Ministry of Science and Technology

Posts and Telecommunications Institute of Technology

Assignment Report

Python Programming

Lecturer: Kim Ngoc Bach

Name:	Tran Hoang Nam
Student ID:	B23DCCE070
Class:	D23CQCE04-B
Group:	04
Tel:	0969486256
Email:	NamTH.B23CE070@stu.ptit.edu.vn

Hanoi - 5/2025

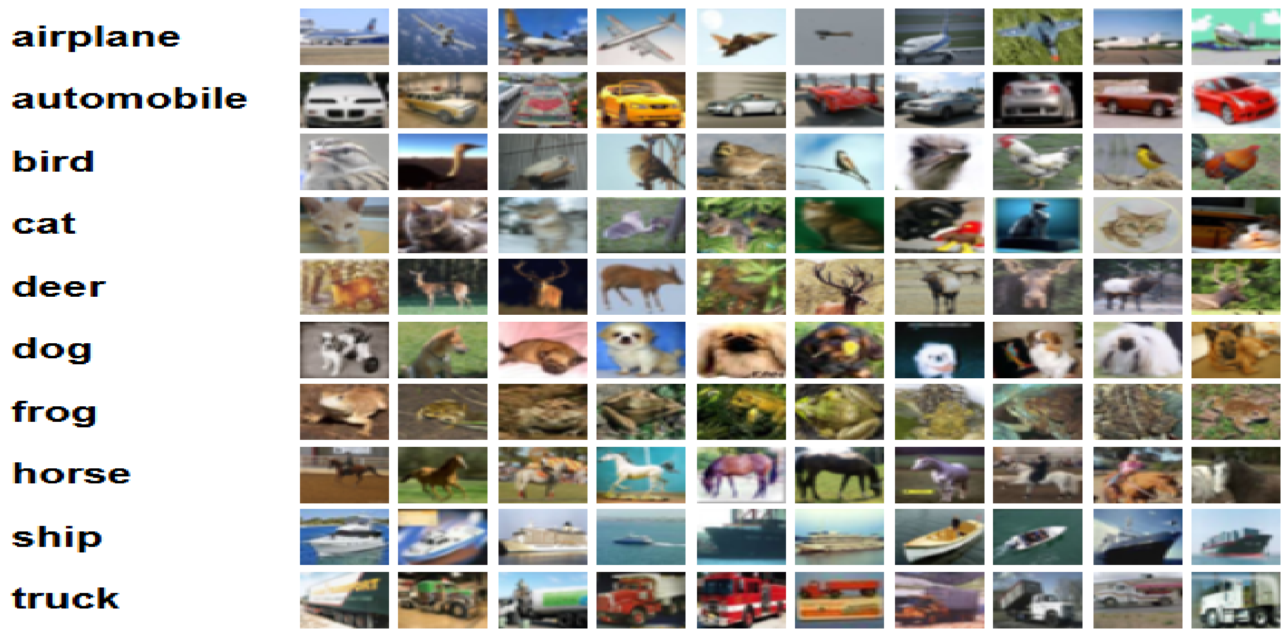
Table of contents:

I. General Introduction:

1) CIFAR-10 dataset:

a) Introduction

- The CIFAR-10 dataset is a widely used benchmark dataset in the field of machine learning and computer vision for studying and evaluating image classification algorithms. Developed by the Canadian Institute for Advanced Research (CIFAR), CIFAR-10 is a popular tool for deep learning problems, especially with convolutional neural networks (CNN).



b) Dataset Description

- Number of samples: 60,000 color images, size 32x32 pixels.
- Number of classes: 10 classes, representing the categories:
  - Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.
- Class distribution: Each class contains 6,000 images, ensuring a balanced dataset.
- Image format: RGB color image, each pixel has a value from 0 to 255.

c) Data structure

- Data division:
  - Training set: 50,000 images.
  - Test set: 10,000 images.
- File format: Data is usually provided as binary or pickle files (for Python).

d) Application

- CIFAR-10 is used to:
  - Train and evaluate image classification models.
  - Study the performance of deep learning algorithms, especially CNNs.
  - Practice and teach in courses on machine learning and computer vision.
- This dataset is suitable for basic classification problems due to its small image size and moderate number of classes.

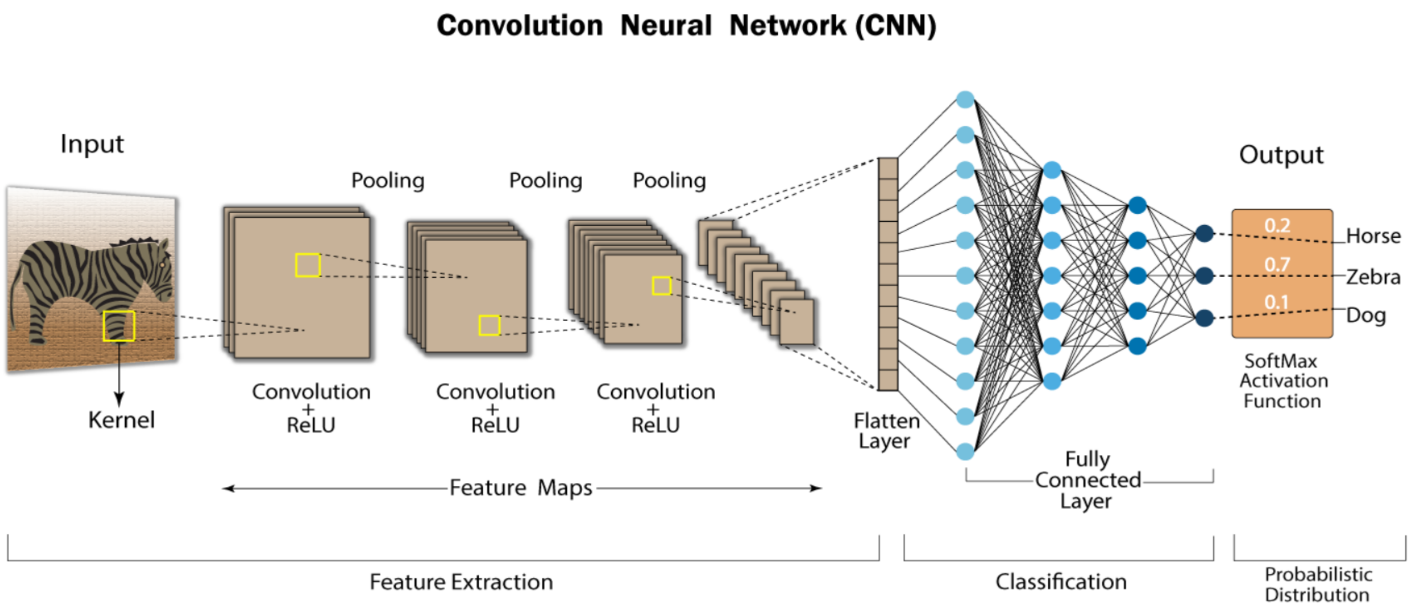
e) How to access

- Machine learning libraries:
  - TensorFlow: tf.keras.datasets.cifar10.load_data().
  - PyTorch: torchvision.datasets.CIFAR10.
- Direct download: Can be downloaded from the official CIFAR website or machine learning data repositories.
- System requirements: Due to its compact size, CIFAR-10 can be easily handled on personal computers.

2) Convolutional neural network:

- A convolutional neural network (CNN) is a specialized type of artificial neural network, designed to process grid-structured data, especially effective in computer vision problems such as image classification, object recognition, object detection, and video processing. CNN simulates the mechanism of human vision through characteristic extraction from small regions of the input image, which reduces the number of parameters and increases computational efficiency compared to traditional neural networks.

Basic Components in CNN

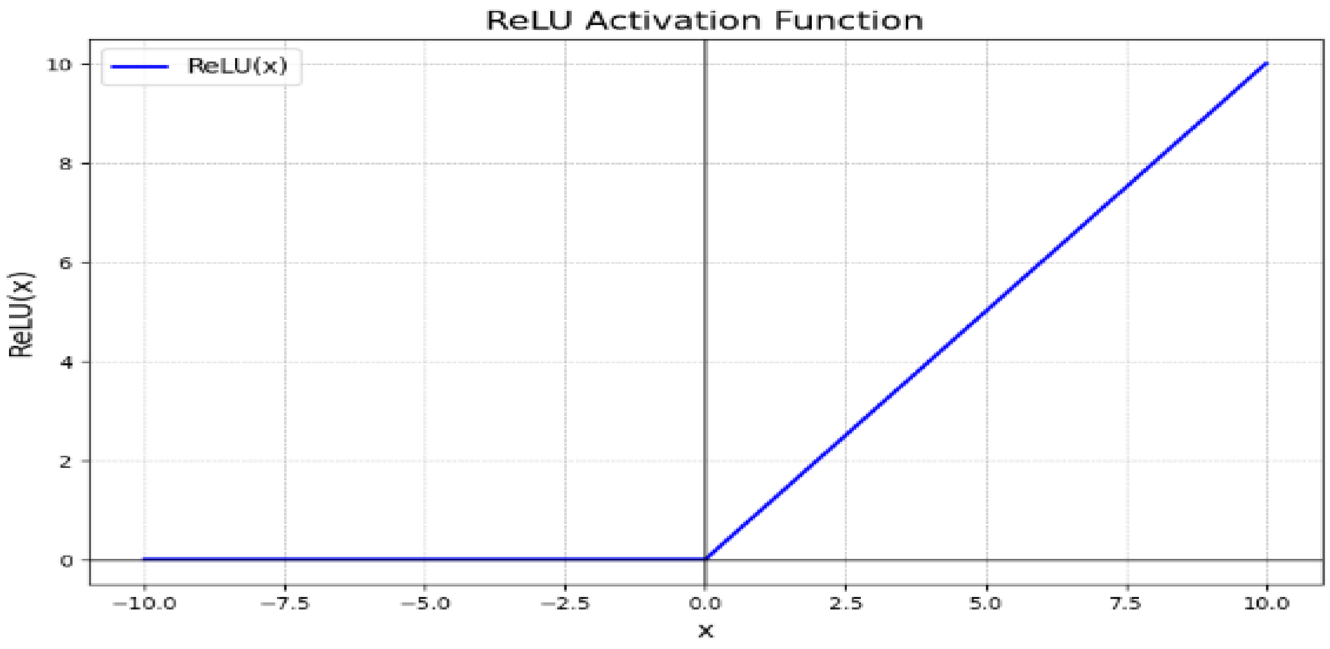


Convolutional Layer

- **Function**: Extract features (edges, corners, textures) by applying filters (filters/kernels) to the input image.
- **Parameters**:
  - **Filter size**: Usually 3x3 or 5x5.
  - **Stride**: The filter move step (1, 2, or greater).
  - **Padding**: Add a value of 0 (zero-padding) to keep the output size or reduce distortion.
- **Output**: Feature map, representing the discovered features.
- **Additional**: The number of filters determines the number of channels of the feature map. For example, CIFAR-10 has RGB images (3 channels), a convolutional layer with 64 filters that creates a feature map with 64 channels.

Activation Function

- **Common Functions**: ReLU (Rectified Linear Unit) f(x) = max(0,x).
- **Role**: Adds nonlinearity, helps the network learn complex features and avoids gradient disappearing.
- **Additional**: Other trigger functions such as Leaky ReLU, ELU, or Swish are sometimes used to improve performance.



