POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF PYTHON PROGRAMMING LANGUAGE

FINAL ASSIGNMENT
PERFORM IMAGE CLASSIFICATION

INSTRUCTOR	: KIM NGUYEN BACH
CLASS	: D23CQCE04-B
GROUP	: 04
TRAN TRUNG KIEN	: B23DCCE058
CAO NGOC HUY	: B23DCCE043

Hanoi – 2025

Table of content

Overview and Objectives

In recent years, image classification and processing have become fundamental tasks in the field of computer vision and machine learning. The ability to automatically recognize and categorize visual information is critical in various real-world applications such as facial recognition, autonomous driving, medical imaging, and security systems. Among the many techniques used to solve these tasks, neural networks—especially Convolutional Neural Networks (CNNs)—have proven to be highly effective due to their ability to extract and learn hierarchical features from images.

This project aims to explore the concepts of image classification and image processing through the practical implementation and evaluation of two types of neural networks: a basic Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN). The CIFAR-10 dataset, a widely-used benchmark in computer vision, serves as the foundation for our experiments, providing a diverse collection of labeled images representing various object categories commonly encountered in real-world visual recognition tasks.

The main objectives of this assignment are:

To understand the differences between fully connected networks and convolutional architectures in handling image data.

To build, train, validate, and test both MLP and CNN models using the PyTorch deep learning framework.

To evaluate model performance using learning curves and confusion matrices.

To analyze and compare the effectiveness of the two approaches in the context of image classification.

The ultimate goal of this assignment is not only to develop a functional image classification system but also to enhance our understanding of image data, strengthen our ability to design and evaluate neural network models, and develop the skills to interpret and communicate visual recognition results effectively. These are essential competencies for anyone aiming to work in computer vision, machine learning, or AI-driven application development.

Acknowledgements

We would like to express our sincere gratitude to Mr. Kim Nguyen Bach, instructor of the Python Programming course, for his dedicated teaching and continuous support throughout the semester. His guidance has been instrumental in helping us understand both the theoretical foundations and practical applications of Python in the field of artificial intelligence.

Through this assignment, we had the opportunity to engage deeply with key concepts in image processing and classification. By implementing and comparing a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN), we not only strengthened our understanding of neural network architectures but also gained valuable experience in model training, evaluation, and performance analysis. Working with the CIFAR-10 dataset allowed us to confront real-world challenges such as image normalization, overfitting, and hyperparameter tuning—skills that are essential in the development of modern computer vision systems.

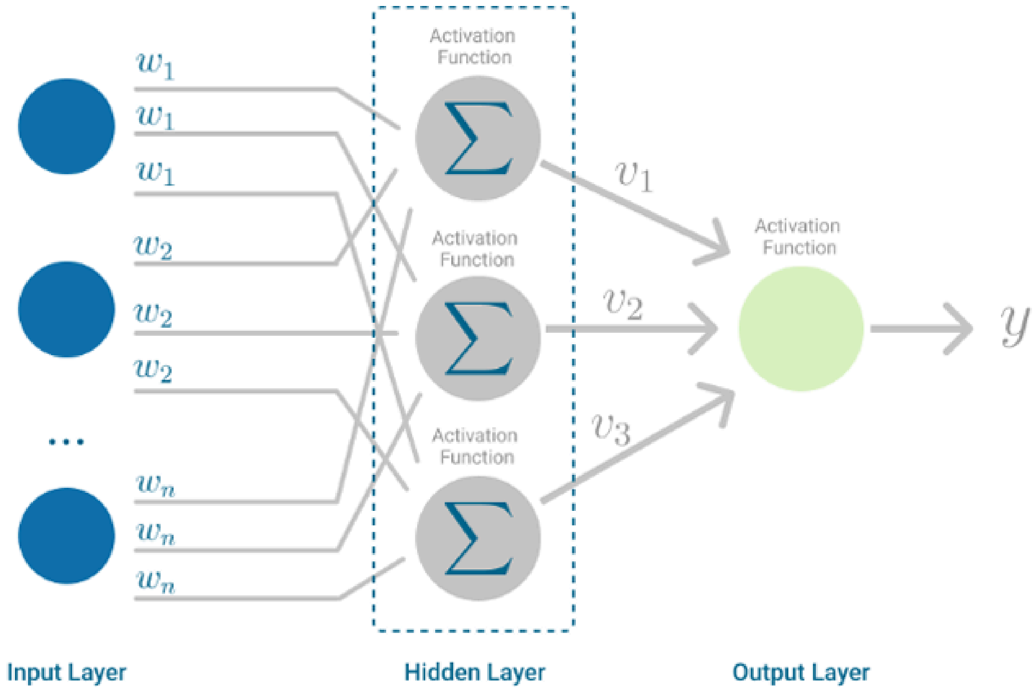
This project also helped us improve our collaboration and project management skills. From planning the workflow to debugging code and interpreting results, we learned to communicate effectively, divide responsibilities, and solve problems as a team. These are crucial abilities for any future career in software development, machine learning, or research.

While we strived to complete the assignment as thoroughly as possible, we recognize that there is always room for growth. We welcome constructive feedback from our instructor and peers, and we look forward to applying these lessons to future projects.

Thank you very much for your support and encouragement!

1. Introduction to MLP and CNN

1.1. Multi-Layer Perceptron (MLP)



The Multi-Layer Perceptron (MLP) is a type of feedforward neural network widely used in machine learning. The basic structure of an MLP includes at least three layers: an input layer, one or more hidden layers, and an output layer. Except for the input layer, each neuron in the network performs computations and applies a nonlinear activation function to increase the network's ability to learn complex relationships in the data.

Structure of MLP:

Input Layer
This layer receives the input data. For image data, the image is usually flattened from a 2D matrix (or 3D if multiple color channels) into a one-dimensional vector. Each element of this vector corresponds to a neuron in the input layer. This layer does not perform any computation but simply passes the data to the first hidden layer.

Hidden Layers
Each hidden layer consists of neurons fully connected to the previous layer. Each connection carries a unique weight, along with a bias term for each neuron. The weighted sum of inputs is passed through a nonlinear activation function such as ReLU, Sigmoid, or Tanh. Increasing the number of hidden layers or neurons can

improve the network's representation capacity but also increases complexity and the risk of overfitting.

Output Layer

The output layer is the last layer of the network and generates the predictions. It is also fully connected to the preceding hidden layer. The number of neurons in the output layer depends on the desired output, for example, 10 neurons for a 10-class classification problem (like CIFAR-10). The softmax activation function is commonly used here to produce a probability distribution, helping the model make more accurate predictions.

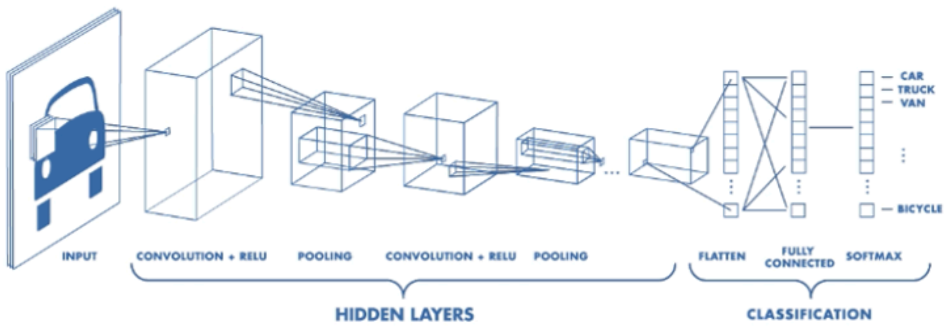
Activation Functions

Activation functions play a critical role in enabling the network to learn nonlinear mappings. In MLPs, ReLU (f(x) = max(0, x)) is commonly used because it is simple and effective in training. Without nonlinear activation functions, no matter how many layers the network has, it would be equivalent to a simple linear model.

Training and Applications

MLPs are trained using backpropagation combined with optimizers such as SGD or Adam to update weights based on a loss function. Although MLPs are powerful for tabular or sequential data, they have a major limitation when working with images — flattening images destroys spatial relationships between pixels, making the model unable to exploit geometric structures like edges, boundaries, or shapes.

1.2. Convolutional Neural Network (CNN)



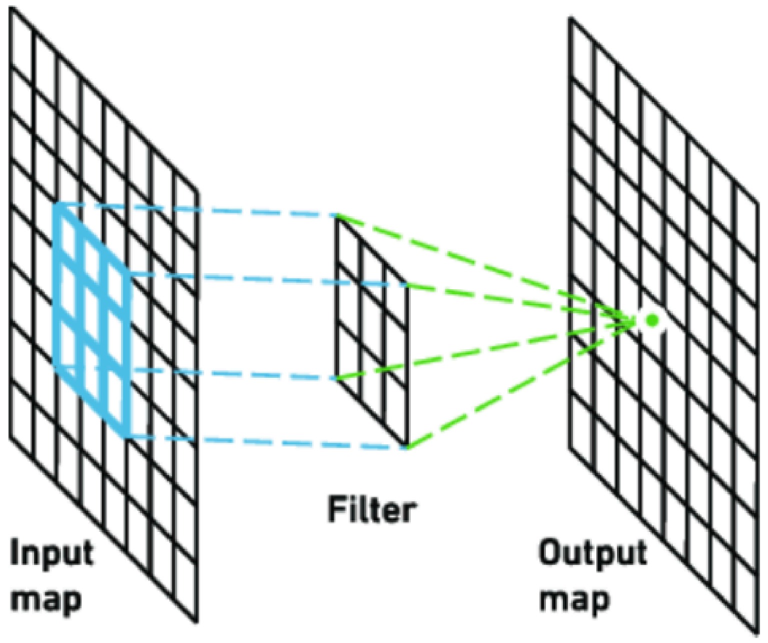
The Convolutional Neural Network (CNN) is a deep learning architecture specifically designed for data with spatial structure such as images. CNNs can learn local features and aggregate them into higher-level abstract features through multiple stacked layers. This allows CNNs to achieve very high performance in computer vision tasks like image classification, object detection, and face recognition.

Main components of CNN:

Convolutional Layer

This is the core layer of CNNs, where filters (kernels) slide over the input image to extract local features such as edges, corners, or textures. Each filter is learned during training and produces a feature map. Important parameters include:

- **Stride:** the step size of the filter as it moves across the image.
- **Padding:** adding fake pixels at the edges to control output size.
- **Number of filters:** determines the depth of output and the number of features learned.



ReLU Activation Function

ReLU is applied after each convolution operation to introduce nonlinearity into the network. It speeds up training and helps prevent the vanishing gradient problem by keeping positive values and discarding negative ones.

