Posts and Telecommunications
Institute of Technology

Report on Project Assignment 2

PYTHON PROGRAMMING LANGUAGE

Supervisor: Kim Ngoc Bach
Student: Chien Dang(B23DCCE012)
Student: Hieu Vu(B23DCCE036)
Class: D23CQCE06-B
Academic Term: 2023 – 2028

TABLE OF CONTENTS

TASK 1

1.Objectives

Build a 3-layer MLP (Multi-Layer Perceptron) network using PyTorch to process CIFAR-10 data, comprising:

Input layer: Accepts flattened image input (32x32x3 → 3072 features).

Hidden layer: A hidden layer with an optional size (e.g., 512 neurons) + ReLU activation.

Output layer: An output layer with 10 neurons (corresponding to the 10 classes of CIFAR-10).

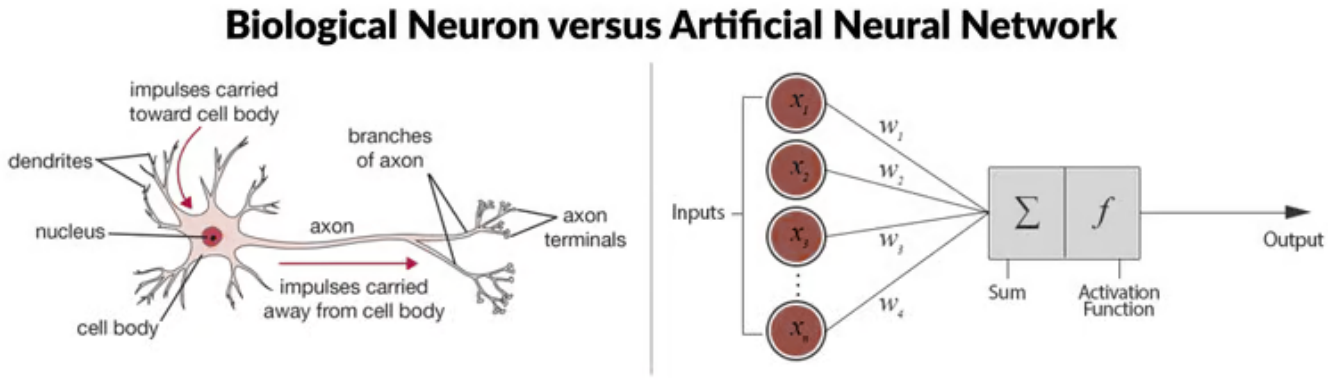
Note: In this task, we will build 5 layers to increase accuracy.

2.Basic Knowledge

Introduction to Neural Networks and Multilayer Perceptron (MLP)

An Artificial Neural Network (ANN) is a machine learning model inspired by the structure and function of neural networks in the human brain. An ANN consists of artificial neurons organized into layers, where information is transmitted from the input layer, through hidden layers, to the output layer. Each neuron receives input signals, performs a calculation with an activation function, and transmits the output signal to other neurons.

A Multilayer Perceptron (MLP) is a type of feedforward neural network, consisting of multiple fully connected layers of neurons with a nonlinear activation function. MLPs have the ability to learn nonlinear relationships in data, helping to solve complex problems such as classification, regression, and pattern recognition.



Biological neuron vs. artificial neural network (Source: **ResearchGate**)

Components of a Multilayer Perceptron (MLP)

Input Layer

- Function: Receives input data, where each neuron represents a feature or dimension of the data.
- Characteristics: Does not perform calculations, only transmits the input value to the first hidden layer.
- Number of Neurons: Depends on the dimensionality of the input data.

Hidden Layers

- Function: Performs calculations on the input data through weighted connections (weights).
- Mechanism:
  - Each neuron in the hidden layer receives input from all neurons of the previous layer, multiplied by the corresponding weight, plus a bias, and applies an activation function.

TASK 1

Calculation Formula: $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$ Where:

w_i: Weight of the i^{th} connection.

x_i: i^{th} input value.

b: Bias of the neuron.

z: Weighted sum, which is passed through an activation function f(z) to create the output.

Number of Layers and Neurons: These are hyperparameters that need to be determined during the model design process.

Output Layer

Function: Generates the final prediction or output of the network.

Number of Neurons: Depends on the problem:

Binary classification: Usually has 1 or 2 neurons.

Multi-class classification: The number of neurons is equal to the number of classes.