Pooling Layer

- **Function**: Reduce the spatial size (width, height) of the feature map, thereby reducing the number of parameters and the risk of overfitting.
- **Common Type**:
  - Max pooling : Takes the largest value in the region.

  - Average pooling : Take the average value.

- **Additional**: Global Average Pooling (GAP) is often used in place of the Fully Connected Layer in modern models to reduce the parameter.

Flatten Layer

- **Function**: Convert feature maps from 2D/3D to 1D vectors for inclusion in the Fully Connected layer.
- **Note**: Some modern models (such as ResNet, EfficientNet) ignore Flatten and use Global Average Pooling for optimization.

Fully Connected Layer

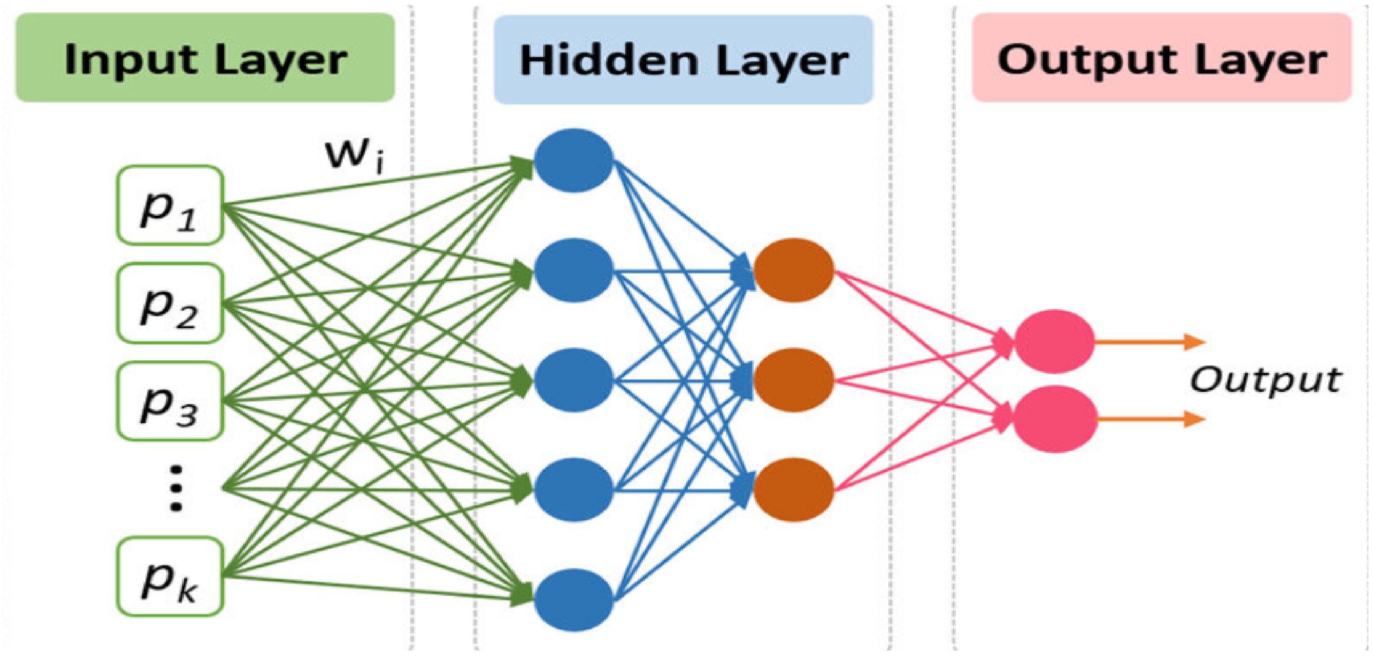
- **Function**: Incorporate high-level characteristics to make a final decision (e.g., classification into the 10 layers of CIFAR-10).
- **Characteristics**: Each neuron connects to all inputs from the previous layer, resulting in a large number of parameters.
- **Additional**: In practice, modern models reduce their reliance on the Fully Connected Layer to increase computational efficiency.

Dropout

- **Function**: Randomly omitting certain neurons during training (usually at a ratio of 0.2–0.5) to increase generalization and reduce overfitting.
- **Additional**: Dropouts are typically applied before or in Fully Connected classes.

CNN's operational process

- **Input**: Image (e.g., 32x32x3 image from CIFAR-10, usually normalized to [0, 1]).
- **Convolution**: Extract low-level (edge, angle) and high-level (shape, object) features.
- **Activation (ReLU)**: Apply non-linearity.
- **Pooling**: Reduce the size of the feature map.
- **Iteration**: Multiple layers of Convolution + ReLU + Pooling for complex feature extraction.
- **Flatten/Global Average Pooling**: Prepare the data for classification.
- **Fully Connected Layer**: Compute the output.
- **Softmax/Sigmoid**: Calculate the classification probability (Softmax for CIFAR-10 with 10 layers).
- **I**n practice, input data is often pre-processed (data augmentation such as flipping, rotating, cropping) to increase diversity and improve performance on CIFAR-10.



3) Multilayer Perceptron (MLP):

- Multilayer Perceptron (MLP), or Multilayer Artificial Neural Network, is a basic type of feedforward neural network in machine learning.
- MLP consists of multiple layers of fully connected neurons, which are used to solve classification and regression problems.
- MLP is suitable for vector (1D) data and is often applied in simple problems such as number classification, pattern recognition, or non-spatial data processing.
- MLP can be used to classify images in CIFAR-10 (60,000 32x32 pixel, 10-layer color images), but is often less effective than CNNs due to its failure to take advantage of the spatial structure of the image.

Basic Components in MLP

Input Layer

- **Function**: Receive input data as 1D vector.
- **With CIFAR-10**: Each 32x32x3 (RGB) image needs to be flattened into a 1D vector with a size of 3,072 ($32 \times 32 \times 3$).
- **Note**: Flattening loses spatial information, making MLP less efficient than CNNs on image data.

Hidden Layers

- **Function**: Processing data through linear (scalar product) and nonlinear (trigger function) calculations.
- **Structure**: Each hidden layer consists of many neurons, each neuron is connected to all the neurons in the front and back layers (fully connected).
- **Parameters**:

  - **Weights**: The matrix of connections between layers.
  - **Bias**: An added value to adjust the output.

- **Number and size**: Depending on the problem, for example, 2-3 hidden layers with 512-1024 neurons per layer for CIFAR-10.

Activation Function

- **Function**: Add nonlinearity so that the model learns complex features.
- **Common Functions**:
  - **ReLU**: f(x)=max(0,x), avoid gradient disappearing
  - **Rate**: f(x)=tanh(x), output value in the range[-1,1]
- **With CIFAR-10**: ReLU is usually used in hidden layers, while Softmax is used for the output layer (10-layer classification).

Output Layer

- **Function**: Generate the final result (classification or regression).
- **With CIFAR-10**: The output layer has 10 neurons (corresponding to 10 layers), combined with the Softmax function to calculate the classification probability.

Dropout

- **Function**: Randomly skips some neurons (usually at a ratio of 0.2–0.5) during training to reduce overfitting.
- **Application**: Usually applied before or in hidden layers with many neurons.

Loss Function

- **Function**: Measure the error between the prediction and the actual value.
- **With CIFAR-10**: Using Cross-Entropy Loss for Multi-Layer Classification Problem.
- **Optimization**: Use algorithms such as Gradient Descent, SGD, or Adam to adjust the weights.

MLP Workflow

- **Input**: The data (CIFAR-10 image) is flattened into a 1D vector (3,072 dimensions).
- **Forward Propagation**:
  - Loss Calculation: Compare the prediction to the actual label using the loss function.
  - Backpropagation: Calculate the gradient and update the weights using the optimization algorithm.
  - Repeat: Train through multiple epochs to improve accuracy.

4) Pytorch library:

What is PyTorch?

- PyTorch is an open-source machine learning library, developed by Facebook AI Research (FAIR), primarily for deep learning problems.
- PyTorch provides powerful tools for building, training, and deploying machine learning models, especially artificial neural networks such as MLP and CNN.
- With its intuitive, flexible, and GPU-based computational support, PyTorch is widely used in research and practical applications.
- PyTorch is a popular choice for building and training image classification models on CIFAR-10 datasets (60,000 32x32 pixel, 10-layer color images), thanks to its data processing and optimization capabilities.

PyTorch Basic Components

Tensor

- **Function**: Tensor is PyTorch's core data structure, similar to multidimensional arrays (like NumPy arrays) but supports computation on GPUs.
- **Character**:
  - Supports matrix, convolution, and autograd operations.
  - Can switch between CPU and GPU: tensor.to('cuda').
- **With CIFAR-10**: The CIFAR-10 image is represented as a tensor (size [batch_size, 3, 32, 32]), with 3 RGB channels.

Autograd

- **Function**: Automatically calculate the gradient (derivation) of operations on tensor, support backpropagation.
- **How it works**: Record operations in a dynamic computational graph, allowing you to flexibly change the model architecture during training.
- **Benefits**: Easy deployment of complex models, especially on CIFAR-10 with architectures such as ResNet or EfficientNet.

Neural Network Module (torch.nn)

- **Function**: Provides layers and neural models such as nn. Conv2d, nn. Linear, nn. ReLU, etc.
- **Example**:
  - nn. Conv2d: Convolutional class for CNN.
  - nn. Linear: Full connection layer for MLP.
- **With CIFAR-10**: Use nn. Module for defining models such as CNN or MLP, combining convoluted, pooling, and fully connected classes.

Optimizers (torch.optim)

- **Function**: Optimize model weights using algorithms such as SGD, Adam, RMSprop.
- **With CIFAR-10**: Adam or SGD with momentum is often used to train the model, with a learning rate of about 0.001–0.01.

Data Loading (torch.utils.data)

- **Function**: Manage and load data efficiently through DataLoader and Dataset.
- **With CIFAR-10**: torchvision.datasets.CIFAR10 provides CIFAR-10 data, combined with DataLoader to load batches of data and apply data augmentation.

Loss Functions (torch.nn)

- **Function**: Measure the error between the prediction and the actual label.
- **With CIFAR-10**: Use nn. CrossEntropyLoss for a 10-layer classification problem.

