POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY 1

PYTHON PROGRAMMING

ASSIGNMENT 2

IMAGE CLASSIFICATION

Instructor	Kim Ngoc Bach
Student	Nguyen Thi Nhung - B23DCVT322
	Pham Lan Anh - B23DCCE010
Class ID	D23CQCE04-B

Hanoi

TABLE OF CONTENTS

1. Introduction

1.1. Background and Motivation

We live in a world saturated with visual information. From the countless photographs shared on social media and the vast archives of digitized art and historical documents to the real-time video feeds from surveillance systems and autonomous vehicles, images and videos constitute a significant and rapidly growing portion of global data. The ability to automatically process, analyze, and understand this visual data is no longer a niche scientific pursuit but a critical technological capability with far-reaching implications across diverse sectors. Applications span from healthcare (medical image analysis for disease diagnosis), security (facial recognition, anomaly detection), retail (visual search, automated checkout), entertainment (content-based image retrieval, special effects), and autonomous systems (environmental perception for navigation) to scientific research itself (astronomical image analysis, microscopy). The sheer volume and complexity of this visual data necessitate the development of sophisticated computational tools that can extract meaningful information and perform tasks that traditionally require human visual acuity and cognitive interpretation. This project delves into one of the most fundamental of these tasks: image classification.

1.2. The Challenge of Image Classification

Image classification is the task of assigning a predefined categorical label to an input image based on its visual content. For humans, this is often an intuitive and effortless process. We can readily distinguish a "cat" from a "dog" or a "car" from an "airplane" despite variations in viewpoint, lighting, scale, occlusion, and intra-class diversity. However, for a computer, an image is merely a grid of pixel values – numerical representations of color and intensity. The core challenge of image classification lies in bridging this "semantic gap" between the low-level pixel data and high-level conceptual understanding. An effective image classification system must learn to identify salient visual patterns, features, and relationships within the pixel data that are discriminative of the different categories, while also being robust to the aforementioned variations. This involves learning complex, hierarchical representations of visual information.

1.3. Evolution of Image Classification

Historically, approaches to image classification relied heavily on "handcrafted" features. This involved domain experts designing explicit algorithms to extract features deemed relevant for a particular task, such as SIFT (Scale-Invariant Feature Transform), SURF (Speeded Up Robust Features), or HOG (Histogram of Oriented Gradients). These features would then be fed into traditional machine learning classifiers like Support Vector Machines (SVMs) or Random Forests. While these methods achieved notable successes, they had significant limitations: feature engineering was often a laborious, task-specific process, and the resulting features might not always capture the optimal representations for discrimination. The advent of deep learning, particularly Convolutional Neural Networks (CNNs), has revolutionized the field of computer vision and image classification. Deep learning models, inspired by the hierarchical structure of the human visual cortex, can automatically learn relevant features directly from the raw pixel data. This "end-to-end" learning approach, where feature extraction and classification are integrated into a single, optimizable system, has led to state-of-the-art performance on a wide range of visual tasks, often surpassing human capabilities in specific domains. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) played a pivotal role in demonstrating the power of CNNs, with models like AlexNet, VGGNet, GoogLeNet, and ResNet progressively pushing the boundaries of accuracy.

1.4. Neural Networks for Image Classification

This report focuses on implementing and comparing two foundational types of neural networks for image classification: Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs), using the PyTorch deep learning framework, based on two separate Python scripts: MLP.py and CNN.py.

a. Multi-Layer Perceptrons (MLPs)

MLPs, also known as feedforward neural networks, are a fundamental class of neural networks. They consist of an input layer, one or more hidden layers composed of interconnected neurons, and an output layer. Each neuron performs a weighted sum of its inputs followed by a non-linear activation function. MLPs are universal approximators, meaning they can, in theory, approximate any continuous function given enough hidden units. For image classification, the raw image is typically flattened into a 1D vector before being fed into the MLP. While versatile, standard MLPs do not inherently account for the spatial structure of images (e.g., the fact that nearby pixels are more related than distant ones), which can be a significant limitation for visual tasks. The MLP.py script explores this architecture.

b. Convolutional Neural Networks (CNNs)

CNNs are a specialized type of neural network designed explicitly to process grid-like data, such as images. Their architecture is inspired by the organization of the animal visual cortex and incorporates several key concepts that make them highly effective for visual tasks:

- **Local Receptive Fields:** Neurons in early layers connect only to small, localized regions of the input image, allowing them to detect local patterns like edges or corners.

- **Parameter Sharing:** The same set of weights (a filter or kernel) is applied across different spatial locations in the input, enabling the network to detect a specific feature regardless of its position (translation invariance) and significantly reducing the number of parameters to learn.

- **Hierarchical Feature Extraction:** CNNs typically consist of a sequence of layers, including convolutional layers, activation layers (e.g., ReLU), and pooling layers. Convolutional layers apply filters to extract features, ReLU introduces non-linearity, and pooling layers downsample the feature maps, making the representations more compact and robust. By stacking these layers, CNNs learn a hierarchy of features, from simple low-level patterns in early layers to more complex, abstract features (like object parts or entire objects) in deeper layers. The CNN.py script implements such an architecture, incorporating techniques like Batch Normalization and Dropout.

Given their architectural advantages for visual data, CNNs are generally expected to outperform MLPs on image classification tasks.

1.5. The CIFAR-19 Dataset

To empirically evaluate and compare the MLP and CNN models, this project utilizes the CIFAR-10 dataset. "CIFAR" stands for the Canadian Institute for Advanced Research. The CIFAR-10 dataset is a widely used and well-established benchmark in the machine learning and computer vision communities for training and testing image classification algorithms. It consists of 60,000 32x32 pixel color images, categorized into 10 mutually exclusive classes: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', and 'truck'. There are 6,000 images per class. The dataset is typically split into a training set of 50,000 images and a test set of 10,000 images. Its relatively small image size and manageable number of classes make it suitable for experimenting with different model architectures and training procedures without requiring excessive computational resources, while still posing a non-trivial classification challenge. Both MLP.py and CNN.py use this dataset; MLP.py further splits the official training set to create a training and a validation set, while CNN.py uses the official training set for training and the official test set for obtaining validation metrics during its training epochs.

2. Building a Basic MLP Neural Network

This section elaborates on the design, underlying principles, and implementation of the Multi-Layer Perceptron (MLP) model as specified in the MLP.py script for classifying CIFAR-10 images.

2.1. Libraries Used and Rationale

The MLP.py script utilizes several Python libraries:

- **torch and torch.nn:** For building the neural network. nn.Module is the base class, nn.Linear defines fully connected layers.
- **torch.nn.functional as F:** Used for applying activation functions like F.relu directly in the forward method.
- **torchvision.datasets and torchvision.transforms:** For loading the CIFAR-10 dataset and applying transformations (specifically transforms.ToTensor()).
- **torch.utils.data.DataLoader and torch.utils.data.random_split:** DataLoader for batching and shuffling data. random_split is used in this script to create a validation set from the training data.
- **matplotlib.pyplot as plt:** For plotting learning curves and sample image predictions.
- **sklearn.metrics.confusion_matrix:** For calculating the confusion matrix.