Regression: Usually has 1 neuron.

Activation Function: Often different from the activation function in the hidden layers, e.g., softmax for multi-class classification or a linear function for regression.

Weights and Bias

Weights: Determine the degree of influence of one neuron on another. Each connection between neurons has a weight, which is learned during training.

Bias: A value added to the weighted sum of a neuron, which helps adjust the activation threshold. Bias is also learned during training.

Activation Functions

Role: Introduces non-linearity into the network, allowing the MLP to learn complex patterns.

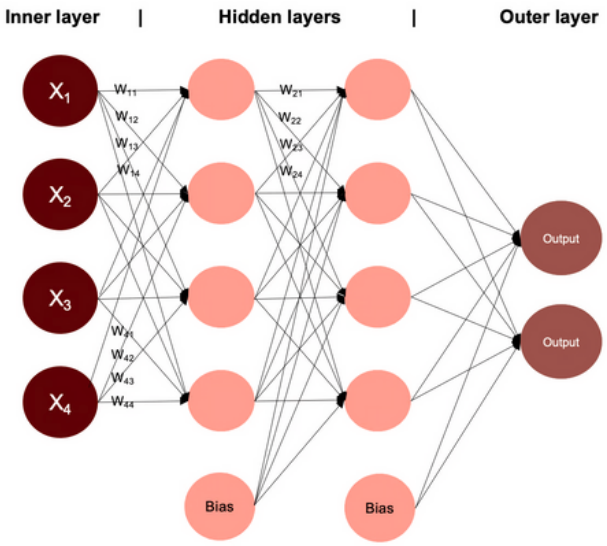
Common Types:

Sigmoid: Maps the output to the range (0, 1), suitable for binary classification.

Tanh: Maps the output to the range (-1, 1).

ReLU (Rectified Linear Unit): The output is 0 if the input is negative, or remains the same if positive, helping to speed up training.

Softmax: Used for multi-class classification, generating probabilities for each class.



Example of an MLP with two hidden layers. Image by Author

TASK 1

3.Flow chart

- INITIALIZATION
- DATA PREPARATION
- MODEL BUILDING
- TRAINING SETUP
- TRAINING LOOP
- EVALUATION AND RESULT DISPLAY

4.Source Code

https://github.com/Hieuvu4438/Python-Assignment-02/blob/main/SOURCE%20CODE/MLP.ipynb

5.Explain the code

Libraries Used

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

PyTorch (torch)

Role: Main framework for the entire model building and training process.

Key Modules Used:

torch.device: Manages the computational device (CPU/GPU).

torch.save: Saves the model.

torch.max: Finds the maximum value (used to calculate accuracy).

TASK 1

Torch Neural Network (torch.nn)

Role: Builds the neural network architecture.

Key Components:

- nn.Sequential: Container for network layers.
- nn.Linear: Fully-connected layer.
- nn.LeakyReLU: Activation function.
- nn.Dropout: Regularization technique.
- nn.Flatten: Converts a tensor into a vector.

Torch Optimizers (torch.optim)

Role: Updates model weights.

Components:

- optim.Adam: Adam optimizer.
- optim.lr_scheduler: Dynamically adjusts the learning rate.

TorchVision

Role: Supports loading and processing image datasets.

Key Functions:

- datasets.CIFAR10: Loads the CIFAR-10 dataset.
- transforms: Image transformation pipeline.
- utils.make_grid: Displays a batch of images.

Matplotlib (matplotlib.pyplot)

Role: Visualizes results.

Important Functions:

- plt.plot(): Plots loss/accuracy graphs.
- plt.imshow(): Displays images.
- plt.figure(): Creates a new figure.

NumPy (numpy)

Role: Processes tensors and converts data.

Specific Application:

Converts PyTorch tensors to NumPy arrays for image display.

Manipulates multi-dimensional arrays.

Device Setup

```
device = torch.device("cuda:0" if torch.cuda.is_available()
    else "cpu")
```

Automatically detects and utilizes the GPU if available; otherwise, utilizes the CPU.

Ensures computations are performed on the most powerful available hardware.

Data Transformations

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247,
        0.243, 0.261))
])
```

TASK 1

Data augmentation:

RandomCrop: Randomly crops the image to 32x32 pixels with 4-pixel padding for increased diversity.

RandomHorizontalFlip: Randomly flips the image horizontally.

Normalization:

Converts the image to a tensor.

Normalizes with mean and standard deviation specific to CIFAR-10.

