment: The accuracy on the training set (Train Acc) continues to increase and

reaches a fairly high level, while the accuracy on the validation set (Validation Acc) increases more slowly and tends to saturate after the first few epochs. This is also a sign of overfitting, when the model learns too specific examples of the training set.

5.1.3 Classification Report

The classification report provides detailed metrics (Precision, Recall, F1-score) for each class and overall on the test set:

```
--- MLP Classification Report ---
              precision    recall  f1-score   support

       plane       0.61      0.55      0.58      1000
         car       0.63      0.68      0.65      1000
        bird       0.46      0.31      0.37      1000
         cat       0.34      0.46      0.39      1000
        deer       0.46      0.45      0.45      1000
         dog       0.45      0.38      0.41      1000
        frog       0.56      0.63      0.59      1000
       horse       0.69      0.53      0.60      1000
        ship       0.59      0.70      0.64      1000
       truck       0.57      0.60      0.59      1000

    accuracy                           0.53     10000
   macro avg       0.54      0.53      0.53     10000
weighted avg       0.54      0.53      0.53     10000
```

Figure 6: MLP Classification Report

5.1.4 Confusion Matrix

The confusion matrix visualizes the number of correct and incorrect predictions for each class:

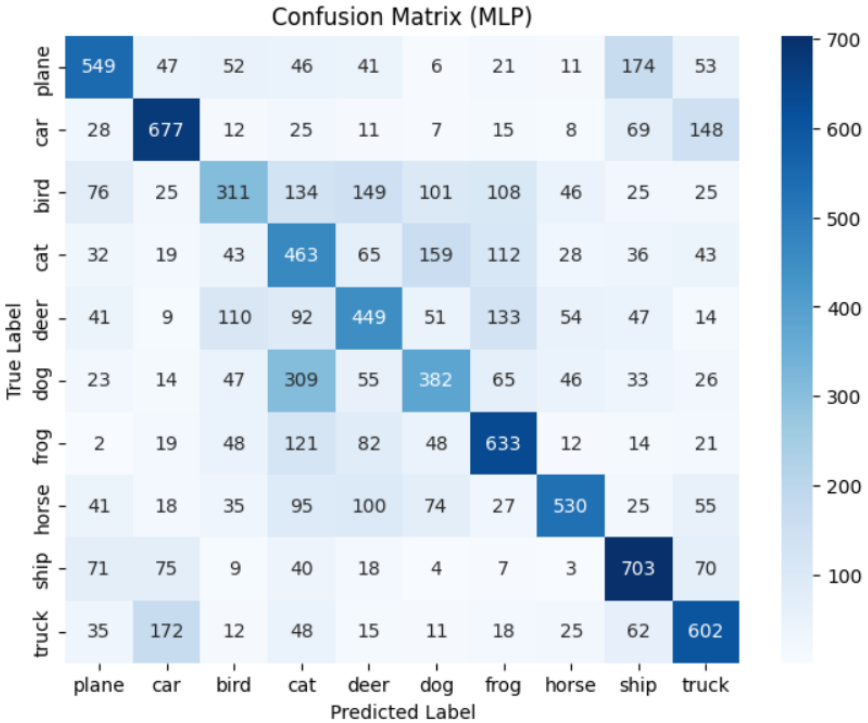

Figure 7: Confusion Matrix (MLP)

Comment: The confusion matrix shows that the MLP has difficulty distinguishing classes with similar visual features. Specifically, the number of false predictions between pairs like “cat” and “dog”, or “bird” and “plane” is quite high. The values on the main diagonal (correct predictions) are relatively low for some classes, confirming the limited performance of this model on complex datasets like CIFAR-10.