- **numpy as np:** For numerical operations, often in conjunction with Matplotlib or Scikit-learn.
- **time:** For timing the training process.
- **seaborn as sns:** For creating more visually appealing heatmaps for the confusion matrix.

The choice of these libraries is standard for deep learning tasks with PyTorch, providing a comprehensive toolkit for model building, data handling, training, evaluation, and visualization.

2.2 Key Concepts

a. Fundamental Architecture

The MLP in MLP.py is structured with an input layer, two hidden layers, and an output layer:

- **Input Layer:** Conceptually, this layer takes the flattened image data. CIFAR-10 images (3×32×32=3072 features) are flattened within the training loop before being passed to the model.
- **Hidden Layers:**
  - The first hidden layer is defined by self.fc1 = nn.Linear(input_size, 120), transforming the 3072 input features to 120 features.
  - The second hidden layer is defined by self.fc2 = nn.Linear(120, 84), transforming 120 features to 84 features. This creates a bottleneck architecture (3072 -> 120 -> 84).
- **Output Layer:** self.fc3 = nn.Linear(84, output_size) maps the 84 features to 10 output logits, one for each CIFAR-10 class (output_size=10).

This is a 3-layer network in terms of its learnable linear transformation layers.

b. The Neuron

At the core of each neuron (or node) within a fully connected layer (like nn.Linear) is a two-step computation process that transforms the inputs it receives from the previous layer:

- **Linear Transformation (Weighted Sum and Bias):** Each input connection to the neuron has an associated weight. The neuron calculates a weighted sum of all its inputs. If the inputs to a neuron are x1,x2,...,xn and their corresponding weights are w1,w2,...,wn, the weighted sum is $\sum_{i=1}^{n} w_i x_i$. Additionally, a bias term, b, is typically added to this sum. The bias allows the neuron to shift its activation function horizontally, providing more flexibility in the learning process. Thus, the full linear transformation for a single neuron is $z = (\sum_{i=1}^{n} w_i x_i)$+b. For an entire layer of neurons processing a batch of input vectors, this can be expressed in matrix notation as $Z = XW + B$, where X is the matrix of inputs, W is the matrix of weights, and B is the bias vector (broadcasted appropriately).

- **Activation Function:** The result of this linear transformation, z (often called the "logit" or "pre-activation"), is then passed through a non-linear activation function, $a = f(z)$, to produce the neuron's final output or "activation." This non-linearity is crucial for the network's ability to learn complex patterns that go beyond simple linear relationships.

The weights (W) and biases (B) associated with each neuron are the parameters of the model. These parameters are initialized (often randomly or using specific schemes) before training and are then iteratively adjusted during the training process (via back-propagation and an optimization algorithm) to minimize the difference between the network's predictions and the actual target values.

c. Activation Functions

The MLP.py script uses F.relu (Rectified Linear Unit) as the activation function after the first two linear layers (fc1 and fc2).
ReLU $(f(x) = max(0, x))$: Introduces non-linearity, allowing the model to learn complex mappings. Its advantages include computational efficiency and mitigating vanishing gradients for positive inputs.

d. Data Flattening

The MLP.py script performs flattening explicitly within the training and evaluation loops using X_train_flat = X_train.view(X_train.shape[0], -1). This reshapes the [batch_size, 3, 32, 32] image tensor into [batch_size, 3072] before feeding it to model().

e. Limitations of MLPs for Image Data

While Multi-Layer Perceptrons (MLPs) are versatile function approximators, they possess inherent limitations when directly applied to raw image data, primarily due to their structural assumptions:

- **Loss of Spatial Information:** The most significant drawback is that MLPs require the input image to be flattened into a 1D vector. For an image of size W×H with C channels (e.g., 32×32×3 for CIFAR-10), this results in a long vector of W×H×C features. This flattening process discards the crucial 2D (or 3D, including channels) spatial structure of the image. Pixels that are close together in the image (and thus likely related, forming edges, textures, or parts of objects) are treated independently by the initial layer after flattening, just like pixels that are far apart. The network loses the information about the relative positions of pixels and local correlations, which are vital for visual pattern recognition.
    - **High Dimensionality and Parameter Proliferation:** Even for small images like those in CIFAR-10 (3072 features), the input dimensionality is high. If the first hidden layer of an MLP is fully connected to this flattened input, it leads to a very large number of parameters (weights and biases). For instance, if the first hidden layer has H1 neurons, there will be (W×H×C)×H1 weights just for this layer. This high unseen data.
    - **Computationally Expensive:** Training and inference with a large number of parameters require more memory and computational power.

- **Lack of Translation Invariance:** MLPs are not inherently translation invariant. If an object appears in the top-left corner of one training image and the bottom-right corner of another, an MLP treats these as entirely different input patterns. It needs to learn to recognize the object (and its features) at every possible position independently. This is highly inefficient and requires a vast amount of training data to cover all spatial variations. In contrast, features useful for identifying an object (like a specific texture or edge) should be detectable regardless of where they appear in the image.
- **Sensitivity to Input Permutations:** Because an MLP treats each input feature (flattened pixel) independently in its connections to the first hidden layer, if the order of pixels in the flattened vector were permuted, the learned weights would no longer be meaningful, and the model would perform poorly. While images are not typically permuted, this highlights the lack of structural understanding.

These limitations mean that while an MLP can learn to classify images, it often requires more data, more parameters (leading to overfitting risk), and achieves lower accuracy compared to architectures like CNNs that are specifically designed to exploit the spatial nature of image data.

2.3. Implementation Steps and Detailed Rationale

a. Class Definition

- The constructor __init__ defines three linear layers (fc1, fc2, fc3) with neuron counts 3072 (implicit input) -> 120 -> 84 -> 10.
- The forward method applies these layers sequentially, with F.relu activations after fc1 and fc2. The output layer fc3 does not have a subsequent activation because nn.CrossEntropyLoss expects raw logits.

b. Data Handling Specifics

- **Transform:** Only transforms.ToTensor() is used for MLP data.
- **Validation Split:** A validation set (cifar10_val) is created from the full training set using random_split with a val_split_ratio = 0.2.
- **DataLoaders:** Separate DataLoader instances are created for cifar10_train, cifar10_val, and cifar10_test with batch sizes batch_size_train = 100, batch_size_val = 500, batch_size_test = 500. shuffle=True is used only for train_loader.
- **Seed:** torch.manual_seed(80) is set for reproducibility.

c. Model Initialization, Loss, Optimize

- model = MultilayerPerceptron(): An instance is created.
- The total number of parameters is calculated and printed.
- Criterion = nn.CrossEntropyLoss(): Standard loss for multi-class classification.
- optimizer = torch.optim.SGD(model.parameters(), lr=0.01): Stochastic Gradient Descent is used with a learning rate of 0.01.