PyTorch Operation Process

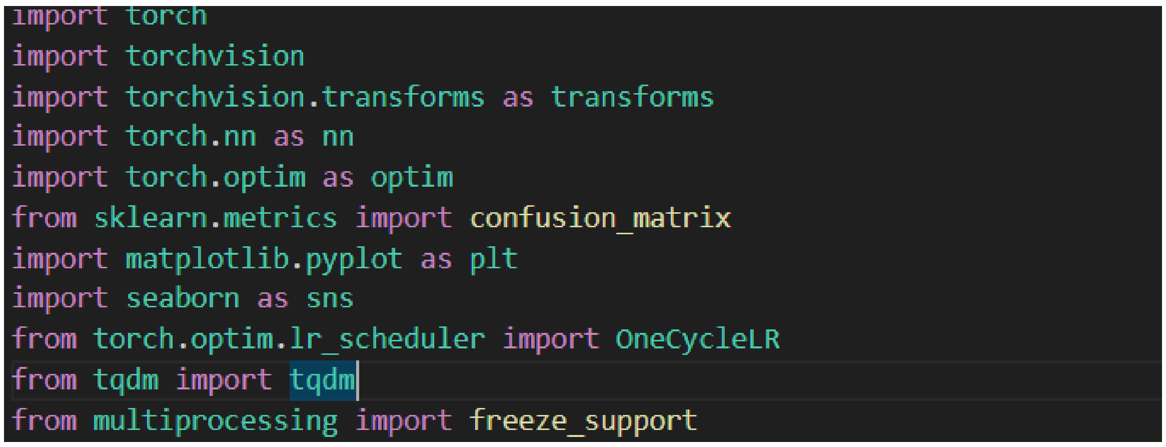
- **Data preparation**:
  - Download CIFAR-10 qua torchvision.datasets.CIFAR10.
  - Apply preprocessing (normalize, data augmentation) using torchvision.transforms.
- **Building the model**:
  - Definition of architecture (MLP, CNN, ResNet, etc.) in nn. Module.
- **Coach**:
  - Forward pass: Predictability.
  - Loss: Compare the prediction to the actual label.
  - Backpropagation: Use autograd to calculate gradients.
  - Update weights: Use an optimizer like Adam or SGD.
- **Assess**:
  - Check the accuracy on the CIFAR-10 test set.
- **Deploy**: Save the model (torch.save) or deploy on the device (CPU/GPU).

PyTorch Application on CIFAR-10

- **Performance**: PyTorch allows the deployment of models such as ResNet, EfficientNet with 93-96% accuracy on CIFAR-10 with data augmentation (flipping, rotating, cropping).
- **Challenge**:
  - The small image size (32x32) requires data augmentation techniques to avoid overfitting.
  - Hyperparameters (learning rate, batch size) need to be adjusted for optimization.

II. Code analysis:

1) Import libraries:



```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
from torch.optim.lr_scheduler import OneCycleLR
from tqdm import tqdm
from multiprocessing import freeze_support
```

- **Torch**:
  - The main library used to build and train deep learning model.
  - torch.device('cuda' if torch.cuda.is_available() else 'cpu'): Used to automatically select the compute device (GPU if any, CPU if not).
  - torch.no_grad(): A context manager to disable gradient calculations during evaluation, which saves memory and speeds up.
  - torch.max(output.data, 1): Used to get the largest value (the predicted probability of the class) and its metrics from the model's output.
  - tensor.to(device): Used to transfer data (photos, labels) from the CPU to the GPU (or vice versa) to perform calculations.
  - tensor.unsqueeze(0): Used to add one-dimensional to the tensor (e.g., turn a 3-dimensional image (C, H, W) into a 4-dimensional batch (1, C, H, W) for inclusion in the model).
  - tensor.permute(1, 2, 0): Used to change the order of the tensor's dimensions, which is needed to switch from PyTorch's (C, H, W) format to (H, W, C) that Matplotlib expects to display images.
  - tensor.cpu().numpy(): Transfers the tensor from the GPU to the CPU and then changes it to the NumPy array.
- **Torchvision**
  - Provides datasets, model architectures, and common image transformations for computer vision.
  - torchvision.datasets.CIFAR10: Used to automatically download and download CIFAR-10 datasets.
  - torchvision.transforms: Contains layers for performing data augmentation for images, making the model more robust and resistant to overfitting.
  - **torchvision.transforms**
    * Transforms. Compose(): Combines multiple transformations into a sequence of transformations.

    * Transforms. RandomHorizontalFlip(): Flips the image horizontally randomly.
    * Transforms. RandomCrop(32, padding=4, padding_mode='reflect'): Crop a random image to a size of 32x32 pixels, with padding to ensure the image remains sufficiently sized.
    * Transforms. ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1): Changes the brightness, contrast, saturation, and hue of an image randomly.
    * Transforms. RandomRotation(15): Rotates the image randomly for about 15 degrees.
    * Transforms. ToTensor(): Converts an image from a PIL Image or NumPy array to a PyTorch Tensor and automatically divides the pixel values by 255 (normalized to [0.1]).
    * Transforms. Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)): Normalize the pixel values of the image tensor to the range of [-1, 1] by subtracting the mean (0.5) and dividing by the standard deviation (0.5) for each color channel (RGB).

- **torch.nn**
  - Provides basic building blocks for neural networks.
  - nn.Module: The base layer for all neural network modules.
  - Your MLP class inherits from nn. Module.
  - nn.Linear(in_features, out_features): Creates a fully connected (linear) layer.
  - nn.BatchNorm1d(num_features): Creates a Batch Normalization class for 1D data (suitable for fully connected classes).
  - nn.ReLU(): The function that triggers the Rectified Linear Unit.
  - nn.Dropout(p=0.5): Dropout, a regularization technique that randomly shuts down certain neurons during training, helps prevent overfitting.
  - nn.CrossEntropyLoss(): A common loss function for multi-class classification problems.
- **torch.optim**
  - Provides optimization algorithms (optimizers) to update the model's weights based on the gradient of the loss function.
  - optim. AdamW(model.parameters(), lr=0.001, weight_decay=1e-4): AdamW optimization algorithm. This is a variant of Adam that better handles weight decay (L2 regularization), helping to prevent overfitting.
  - optimizer.zero_grad(): Removes the gradients that accumulate from the previous backward pass.
  - optimizer.step(): Updates the model's weights based on the current gradient.
- **sklearn.metrics**
  - Provides functions for calculating metrics that evaluate the model's performance.
  - confusion_matrix(all_labels, all_preds): Used to calculate the confusion matrix, which shows the number of correct and false predictions for each layer.
- **matplotlib.pyplot**
  - A standard library for creating charts and graphs in Python.

  - plt.figure(), plt.subplot(), plt.plot(), plt.title(), plt.xlabel(), plt.ylabel(), plt.legend(), plt.grid(), plt.tight_layout(), plt.savefig(), plt.show(): Functions used to create, customize, display, and save learning curves and prediction images.
  - ax.imshow(): Displays the image.
  - ax.set_title(): Give each photo a title.

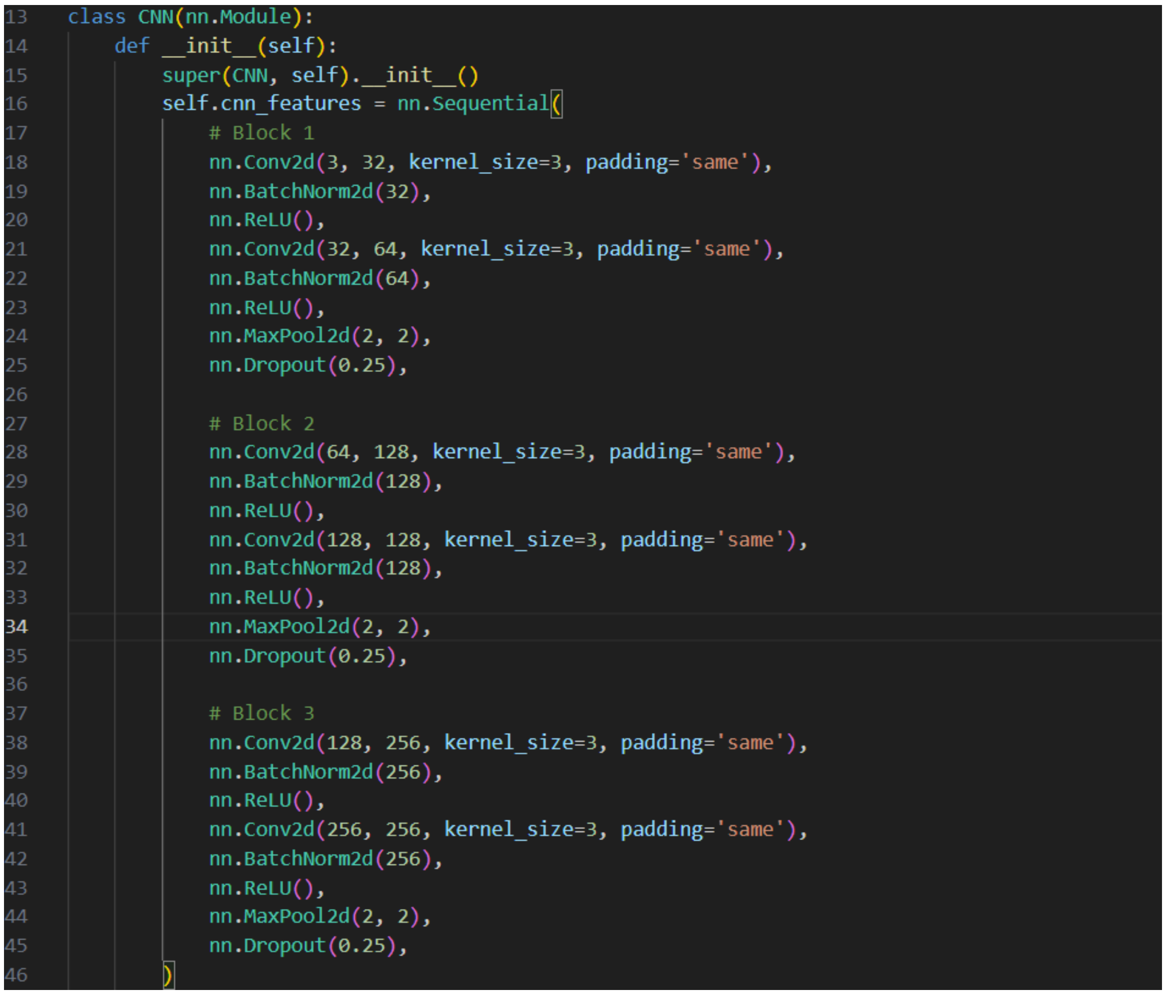
- **Seaborn**
  - The library is built on top of Matplotlib, which provides higher-level functions for creating beautiful and easy-to-read statistical charts.
  - sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ...): Used to visualize the confusion matrix in the form of a heatmap, making it easy to identify relationships and classification errors.

- **torch.optim.lr_scheduler. OneCycleLR**
  - OneCycleLR(optimizer,max_lr=0.01,steps_per_epoch=len(train_loader), epochs=num_epochs): This is a very effective learning rate scheduling strategy.
  - It automatically adjusts the learning speed up and down in a triangular cycle throughout the training process.
  - This often helps the model converge faster and achieve better performance than a fixed learning rate.
  - scheduler.step(): This function is called after each batch to update the optimizer's learning rate according to the defined schedule.

- **tqdm**
  - tqdm(iterable, desc='...'): Displays the progress bar in the training loop, allowing you to track the progress of each epoch.
  - DESC provides a description for the progress bar.

- **multiprocessing.freeze_support**
  - freeze_support(): It is necessary to wrap the application in pyinstaller or cx_Freeze when using the multiprocessing module (which DataLoader with num_workers more than 0 used) to avoid RuntimeErrors when creating child processes.