```
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247,
        0.243, 0.261))
])
```

Data augmentation is not applied to the test set to ensure accurate evaluation.

Data Loading

trainset=torchvision.datasets.CIFAR10(root='./data',train=True,download=True,transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

Batch size of 128: Balances speed and memory usage.
Shuffles training data to prevent order from influencing learning.
Utilizes 2 workers to load data in parallel.

Model Architecture: MLP

MLP Layers

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(32 * 32 * 3, 512),
            nn.LeakyReLU(0.1),
            nn.Dropout(0.3),
            nn.Linear(512, 512),
            nn.LeakyReLU(0.1),
            nn.Dropout(0.3),
            nn.Linear(512, 512),
            nn.LeakyReLU(0.1),
            nn.Dropout(0.3),
            nn.Linear(512, 512),
```

TASK 1

```
            nn.LeakyReLU(0.1),
            nn.Dropout(0.3),
            nn.Linear(512, 512),
            nn.LeakyReLU(0.1),
            nn.Dropout(0.3),
            nn.Linear(512, 10)
        )
```

Flatten: Transforms the 3x32x32 image into a 3072-dimensional vector.
5 hidden layers with 512 units each.
LeakyReLU activation (slope 0.1) is used instead of standard ReLU to prevent "dying neurons."
Dropout of 0.3 to reduce overfitting.
Weights initialized with He initialization (kaiming_uniform_), which is suitable for ReLU/LeakyReLU activations.

Model Verification

```
input_tensor = torch.rand(5, 3, 32, 32).to(device)
output = model(input_tensor)
```

Forward pass verified with a dummy input.
Confirmed that the output has the correct shape (5x10 for 5 images and 10 classes).

Model Training

Training Setup

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001,
    weight_decay=1e-4)
scheduler=optim.lr_scheduler.ReduceLROnPlateau(optimizer,
    mode='max',factor=0.1, patience=10)
```

Loss Function: CrossEntropy, suitable for classification problems.
Optimizer: Adam with a learning rate of 0.001 and weight decay of 1e-4 for regularization.
Scheduler: Reduces the learning rate when accuracy does not improve after 10 epochs.

Training Process

```
for epoch in range(max_epoch):
    # Training phase
    model.train()
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # Forward, backward, optimize
        # ...

    # Evaluation phase
    test_loss, test_accuracy = evaluate(model, testloader,
        criterion)
```

TASK 1

```
    # Early stopping
    if test_accuracy > best_test_acc:
        best_test_acc = test_accuracy
        patience_counter = 0
        torch.save(model.state_dict(), 'best_model.pth')
    else:
        patience_counter += 1
        if patience_counter >= patience:
            break
```

Early Stopping: Training is halted if test accuracy does not improve after 20 epochs.
Best Model Saving: Based on the highest test accuracy achieved.
Model Evaluation

```
def evaluate(model, testloader, criterion):
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
            images, labels = images.to(device), labels.to(
                device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    test_loss = test_loss / len(testloader)
    return test_loss, accuracy
```

eval() mode is used to disable dropout/batch normalization layers.
no_grad() context is used to disable gradient calculation for memory efficiency.

Result Visualization