5.1.5 Show random predictions and initial comments on MLP performance

To get a visual idea of the model’s predictive ability, we’ve shown some random images from the test set along with the actual labels and the MLP’s predicted labels: **Random test image and MLP prediction:**


```
GroundTruth:  cat   ship  ship  plane frog  frog  car   frog  cat   car   plane truck dog   horse truck ship  dog
Predicted:  cat   truck ship  plane deer  frog  car   frog  dog   car   ship  car   cat   horse truck ship  cat
```

Figure 8: Random Test Image and MLP Prediction

Comment: Observing the images shows that the MLP can correctly predict some obvious objects like “car” or “ship”. However, it frequently makes mistakes with more complex or similar shaped objects like “cat”, “dog”, “bird” or “deer”. This reflects the low overall accuracy and limited feature learning ability of the MLP when processing raw images.

5.1.6 Save the trained model (Save Model)

After training, the state (`state_dict`) of the MLP model is saved to the `mlp_cifar10.pth` file. This allows us to reload the model without retraining, which is useful for future deployment or continued training. Save path: `./mlp_cifar10.pth`.

5.2 CNN model results

5.2.1 Training and testing process

The CNN model was trained for 20 epochs with the same configuration as MLP. Below is the summary training log of each epoch and print test accuracy:

```
--- Training CNN Model ---
Starting training for CNN model for 20 epochs...
Epoch [1/20], Train Loss: 1.5605, Train Acc: 43.50%, Val Loss: 1.3141, Val Acc: 53.32%
Epoch [2/20], Train Loss: 1.2110, Train Acc: 56.84%, Val Loss: 1.1307, Val Acc: 60.18%
Epoch [3/20], Train Loss: 1.0426, Train Acc: 63.29%, Val Loss: 1.0390, Val Acc: 64.56%
Epoch [4/20], Train Loss: 0.9460, Train Acc: 66.86%, Val Loss: 0.8916, Val Acc: 69.52%
Epoch [5/20], Train Loss: 0.8797, Train Acc: 69.55%, Val Loss: 0.8641, Val Acc: 70.68%
Epoch [6/20], Train Loss: 0.8296, Train Acc: 71.23%, Val Loss: 0.7993, Val Acc: 72.90%
Epoch [7/20], Train Loss: 0.7913, Train Acc: 72.46%, Val Loss: 0.7793, Val Acc: 73.42%
Epoch [8/20], Train Loss: 0.7570, Train Acc: 73.70%, Val Loss: 0.7663, Val Acc: 73.48%
Epoch [9/20], Train Loss: 0.7282, Train Acc: 74.74%, Val Loss: 0.7212, Val Acc: 75.44%
Epoch [10/20], Train Loss: 0.7064, Train Acc: 75.55%, Val Loss: 0.7242, Val Acc: 75.66%
Epoch [11/20], Train Loss: 0.6829, Train Acc: 76.37%, Val Loss: 0.7312, Val Acc: 75.26%
Epoch [12/20], Train Loss: 0.6637, Train Acc: 77.02%, Val Loss: 0.6763, Val Acc: 76.32%
Epoch [13/20], Train Loss: 0.6511, Train Acc: 77.32%, Val Loss: 0.6673, Val Acc: 77.26%
Epoch [14/20], Train Loss: 0.6316, Train Acc: 78.00%, Val Loss: 0.6747, Val Acc: 77.36%
Epoch [15/20], Train Loss: 0.6231, Train Acc: 78.34%, Val Loss: 0.6980, Val Acc: 76.50%
Epoch [16/20], Train Loss: 0.6031, Train Acc: 79.13%, Val Loss: 0.6533, Val Acc: 77.42%
Epoch [17/20], Train Loss: 0.6044, Train Acc: 78.95%, Val Loss: 0.6318, Val Acc: 78.38%
Epoch [18/20], Train Loss: 0.5821, Train Acc: 79.81%, Val Loss: 0.6380, Val Acc: 78.46%
Epoch [19/20], Train Loss: 0.5752, Train Acc: 79.86%, Val Loss: 0.6348, Val Acc: 78.82%
Epoch [20/20], Train Loss: 0.5628, Train Acc: 80.32%, Val Loss: 0.6049, Val Acc: 79.24%
Training complete.
CNN model saved to ./cnn_cifar10.pth
Starting testing for CNN model...
Test accuracy for CNN model: 80.03%
```