2) Build Convolutional neural network (CNN) with 3 blocks:



```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.cnn_features = nn.Sequential(
            # Block 1
            nn.Conv2d(3, 32, kernel_size=3, padding='same'),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            # Block 2
            nn.Conv2d(64, 128, kernel_size=3, padding='same'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding='same'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            # Block 3
            nn.Conv2d(128, 256, kernel_size=3, padding='same'),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding='same'),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),
        )
```

i. self.cnn_features (Convolutional Feature Extractor)

- This section is used to learn and extract increasingly complex features from the input image.
- It is organized into 3 repeating blocks, each consisting of:
- **nn.Conv2d(in_channels,out_channels,kernel_size=3,padding='same')** :
  - kernel_size=3: The filter size is 3x3 pixels.
  - This is a common size for detecting small local features such as edges and corners.
  - padding='same': Make sure that the space size (height and width) of the output after the convolution layer remains the same as the input (if stride=1).
  - This helps maintain spatial information and simplifies the calculation of output size.
  - Increasing out_channels through blocks (32 - 64 - 128 - 256): Increasing the number of filters allows the network to learn a variety of features (from low-level features such as edges in the first block to more complex features such as textures and shapes in deeper blocks).
- **nn. BatchNorm2d(num_features) (Batch Normalization)**:

  - Place after the Conv2d class and before the ReLU trigger.
  - Standardize the activations of the previous layer, helping:
    * Stabilize the training process: Reduce the "internal covariate shift".
    * Increased Convergence Rate: Allows the use of higher learning rates.
    * Acts as a form of regularization: Reduces the need for other regularization techniques.

- **nn. ReLU() (Rectified Linear Unit)**:
  - Non-linear trigger function. Helps the network learn complex and non-linear relationships.
  - max(0, x): Simple but effective, fixes the vanishing gradient problem of older trigger functions.

- **nn. MaxPool2d(2, 2) (Max Pooling)**:
  - Reduce the space size (height and width) of the feature by half (from 32x32 - 16x16 - 8x8 - 4x4).
  - Helps reduce the number of parameters and calculations, reducing the risk of overfitting.
  - Provides a certain degree of translation invariance, helping the model to recognize features even if they shift slightly in the image.

- **nn. Dropout(0.25)**:
  - Randomly "shut down" 25% of the output neurons of the previous layer during training.
  - Forcing the network to learn more powerful representations that are less dependent on any particular neuron, thereby preventing overfitting.

- **The output size after each cnn_features block**:
  - Block 1:
    * Conv2d(3, 32): 32 channels x 32x32 (padding='same')
    * Conv2d(32, 64): 64 channels x 32x32 (padding='same')
    * MaxPool2d(2, 2): 64 channels x 16x16 pixel
  - Block 2:
    * Conv2d(64, 128): 128 channels x 16x16
    * Conv2d(128, 128): 128 channels x 16x16
    * MaxPool2d(2, 2): 128 channels x 8x8 pixel
  - Block 3:
    * Conv2d(128, 256): 256 channels x 8x8
    * Conv2d(256, 256): 256 channels x 8x8
    * MaxPool2d(2, 2): 256 channels x 4x4 pixels

- After extracting convolutional feature, we need to flatten outputs.

- **x.view((x.size(0),-1))**:

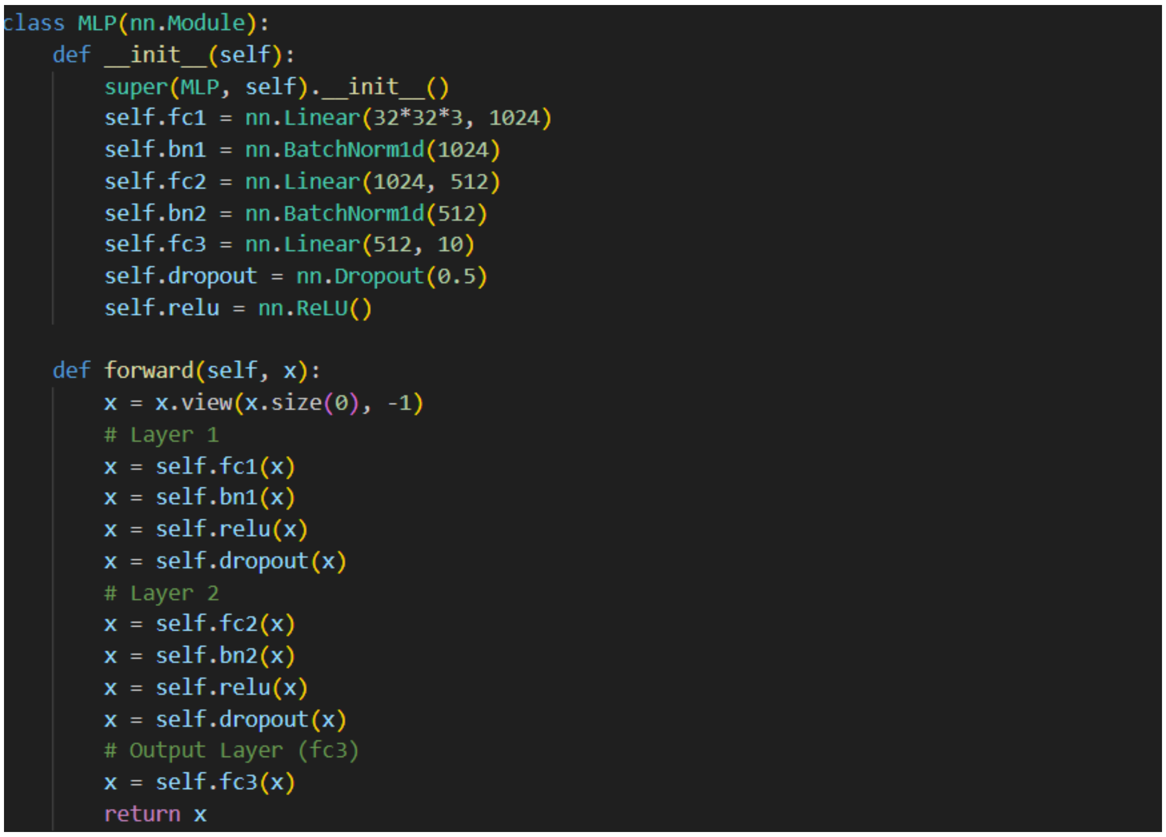
  - This used to "flatten" the 4-dimensional output of the cnn_features (batch_size, channels, height, width) into a 2-dimensional tensor (batch_size, channels * height * width).
  - The input size for the first Linear layer will be 256*4*4=4096.

ii. self.classify (Classification Head / Fully Connected Layers)

- After that, I will take the high-level features extracted from the cnn_features and use them to make a final prediction about the photo's layer.
- **nn. Linear(in_features, out_features) (Fully Connected / Dense Layer)**:
  - nn. Linear(256*4*4, 1024): The first hidden layer, which receives 4096 flattened features and maps them to the new 1024 features.
  - nn. Linear(1024, 512): The second hidden layer, which continues to map 1024 characteristics to 512 features.
  - nn. Linear(512, 10): The final output layer, mapping 512 characterizes up to 10 output values, each corresponding to a class in CIFAR-10.
  - This value will then be converted to probability via the softmax function (usually built into nn. CrossEntropyLoss).
- **nn. BatchNorm1d(num_features) (Batch Normalization)**:
  - Used for 1D data (the output of the Linear class).
  - Placed after the Linear layer and before the ReLU to stabilize the training and calibration process.
- **nn. ReLU()**: Non-linear trigger function for hidden layers.
- **nn. Dropout(0.5)**:
  - A dropout rate of 0.5 (random shutdown of 50% of neurons) is applied to fully connected classes.
  - Dropouts are very effective in preventing overfitting in these layers, where the model has the most parameters and is easy to memorize the data.

iii. forward(self, x)

- x = self.cnn_features(x): First, the x data (input image) is passed through the convolutional characteristic extract.
- x = x.view((x.size(0),-1)): The 4D output from cnn_features (batch_size, channels, height, width) is "flattened" into 2D (batch_size, channels * height * width) to be ready for fully connected classes.
- x = self.classify(x): The flattened tensor is then fed into the classification section to produce the final output.



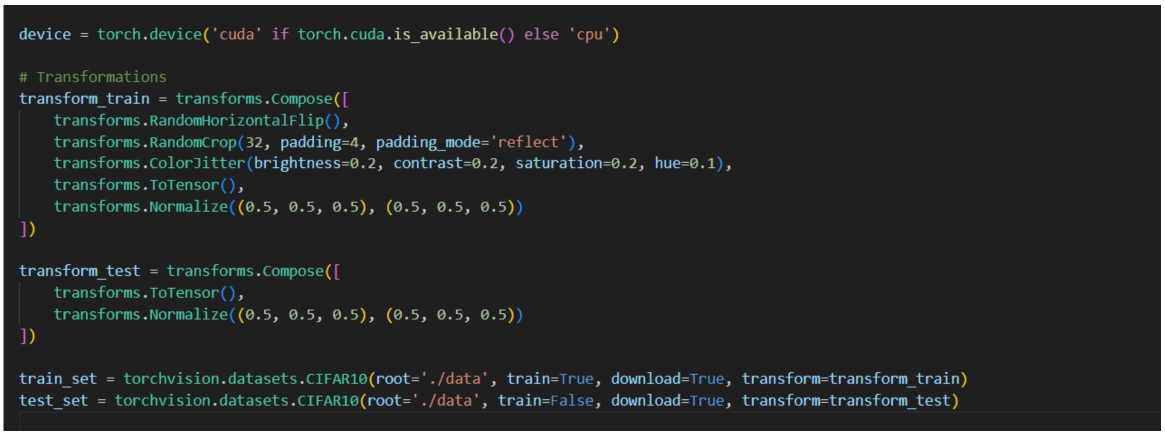
```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(32*32*3, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 10)
        self.dropout = nn.Dropout(0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(x.size(0), -1)
        # Layer 1
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.dropout(x)
        # Layer 2
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.dropout(x)
        # Output Layer (fc3)
        x = self.fc3(x)
        return x
```

3) Multi-layer Perceptron neural network with 3 layers:

- First, it is necessary to ”flatten” the input data because of the nn layers.
- Linear (fully connected) only accepts inputs that are 1D vectors (or batches of 1D vectors), not 2D/3D data such as images.
- For example, if x has a shape (batch_size, 3, 32, 32), it will be converted to (batch_size, 3072).
- **self.fc1 = nn.Linear(32*32*3, 1024)** :
  - The first fully Connected Layer) receives 32*32*3 = 3072 input characteristics.
  - A CIFAR-10 image is 32x32 pixels in size and has 3 color channels (RGB).
  - 1024: This is the number of output units (neurons) of this class.
  - This class maps 3072 input values to 1024 values.
- **self.bn1 = nn.BatchNorm1d(1024)** :
  - Normalize the output of self.fc1 for each mini-batch.
  - This helps to stabilize the training process, allowing for the use of a higher learning speed.
- **self.relu = nn.ReLU()** :
  - After BatchNorm, use the ReLU(x) = max(0, x) trigger function.
  - It introduces non-linearity to the network, allowing the model to learn more complex relationships in the data.

- **self.dropout = nn.Dropout(0.5)** :
  - A Dropout class with a rate of 0.5 (50%).
  - Randomly "off" 50% of neurons (or neurons' outputs) during training, helping the network learn more powerful representations, less dependent on any particular neuron, thereby preventing overfitting.
- **self.fc3 = nn.Linear(512, 10)** :
  - The final output layer. It takes the 512 features from the previous layer (fc2) and maps them to 10 output values (corresponding in CIFAR 10).