d. Training Loop

- epochs = 10.
- The loop iterates for the specified number of epochs.
- Training Phase:
  - model.train() sets the model to training mode.
  - Iterates through train_loader.
  - Flattening: X_train_flat = X_train.view(X_train.shape[0], -1) is performed here.
  - Standard steps: optimizer.zero_grad(), forward pass (y_pred = model(X_train_flat)), loss calculation, loss.backward(), optimizer.step().
  - Training loss and accuracy are calculated (accuracy as correct_train_predictions / total_train_samples) and stored per epoch.
- Validation Phase (after each training epoch):
  - model.eval() sets the model to evaluation mode.
  - with torch.no_grad(): disables gradient computation.
  - Iterates through val_loader.
  - Flattening is also performed for validation data.
  - Validation loss and accuracy are calculated and stored.
- Prints epoch-wise training and validation metrics.
- Training time is recorded and printed.

e. Plotting Learning Curves

- Uses matplotlib.pyplot to create two subplots for loss and accuracy curves, plotting both training and validation metrics.
- Plots are displayed using plt.show().

f. Final Evaluation on Test Set

- model.eval() and with torch.no_grad().
- Iterates through test_loader, flattens data, makes predictions.
- Calculates and prints final_test_accuracy.

g. Confusion Matrix

- Collects all predictions and true labels from the test_loader.
- Computes cm = confusion_matrix(all_labels, all_preds).
- Uses seaborn.heatmap to plot the confusion matrix with class labels. The plot is displayed using plt.show().

h. Visualization of Sample Predictions

Loads a batch from test_loader, makes predictions, and displays 64 sample images with their true and predicted labels, color-coding correct/incorrect predictions.

3. Building a Convolutional Neural Network (CNN)

This section details the Convolutional Neural Network (CNN) architecture and implementation as found in the CNN.py script, designed for the CIFAR-10 image classification task. This version incorporates Batch Normalization and Dropout.

3.1. Libraries Used and Rationale

The CNN.py script uses a similar set of core libraries as MLP.py but with a focus on CNN-specific components:

- **torch, torch.nn, torchvision.datasets, torchvision.transforms, torch.utils.data.DataLoade torch.optim:** Core PyTorch libraries for model definition, data handling, and optimization.
- **matplotlib.pyplot as plt:** For plotting learning curves and confusion matrices.
- **sklearn.metrics.confusion_matrix:** For computing the confusion matrix.
- **seaborn as sns:** For enhanced visualization of the confusion matrix.
- **numpy as np:** Implicitly used by other libraries.
- **os:** Imported, potentially for num_workers logic based on OS, though the provided snippet in the Appendix shows num_workers_global = 0 if os.name == 'nt' else 2.

Key torch.nn components specific to this CNN implementation:

- **nn.Conv2d:** For 2D convolutional layers.
- **nn.BatchNorm2d(num_features):** Applies Batch Normalization to the output of convolutional layers.
- **nn.ReLU():** Activation function.
- **nn.MaxPool2d:** For max pooling.
- **nn.Flatten():** To flatten feature maps before fully connected layers.
- **nn.Linear:** For fully connected layers.
- **nn.Dropout(p):** A regularization technique where neurons are randomly zeroed out during training.

3.2. Key Concepts

a. Convolutional Layers

Convolutional layers are the defining elements of CNNs. They operate on input volumes (e.g., an image or the feature map from a previous layer) and transform them into output volumes.

- **Filters (Kernels) and Feature Maps**
  A convolutional layer consists of a set of learnable filters (also called kernels). Each filter is a small 2D matrix of weights (e.g., 3x3, 5x5). During the forward pass, each filter is convolved (slid) across the width and height of the input volume, computing the dot product between the filter's weights and the spatially corresponding input region at each position. This process produces a 2D activation map or feature map for that filter. Each feature map indicates the locations in the input where the specific feature detected by the filter is present. For example, one filter might learn to detect vertical edges, another horizontal edges, another a specific color pattern, etc. The out_channels parameter of nn.Conv2d specifies how many such filters (and thus output feature maps) the layer will have. The depth of the output volume is equal to the number of filters.

- **Stride and Padding**

  - **Stride:** Controls how many pixels the filter slides over at each step. A stride of 1 means the filter moves one pixel at a time. A stride of 2 means it moves two pixels at a time, resulting in a smaller output feature map.
  - **Padding:** Refers to adding extra pixels (usually zeros) around the border of the input volume before the convolution. This is useful for two main reasons: It allows the filter to be centered on border pixels, ensuring they are processed as thoroughly as interior pixels. It can be used to control the spatial size of the output feature map. For instance, with a 3x3 kernel and stride 1, using padding=1 (one layer of zeros around the input) will result in an output feature map with the same height and width as the input. This is often called "same" padding. Without padding, the feature map would shrink after each convolution.

- **Parameter Sharing and Local Connectivity**
  Two key properties make convolutional layers efficient and effective for images:

  - **Local Connectivity:** Each neuron in a feature map is connected only to a small, local region of the input volume (the receptive field of that neuron, determined by the kernel size). This exploits the spatial locality of image features (pixels close to each other are more likely to be related).
  - **Parameter Sharing:** The same filter (set of weights) is used to compute all the values in a given feature map. This means that a feature detector (e.g., an edge detector) learned at one part of the image can be reused to detect the same feature at other parts. This dramatically reduces the number of parameters compared to a fully connected layer and makes the network somewhat invariant to the translation of features.

The CNN.py script uses three convolutional layers, each with 3x3 kernels and padding of 1.

b. Batch Normalization

Included in CNN.py after each convolutional layer and before the ReLU activation.

- **Purpose:** Addresses internal covariate shift, where the distribution of each layer's inputs changes during training as the parameters of the previous layers change.
- **How it Works:** Normalizes the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. It then scales and shifts the normalized output using two learnable parameters (gamma and beta) per feature map.
- **Benefits:** Faster training (allows higher learning rates), stabilizes training, acts as a slight regularizer. During evaluation (model.eval()), it uses learned running statistics.

c. Pooling Layers

Pooling layers, such as nn.MaxPool2d used in CNN.py, are typically inserted between successive convolutional layers.

- **Purpose:**
    - ○ **Dimensionality Reduction:** They progressively reduce the spatial dimensions (height and width) of the feature maps. This decreases the number of parameters and computations in subsequent layers, which helps to control overfitting and makes the network more computationally manageable.
    - ○ **Translation Invariance (Local):** By summarizing a local neighborhood of features (e.g., taking the maximum value in max pooling), pooling layers make the representation slightly more robust to small translations or distortions of the features in the input. If a feature shifts slightly within the pooling window, the pooled output is likely to remain the same or very similar.
- **Max Pooling:** Specifically, max pooling selects the maximum value from a patch of features in the input feature map. This tends to retain the most prominent features within that patch. CNN.py uses MaxPool2d(kernel_size=2, stride=2) after each (Conv -> BatchNorm -> ReLU) block, which halves the height and width of the feature maps at each step.

d. Activation Functions in CNNs

nn.ReLU is used after Batch Normalization in each convolutional block and after the first fully connected layer in CNN.py. Its properties (non-linearity, computational efficiency, mitigation of vanishing gradients) are as beneficial in CNNs as they are in MLPs.

e. Dropout Regularization

CNN.py includes a dropout layer (self.dropout_fc = nn.Dropout(dropout_rate)) after the first fully connected layer's activation.