Pooling Layer

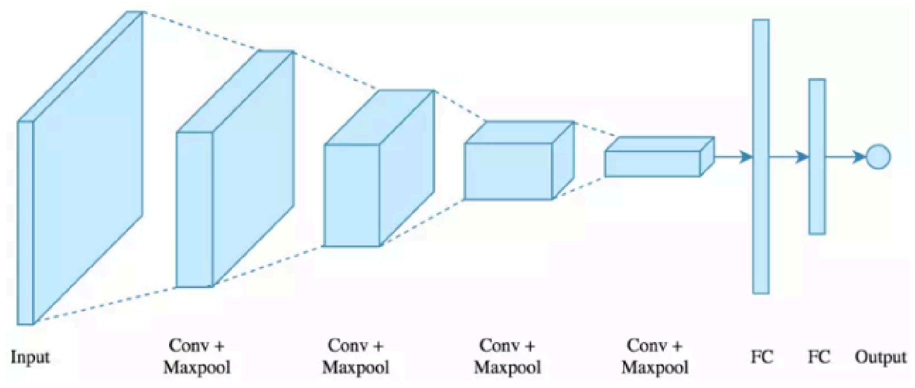
Pooling layers reduce the spatial dimensions of feature maps, which decreases the number of parameters and computational cost. Common pooling methods are:

- **Max Pooling:** selects the maximum value in each region.

- **Average Pooling:** calculates the average value in each region.

A typical configuration uses a 2×2 window with stride 2, halving the width and height of the input.

Fully Connected Layer
After convolution and pooling layers, the output is flattened and passed to fully connected layers for classification. Each neuron here connects to every neuron in the previous layer. The final layer usually has as many neurons as the number of classes.

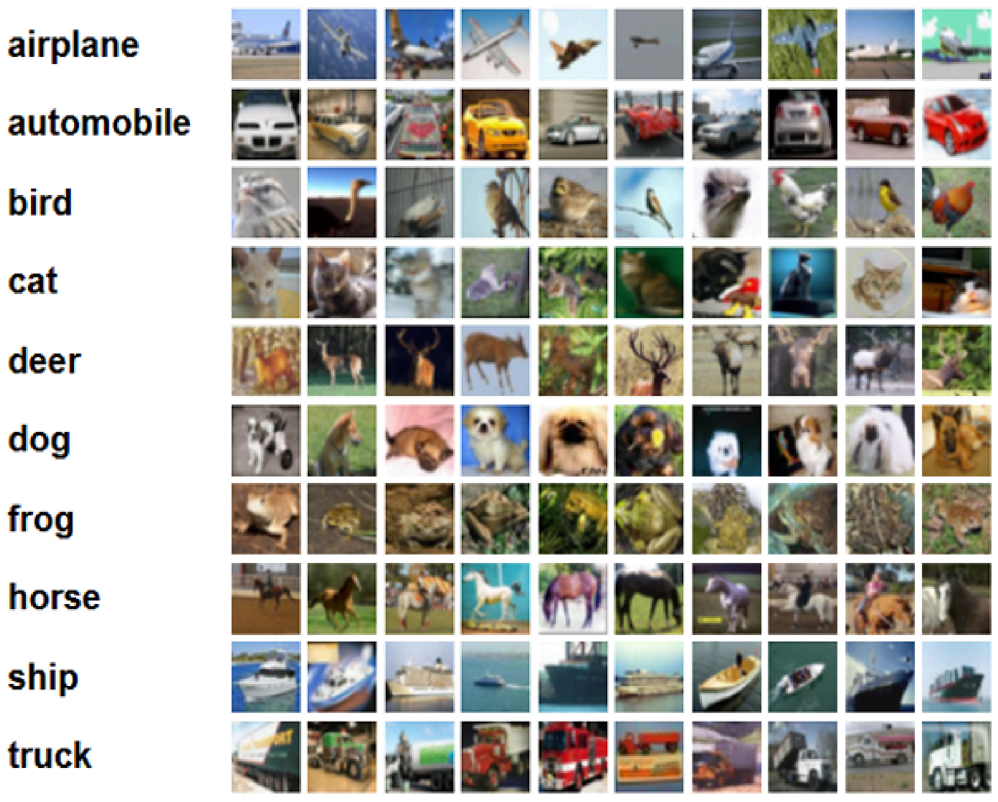


Standard and Effective Architectures
Modern CNN architectures may have from 3 up to more than 10 convolutional layers, depending on the problem complexity and data size. With large datasets like ImageNet, famous CNN models such as LeNet, AlexNet, VGG, and ResNet have been developed, achieving very high accuracy in competitions like ILSVRC.

Today, CNNs are the dominant tool in image processing thanks to their hierarchical feature learning, computational efficiency, and superior accuracy compared to traditional models.

2. Introduction to CIFAR-10

CIFAR-10 is a standard dataset commonly used for image classification tasks. It contains 60,000 color images, each with a resolution of 32×32 pixels. The dataset is divided into 10 classes, with 6,000 images per class. These classes include: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are mutually exclusive. The dataset is split into 50,000 training images and 10,000 test images.



3. Programming Languages, Libraries, and Tools Used

To complete this assignment, we used the following programming language, libraries, and tools as seen in the notebook:

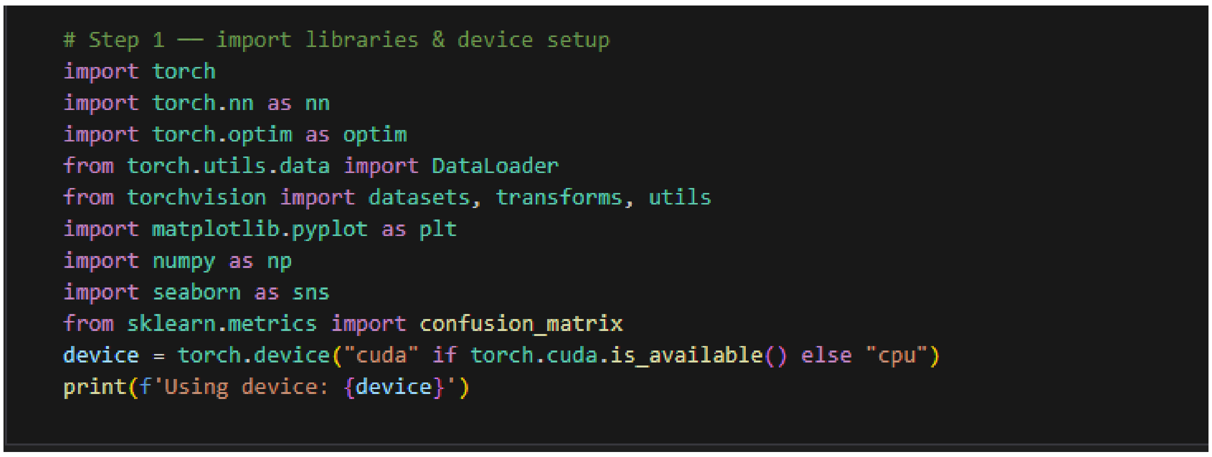
- **Programming Language**: Python
- **Deep Learning Framework: PyTorch**
    - torch: Core tensor operations and model utilities.
    - torch.nn: Tools for building neural network layers and defining loss functions.
    - torch.optim: Optimizers such as SGD, Adam for training.
    - torch.utils.data.DataLoader: Efficient batch data loading for training and evaluation.
- **Computer Vision Library: Torchvision**
    - torchvision.datasets: Access to image datasets like CIFAR-10.
    - torchvision.transforms: Preprocessing utilities (e.g., ToTensor, Normalize).
    - torchvision.utils: Miscellaneous vision-related helper functions.
- **Visualization Library: Matplotlib**
    - matplotlib.pyplot: Used for plotting learning curves, accuracy/loss graphs, and sample images.
- **Numerical Library: NumPy**
    - numpy: Efficient numerical operations on arrays and matrices, used for data handling and preprocessing.
- **Device Setup**:

    - The computation is set to run on a GPU (cuda) if available, otherwise defaults to CPU

4. Build Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN)

4.1. Import libraries and set up devices

We used essential libraries in PyTorch and for data handling:

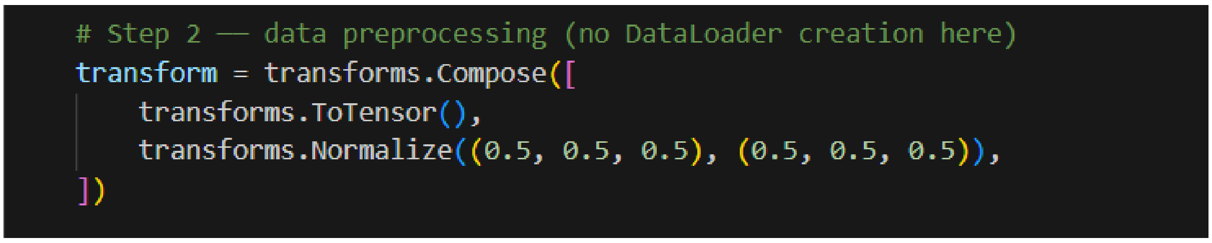


```
# Step 1 — import libraries & device setup
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, utils
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Using device: {device}')
```

The computation device was identified (GPU if available)

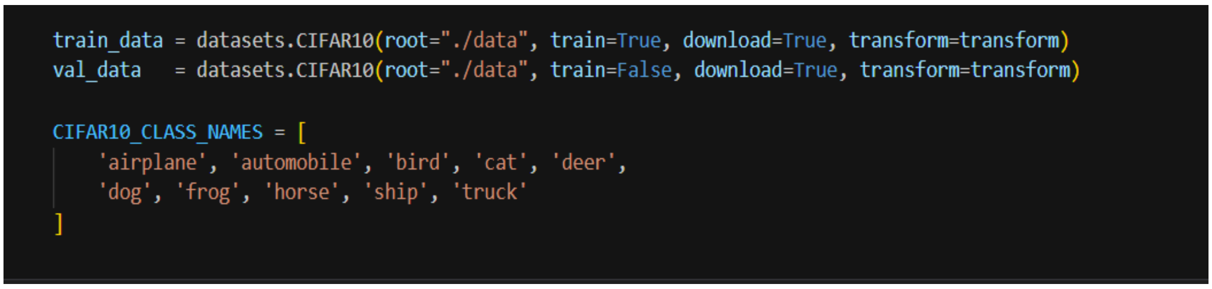
4.2. Data Loading and Preprocessing

Image transformations were applied as follows:



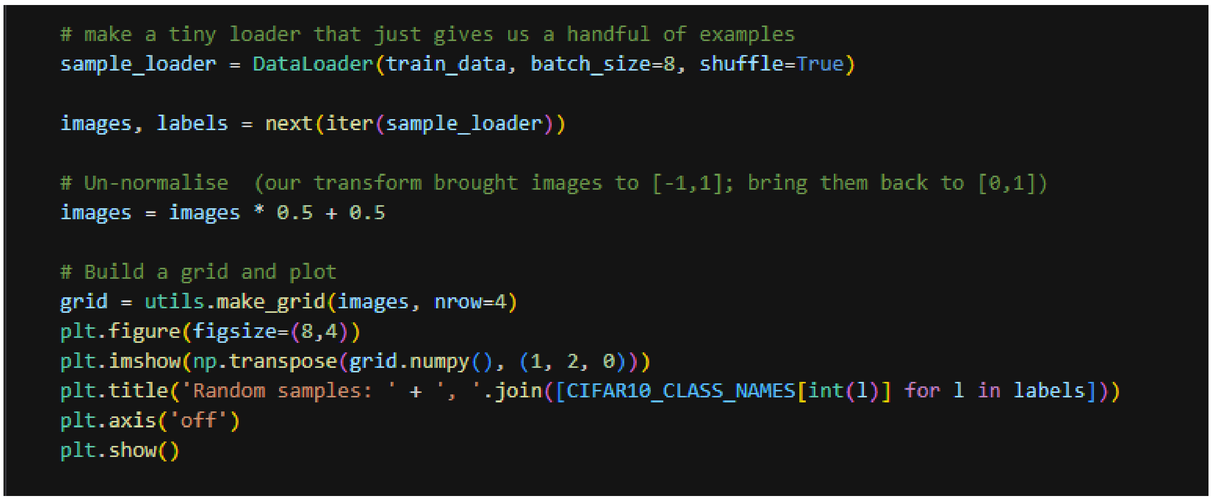
```
# Step 2 — data preprocessing (no DataLoader creation here)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

CIFAR-10 data loading:



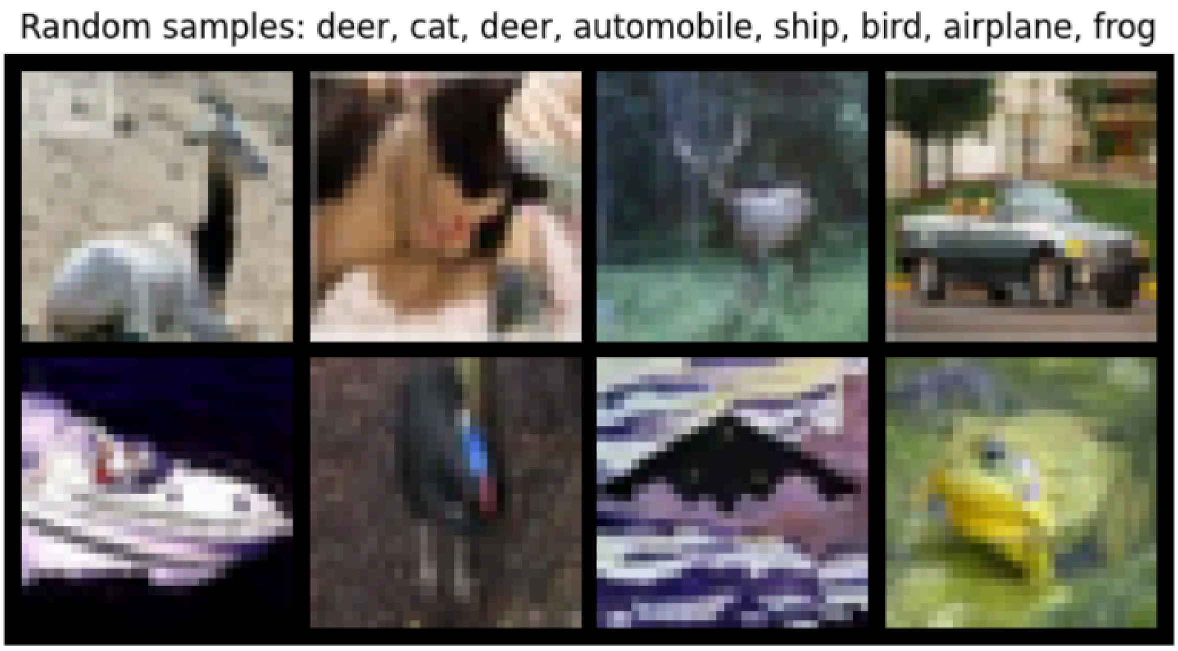
```
train_data = datasets.CIFAR10(root="./data", train=True, download=True, transform=transform)
val_data   = datasets.CIFAR10(root="./data", train=False, download=True, transform=transform)

CIFAR10_CLASS_NAMES = [
    'airplane', 'automobile', 'bird', 'cat', 'deer',
    'dog', 'frog', 'horse', 'ship', 'truck'
]
```

A sample image visualization function was implemented:



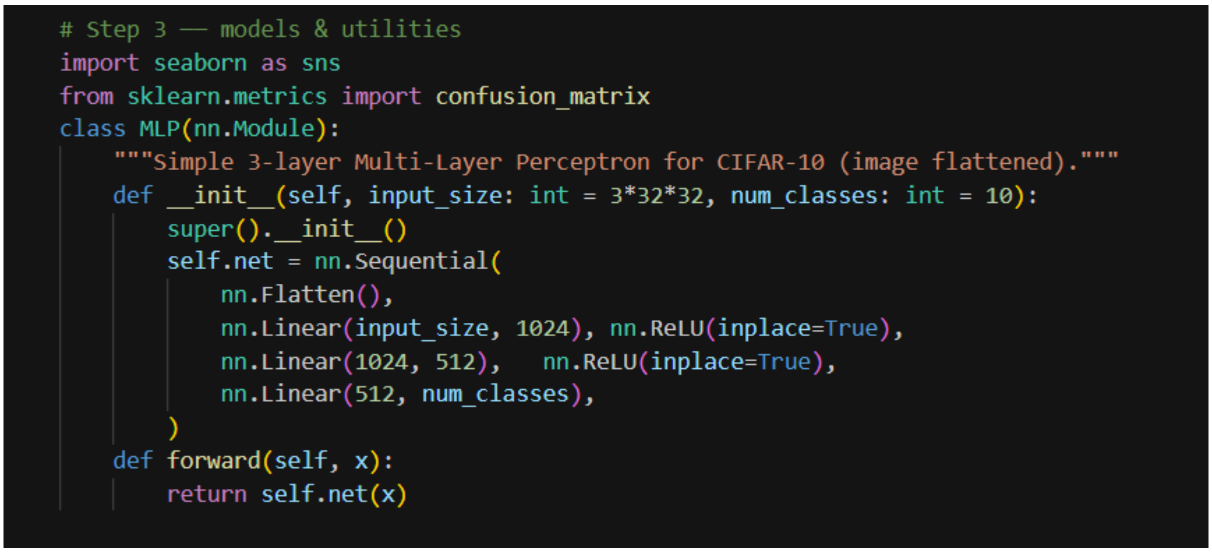
```
# make a tiny loader that just gives us a handful of examples
sample_loader = DataLoader(train_data, batch_size=8, shuffle=True)

images, labels = next(iter(sample_loader))

# Un-normalise  (our transform brought images to [-1,1]; bring them back to [0,1])
images = images * 0.5 + 0.5

# Build a grid and plot
grid = utils.make_grid(images, nrow=4)
plt.figure(figsize=(8,4))
plt.imshow(np.transpose(grid.numpy(), (1, 2, 0)))
plt.title('Random samples: ' + ', '.join([CIFAR10_CLASS_NAMES[int(l)] for l in labels]))
plt.axis('off')
plt.show()
```

Result :


Random samples: deer, cat, deer, automobile, ship, bird, airplane, frog

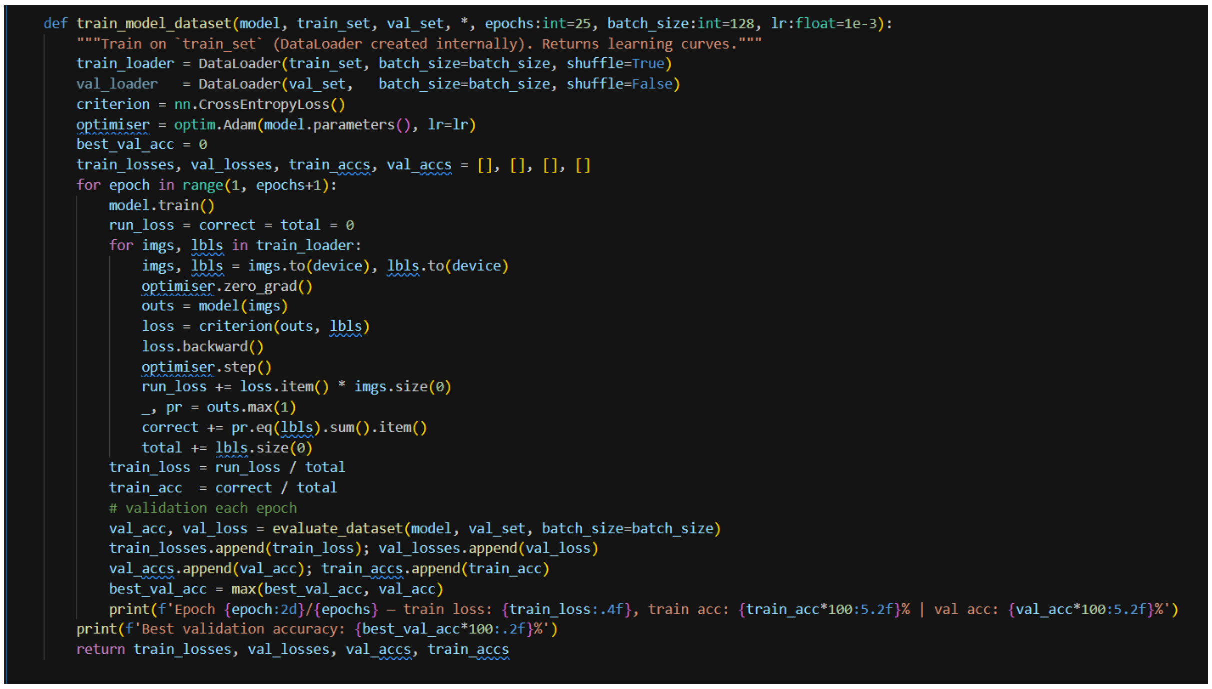
4.3. Multi-Layer Perceptron (MLP) Implementation

We defined the MLP model as follows:

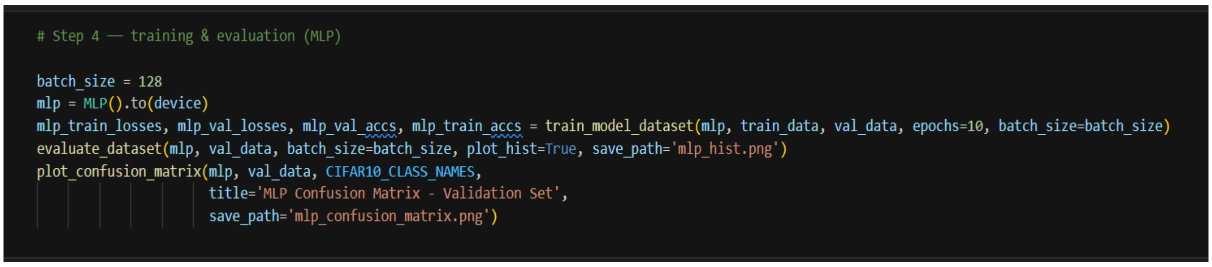