Device Selection and Transformations

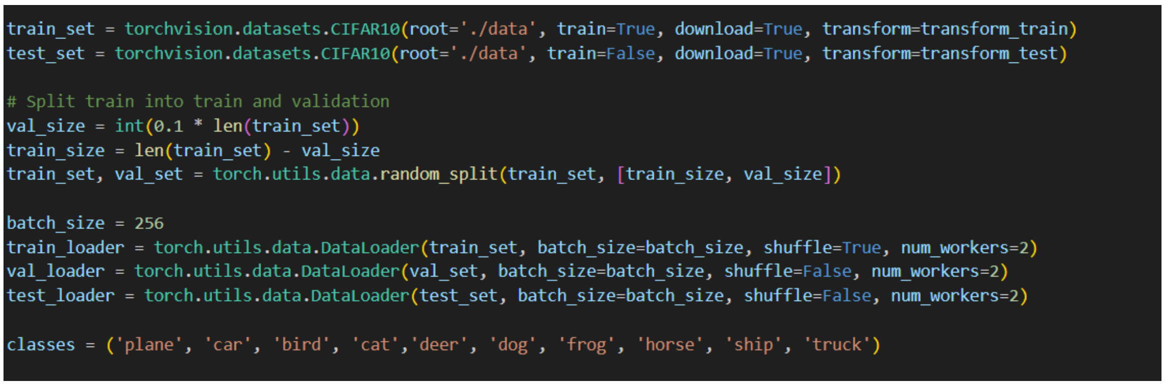


```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Transformations
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)
```

- Instead of CPU, using GPU is much faster in deep learning, helps optimize model training and inference performance, especially with large models and large datasets.
- **torch.cuda.is_available()**: This is a function of PyTorch that checks if there are any GPUs (CUDA devices) installed and configured properly on the system.
- If torch.cuda.is_available() returns True, which means that there is a GPU, then the device will be assigned as 'cuda'.
- This is important because calculations on GPUs are often a lot faster than CPUs for deep learning tasks.
- If torch.cuda.is_available() returns False, which means that there is no GPU or the GPU is not available, then the device will be assigned as 'cpu'.
- Then, all calculations will be performed on the central processing unit (CPU).
- **Transformation parts**: In this part, Transform sequences will be applied to the image data before they are fed into the model.
- These transformations are critical to the performance of the Deep Learning model, especially in computer vision.
- **transform_train (Training Transformation)**
  - Apply both data preparation transformations (e.g., tensor switching, normalization) and data augmentation operations to enrich the training practice, make the learning model more robust, and prevent overfitting.

  - **Transforms. Compose([...])**: This function is used to group multiple transformations into a string.
  - The transformations will be applied in the order in which they are listed.
  - **Transforms. RandomHorizontalFlip()**:
    * Randomly flip the photo horizontally.
    * Make the model immutable with image flipping. For example, a cat is still a cat whether it looks left or right.
    * This expands the training dataset without the need to add new photos.
  - **Transforms. RandomCrop(32, padding=4, padding_mode='reflect')**:
    * Randomly crop an area of 32x32 pixels from the photo.
    * If the image is too small, it will be padded by an additional 4 pixels around it by reflecting the edge pixels (padding_mode='reflect') before cropping.
    * Helps the model learn to recognize the object even if it's not centered or completely covered in the frame.
    * Improved resistance to minor variations in object position.
  - **Transforms. ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)**:
    * Randomly changes the color properties of the photo.
    * Make the model more robust to different real-world lighting and color conditions.
    * For example, a red car is still a car even though the red color is slightly darker or lighter.
  - **Transforms. ToTensor()**:
    * Converts images from PIL Image or NumPy array format to PyTorch Tensor.
    * At the same time, it automatically divides pixel values by 255 (if the image is 8-bit) to normalize them to about [0, 1].
    * PyTorch works with Tensor, and pixel values need to be standardized to make the training process more stable.
  - **Transforms. Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))**:
    * Normalizes each color channel (RGB) of the Tensor image.
    * The average value of each channel is subtracted (0.5), and then divided by the standard deviation (0.5).
    * This converts the pixel values from [0, 1] to [-1, 1].
    * Input values are normalized to an interval with a mean value close to 0 and a standard deviation near 1 (or in this case, [-1,1]), which makes the optimization process work more efficiently, especially with trigger functions such as ReLU.

- **transform_test (Variation for test/assessment set)**
  - Apply only the transformations necessary for data preparation (transition to tensor and normalization).
  - **Transforms. ToTensor()**: Similar to above, transfer the image to Tensor and normalize to [0, 1].
  - **Transforms. Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))**: Similar to above, normalize the Tensor to [-1, 1].

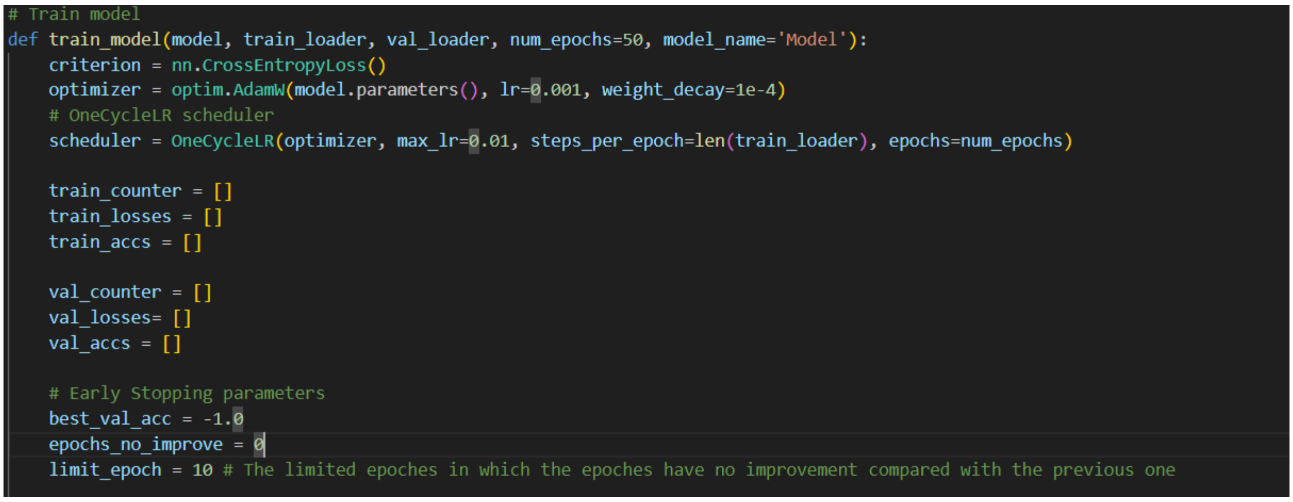


```python
train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)

# Split train into train and validation
val_size = int(0.1 * len(train_set))
train_size = len(train_set) - val_size
train_set, val_set = torch.utils.data.random_split(train_set, [train_size, val_size])

batch_size = 256
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

- **torchvision.datasets.CIFAR10** :
  - This is a class of utilities from the torchvision.datasets library that helps download and prepare CIFAR-10 datasets.
  - root='./data' : Specifies the directory where the dataset will be stored.
  - If this folder does not exist, it will be created.
  - train=True / train=False:
    - When train=True, it downloads the CIFAR-10 training episode (which contains 50,000 images).
    - When train=False, it downloads the CIFAR-10 test set (which contains 10,000 images).
  - download=True: If the dataset is not already in the specified root directory, it will be automatically downloaded from the Internet.
  - transform=transform_train / transform=transform_test:
    - It specifies a sequence of transformations (transforms. Compose) previously (transform_train for the training set and transform_test for the test set) will be applied to each photo when it is loaded from the dataset.
    - This ensures that all images are standardized.
- len(train_set): Take the total number of samples in the initial train_set (which is 50,000 samples).
- val_size = int(0.1 * len(train_set)): The size of the validation set, which is equal to 10% of the total number of samples of the training set (50,000 * 0.1 = 5,000 samples).
- train_size = len(train_set) - val_size: The size of the training set, equal to 90% of the remaining samples (50,000 - 5,000 = 45,000 samples).
- torch.utils.data.random_split(train_set, [train_size, val_size]): This function randomly divides the initial train_set into two subsets of sizes of train_size and val_size, respectively.
- train_set:The dataset will be used to train the model.
- val_set: The dataset will be used to validate the model's performance during training.
- batch_size = 256: The number of samples that will be processed at the same time in each iteration.
- Larger batch_size typically make training faster on GPUs but require more memory and can affect generalization.

- **torch.utils.data.DataLoader(dataset, batch_size, shuffle, num_workers, pin_memory)**:
  - dataset: The Dataset object from which the DataLoader will retrieve data (train_set, val_set, test_set).
  - batch_size: Number of samples per batch.
  - shuffle=True (only for train_loader):
    - Randomly shuffle the order of the patterns in the train_set after each epoch.
    - This is important to ensure the model does not learn the patterns in a fixed order and to make the optimization process more stable.
    - For val_loader and test_loader, shuffle=False because the order of the data doesn't matter, and the shuffling will only take unnecessary time.
  - num_workers=2:
    - Specifies the number of subprocesses used to load the data.
    - By using multiple workers, data can be loaded in parallel with the GPU's training process, which avoids the GPU having to wait for data from the CPU.

Train model: (Use both for CNN and MLP to compare them)



```python
# Train model
def train_model(model, train_loader, val_loader, num_epochs=50, model_name='Model'):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-4)
    # OneCycleLR scheduler
    scheduler = OneCycleLR(optimizer, max_lr=0.01, steps_per_epoch=len(train_loader), epochs=num_epochs)

    train_counter = []
    train_losses = []
    train_accs = []

    val_counter = []
    val_losses= []
    val_accs = []