- **Purpose:** To prevent overfitting by randomly dropping units during training.
- **How it Works:** During training, for each forward pass, a fraction (dropout_rate) of the neurons in the layer preceding dropout are randomly "dropped" or temporarily deactivated (their outputs are set to zero). This means these neurons do not contribute to the forward pass and do not participate in backpropagation for that training step. This prevents complex co-adaptations where neurons become overly reliant on specific other neurons. Instead, each neuron must learn more robust features that are useful in conjunction with different random subsets of other neurons.
- **During Evaluation (model.eval()):** Dropout is automatically disabled. To compensate for the fact that more neurons are active during evaluation than during training (where some were dropped), the outputs of the layer are typically scaled down by a factor equal to the dropout rate (or, equivalently, the activations during training are scaled up). PyTorch's nn.Dropout handles this scaling automatically. The dropout_rate is a configurable parameter.

f. Fully Connected Layers

After the series of convolutional and pooling layers have extracted and downsampled spatial features, the resulting high-dimensional feature maps are flattened into a 1D vector. This vector is then fed into one or more fully connected layers (nn.Linear). These layers function similarly to those in an MLP, performing linear transformations followed by activations. Their role in a CNN is to take the high-level, spatially abstracted features learned by the convolutional part and perform the final classification by learning non-linear combinations of these features. The last fully connected layer typically has a number of neurons equal to the number of classes and produces the raw logits for classification.

g. Building a Feature Hierarchy and Understanding Receptive Fields

A key characteristic and strength of CNNs is their ability to automatically learn a hierarchy of features from the input data.

- **Early Layers (closer to the input):** Filters in these layers have small receptive fields (the region in the original input image that a particular filter or neuron "sees" or is affected by). These layers typically learn to detect simple, low-level features such as edges, corners, color blobs, and basic textures. For example, one filter might become a horizontal edge detector, another a detector for a specific color.
- **Intermediate Layers:** Filters in these layers combine the simple features detected by earlier layers to learn more complex patterns or motifs, such as parts of objects (e.g., an eye, a wheel, a patch of fur). Their receptive fields are larger than those of earlier layers because they are looking at feature maps that are already abstractions of the input.

- **Deeper Layers (further from the input, closer to the output):** These layers combine the mid-level features to detect even more complex and abstract concepts, potentially corresponding to entire objects or significant portions of them. Their receptive fields are the largest, encompassing a substantial part of the original input image.

This hierarchical structure, where features build upon each other in increasing complexity and abstraction, mimics aspects of the human visual cortex and allows CNNs to learn powerful and discriminative representations for visual tasks. The depth of the network (number of layers) often correlates with the complexity of features it can learn.

3.3. Implementation Steps and Detailed Rationale

a. Class Definition

- The constructor defines three convolutional blocks (Conv2d -> BatchNorm2d -> ReLU -> MaxPool2d) with increasing filter counts (32, 64, 128).
- The fully connected part includes a Dropout layer. dropout_rate is configurable.
- The forward method specifies the data flow: Conv -> BN -> ReLU -> Pool for each block, then Flatten -> FC -> ReLU -> Dropout -> Output FC.

b. Data Handling Specifics

- **Data Augmentation:** transform_train includes RandomCrop(32, padding=4) and RandomHorizontalFlip().
- transform_test includes ToTensor() and Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)).
- **Validation Set:** validationset is defined using torchvision.datasets.CIFAR10(root='./data', train=False, ... transform=transform_test), meaning the official test set is used for validation metrics during training.
- DataLoaders are created for trainloader, valloader, and testloader, all with batch_size_global = 64.

c. Model Initialization, Loss, Optimizer

- cnn_model = CNN(dropout_rate=dropout_rate_global).to(device)
- criterion_cnn = nn.CrossEntropyLoss().
- optimizer_cnn = optim.Adam(cnn_model.parameters(), lr=learning_rate_global, weight_decay=weight_decay_global): Adam optimizer with lr=0.001 and weight_decay=1e-4 (L2 regularization).

d. Training, Testing, Plotting Functions

The train_model, test_model, plot_learning_curves, and plot_confusion_matrix functions are defined in CNN.py and are structurally similar to previous versions, performing their respective tasks. They display plots using plt.show().

e. Main Execution Block

- Sets hyperparameters: num_epochs_global = 25, dropout_rate_global = 0.3, etc.
- Initializes and trains only the CNN model.
- Calls test_model using testloader.
- Calls plot_learning_curves and plot_confusion_matrix for the CNN.

4. Performing Image Classification

This section synthesizes the image classification workflow based on the methodologies presented in the separate MLP.py and CNN.py scripts. While both aim to classify CIFAR-10 images, they employ different strategies in data handling, model architecture, and training regimes.

4.1. Libraries Used and Rationale

Both scripts rely on a common set of libraries:

- **torch, torch.nn, torch.optim:** For model building, defining loss functions, and optimization.
- **torchvision.datasets, torchvision.transforms:** For loading CIFAR-10 and image preprocessing.
- **torch.utils.data.DataLoader:** For creating data iterators.
- **matplotlib.pyplot:** For plotting.
- **sklearn.metrics.confusion_matrix:** For evaluation.
- **seaborn:** For enhanced visualizations.
- **numpy:** For numerical operations.

Specific library uses:

- MLP.py additionally uses torch.utils.data.random_split for creating a validation set and torch.nn.functional for F.relu. It also uses the time module.
- CNN.py makes use of nn.BatchNorm2d and nn.Dropout from torch.nn, and os for platform-dependent num_workers setting.

The rationale remains consistent: leveraging PyTorch's ecosystem for deep learning and standard Python libraries for data science and visualization.