Figure 9: CNN Training Log

5.2.2 Learning Curves

The learning curve graph for the CNN shows a significant improvement in performance:
Loss Chart:

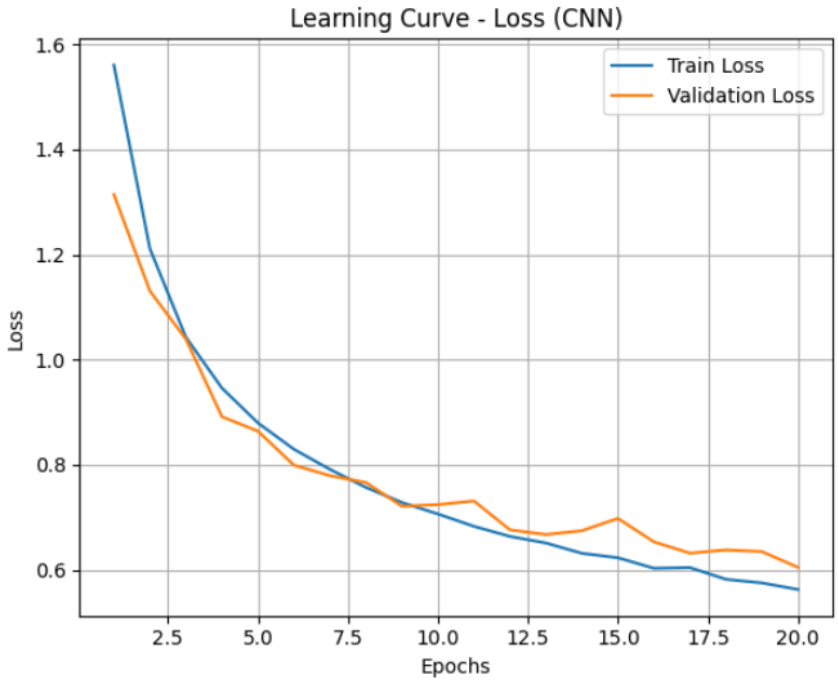

Figure 10: Learning Curve - Loss (CNN)

Comment: The CNN's Training Loss and Validation Loss decrease very quickly and converge to a much lower value than the MLP. The gap between the two lines is also sig-

nificantly smaller, indicating that the model learns efficiently and has good generalization ability without much overfitting.

Accuracy Chart:

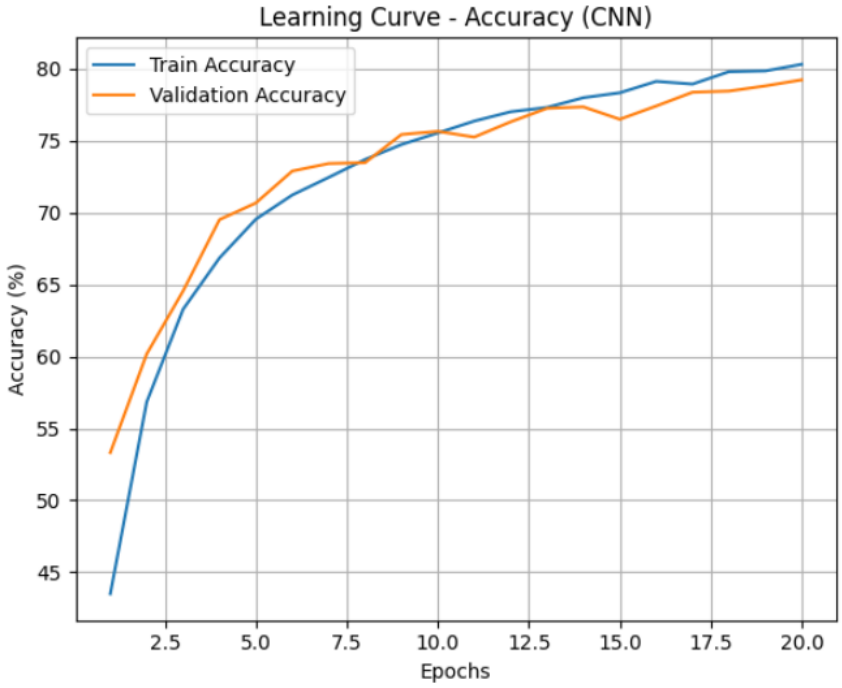

Figure 11: Learning Curve - Accuracy (CNN)