```
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
# ...
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
```

Plots graphs tracking loss and accuracy across epochs.
Aids in analyzing the learning process and detecting overfitting/underfitting.

TASK 2

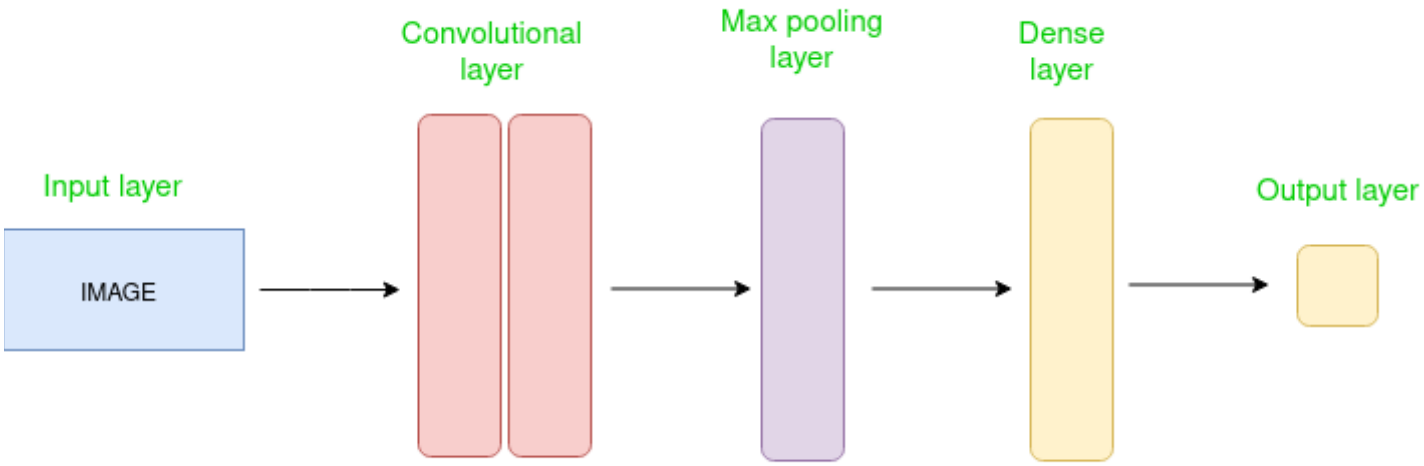
1.Objectives

Build a 22-layer CNN using PyTorch for CIFAR-10 classification, including:
Input layer: 32x32x3 RGB images.
Convolutional blocks: Multiple 3x3 conv layers + BatchNorm + ReLU + MaxPooling.
Output layer: 10 neurons with Softmax.
Goal: Leverage spatial hierarchies to outperform MLP in accuracy.

2.Basic Knowledge

Introduction to Convolution Neural Network

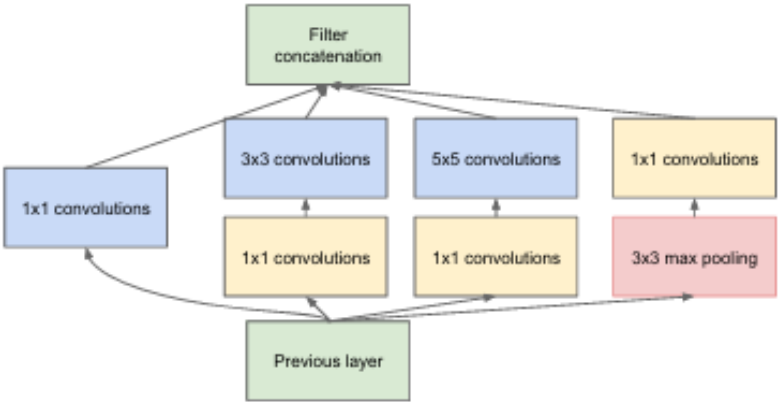
Convolutional Neural Network (CNN) is an advanced version of artificial neural networks (ANNs), primarily designed to extract features from grid-like matrix datasets. This is particularly useful for visual datasets such as images or videos, where data patterns play a crucial role. CNNs are widely used in computer vision applications due to their effectiveness in processing visual data.
CNNs consist of multiple layers like the input layer, Convolutional layer, pooling layer, and fully connected layers. Let's learn more about CNNs in detail.



Simple CNN architecture

Introduction to Inception Modules

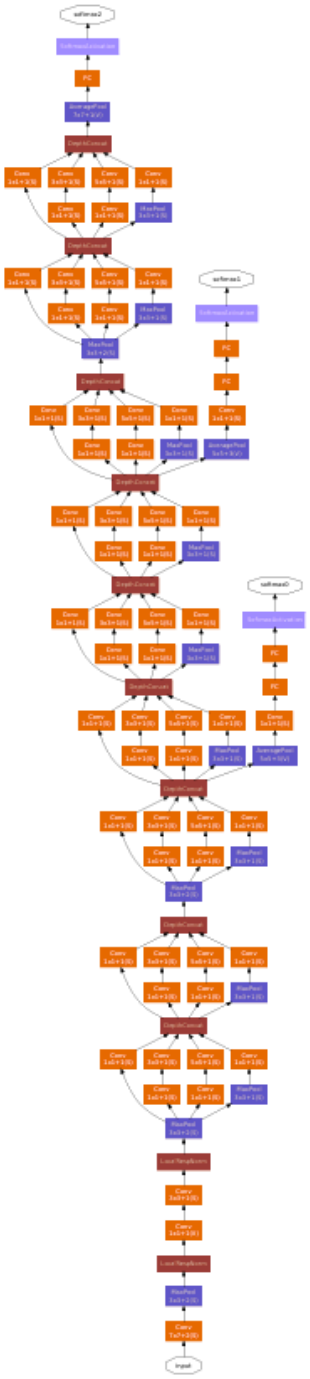
Inception is a family of convolutional neural network (CNN) for computer vision, introduced by researchers at Google in 2014 as GoogLeNet (later renamed Inception v1). The series was historically important as an early CNN that separates the stem (data ingest), body (data processing), and head (prediction), an architectural design that persists in all modern CNN.