```python
# Step 3 — models & utilities
import seaborn as sns
from sklearn.metrics import confusion_matrix
class MLP(nn.Module):
    """Simple 3-layer Multi-Layer Perceptron for CIFAR-10 (image flattened)."""
    def __init__(self, input_size: int = 3*32*32, num_classes: int = 10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_size, 1024), nn.ReLU(inplace=True),
            nn.Linear(1024, 512),   nn.ReLU(inplace=True),
            nn.Linear(512, num_classes),
        )
    def forward(self, x):
        return self.net(x)
```

Training details:

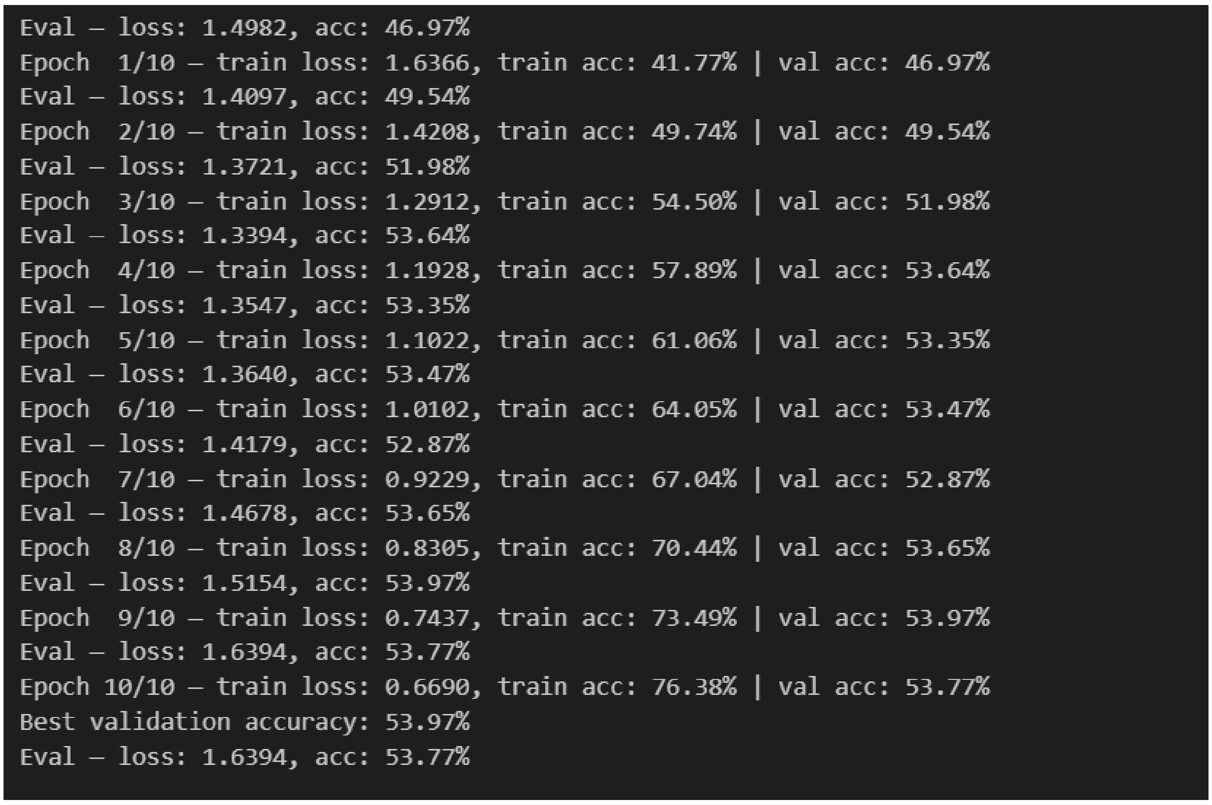


```python
def train_model_dataset(model, train_set, val_set, *, epochs:int=25, batch_size:int=128, lr:float=1e-3):
    """Train on `train_set` (DataLoader created internally). Returns learning curves."""
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val_loader   = DataLoader(val_set,   batch_size=batch_size, shuffle=False)
    criterion = nn.CrossEntropyLoss()
    optimiser = optim.Adam(model.parameters(), lr=lr)
    best_val_acc = 0
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    for epoch in range(1, epochs+1):
        model.train()
        run_loss = correct = total = 0
        for imgs, lbls in train_loader:
            imgs, lbls = imgs.to(device), lbls.to(device)
            optimiser.zero_grad()
            outs = model(imgs)
            loss = criterion(outs, lbls)
            loss.backward()
            optimiser.step()
            run_loss += loss.item() * imgs.size(0)
            _, pr = outs.max(1)
            correct += pr.eq(lbls).sum().item()
            total += lbls.size(0)
        train_loss = run_loss / total
        train_acc  = correct / total
        # validation each epoch
        val_acc, val_loss = evaluate_dataset(model, val_set, batch_size=batch_size)
        train_losses.append(train_loss); val_losses.append(val_loss)
        val_accs.append(val_acc); train_accs.append(train_acc)
        best_val_acc = max(best_val_acc, val_acc)
        print(f'Epoch {epoch:2d}/{epochs} — train loss: {train_loss:.4f}, train acc: {train_acc*100:5.2f}% | val acc: {val_acc*100:5.2f}%')
    print(f'Best validation accuracy: {best_val_acc*100:.2f}%')
    return train_losses, val_losses, val_accs, train_accs
```

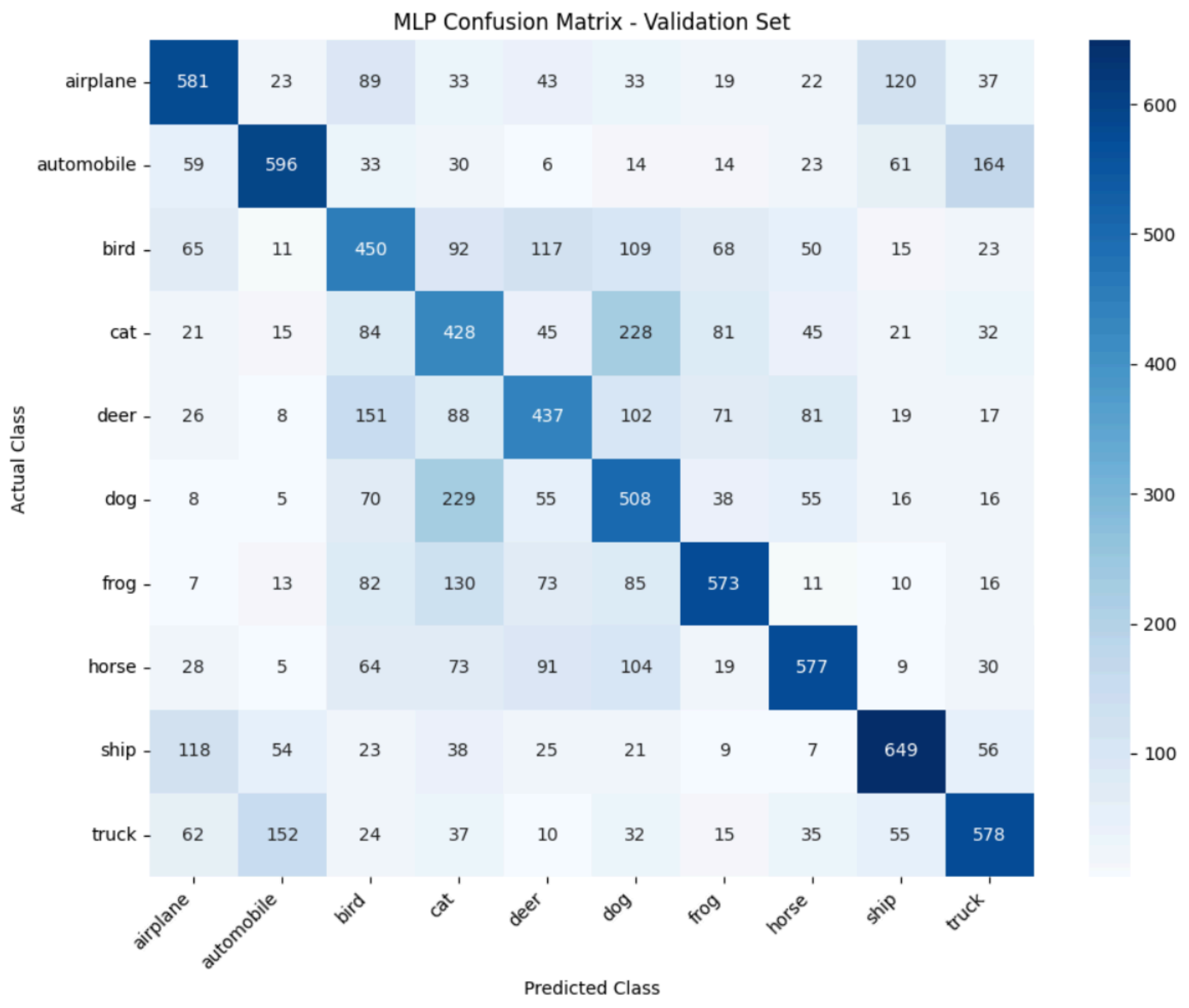


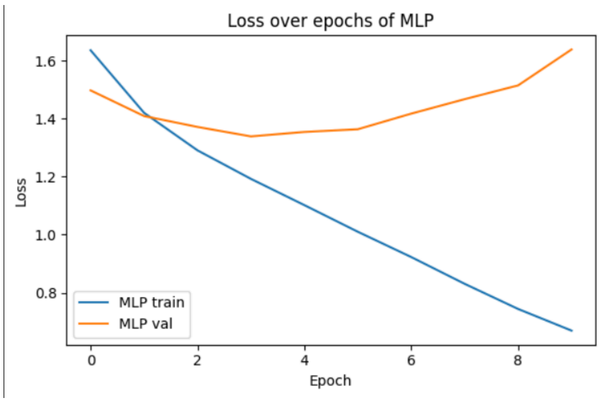
```python
# Step 4 — training & evaluation (MLP)

batch_size = 128
mlp = MLP().to(device)
mlp_train_losses, mlp_val_losses, mlp_val_accs, mlp_train_accs = train_model_dataset(mlp, train_data, val_data, epochs=10, batch_size=batch_size)
evaluate_dataset(mlp, val_data, batch_size=batch_size, plot_hist=True, save_path='mlp_hist.png')
plot_confusion_matrix(mlp, val_data, CIFAR10_CLASS_NAMES,
                      title='MLP Confusion Matrix - Validation Set',
                      save_path='mlp_confusion_matrix.png')
```

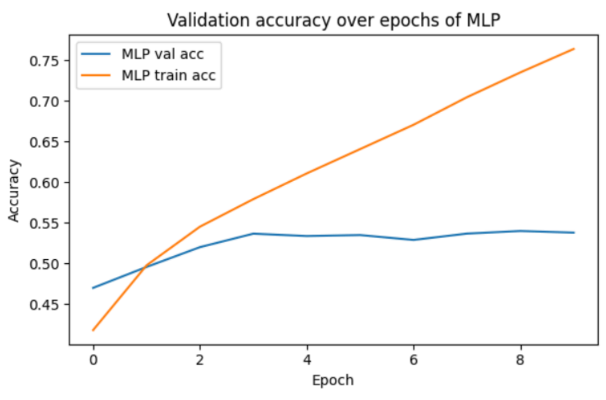
Results obtained :