Comment: The accuracy on both the training and validation sets of the CNN increases rapidly and reaches very high levels. The validation accuracy curve is almost parallel and close to the training curve, indicating that the model is learning well and generalizing effectively on new data.

5.2.3 Classification Report

The report details the CNN performance on the test set:

```
--- CNN Classification Report ---
              precision    recall  f1-score   support

       plane       0.83      0.80      0.81      1000
         car       0.91      0.91      0.91      1000
        bird       0.73      0.69      0.71      1000
         cat       0.63      0.65      0.64      1000
        deer       0.78      0.76      0.77      1000
         dog       0.72      0.71      0.72      1000
        frog       0.84      0.86      0.85      1000
       horse       0.86      0.83      0.84      1000
        ship       0.84      0.93      0.88      1000
       truck       0.87      0.87      0.87      1000

    accuracy                           0.80     10000
   macro avg       0.80      0.80      0.80     10000
weighted avg       0.80      0.80      0.80     10000
```

Figure 12: CNN Classification Report

5.2.4 Confusion Matrix

CNN confusion matrix:

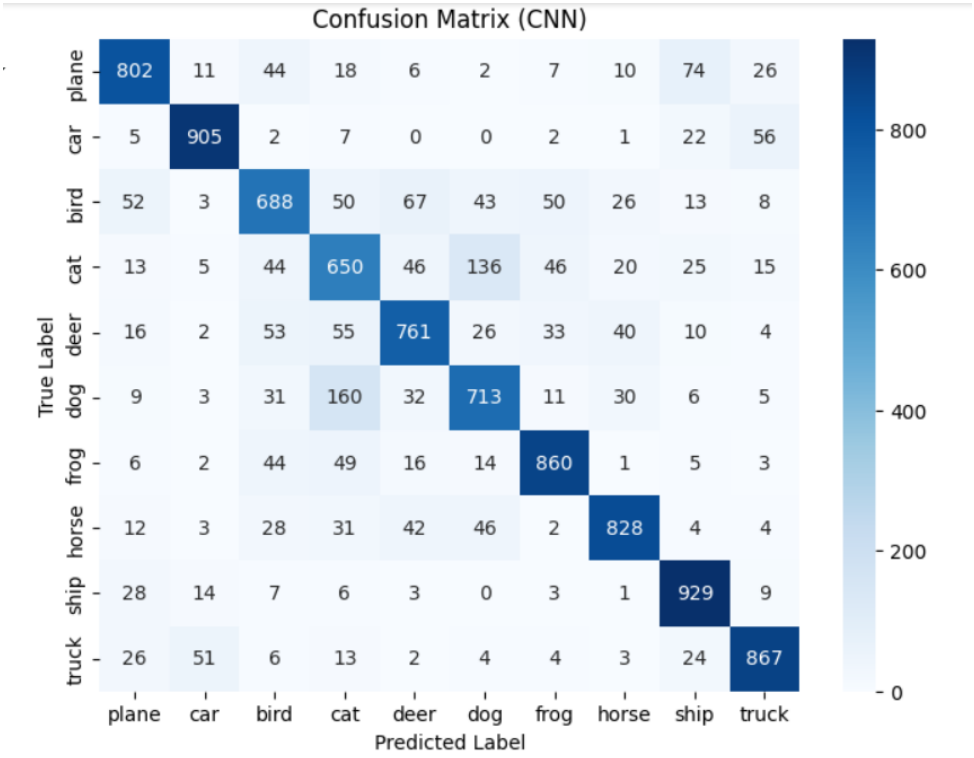

Figure 13: Confusion Matrix (CNN)

Comment: The CNN confusion matrix shows that the values on the main diagonal (correct predictions) are very high for most of the classes. This confirms the CNN's ability to classify accurately. Although there are still some small confusions between similar classes like "cat" and "dog", or "bird" and "plane", the number of these confusions is significantly less than that of MLP.