    # Early Stopping parameters
    best_val_acc = -1.0
    epochs_no_improve = 0
    limit_epoch = 10 # The limited epoches in which the epoches have no improvement compared with the previous one
```

- **Num_epochs=50**:
  - Enough time to converge:
    - Deep Learning models, especially CNNs or MLPs with multiple layers and standardized techniques such as Batch Normalization, Dropout, and Data Augmentation.
    - 50 epochs is usually enough for the model to learn the features and achieve good performance on many medium-sized datasets such as CIFAR-10.
    - With techniques such as OneCycleLR, the learning rate changes throughout the training process, and stretching this process over 50 epochs allows the scheduler enough time to complete the cycle, making the model converge efficiently.
  - With Early Stopping:
    - When setting num_epochs = 50 (or a larger number), it is not necessary for the model to run all 50 epochs.

    * Instead, num_epochs acts as an *upper bound*. The aim is to ensure the model has *enough maximum time* to learn and achieve the best performance.
    * Early Stopping (limit_epoch = 10) will be the mechanism that determines when to stop training.
    * For example, if the model reaches its best performance on its validation set at epoch 30, and then does not improve in the next 10 epochs, then the training will automatically stop at epoch 40.
    * If setting the num_epochs too low (e.g., 15-20), it's possible that the model hasn't reached its best performance and has run out of allowed epochs, leading to underfitting.
  - Calculation cost:
    * 50 epochs is a "moderate" number in terms of computational cost.
    * It's not too little to underfit the pattern, and also not so much as to waste resources if the Early Stopping is properly established.

- **criterion = nn.CrossEntropyLoss()**:
  - The loss function is used to measure the difference between the model's prediction and the actual label.
  - CrossEntropyLoss is the standard choice for multi-layer classification problems. It combines LogSoftmax and NLLLoss (Negative Log Likelihood Loss) in a single function.

- **optimizer=optim.AdamW(model.parameters(),lr=0.001,weight_decay=1e-4)**:
  - The optimization algorithm is used to update the model's weights.
  - model.parameters(): Passes all the model's trainable parameters to the optimizer.
  - lr=0.001: Initial learning rate. This is the factor that determines how big or small the weighting update step is in each iteration.
  - weight_decay=1e-4: A form of L2 regularization. Adding a term to the proportional loss function to the square of the weight, encourages smaller weights and helps prevent overfitting.

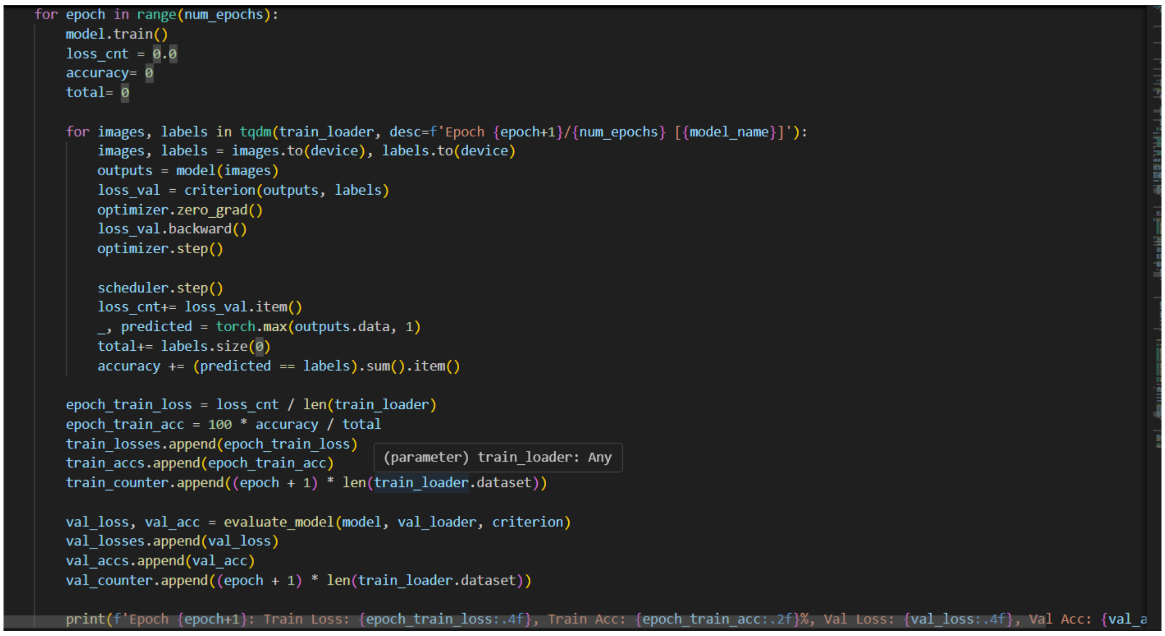
- **scheduler=OneCycleLR(optimizer,max_lr=0.01,steps_per_epoch=len(train_loader), epochs=num_epochs)**:
  - A very effective learning rate scheduler, which automatically adjusts the learning rate of the optimizer.
  - max_lr=0.01: The maximum learning rate that the scheduler will reach during its cycle.
  - steps_per_epoch=len(train_loader): Number of steps (number of batches) in each epoch.
  - The scheduler needs to know this in order to correctly distribute the learning rate changes throughout the epochs.
  - epochs=num_epochs: The total number of epochs that the scheduler will run through its cycle.
  - OneCycleLR usually increases the learning rate from a low value to max_lr and then gradually decreases to a very low value (usually 0) throughout the training process.

  - Helps the model explore a wider parametric space at high learning rates and then converge more accurately at low learning speeds, resulting in better results and faster convergence.

- **train_counter, val_counter**: Stores the number of training patterns that have been seen (on the X-axis of the graph).(Use to plot learning curves)

- **Initializing the Early Stopping Parameters**

  - best_val_acc = -1.0: Stores the best validation accuracy that the model has ever achieved.
  - Initialize with a negative value to ensure that any actual accuracy will be "better".
  - epochs_no_improve = 0: Counts the number of consecutive epochs where the validation accuracy does not improve compared to best_val_acc.
  - limit_epoch = 10: This is the tolerance threshold. If epochs_no_improve reaches this value, the training will stop early to avoid overfitting.



```
for epoch in range(num_epochs):
    model.train()
    loss_cnt = 0.0
    accuracy= 0
    total= 0

    for images, labels in tqdm(train_loader, desc=f'Epoch {epoch+1}/{num_epochs} [{model_name}]'):
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss_val = criterion(outputs, labels)
        optimizer.zero_grad()
        loss_val.backward()
        optimizer.step()

        scheduler.step()
        loss_cnt+= loss_val.item()
        _, predicted = torch.max(outputs.data, 1)
        total+= labels.size(0)
        accuracy += (predicted == labels).sum().item()

    epoch_train_loss = loss_cnt / len(train_loader)
    epoch_train_acc = 100 * accuracy / total
    train_losses.append(epoch_train_loss)
    train_accs.append(epoch_train_acc)        (parameter) train_loader: Any
    train_counter.append((epoch + 1) * len(train_loader.dataset))

    val_loss, val_acc = evaluate_model(model, val_loader, criterion)
    val_losses.append(val_loss)
    val_accs.append(val_acc)
    val_counter.append((epoch + 1) * len(train_loader.dataset))