```
Eval — loss: 1.4982, acc: 46.97%
Epoch  1/10 — train loss: 1.6366, train acc: 41.77% | val acc: 46.97%
Eval — loss: 1.4097, acc: 49.54%
Epoch  2/10 — train loss: 1.4208, train acc: 49.74% | val acc: 49.54%
Eval — loss: 1.3721, acc: 51.98%
Epoch  3/10 — train loss: 1.2912, train acc: 54.50% | val acc: 51.98%
Eval — loss: 1.3394, acc: 53.64%
Epoch  4/10 — train loss: 1.1928, train acc: 57.89% | val acc: 53.64%
Eval — loss: 1.3547, acc: 53.35%
Epoch  5/10 — train loss: 1.1022, train acc: 61.06% | val acc: 53.35%
Eval — loss: 1.3640, acc: 53.47%
Epoch  6/10 — train loss: 1.0102, train acc: 64.05% | val acc: 53.47%
Eval — loss: 1.4179, acc: 52.87%
Epoch  7/10 — train loss: 0.9229, train acc: 67.04% | val acc: 52.87%
Eval — loss: 1.4678, acc: 53.65%
Epoch  8/10 — train loss: 0.8305, train acc: 70.44% | val acc: 53.65%
Eval — loss: 1.5154, acc: 53.97%
Epoch  9/10 — train loss: 0.7437, train acc: 73.49% | val acc: 53.97%
Eval — loss: 1.6394, acc: 53.77%
Epoch 10/10 — train loss: 0.6690, train acc: 76.38% | val acc: 53.77%
Best validation accuracy: 53.97%
Eval — loss: 1.6394, acc: 53.77%
```







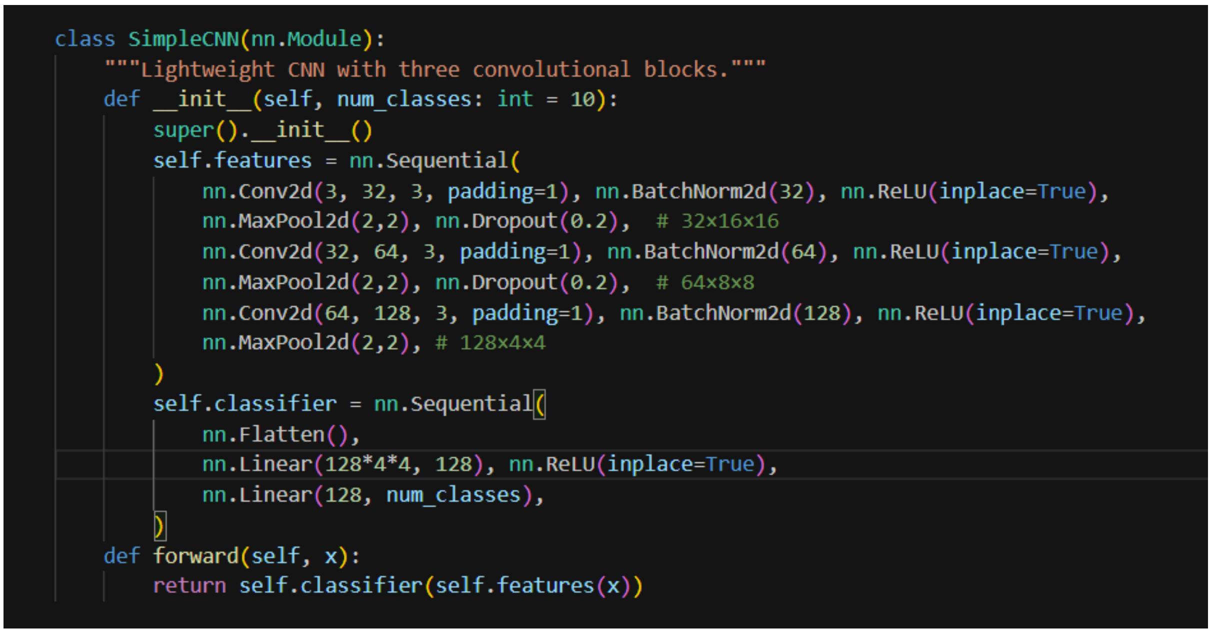
Best validation accuracy achieved by the MLP model was 53.97%.

It performed well on classes like ship, automobile, and airplane.

The model struggled to distinguish between visually similar classes such as cat, dog, bird,   and deer.

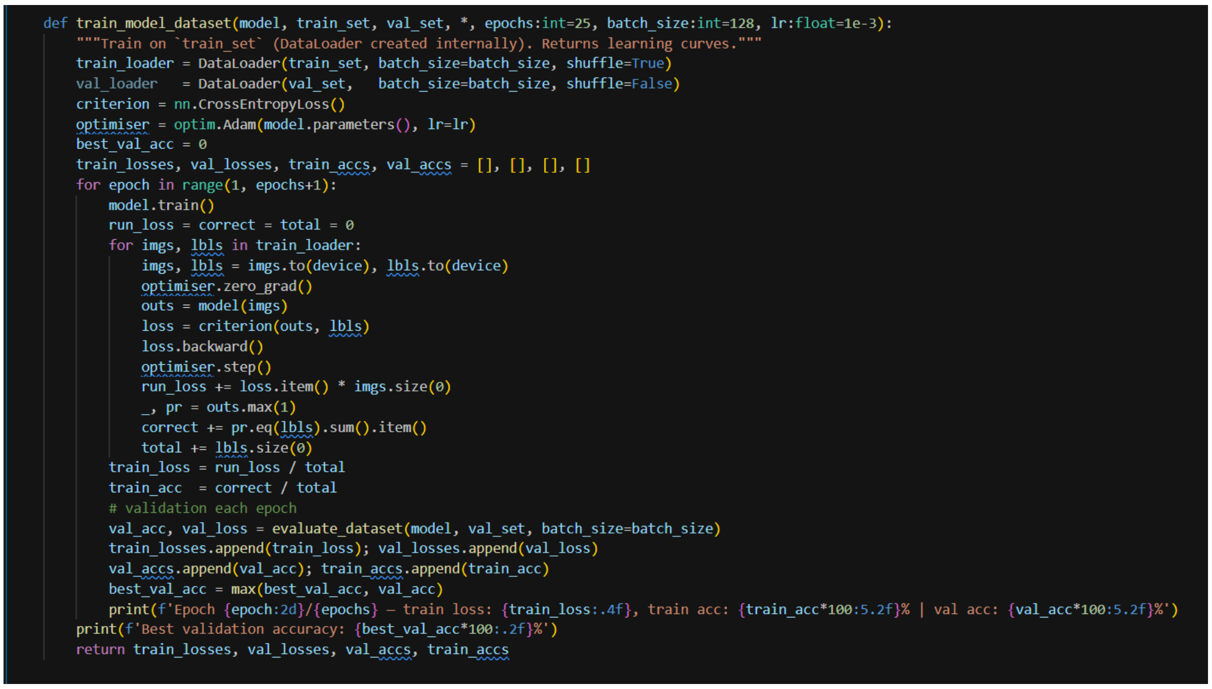
4.4. Convolutional Neural Network (CNN) Implementation

We implemented a simple CNN architecture as follows:

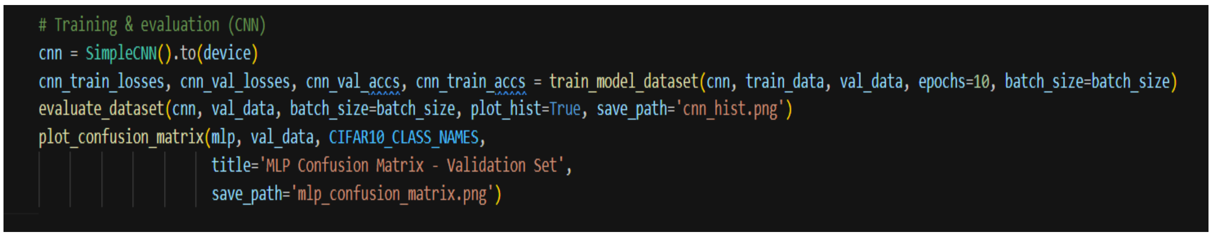


```python
class SimpleCNN(nn.Module):
    """Lightweight CNN with three convolutional blocks."""
    def __init__(self, num_classes: int = 10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32), nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2), nn.Dropout(0.2),  # 32×16×16
            nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2), nn.Dropout(0.2),  # 64×8×8
            nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2), # 128×4×4
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128*4*4, 128), nn.ReLU(inplace=True),
            nn.Linear(128, num_classes),
        )
    def forward(self, x):
        return self.classifier(self.features(x))
```

Training details:

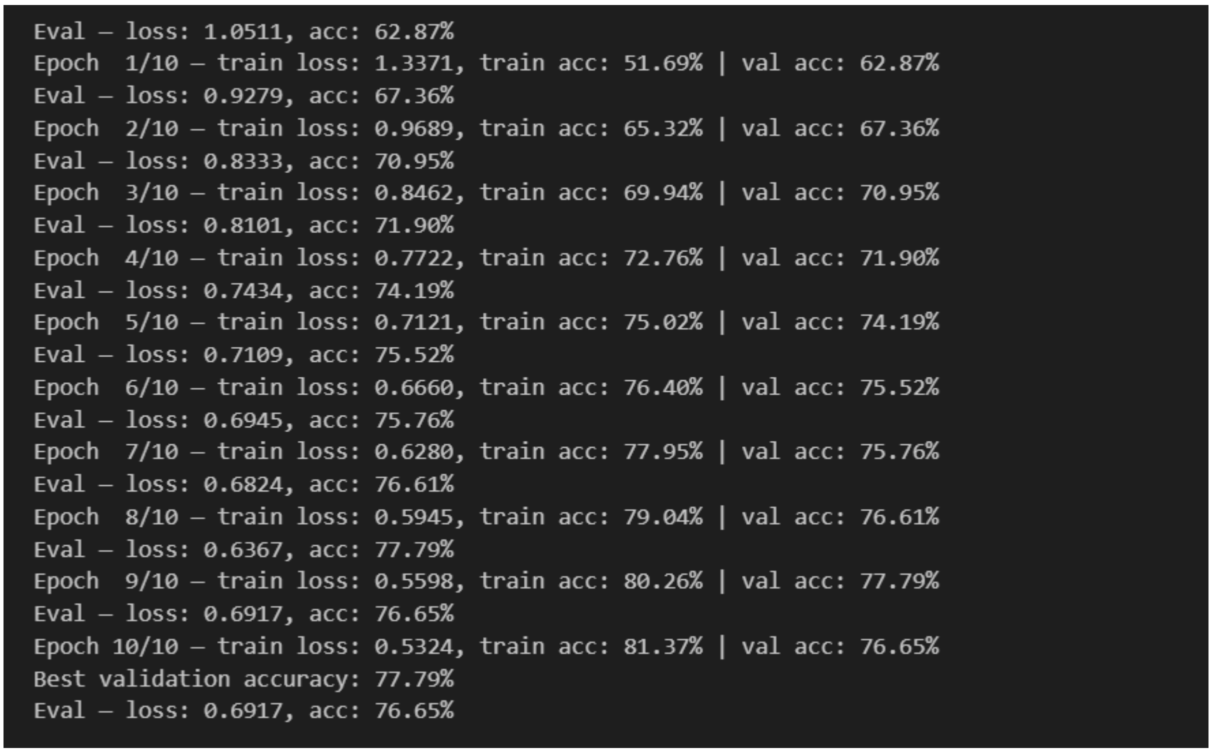