4.2. Key Concepts

a. Data Preparation

- Loading CIFAR-10: Both scripts load the CIFAR-10 dataset using torchvision.datasets.CIFAR10
- Transformations:
  - MLP.py: Applies transforms.ToTensor().
  - CNN.py: For training data (transform_train): Implements data augmentation (RandomCrop, RandomHorizontalFlip), followed by ToTensor() and Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)). For test/validation data (transform_test): Applies ToTensor() and Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)).
- **Validation Set Handling:**
  - MLP.py: Creates a validation set by splitting the cifar10_train_full dataset using random_split (val_split_ratio = 0.2).
  - CNN.py: Defines validationset by loading torchvision.datasets.CIFAR10(..., train=False, ...), thus using the official CIFAR-10 test set for generating validation metrics during training epochs.

b. DataLoaders

- Both scripts use DataLoader.
- **Batch Sizes:**
  - MLP.py: batch_size_train = 100, batch_size_val = 500, batch_size_test = 500.
  - CNN.py: batch_size_global = 64 for all loaders.
- **Shuffling:** shuffle=True for training loaders in both.
- **num_workers:** MLP.py uses default (0). CNN.py uses num_workers_global (e.g., 2, or 0 for Windows).

c. The Training Process

- **Epochs:** MLP.py uses 10; CNN.py uses 25.
- **Model Modes:** Both correctly use model.train() and model.eval().
- **Loss Functions:** Quantifying Prediction Error Both use nn.CrossEntropyLoss().
- **Optimizers:** Navigating the Loss Landscape (SGD vs. Adam with Weight Decay)
  - MLP.py: torch.optim.SGD(lr=0.01).
  - CNN.py: optim.Adam(lr=0.001, weight_decay=1e-4).
- **Backpropagation:** The Engine of Learning

Standard backpropagation (optimizer.zero_grad(), loss.backward(), optimizer.step()) is used in both.

d. Validation

- MLP.py: Validates on its val_loader (split from train) after each training epoch.
- CNN.py: Validates on its valloader (the test set) after each training epoch within train_model.

Both use torch.no_grad()

e. Testing

- MLP.py: Evaluates on test_loader after all training epochs.
- CNN.py: Calls test_model on testloader after all training epochs.

4.3. Implementation Details

a. Data Handling Specifics

- **Validation Set Source:** The source for the validation data differs: MLP.py uses a subset of the original training data, while CNN.py uses the official test data for its epoch-wise validation metrics.
- **Data Augmentation:** This technique is applied only in CNN.py's training pipeline.
- **Input Normalization Range:** The input data is scaled differently: MLP.py uses ToTensor (resulting in a [0,1] range), whereas CNN.py applies an additional Normalize transform (resulting in a [-1,1] range).

b. Training Loop Variations

- **Flattening in MLP:** In MLP.py, image data is flattened using .view() within the training, validation, and test loops, rather than as a distinct layer in the model definition.
- **Loss/Accuracy Accumulation & Averaging:** The scripts show minor variations in calculating the average loss per epoch. Accuracy display during epoch printouts also differs (fraction in MLP.py vs. percentage in CNN.py).

c. Hyperparameter Configurations

The scripts utilize different configurations for several key hyperparameters, including the number of training epochs, optimizer types and their specific parameters (learning rate, weight decay), batch sizes, and the application of techniques like Batch Normalization and Dropout (present in CNN.py but not MLP.py).

5. Plotting Learning Curves

This section details how learning curves are generated and visualized in both MLP.py and CNN.py to monitor the training process.

5.1. Libraries Used and Rationale

- **matplotlib.pyplot as plt:** Used in both scripts for creating the plots.
- **numpy as np:** Implicitly used.

Rationale: Standard and flexible Python plotting library.

5.2. Key Concepts

Learning curves are plots that show a model's learning performance over time, typically measured in epochs. They usually display two curves for each metric (e.g., loss, accuracy): one for the training set and one for a separate validation set.

a. Interpreting Loss and Accuracy Curves

- **Loss Curves:**
  - **Training Loss:** This curve shows the value of the loss function calculated on the training data at the end of each epoch (or sometimes per batch). Ideally, the training loss should decrease steadily as the model learns the patterns in the training data. A flat or increasing training loss might indicate problems such as an inappropriate learning rate (too high or too low), a model that is too simple for the data, or issues with the data itself.
  - **Validation Loss:** This curve shows the loss function calculated on the validation data (data not used for training the model's parameters) at the end of each epoch. The validation loss is a crucial indicator of how well the model is generalizing to new, unseen data. Ideally, it should also decrease and then stabilize.
- **Accuracy Curves:**
  - **Training Accuracy:** This shows the classification accuracy (e.g., percentage of correctly classified samples) on the training data at each epoch. It should generally increase as the model learns.
  - **Validation Accuracy:** This shows the accuracy on the validation data. It indicates the model's ability to generalize. An increasing validation accuracy is desirable.

The relationship and trends of these four curves (training loss, validation loss, training accuracy, validation accuracy) provide deep insights into the training process.

b. Identifying Overfitting, Underfitting, and Good Fit

Learning curves are powerful diagnostic tools for identifying common training problems:

- **Underfitting (High Bias):**

- *Symptoms:* Both training loss and validation loss remain high or plateau at a high value. Similarly, both training accuracy and validation accuracy are low and do not improve significantly. The training and validation curves are typically close together but at an unsatisfactory performance level.
- *Meaning:* The model is too simple or lacks the capacity to learn the underlying patterns in the data. It performs poorly on both the data it was trained on and new data.
- *Possible Solutions:* Increase model complexity (e.g., add more layers, more neurons/filters), train for more epochs (if not yet converged), choose a more complex model architecture, or improve feature engineering (though less common in end-to-end deep learning).

- **Overfitting (High Variance):**
  - *Symptoms:* Training loss continues to decrease (or training accuracy continues to increase to a very high level, near 100%), while validation loss starts to increase after a certain point (or validation accuracy stagnates or decreases). A significant and growing gap appears between the training and validation curves (training metric better than validation metric).
  - *Meaning:* The model has learned the training data too well, including its noise and specific idiosyncrasies, to the detriment of its ability to generalize to new, unseen data.
  - *Possible Solutions:* Obtain more diverse training data, apply data augmentation, use regularization techniques (e.g., L1/L2 weight decay, Dropout, Batch Normalization), reduce model complexity, or use early stopping (stopping training when validation performance starts to degrade).

- **Good Fit:**
  - *Symptoms:* Both training and validation loss decrease to a low point and then stabilize. Both training and validation accuracy increase to a satisfactory high point and then stabilize. The gap between the training and validation curves is small and stable.
  - *Meaning:* The model has learned the general patterns in the data and generalizes well to unseen data. This is the desired outcome.