5.2.5 Random Prediction Display and Initial Comments on CNN Performance

Show some random images from the test set along with the actual label and the CNN's predicted label: **Random test image and CNN prediction:**

Figure 14: Random Test Image and CNN Prediction

Comment: CNN predictions on random images are much more accurate than MLP. Most of the objects, whether animals or vehicles, are classified correctly. This illustrates the powerful feature extraction and efficient classification capabilities of the CNN architecture.

5.2.6 Save the trained model (Save Model)

After training, the state (`state_dict`) of the CNN model has also been saved to the `cnn_cifar10.pth` file. This storage is important to reuse the model without having to train it from scratch. Save path: `./cnn_cifar10.pth`.

5.3 Comparison and discussion of results between MLP and CNN

5.3.1 Overview comparison table

To summarize, here is a table comparing the final accuracy of the two models on the test set:

Table 1: Comparison of Test Accuracy for MLP and CNN

Model	**Test Set Accuracy**
MLP	52.99%
CNN	80.03%

5.3.2 Analysis of causes of performance differences

The significant performance difference between MLP and CNN in the CIFAR-10 image classification problem can be explained by the following core reasons:

- **Image data nature:** Images have significant spatial structure (e.g. neighboring pixels have close relationships).

  - **MLP:** Processes an image by flattening it into a 1D vector. This process completely loses information about the relative positions between pixels. Therefore, MLPs have to learn from scratch the complex relationships between spatially unrelated pixels, making it difficult to extract meaningful features from the image.
  - **CNN:** The convolutional layers of a CNN are designed to take advantage of this spatial structure. They use filters (kernels) to scan through the image, detecting local patterns (like edges, textures, shapes). Sharing the weights of these filters across the entire image makes the CNN much more efficient at learning image features and also greatly reduces the number of parameters to learn.

- **Automatic feature extraction capabilities:**
  - **MLP:** As a "shallow learning" network, the hidden layers of an MLP are not capable of automatically learning high-level features from raw data. It mainly learns simple linear or nonlinear relationships between input pixels.
  - **CNN:** CNNs are capable of building a hierarchy of features. Early convolutional layers learn low-level features (e.g., edges, lines), while deeper layers combine these features to create higher-level features (e.g., eyes, ears, wheels). This ability to extract features automatically and hierarchically is key to CNNs' superior performance in computer vision tasks.

- **Translation Invariance:** The Max Pooling layers in CNN make the model less sensitive to small displacements of objects in the image. That is, no matter where the object appears in the image, the CNN is still able to recognize it. MLPs do not have this property; if an object moves position in the image, the input pixels will change significantly, and the MLP needs to relearn the pattern at the new position.

6 CONCLUSION

This paper has successfully performed image classification on the CIFAR-10 dataset by constructing and comparing two neural network models: Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN). The main results can be summarized as follows:

- **MLP Model:** The accuracy achieved on the test set is 52.99%. Although the model learns and the loss decreases, and the accuracy increases over epochs, the overall performance is limited. The learning curves show early saturation, and the classification report indicates difficulty in distinguishing classes with similar visual features (e.g., cat, dog, bird).
- **CNN model:** The accuracy achieved on the test set is 80.03%. This is a significant improvement, outperforming MLP by 27.04%. The CNN learning curves demonstrate fast learning, stability, and good generalization ability. The classification report and confusion matrix show that the CNN has a better ability to discriminate between classes, significantly reducing confusion errors.

This clear difference in performance confirms the superiority of CNNs in computer vision problems. CNNs effectively exploit the spatial structure of images through convolutional layers, weight sharing mechanisms, and the ability to learn hierarchical features, which MLPs cannot do due to data flattening.