```python
def train_model_dataset(model, train_set, val_set, *, epochs:int=25, batch_size:int=128, lr:float=1e-3):
    """Train on `train_set` (DataLoader created internally). Returns learning curves."""
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val_loader   = DataLoader(val_set,   batch_size=batch_size, shuffle=False)
    criterion = nn.CrossEntropyLoss()
    optimiser = optim.Adam(model.parameters(), lr=lr)
    best_val_acc = 0
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    for epoch in range(1, epochs+1):
        model.train()
        run_loss = correct = total = 0
        for imgs, lbls in train_loader:
            imgs, lbls = imgs.to(device), lbls.to(device)
            optimiser.zero_grad()
            outs = model(imgs)
            loss = criterion(outs, lbls)
            loss.backward()
            optimiser.step()
            run_loss += loss.item() * imgs.size(0)
            _, pr = outs.max(1)
            correct += pr.eq(lbls).sum().item()
            total += lbls.size(0)
        train_loss = run_loss / total
        train_acc  = correct / total
        # validation each epoch
        val_acc, val_loss = evaluate_dataset(model, val_set, batch_size=batch_size)
        train_losses.append(train_loss); val_losses.append(val_loss)
        val_accs.append(val_acc); train_accs.append(train_acc)
        best_val_acc = max(best_val_acc, val_acc)
        print(f'Epoch {epoch:2d}/{epochs} – train loss: {train_loss:.4f}, train acc: {train_acc*100:5.2f}% | val acc: {val_acc*100:5.2f}%')
    print(f'Best validation accuracy: {best_val_acc*100:.2f}%')
    return train_losses, val_losses, val_accs, train_accs
```

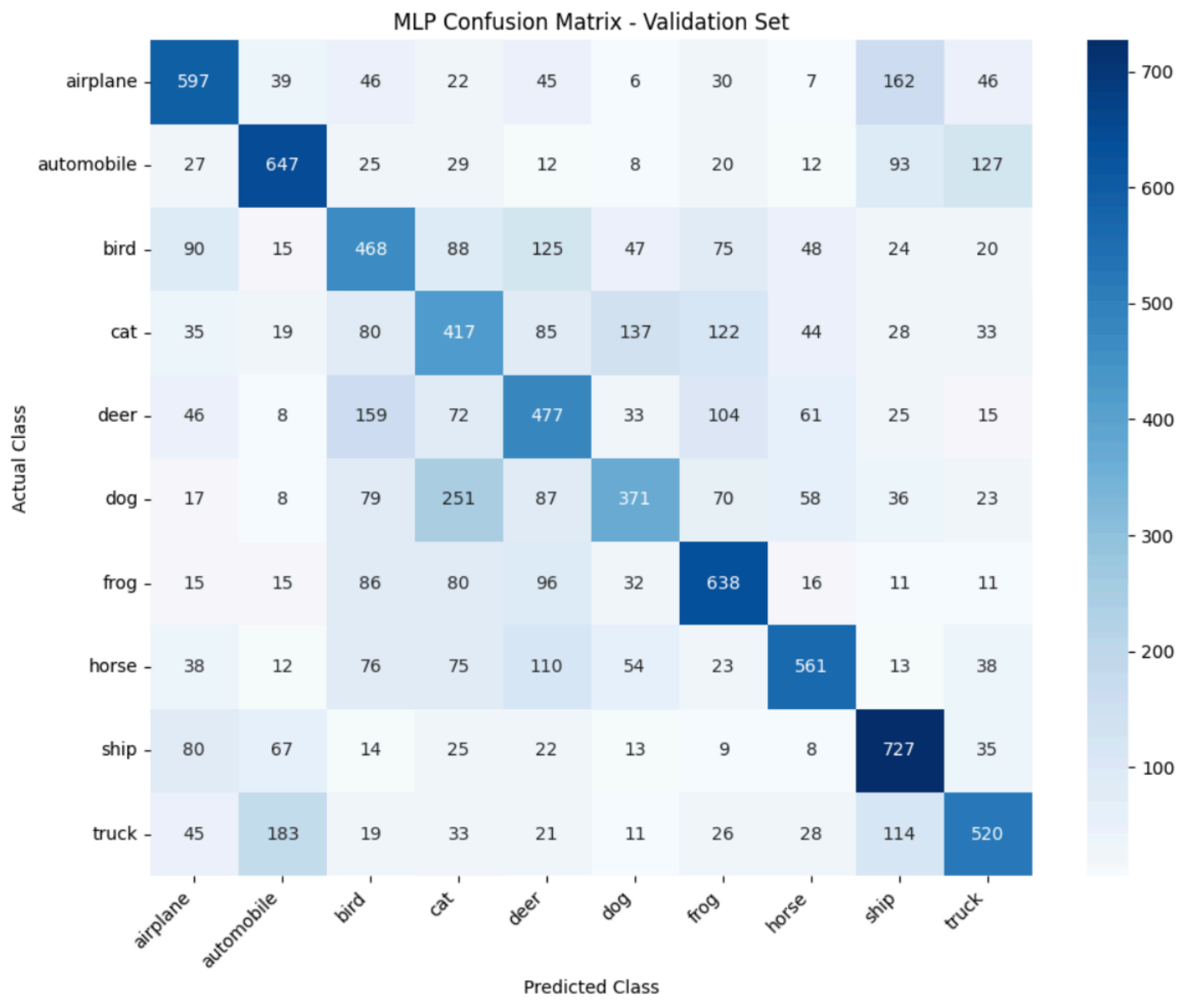