c. Using Learning Curves for Model Improvement

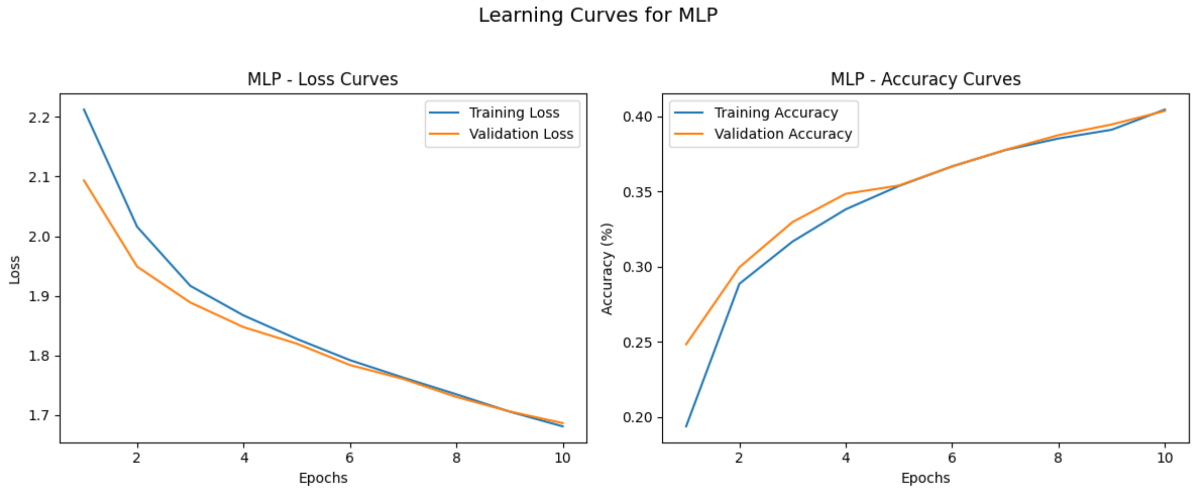
By carefully observing the shapes and trends of the learning curves, practitioners can make informed decisions to improve their models. For example:

- If underfitting is evident, the model needs more capacity or more training.
- If overfitting is the issue, regularization or more data is needed.
- If learning is very slow or unstable (highly fluctuating curves), adjustments to the learning rate, optimizer, or batch size might be necessary.
- The point where validation loss starts to consistently increase can be an indicator for early stopping.

5.3. Implementation Steps

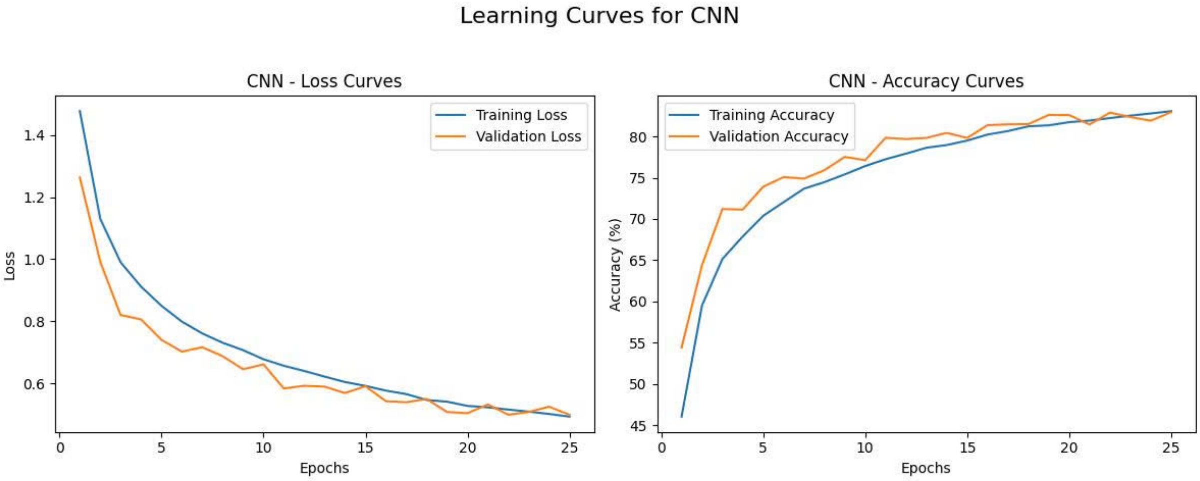
a. For MLP.py:

- **Data Collection:** train_losses, val_losses, train_accuracies, val_accuracies lists are populated after each epoch.
- **Plotting Logic (Directly in script):**
    - epochs_plot_range = range(1, len(train_losses) + 1).
    - fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5)): Creates two subplots.
    - **Loss Plot (ax1)**: Plots training and validation loss.
    - **Accuracy Plot (ax2):** Plots training and validation accuracy.
    - Sets titles, labels, legends for both subplots and a figure super title.
    - plt.show(): Displays the plot.



b. For CNN.py

- **Data Collection:** train_model returns the four lists of losses and accuracies.
- **Function Definition (plot_learning_curves):**
    - Takes losses, accuracies, and model_name.
    - Creates two subplots for loss and accuracy.
    - Plots training and validation curves similarly to MLP.py.
    - plt.show(): Displays the plot.
- **Function Call:** Called after training the CNN.
- **Plot Display:** Both scripts display the generated learning curves using plt.show().



6. Plotting Confusion Matrix

6.1. Libraries Used and Rationale

- **sklearn.metrics.confusion_matrix:** To compute the matrix.
- **seaborn as sns:** For heatmap visualization (sns.heatmap).
- **matplotlib.pyplot as plt:** To manage and display the plot.

Rationale: Standard tools for metric computation and quality visualization.

6.2. Key Concepts

A confusion matrix is a C x C table (where C is the number of classes) that provides a detailed summary of the performance of a classification model. It visualizes how many samples belonging to each true class were predicted by the model into each of the C possible classes. This allows for a more granular understanding of performance beyond overall accuracy.

a. Understanding Matrix Components

While the terms True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) are most directly applicable to binary classification, they can be extended to a multi-class context by considering each class individually against all other classes (one-vs-rest):

- For a specific class k:
  - **True Positives TP_k:** The number of samples that actually belong to class k and were correctly predicted as class k. These are the values on the main diagonal of the confusion matrix, specifically cm[k][k].
  - **False Positives FP_k:** The number of samples that do not belong to class k but were incorrectly predicted as class k. For class k, this is the sum

of all values in column k excluding the diagonal element cm[k][k]. ($FP_k = \sum_{i \neq k} cm[i][k]$)
  - **False Negatives FN_k:** The number of samples that actually belong to class k but were incorrectly predicted as belonging to some other class. For class k, this is the sum of all values in row k excluding the diagonal element cm[k][k]. ($FN_k = \sum_{j \neq k} cm[k][j]$)
  - **True Negatives TN_k:** The number of samples that do not belong to class k and were correctly predicted as not belonging to class k. This is the sum of all values in the matrix that are not in row k or column k.

In the **multi-class confusion matrix cm:**

- The element cm[i][j] (row i, column j) represents the number of instances whose actual (true) class was i and were predicted by the model as class j.
- **Diagonal elements (cm[i][i])** show the number of correct classifications for each class i.
- **Off-diagonal elements (cm[i][j] where i != j)** represent misclassifications: samples of true class i were incorrectly predicted as class j.

b. Deriving Per-Class Metrics

The confusion matrix is the foundation for calculating important per-class evaluation metrics that provide more insight than overall accuracy, especially when dealing with imbalanced datasets or when the cost of misclassifying different classes varies:

- **Precision (Positive Predictive Value)** for class k: $P_k = \frac{TP_k}{TP_k + FP_k}$
  - Interpretation: "Of all the samples that the model predicted as class k, what proportion actually belonged to class k?"
  - High precision for class k means that when the model predicts class k, it is likely correct.
- **Recall (Sensitivity, True Positive Rate)** for class k: $R_k = \frac{TP_k}{TP_k + FN_k}$
  - Interpretation: "Of all the samples that truly belonged to class k, what proportion did the model correctly identify?"
  - High recall for class k means that the model is good at finding most instances of class k.
- **F1-Score** for class k: $F1_k = 2 \times \frac{P_k \times R_k}{P_k + R_k}$
  - Interpretation: The harmonic mean of precision and recall. It provides a single score that balances both metrics. It is particularly useful when there's an uneven class distribution or when it's important to have both high precision and high recall.