    print(f'Epoch {epoch+1}: Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.2f}%, Val Loss: {val_loss:.4f}, Val Acc: {val_a
```

- **The main loop of the training process (for epoch in range(num_epochs):)**

- model.train(): Puts the model into training mode, enabling features such as Dropout and Batch Normalization.

- loss_cnt, accuracy, total: These variables are reinitialized at the beginning of each epoch to accumulate the total loss, the total number of correct predictions, and the total number of samples in that epoch.

- **Batch Loop (for images, labels in tqdm(train_loader, ...):)**

- tqdm(...): Displays the progress bar in the console, tracking each batch.

- images, labels = images.to(device), labels.to(device): Passes the image tensors and labels of the current batch to the selected compute device (GPU or CPU).

- **outputs = model(images)**

  - The data travels through the neural network. Tensor images are included in the first layer of the model.
  - Each layer in the model (e.g., Conv2d, BatchNorm2d, ReLU, MaxPool2d, Linear) will perform its calculations on the input from the previous layer, until the data reaches the final output layer.
  - outputs will be "logits" (raw numerical values, not normalized to probability) for each output class.
  - For CIFAR-10, there are 10 layers, so the output will have 10 values for each image in the batch.
  - images (input): tensor with shape (256, 3, 32, 32), device='cuda'.
  - After model(images):
    * outputs: tensor with shape(256, 10) (batch_size, num_classes), containing logits (e.g. [1.2, -0.5, 3.1, ..., 0.8]), device='cuda'

- **loss_val = criterion(outputs, labels)**
  - The loss function calculates a unique numeric value loss_val represent how "wrong" the model's prediction is for the current batch.
  - The lower the loss_val value, the better.
  - Example:
    * outputs: tensor with shape (256, 10), device='cuda:0'.
    * labels: tensor with shape(256,), device='cuda:0' (e.g. [3, 7, 0, ..., 5]).
  - After criterion(outputs, labels):
    * loss_val: tensor has shape() (scalar - a single value), device='cuda:0' (e.g. 0.7853).

- **optimizer.zero_grad()**
  - Before calculating the gradient for the current batch, we need to "zero out" the gradients that have been accumulated from the previous training batch.
  - In PyTorch, gradients are accumulated by default. If we don't call zero_grad(), the gradients from previous batches will add up to the gradient of the current batch, resulting in skewed weighted update steps and inefficient learning.

- **loss_val.backward()**
  - This is the "backpropagation" step. PyTorch will use chain rules to automatically calculate loss_val gradients for *all trainable parameters* in your model (e.g., weights and biases in nn classes. Linear and nn. Conv2d).
  - These gradients are stored in the .grad attribute of each parameter.

- **optimizer.step()**
  - After the gradients of all parameters have been calculated (loss_val.backward()), optimizer.step() will use the selected optimization algorithm (AdamW) to adjust the parameters of the model based on those gradients and the current learning rate.
  - This is where the model's weights are actually updated to reduce the loss function.
  - Examples:

    - Assuming that model.fc1.weight is originally tensor([[1.0, 2.0], ...]).
    - And model.fc1.weight.grad is tensor([[0.01, -0.05], ...]).
  - Or optimizer.step():
    - model.fc1.weight will be updated (e.g. tensor([[0.999, 2.005], ...]), depending on the learning rate and the AdamW algorithm).

- **scheduler.step()**
  - Update the learning rate of the optimizer according to the policy defined in the scheduler (OneCycleLR).
  - For OneCycleLR, this step *must be invoked after each batch* so that the learning rate changes smoothly cyclically throughout the epochs.

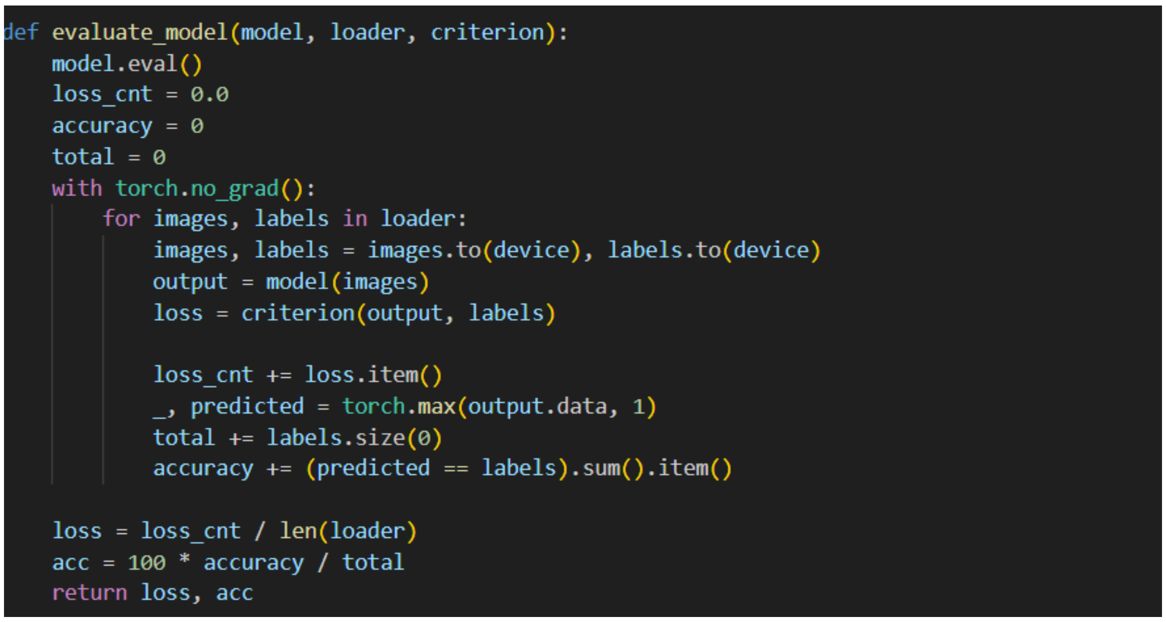
- **loss_cnt += loss_val.item()**
  - Add the loss scalar value of the current batch to the loss_cnt variable, which will sum up the total loss of the entire epoch.

- **_, predicted = torch.max(outputs.data, 1)**
  - Get the highest probability class prediction for each sample in the batch.
  - outputs.data: Access tensor data without gradient tracking (like torch.no_grad()).
  - torch.max(..., 1): The max function returns a tuple (the largest value, the index of the largest value).
  - 1 is the dimension to find the max (dimension of the layers).
  - Examples:
    - outputs (e.g., 3 patterns): tensor([[1.2, -0.5, 3.1], [0.1, 0.9, 0.2], [-0.3, 0.4, 0.1]])
    - predicted: tensor([2, 1, 1]) (the index of the largest value per row)

- **total += labels.size(0)**
  - Add the number of samples in the current batch (for example, 256) to the total variable, summing up the total number of samples processed in the epoch.

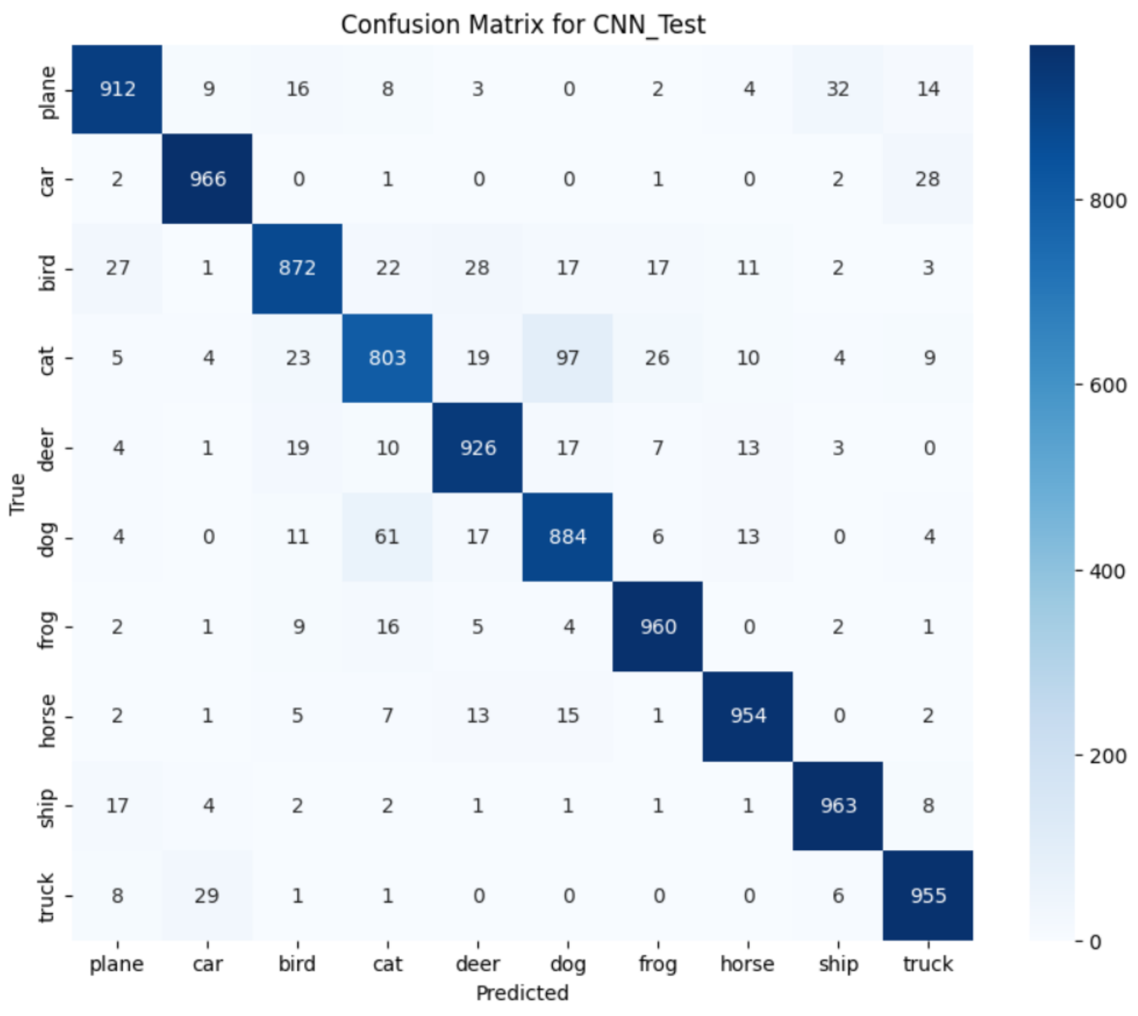
- **accuracy += (predicted == labels).sum().item()**
  - Compare the predicted tensor with the actual tensor labels.
  - (predicted == labels): Creates a boolean tensor, where True if the prediction is correct, False if false.
  - .sum(): Counts the number of True (i.e., the number of correct predictions in the batch).
  - .item(): Takes the Python numeric value.
  - This value is added to the accuracy variable, which sums up the total number of correct predictions in the epoch.
  - Example :
    - predicted: tensor([2, 1, 1])
    - labels: tensor([2, 0, 1])
    - (predicted == labels): tensor([True, False, True])

- **epoch_train_loss = loss_cnt / len(train_loader)**:
  - loss_cnt has accumulated the total loss function of all batches in the current epoch.
  - len(train_loader) is the total number of batches in train_loader. Therefore, the *average loss* of the model over the entire training set for this epoch will be calculated.
  - Provides a stable value and represents training performance throughout the epoch, eliminating interference from individual batches.
- **epoch_train_acc = 100 * accuracy / total**:
  - Accuracy has accumulated the total number of correct predictions, and Total is the total number of samples processed in the epoch.
  - Calculate *the average accuracy (%)* of the model over the entire training set for this epoch.
  - Provides an overview of how good the model is at classifying training patterns in epochs.
- **train_losses.append(epoch_train_loss),train_accs.append(epoch_train_acc)**:
  - These lists will be used to plot the Loss and Accuracy lines of the training session on the learning chart later.
- **train_counter.append((epoch + 1) * len(train_loader.dataset))**:
  - Calculate the total number of training patterns that the model has "seen" up to the end of the current epoch.
  - Provides points on the X-axis of the learning graph, showing the training progress by sample number.
- **val_loss, val_acc = evaluate_model(model, val_loader, criterion)**:
  - Call the evaluate_model function to calculate the loss and accuracy of the model across the entire validation set.



```
def evaluate_model(model, loader, criterion):
    model.eval()
    loss_cnt = 0.0
    accuracy = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            output = model(images)
            loss = criterion(output, labels)

            loss_cnt += loss.item()
            _, predicted = torch.max(output.data, 1)
            total += labels.size(0)
            accuracy += (predicted == labels).sum().item()