```python
# Training & evaluation (CNN)
cnn = SimpleCNN().to(device)
cnn_train_losses, cnn_val_losses, cnn_val_accs, cnn_train_accs = train_model_dataset(cnn, train_data, val_data, epochs=10, batch_size=batch_size)
evaluate_dataset(cnn, val_data, batch_size=batch_size, plot_hist=True, save_path='cnn_hist.png')
plot_confusion_matrix(mlp, val_data, CIFAR10_CLASS_NAMES,
                      title='MLP Confusion Matrix - Validation Set',
                      save_path='mlp_confusion_matrix.png')
```

Results obtained :

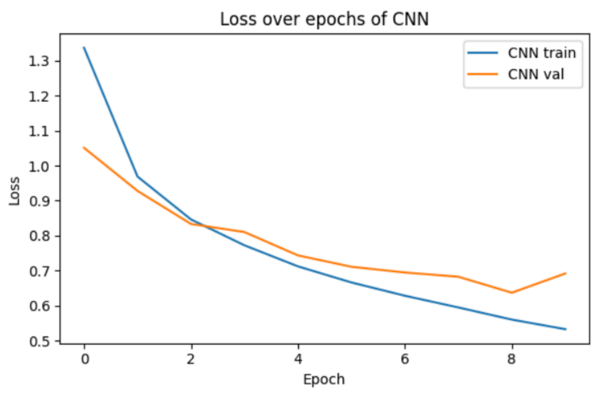


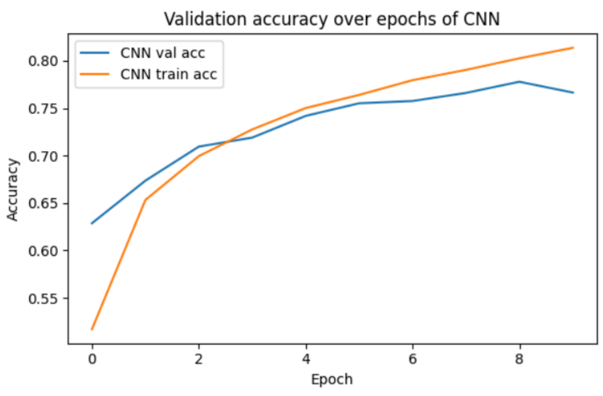
```
Eval — loss: 1.0511, acc: 62.87%
Epoch  1/10 — train loss: 1.3371, train acc: 51.69% | val acc: 62.87%
Eval — loss: 0.9279, acc: 67.36%
Epoch  2/10 — train loss: 0.9689, train acc: 65.32% | val acc: 67.36%
Eval — loss: 0.8333, acc: 70.95%
Epoch  3/10 — train loss: 0.8462, train acc: 69.94% | val acc: 70.95%
Eval — loss: 0.8101, acc: 71.90%
Epoch  4/10 — train loss: 0.7722, train acc: 72.76% | val acc: 71.90%
Eval — loss: 0.7434, acc: 74.19%
Epoch  5/10 — train loss: 0.7121, train acc: 75.02% | val acc: 74.19%
Eval — loss: 0.7109, acc: 75.52%
Epoch  6/10 — train loss: 0.6660, train acc: 76.40% | val acc: 75.52%
Eval — loss: 0.6945, acc: 75.76%
Epoch  7/10 — train loss: 0.6280, train acc: 77.95% | val acc: 75.76%
Eval — loss: 0.6824, acc: 76.61%
Epoch  8/10 — train loss: 0.5945, train acc: 79.04% | val acc: 76.61%
Eval — loss: 0.6367, acc: 77.79%
Epoch  9/10 — train loss: 0.5598, train acc: 80.26% | val acc: 77.79%
Eval — loss: 0.6917, acc: 76.65%
Epoch 10/10 — train loss: 0.5324, train acc: 81.37% | val acc: 76.65%
Best validation accuracy: 77.79%
Eval — loss: 0.6917, acc: 76.65%
```



MLP Confusion Matrix - Validation Set

Actual Class \ Predicted Class	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	597	39	46	22	45	6	30	7	162	46
automobile	27	647	25	29	12	8	20	12	93	127
bird	90	15	468	88	125	47	75	48	24	20
cat	35	19	80	417	85	137	122	44	28	33
deer	46	8	159	72	477	33	104	61	25	15
dog	17	8	79	251	87	371	70	58	36	23
frog	15	15	86	80	96	32	638	16	11	11
horse	38	12	76	75	110	54	23	561	13	38
ship	80	67	14	25	22	13	9	8	727	35
truck	45	183	19	33	21	11	26	28	114	520





Best validation accuracy achieved was 77.79% (at epoch 9).

The model showed strong generalization, with validation accuracy improving steadily from 62.87% to 77.79%.

The learning curves demonstrated consistent loss reduction and accuracy improvement on both training and validation sets.

4.5. Model Evaluation Procedures

We evaluated both models based on:

- Accuracy calculation on the validation set
- Learning curves for loss and accuracy over epochs
- Confusion matrices and per-class accuracy plots

Results:

- **MLP:**
  Achieved moderate performance with a best validation accuracy of 53.97%.
  It struggled particularly with visually similar classes such as cat vs dog and truck vs automobile, as shown in the confusion matrix.
  Per-class accuracy was lower for animals like cat, bird, and deer, indicating difficulty in distinguishing fine-grained visual features.

- **CNN:**
  Achieved significantly better results, with a best validation accuracy of 77.79%.
  The model demonstrated consistent learning improvement across epochs, as illustrated by the loss and accuracy curves.
  It clearly outperformed MLP, leveraging convolution and pooling layers to learn spatial features more effectively and distinguish between complex classes.

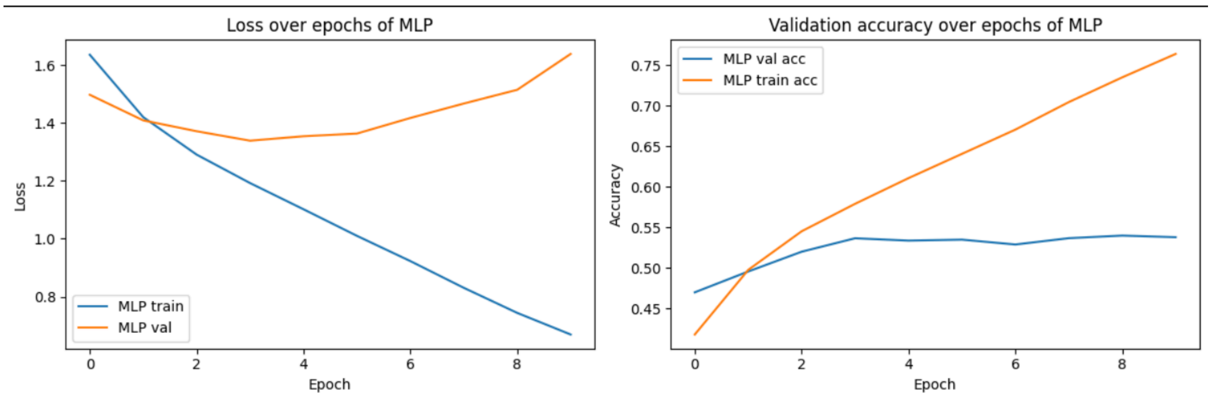
5. Discussion

Compare and discuss the results of the two neural networks

5.1. MLP Model

The MLP model was trained for 10 epochs.

- Its training accuracy started at 41.77% in epoch 1 and reached 76.38% by epoch 10.
- The validation accuracy began at 46.97% and achieved a best of 53.97% in epoch 9, finishing at 53.77% in epoch 10.
- The training loss showed a steady decrease from 1.6366 to 0.6690.
- However, the validation loss, after an initial decrease from 1.4982, started to plateau and then increase towards the end (e.g., 1.3394 in epoch 4 to 1.6394 in epoch 10), indicating overfitting as training progressed. This is clearly visible in the "Loss over epochs of MLP" plot where the validation loss diverges from the training loss.

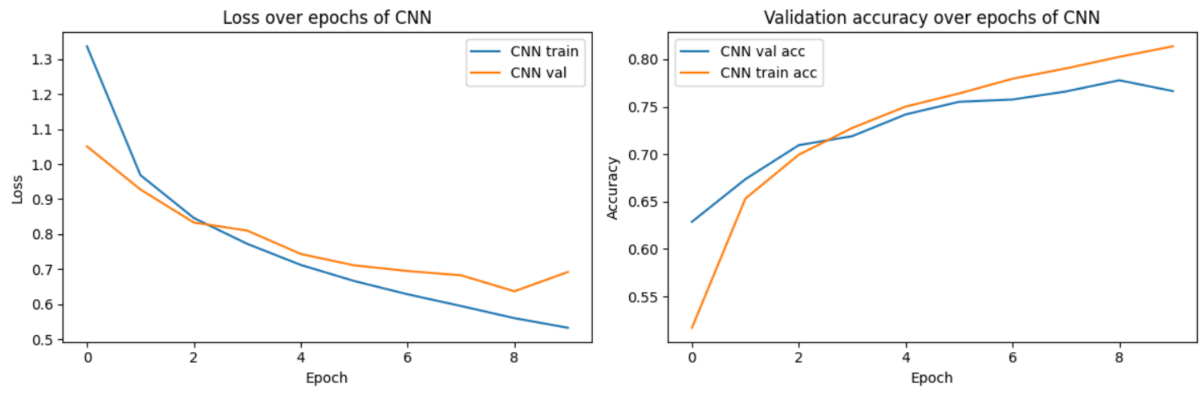


5.2. CNN Model

The CNN model was also trained for 10 epochs.

- It displayed a more effective learning pattern, with training accuracy starting at 51.69% and rising to 81.37%.
- Validation accuracy showed substantial improvement, beginning at 62.87% and reaching a best of 77.79% in epoch 9, before slightly decreasing to 76.65% in epoch 10.
- The training loss for the CNN model reduced significantly from 1.3371 to 0.5324.
- The validation loss also saw a more consistent decrease, from 1.0511 to a low of 0.6367 in epoch 9, ending at 0.6917, suggesting better generalization

compared to the MLP. The "Loss over epochs of CNN" plot shows the validation loss tracking the training loss more closely, with less divergence.



5.3. Error Analysis

MLP Confusion Matrix Insights

Looking at the MLP's confusion matrix:

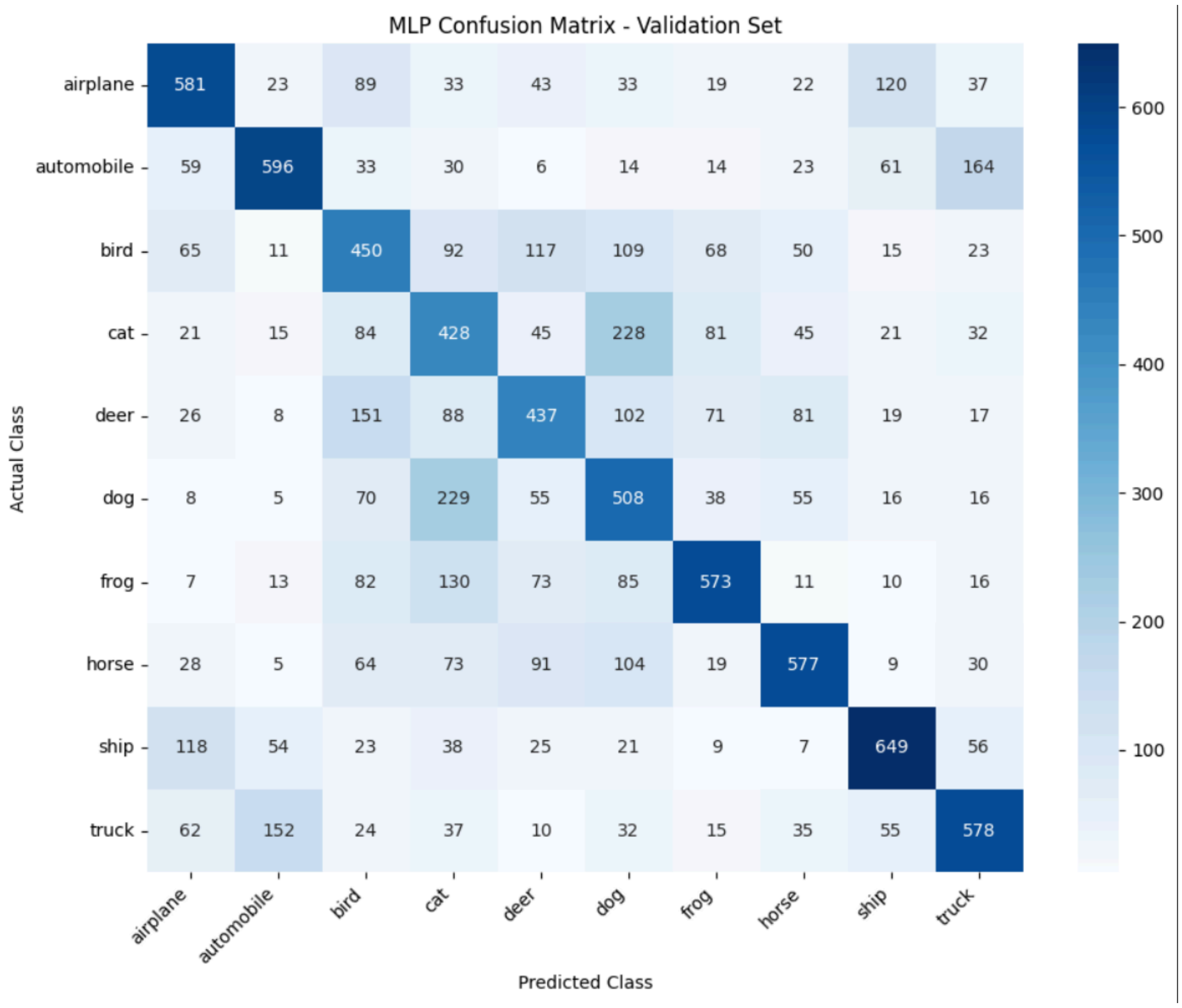
- **Overall Poor Performance**: The MLP misclassifies a significant number of samples across most classes. For example, only 450 out of 1000 "bird" images were correctly classified, and only 428 "cat" images were correct.
- **Common Error Patterns**:
    - **Confusion between visually similar classes**: The model frequently confuses "cat" with "dog" (229 misclassifications of cat as dog, and 228 misclassifications of dog as cat), "automobile" with "truck" (164 misclassifications of automobile as truck, and 152 misclassifications of truck as automobile), and "airplane" with "ship" (120 misclassifications of airplane as ship, and 118 misclassifications of ship as airplane).
    - **Contextual/Shape Confusion**: Classes like "bird" are confused with "deer" (151 times), "cat" (84 times), "dog" (70 times), and "frog" (82 times), likely due to similarities in general shape or common background elements rather than distinct object features. "Deer" is also heavily confused with "bird" (151 times) and "dog" (130 times).
    - **Inability to Capture Fine-Grained Details**: The MLP struggles to differentiate based on subtle but critical features that distinguish classes.

The errors observed in the MLP are characteristic of its architectural limitations when processing image data:

- **Loss of Spatial Information**: By flattening the input image (3x32x32) into a 1D vector (3072 features), the MLP discards the spatial relationships between pixels. Information about edges, corners, and textures, which is vital for object recognition, is lost.

- **Lack of Hierarchical Feature Learning**: The MLP, with its fully connected layers (input to 1024, then to 512, then to 10 classes with ReLU activations), tries to learn direct mappings from raw pixels to classes. It doesn't inherently learn simpler features (like edges) and then combine them into more complex ones (like object parts), a key strength of CNNs.