Inception module with dimension reductions

TASK 2

GoogleNet Model



GoogLeNet network with all the bells and whistles

TASK 2

The GoogleNet model (also known as Inception) is an advanced convolutional neural network (CNN) architecture developed by Google. Introduced in a 2014 paper, GoogleNet stands out for its use of Inception modules, which allow for the parallel processing of different filter sizes, enhancing computational efficiency and feature extraction capabilities. This model won the ImageNet 2014 competition and marked a significant turning point in the field of deep learning, particularly in image recognition.

GoogleNet (Inception v1)

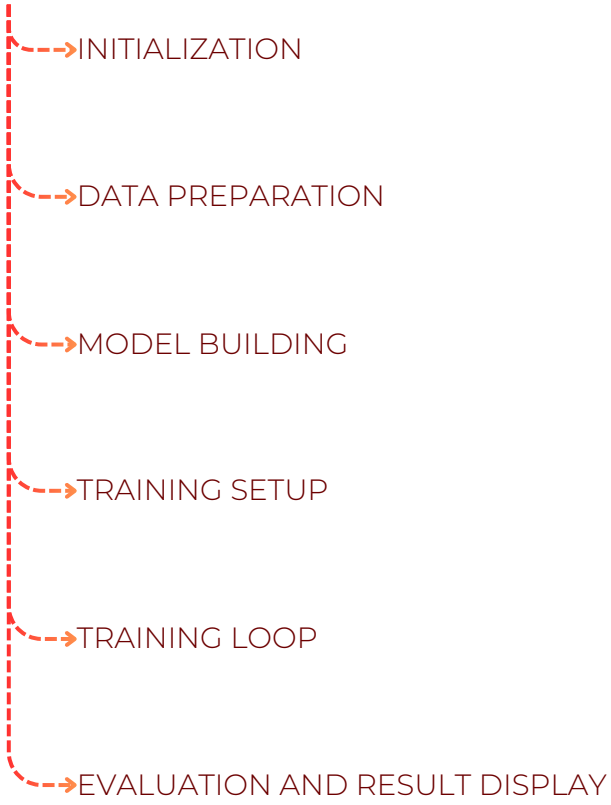
Number of Layers: 22 layers (including convolution, pooling, and Inception modules).
Number of Parameters: ~6.8 million (fewer than AlexNet and VGG due to optimized design).
Key Architecture:

- Inception Module: Combines multiple Convs (1x1, 3x3, 5x5) and Max Pooling in parallel → learns multi-scale features.
- 1x1 Convs: Reduces depth before using larger Convs → saves parameters.
- Global Average Pooling: Replaces FC layers → reduces overfitting.
- 2 Auxiliary Classifiers: Helps avoid vanishing gradients during training.

Advantages: High performance, fewer parameters, suitable for real-world deployment.

TASK 2

3.Flow chart



4.Source Code

https://github.com/Hieuvu4438/Python-Assignment-02/blob/main/SOURCE%20CODE/CNN.ipynb

5.Explain the code

GPU Environment Setup