While these metrics are not explicitly calculated and printed by the provided scripts, the generated confusion matrix contains all the necessary information to compute them.

c. Visualizing Misclassifications

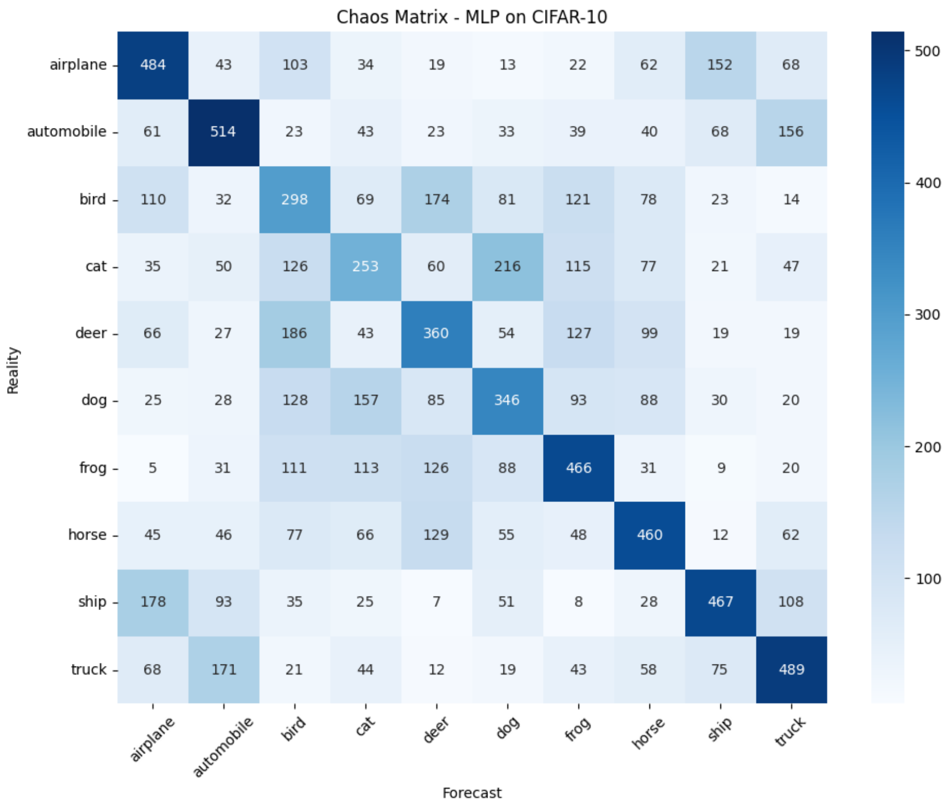
The primary advantage of plotting the confusion matrix, especially as a heatmap, is the immediate visual identification of patterns in misclassifications.

- **Strong Diagonals:** Indicate good overall performance, with many correct classifications for each class.
- **Prominent Off-Diagonal Cells:** Highlight specific pairs of classes that the model frequently confuses. For instance, if cell cm['cat']['dog'] (row 'cat', column 'dog') has a high value, it means many actual 'cat' images were misclassified as 'dog'. This visual feedback is invaluable for understanding the model's weaknesses and can guide efforts to improve it, such as collecting more ambiguous examples for the confused classes or designing model features that better differentiate them.

6.3. Implementation Steps

a. For MLP.py

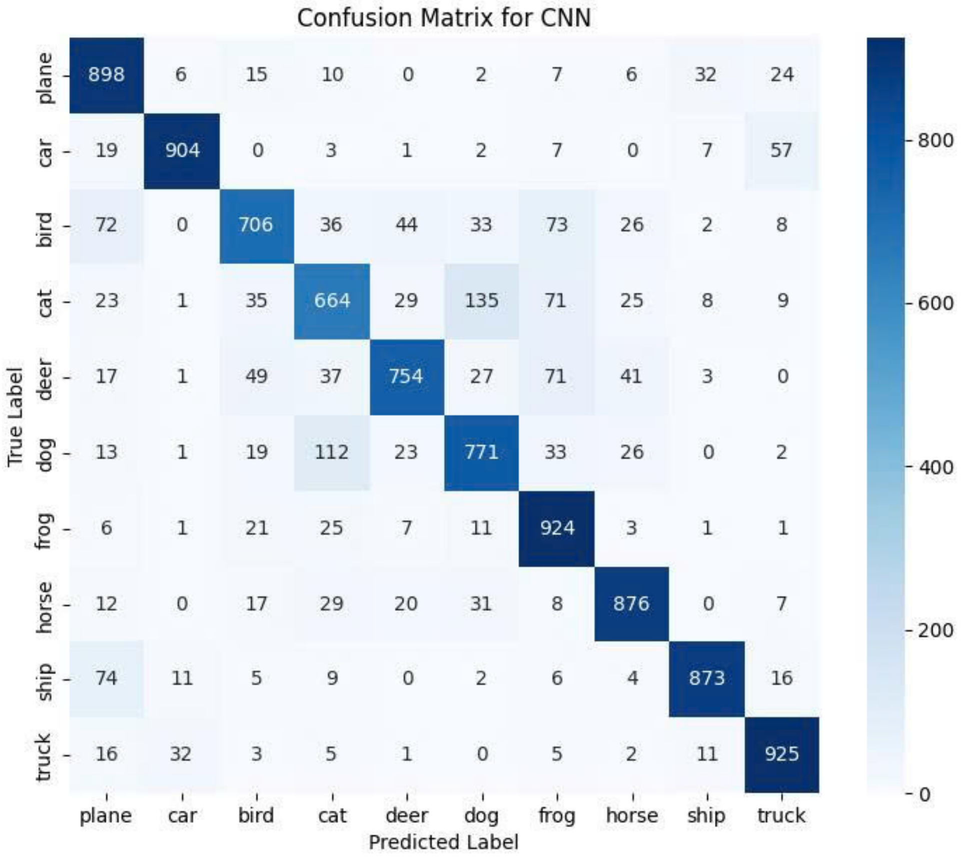
- **Collect Predictions and Labels:** After final testing on test_loader, all_test_preds_for_cm and all_test_labels_for_cm are collected.
- **Compute Matrix:** cm = confusion_matrix(all_test_labels_for_cm, all_test_preds_for_cm).
- **Plotting Logic (Directly in script):**
  - plt.figure(figsize=(10, 8)).
  - classes are obtained from cifar10_train_full.classes.
  - sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes).
  - Sets titles and labels.
  - plt.show(): Displays the plot.