- **No Translation Invariance**: If an object is shifted in the image, the MLP perceives it as a largely new input, making it difficult to generalize.

In essence, the MLP relies on broad statistical correlations in the pixel data, leading to high confusion between classes with general visual or contextual similarities, and it fails to grasp the detailed features needed for accurate image classification.


MLP Confusion Matrix - Validation Set

Actual Class \ Predicted Class	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	581	23	89	33	43	33	19	22	120	37
automobile	59	596	33	30	6	14	14	23	61	164
bird	65	11	450	92	117	109	68	50	15	23
cat	21	15	84	428	45	228	81	45	21	32
deer	26	8	151	88	437	102	71	81	19	17
dog	8	5	70	229	55	508	38	55	16	16
frog	7	13	82	130	73	85	573	11	10	16
horse	28	5	64	73	91	104	19	577	9	30
ship	118	54	23	38	25	21	9	7	649	56
truck	62	152	24	37	10	32	15	35	55	578

CNN Confusion Matrix Insights

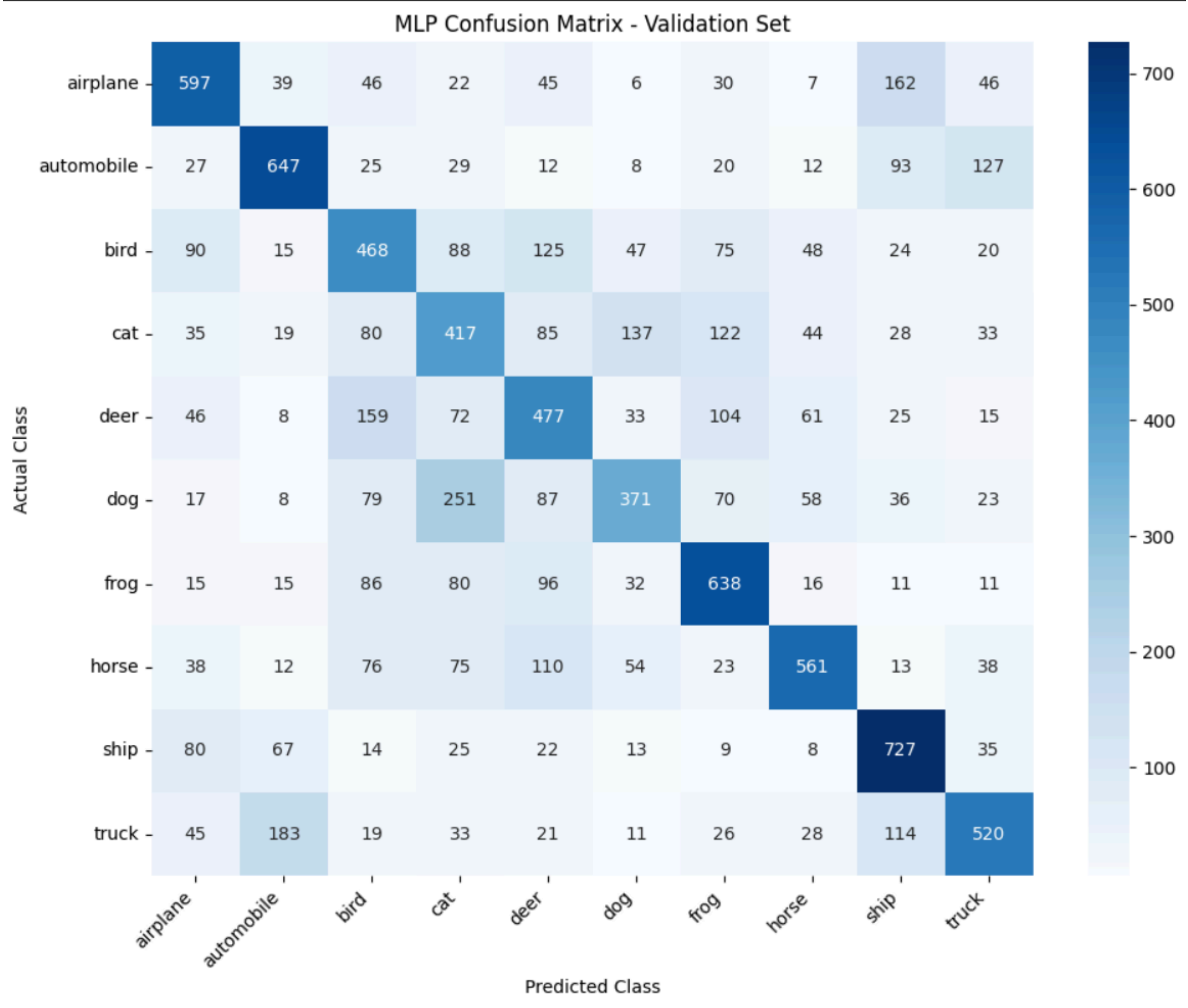
Examining the CNN's confusion matrix:

- **Superior Performance**: The CNN classifies images much more accurately than the MLP. For instance, "airplane" corrections went from 581 (MLP) to 597 (CNN), "automobile" from 596 to 647, and "ship" from 649 to 727.
- **Reduced but Persistent Error Patterns**:
    - **Visually Similar Classes**: Confusion between highly similar classes is reduced but still present. The most notable is "cat" and "dog," where the CNN misclassified 251 "dog" images as "cat" and 137 "cat" images as "dog". "Automobile" and "truck" also still show confusion (127 "automobile" as "truck", 183 "truck" as "automobile").
    - **Fine-Grained Distinctions**: While better, the CNN can still struggle with very subtle differences or high intra-class variation. For example, "bird" is still confused with "airplane" (90 times), "deer" (159 times), "cat" (80 times), and "frog" (86 times). The "dog" class had notably lower correct classifications (371) compared to the MLP (508), with many misclassified as "cat" (251). Similarly, "truck" correct classifications were 520 for CNN versus 578 for MLP.

The significant improvement of this SimpleCNN (with three convolutional blocks, batch normalization, and dropout) over the MLP stems from its architecture effectively addressing the MLP's weaknesses:

- **Preservation and Utilization of Spatial Information**: Convolutional layers (using 3x3 kernels in this CNN) process local pixel regions, thereby preserving spatial relationships. This allows the network to learn local features like edges and textures.
- **Hierarchical Feature Learning**: The stacking of convolutional layers enables the CNN to build a hierarchy of features—from simple edges to more complex patterns and object parts.
- **Translation Invariance (or Local Invariance)**: The use of shared weights in convolutional filters and max pooling layers (2x2 in this CNN) helps the model recognize features even if their position in the image varies slightly.

In summary, the CNN's architecture, which leverages spatial structure and hierarchical feature learning, makes it far more adept at image classification tasks than the MLP. The remaining errors are often concentrated in classes with very high visual similarity (like cat/dog) or those requiring discrimination of extremely fine-grained details, suggesting that while this basic CNN is effective, more complex architectures could yield further improvements.



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	597	39	46	22	45	6	30	7	162	46
automobile	27	647	25	29	12	8	20	12	93	127
bird	90	15	468	88	125	47	75	48	24	20
cat	35	19	80	417	85	137	122	44	28	33
deer	46	8	159	72	477	33	104	61	25	15
dog	17	8	79	251	87	371	70	58	36	23
frog	15	15	86	80	96	32	638	16	11	11
horse	38	12	76	75	110	54	23	561	13	38
ship	80	67	14	25	22	13	9	8	727	35
truck	45	183	19	33	21	11	26	28	114	520

5.4. Architectural Impact on Performance

The performance disparity between the MLP and CNN is primarily due to their differing architectural designs and how well they process image data.

MLP Limitations

The MLP flattens the 3x32x32 image into a 3072-feature vector, which discards crucial spatial relationships between pixels. This makes it difficult for the MLP to learn local patterns or understand the hierarchical structure of visual features that define objects. While its fully connected layers offer capacity, their lack of inductive bias for spatial data makes them less efficient for image tasks, often requiring more parameters for similar receptive fields and potentially leading to overfitting if not well-regularized. The MLP used here has a flatten layer followed by linear layers (3072 -> 1024 -> 512 -> 10) with ReLU activations. Its performance, with a best validation accuracy of 53.97%, indicates its struggle to build robust feature representations from the CIFAR-10 images.

CNN Strengths

CNNs are specifically designed for grid-like data such as images. The SimpleCNN used here consists of three convolutional blocks followed by a classifier.

- **Convolutional Layers**: These layers use learnable filters (3x3 kernels in this CNN) to scan the input, detecting local patterns and creating feature maps that maintain spatial information. Stacking these layers (three blocks in this model, with channels progressing from 3 to 32, then 64, then 128) allows the network to learn a hierarchy of features. Each convolutional layer is followed by BatchNorm2d and ReLU activation.
- **Parameter Sharing**: Filters are shared across spatial locations in a feature map, reducing the total parameters compared to an MLP and making learning more efficient and robust to feature translation.
- **Pooling Layers**: Max pooling layers (2x2 with stride 2) are used after the first two convolutional blocks in this CNN (after the 32-channel and 64-channel convolutions), reducing feature map dimensionality and providing some translation invariance. A max pooling layer is also used after the third convolutional block (128-channel) before flattening for the classifier.
- **Regularization**: Dropout layers (with a rate of 0.2) are included after the first two convolutional blocks, which helps in preventing overfitting.
- **Classifier**: After the feature extraction by convolutional and pooling layers, the output is flattened and passed through fully connected layers (128*4*4 -> 128 -> 10) with ReLU activation for the final classification.

The CNN's architecture is inherently better at learning the rich, spatially organized features in CIFAR-10 images, resulting in significantly higher classification accuracy (best validation accuracy of 77.79%) compared to the MLP.

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 final validation accuracy of **76.65%** with a corresponding validation loss of **0.6917** after 10 epochs (with a best validation accuracy of 77.79% achieved during training). In stark contrast, the MLP model's performance was considerably lower, yielding a final validation accuracy of **53.77%** and a validation loss of **1.6394** after 10 epochs (with a best validation accuracy of 53.97%). 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.

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!