```
import os
os.environ['CUDA_VISIBLE_DEVICES'] = '2'
```

Purpose: Specifies that the GPU with ID 2 will be used to run the program.

Explanation:

CUDA_VISIBLE_DEVICES is an environment variable used to select a specific GPU when multiple GPUs are present on the system.

Here, GPU number 2 is selected (numbering starts from 0).

TASK 2

Import Libraries

```
import torch
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader
from torch.optim import Adam
import torchvision
import torchvision.models as models
from torchsummary import summary
```

Explanation:

torch: PyTorch's main library for working with tensors and neural networks.

torchvision.transforms: Provides data transformation operations such as resize, normalize, etc.

torch.nn: Module containing the layers for building neural networks (e.g., nn.Conv2d, nn.Linear).

torch.nn.functional (F): Contains activation functions (ReLU, sigmoid), loss functions, etc.

numpy (np): Numerical computation library.

matplotlib.pyplot (plt): Used for plotting graphs and displaying images.

CIFAR10: A 10-class color image dataset from torchvision.

DataLoader: Helps load data in batches, supports shuffling and parallelization.

Adam: A popular optimizer for model optimization.

torchvision.models: Contains pre-trained models (ResNet, VGG, etc.).

summary: Prints the model architecture and number of parameters.

Initialize Random Seed and Device

```
torch.manual_seed(1)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Explanation:

torch.manual_seed(1): Sets the seed for the random number generator to ensure reproducible results.

device: Checks if a GPU (CUDA) is available. If so, uses 'cuda'; otherwise, uses 'cpu'.

TASK 2

Batch Size Setup

batch_size = 256

Purpose: Defines the number of data samples processed simultaneously in a single iteration (batch).

Explanation:

A large batch size (256) helps accelerate training but requires more GPU memory.

A suitable value is typically chosen based on GPU memory capacity.

Define Transformations (Transforms)

```
train_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2470,
        0.2435, 0.2616]),
])
```

For the Training Set (train_transform)

Explanation:

ToTensor(): Converts images from PIL Image or NumPy array to PyTorch tensors (with pixel values in the range [0, 1]).

Normalize(mean, std): Normalizes the image tensor by subtracting the mean and dividing by the standard deviation for each color channel (RGB).

Mean: [0.4914, 0.4822, 0.4465] (average values of CIFAR-10 across the 3 channels R, G, B).

Std: [0.2470, 0.2435, 0.2616] (standard deviation of CIFAR-10 across the 3 channels).

For the Validation Set (val_transform)

```
val_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2470,
        0.2435, 0.2616]),
])
```

Note: The validation set uses the same normalization as the training set to ensure consistency.

Load CIFAR-10 Data

Training Set

```
train_set = CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=train_transform
)
```

Parameters:

root='./data': Directory to store the data.

TASK 2

train=True: Loads the training set (50,000 images).

download=True: Automatically downloads the data if it's not already present.

transform=train_transform: Applies the defined transformations.

Validation/Test Set

```
val_set = CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=val_transform
)
```

Differences:

train=False: Loads the test set (10,000 images).

Uses val_transform instead of train_transform (although they are the same in this case).

Create DataLoaders

For the Training Set

```
trainloader = DataLoader(
    train_set,
    batch_size=batch_size,
    shuffle=True,
    num_workers=4,
)
```

Parameters:

shuffle=True: Shuffles the data after each epoch → prevents overfitting.

num_workers=4: Uses 4 processes to load data in parallel → increases speed.

For the Test Set

```
testloader = DataLoader(
    val_set,
    batch_size=batch_size,
    shuffle=False,
    num_workers=4,
)
```

Differences:

shuffle=False: Does not shuffle the test data to evaluate the model consistently.

TASK 2

Inception Layer

This is the fundamental building block of GoogLeNet, combining multiple convolution operations with different kernel sizes to capture multi-scale information.

Inception Layer Structure:

```
class Inception(nn.Module):
    def __init__(self, in_planes, n1x1, n3x3red, n3x3, n5x5red,
        n5x5, pool_planes):
        super(Inception, self).__init__()
```

Parameters:

in_planes: Number of input channels.

n1x1: Number of filters for the 1x1 conv branch.

n3x3red, n3x3: Number of filters for the dimensionality reduction (1x1) + 3x3 conv branch.

n5x5red, n5x5: Number of filters for the dimensionality reduction (1x1) + 5x5 conv branch (actually uses 2 layers of 3x3 to reduce computation).

pool_planes: Number of filters for the max-pooling + 1x1 conv branch.

Four Branches in the Inception Block:

1x1 conv branch:

```
self.b1 = nn.Sequential(
    nn.Conv2d(in_planes, n1x1, kernel_size=1),
    nn.BatchNorm2d(n1x1),
    nn.ReLU(True),
)
```

Used to extract features at a 1x1 scale.

Purpose: Processes features at a 1×1 scale within the Inception model.

Function of Each Layer:

- 1×1 Conv: Changes the number of channels (in_planes → n1x1), keeping the image size the same.
- BatchNorm: Stabilizes the training process.
- ReLU: Adds non-linearity (optimizes memory with inplace=True).

Application: Reduces data dimensionality or combines features before more complex convolution operations (e.g., 3×3).