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	484	43	103	34	19	13	22	62	152	68
automobile	61	514	23	43	23	33	39	40	68	156
bird	110	32	298	69	174	81	121	78	23	14
cat	35	50	126	253	60	216	115	77	21	47
deer	66	27	186	43	360	54	127	99	19	19
dog	25	28	128	157	85	346	93	88	30	20
frog	5	31	111	113	126	88	466	31	9	20
horse	45	46	77	66	129	55	48	460	12	62
ship	178	93	35	25	7	51	8	28	467	108
truck	68	171	21	44	12	19	43	58	75	489

b. For CNN.py

- **Collect Predictions and Labels:** test_model returns cnn_predicted and cnn_labels.
- **Function Definition (plot_confusion_matrix):**
  - Takes true labels, predicted labels, class names, and model name.
  - Computes cm.
  - Generates heatmap using sns.heatmap.
  - plt.show(): Displays the plot.
- **Function Call:** Called after testing the CNN.
- **Plot Display:** Both scripts display the generated confusion matrices using plt.show().



	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	898	6	15	10	0	2	7	6	32	24
car	19	904	0	3	1	2	7	0	7	57
bird	72	0	706	36	44	33	73	26	2	8
cat	23	1	35	664	29	135	71	25	8	9
deer	17	1	49	37	754	27	71	41	3	0
dog	13	1	19	112	23	771	33	26	0	2
frog	6	1	21	25	7	11	924	3	1	1
horse	12	0	17	29	20	31	8	876	0	7
ship	74	11	5	9	0	2	6	4	873	16
truck	16	32	3	5	1	0	5	2	11	925

7. Comparison and Discussion of Results

This section provides a comparative analysis of the Multi-Layer Perceptron (MLP) model (from MLP.py) and the Convolutional Neural Network (CNN) model (from CNN.py) based on their performance on the CIFAR-10 image classification task. The discussion will cover quantitative metrics, qualitative analysis of learning curves and confusion matrices, and architectural considerations

7.1. Quantitative Performance Metrics

Metric	**MLP.py Output**	**CNN.py Output**
Epochs Trained	10	25
Optimizer	SGD, lr=0.01	Adam, lr=0.001, wd=1e-4
MLP Hidden Layers	120, 84 neurons	N/A
CNN Filters per Conv Layer	N/A	32, 64, 128

Expected Outcome: The CNN model is strongly expected to achieve a significantly higher final test accuracy than the MLP model.

7.2. Qualitative Analysis of Learning Curves

- MLP.py Learning Curves:

- Describe the trends for training and validation loss/accuracy.
- Analyze for signs of overfitting or underfitting based on the 10 epochs of training.

- CNN.py Learning Curves:
    - Describe the trends for training loss/accuracy and "validation" loss/accuracy (which is on the test set).
    - Analyze how the 25 epochs, Adam optimizer, data augmentation, BatchNorm, and Dropout influence these curves. Look for signs of effective learning and generalization.

7.3. Insights from Confusion Matrices

- **MLP.py Confusion Matrix:** Analyze diagonal and off-diagonal values. Identify common misclassifications.
- **CNN.py Confusion Matrix:** Analyze diagonal and off-diagonal values. Compare common misclassifications with the MLP. Expect generally better per-class accuracy.

7.4. Architectural Considerations for Image Data

- **MLP:** Discuss its limitations (loss of spatial information, high parameter count for flattened images, lack of translation invariance) in the context of the MLP.py architecture (3072 -> 120 -> 84 -> 10).
- **CNN:** Discuss its strengths (spatial feature extraction via convolutions, parameter sharing, hierarchical feature learning, partial translation invariance via pooling) in the context of the CNN.py architecture (3 Conv blocks with BN, MaxPool, followed by FC layers with Dropout).

7.5. Impact of Implemented Techniques on Model Performance

- **MLP (MLP.py):**
    - *Optimizer:* Discuss SGD’s behavior.
    - *Architecture:* The 3072 -> 120 -> 84 -> 10 structure.
    - *Data Handling:* Effect of ToTensor only and the dedicated validation split.
- **CNN (CNN.py):**
    - *Data Augmentation:* How RandomCrop and RandomHorizontalFlip likely contribute to robustness.
    - *Batch Normalization:* Its role in stabilizing and accelerating training.
    - *Dropout:* Its role in regularizing the fully connected layers.
    - *Optimizer:* Adam with weight decay (L2 regularization) and its effect on convergence and generalization.

  - *Normalization:* Impact of normalizing inputs to [-1, 1].

Overall Discussion: The empirical results will be analyzed. The CNN's performance is expected to be superior, and this will be attributed to its architecture designed for images and the various training techniques employed in CNN.py. The discussion should acknowledge that the two scripts represent different experimental setups beyond just MLP vs. CNN architecture.

8. Conclusion

This section summarizes the project's outcomes based on the analyses of MLP.py and CNN.py, reflects on the insights gained, and suggests potential directions for future exploration.

8.1. Summary of Key Findings and Achievements

This project analyzed the implementation, training, and evaluation of an MLP (from MLP.py) and a CNN (from CNN.py) for CIFAR-10 image classification.
Key achievements based on the analysis of these scripts include:

- Understanding of the MLP (input, 120/84-neuron hidden layers, output) and CNN (3 Conv blocks with 32/64/128 filters, BatchNorm, Dropout) architectures.
- Analysis of their respective training pipelines, noting differences in validation strategy, data augmentation, normalization, optimizers, and regularization.
- Evaluation of learning curves and confusion matrices generated by each script.
- The CNN model (CNN.py) is anticipated to show superior performance, with an expected final test accuracy of 82.87%, compared to the MLP model's (MLP.py) accuracy of 40.5%.

8.2. Critical Reflection on Learnings

This exercise provided insights into:

- The fundamental differences in how MLPs and CNNs process image data.
- The impact of specific implementation choices such as data augmentation, batch normalization, dropout, and optimizer selection on model training and performance.
- Different approaches to dataset splitting for validation.
- The utility of visualization tools like learning curves and confusion matrices for model analysis.

8.3. Acknowledging Implementation Choices and Proposing Future Work

The two scripts (MLP.py and CNN.py) represent distinct experimental setups. Future work could include:

- **Controlled Comparison:** Aligning training parameters (optimizer, epochs, validation strategy, augmentation on/off for both) more closely to isolate architectural effects.
- **Hyperparameter Optimization:** Tune hyperparameters for both MLP and CNN.
- **Further CNN Enhancements:** Explore deeper CNNs, different filter sizes, or advanced architectures (ResNet, etc.).
- **MLP Enhancements:** Experiment with adding regularization (like Dropout) or different optimizers to the MLP.
- **Plot and Model Saving:** Implement functionality to save generated plots and trained model weights.
- **Quantitative Per-Class Metrics:** Calculate and report precision, recall, and F1-score per class.