    loss = loss_cnt / len(loader)
    acc = 100 * accuracy / total
    return loss, acc
```

- **Disable gradient calculation (with torch.no_grad():)**
  - This is a context manager in PyTorch.
  - Any calculations performed inside the block with torch.no_grad(): will *not compute and store the gradient.*
  - During the evaluation, there is no need to update the model's weights, so there is no need for gradients.
  - Disabling gradient calculation helps:
    - Memory Saving: No need to store intermediate values for the backward pass.
    - Increased speed: Calculations run faster because there is no overhead associated with building gradient computational graphs.
  - Evaluate the model's generalization ability on data that it has never seen during training.
  - This value is the primary measure for detecting overfitting.
- **val_losses.append(val_loss)/val_accs.append(val_acc)/val_counter.append(...)**:
  - These lists will be used to plot the Loss and Accuracy representations of the validation set on the learning chart.



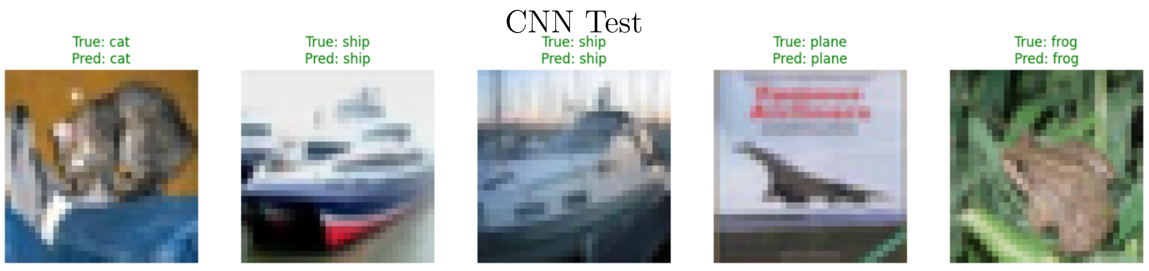
	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	912	9	16	8	3	0	2	4	32	14
car	2	966	0	1	0	0	1	0	2	28
bird	27	1	872	22	28	17	17	11	2	3
cat	5	4	23	803	19	97	26	10	4	9
deer	4	1	19	10	926	17	7	13	3	0
dog	4	0	11	61	17	884	6	13	0	4
frog	2	1	9	16	5	4	960	0	2	1
horse	2	1	5	7	13	15	1	954	0	2
ship	17	4	2	2	1	1	1	1	963	8
truck	8	29	1	1	0	0	0	0	6	955

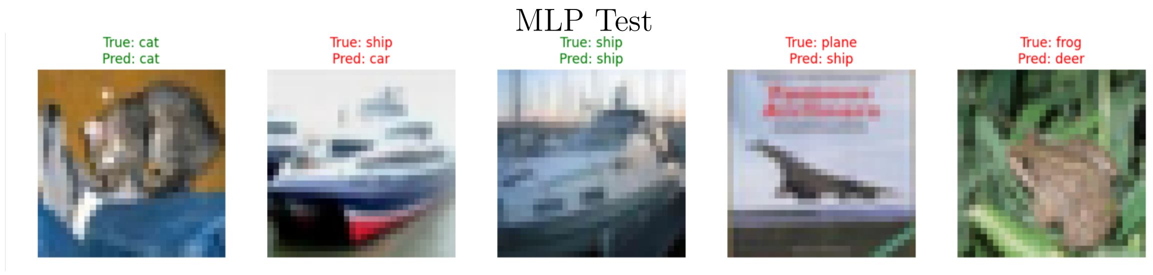
Evaluate on Test_set and plot confusion matrix:

For CNN (similar to MLP):

- After training and validation parts, we need to evaluate the final performance of the model on a completely unseen dataset (test_loader) and plot confusion matrix.
- **confusion_matrix** is a function from the sklearn.metrics library.
- It takes two main inputs: all_labels (true labels) and all_preds (predictive labels).
- This function calculates a square matrix where each row (i) represents cases that are actually labeled as i, and each column (j) represents the cases predicted by the model as j.
- The element cm[i, j] will contain the number of samples that are actually labeled i but are predicted by the model to be j.
- The cells on the main diagonal show the number of samples that the model correctly predicted for each layer.
- Ex: plane: 912, car: 966,...
- The cells outside the main diagonal show the number of samples that were *incorrectly* predicted.
- Ex: True: bird, Predicted: plane: There are 27 photos of birds that were mistaken for airplanes by the model.

Predict image from CIFAR 10:





III. Compare and discuss the results of 2 neural network:

Training process:

- MLP is slightly faster than CNN because it doesn't have complex convolutional calculations.
- At the start, CNN learned and generalized better from the first epoch (Val Acc 48.76% vs. MLP's 34.70%), showing superior characteristic extraction from the start.
- CNN increased steadily in the following epochs, continuing to improve strongly.
- CNN shows impressive and sustained performance growth with each epoch, achieving very high accuracy.
- MLP also improved, but to a much lesser extent and soon saturated in absolute performance.
- Overfitting: Both are well regulated thanks to Batch Normalization, Dropout, both CNNs and MLPs avoid severe overfitting.
- However, with MLP, instead of showing signs of overfitting, it can be seen that MLP is underfitting.
- After 50 epochs, it can be seen that CNN is superior, achieving a validation accuracy that is 38% higher than MLP (91% vs 48%).
- This is a clear indication that CNNs have a much stronger generalization ability on image data than MLPs.

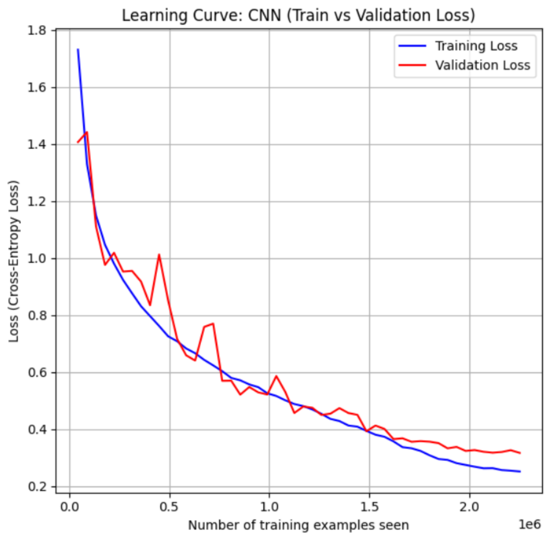
Learning Curves:

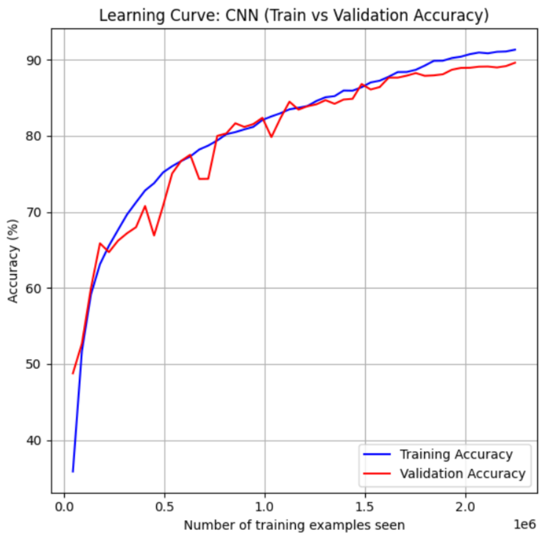
Curve shape (smoothness):

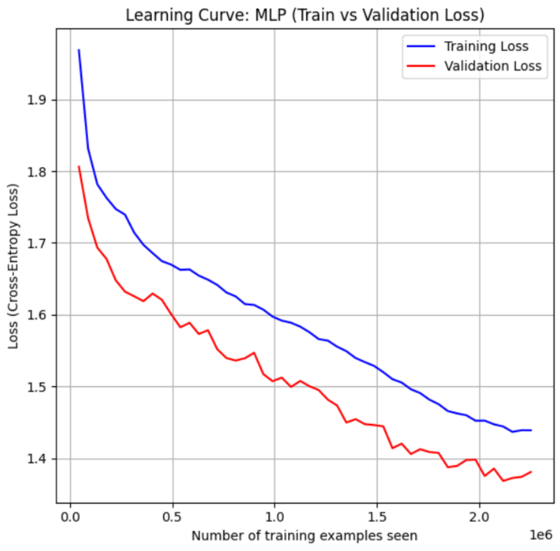
- CNN: Both the Training Loss/Acc and Validation Loss/Acc lines are extremely smooth, with no large fluctuations in batches.
- MLP : Similarly, the curves are also very smooth.
- Comment: Changing the way metrics are recorded to average per epoch has been a great success, eliminating noise and making the chart much easier to read.

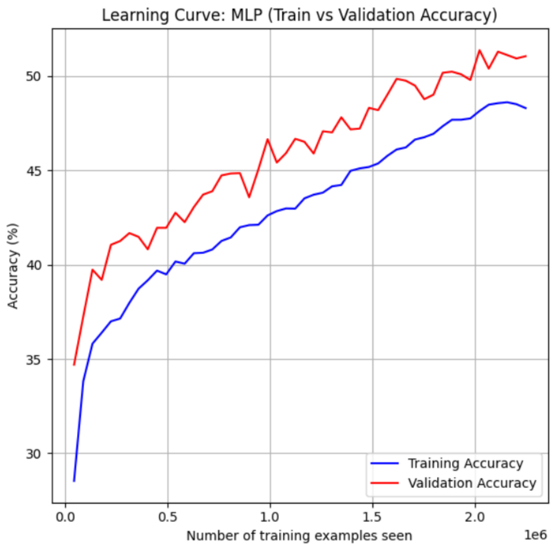
Trend and Gap:

- **CNN**:
  - Loss: Training Loss and Validation Loss both decreased sharply and continuously, maintaining a small and stable distance, moving towards very low levels.
  - Accuracy: Training Accuracy and Validation Accuracy increase steadily, maintaining a small and stable distance (Train Acc is about 1-2% higher than Val Acc).
  - Both reached very high levels (almost 90%).
  - CNN's graph shows an *efficient, stable, and well-generalized learning process.*
  - The model is learning strong characteristics and is not severely overfitting.





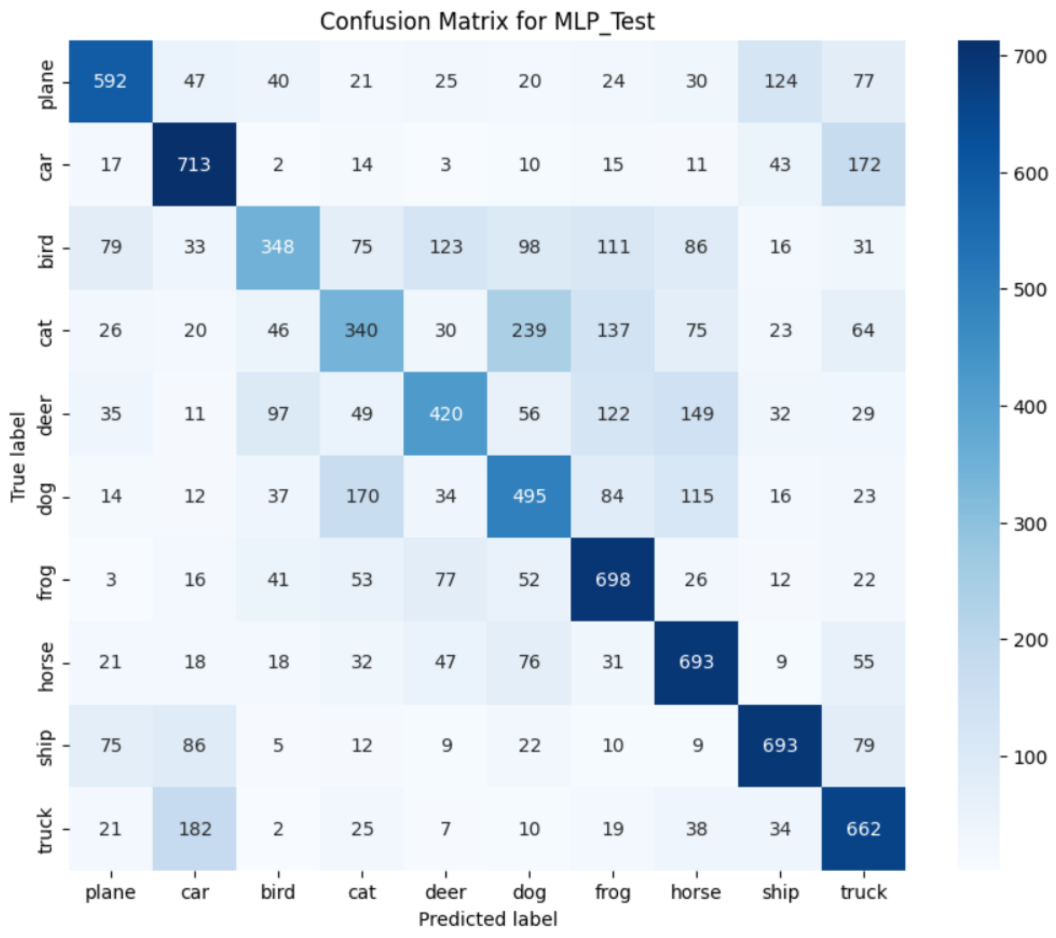




- **MLP**:
  - Loss: Training Loss and Validation Loss gradually decrease, maintaining a small gap.
  - Accuracy: Training Accuracy and Validation Accuracy increase, maintaining a small gap.
  - However, both have slowed down at very low levels (less than 55%).
  - MLP's graph shows that the model is learning *steadily*, and is not heavily overfitting.
  - However, it is *underfitting* – the model is not strong enough to learn complex patterns in the data, resulting in low performance on both the training and validation sets.

Confusion Matrix:

- **CNN**:
  - Main diagonal: Very high values (e.g., plane 912, car 966, frog 960, ship 963, truck 955), indicating high accuracy for most grades.
- **MLP**:



	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	592	47	40	21	25	20	24	30	124	77
car	17	713	2	14	3	10	15	11	43	172
bird	79	33	348	75	123	98	111	86	16	31
cat	26	20	46	340	30	239	137	75	23	64
deer	35	11	97	49	420	56	122	149	32	29
dog	14	12	37	170	34	495	84	115	16	23
frog	3	16	41	53	77	52	698	26	12	22
horse	21	18	18	32	47	76	31	693	9	55
ship	75	86	5	12	9	22	10	9	693	79
truck	21	182	2	25	7	10	19	38	34	662

  - Main diagonal: Significantly lower values (e.g., plane 470, car 713, bird 467, cat 467, dog 595).

Conclusion: For CIFAR-10, the CNN model shows superior performance (91.95 % Test Accuracy) compared to MLP (56.54% Test Accuracy). This confirms that CNN is a more suitable and effective architecture for image classification tasks due to its ability to learn spatial features, although both are optimized using modern techniques.