1x1 → 3x3 conv branch:

```
self.b2 = nn.Sequential(
    nn.Conv2d(in_planes, n3x3red, kernel_size=1),
    nn.BatchNorm2d(n3x3red),
    nn.ReLU(True),
    nn.Conv2d(n3x3red, n3x3, kernel_size=3, padding=1),
    nn.BatchNorm2d(n3x3),
    nn.ReLU(True),
)
```

Used to extract features in a 3x3 region.

TASK 2

Step-by-step Details:

1×1 Convolution:

Reduces the number of channels from in_planes → n3x3red (optimizes computation).

BatchNorm + ReLU: Normalizes and adds non-linearity.

3×3 Convolution:

Processes features within a 3×3 range (with padding=1, maintains image size).

Continues normalization and activation to learn features effectively.

Applications:

Detects local features (edges, textures) in a 3×3 region.

Combines with other branches (1×1, 5×5) in Inception to diversify features.

1x1 → 5x5 conv branch:

```
self.b3 = nn.Sequential(
    nn.Conv2d(in_planes, n5x5red, kernel_size=1),
    nn.ReLU(True),
    nn.Conv2d(n5x5red, n5x5, kernel_size=3, padding=1),
    nn.BatchNorm2d(n5x5),
    nn.ReLU(True),
    nn.Conv2d(n5x5, n5x5, kernel_size=3, padding=1),
    nn.BatchNorm2d(n5x5),
    nn.ReLU(True),
)
```

Used to extract features in a 5x5 region (more efficient with two 3x3 layers).

MaxPool → 1x1 conv branch:

```
self.b4 = nn.Sequential(
    nn.MaxPool2d(3, stride=1, padding=1),
    nn.Conv2d(in_planes, pool_planes, kernel_size=1),
    nn.BatchNorm2d(pool_planes),
    nn.ReLU(True),
)
```

Used to retain information from pooling and reduce dimensionality.

This branch combines two important techniques:

- MaxPooling: Retains the most prominent features.
- Conv1x1: Reduces data dimensionality efficiently.

TASK 2

Analysis of Each Component

MaxPool2d(3, stride=1, padding=1)

- Kernel size of 3x3 helps detect local features.
- Stride=1 and padding=1 maintain spatial dimensions.
- Effective in reducing overfitting and increasing invariance.

Conv2d 1x1

- Reduces the number of channels from in_planes → pool_planes.
- Saves significant model parameters.
- Linearly combines features before passing through ReLU.

BatchNorm + ReLU

BatchNorm helps stabilize the training process.

ReLU adds non-linearity, increasing representational power.

Forward Method:

```
def forward(self, x):
        y1 = self.b1(x)
        y2 = self.b2(x)
        y3 = self.b3(x)
        y4 = self.b4(x)
        return torch.cat([y1,y2,y3,y4], 1)
```

Combines the outputs from the 4 branches using torch.cat() along the channel dimension.

Parallel Processing:

The input x is simultaneously passed through 4 branches (b1, b2, b3, b4).

Each branch extracts features in its own way (1x1 conv, 3x3 conv, maxpool...).

Feature Combination:

The results y1, y2, y3, y4 are concatenated along the channel dimension (dim=1).

Requirement: the tensors must have the same spatial dimensions (H, W).

Advantages:

Integrates multi-scale features (1x1, 3x3, pooling).

Increases the representational capacity of the model.

Maintains the spatial dimensions.

Output:

Number of channels = sum of channels from the 4 branches.

Spatial dimensions (H, W) are maintained.

TASK 2

GoogLeNet Layer (Main Model)

This model consists of:

Preprocessing Layer (pre_layers):

```
self.pre_layers = nn.Sequential(
    nn.Conv2d(3, 192, kernel_size=3, padding=1),   # Conv 3x3
    nn.BatchNorm2d(192),
    nn.ReLU(True),
)
```

Transforms the input image (3 channels) into 192 channels.

Inception Blocks (a3, b3, a4, b4, c4, d4, e4, a5, b5):

Each Inception block has a different configuration regarding the number of filters.

Example:

self.a3 = Inception(192, 64, 96, 128, 16, 32, 32)

self.b3 = Inception(256, 128, 128, 192, 32, 96, 64)

MaxPooling is applied after some blocks to reduce spatial dimensions:

self.maxpool = nn.MaxPool2d(3, stride=2, padding=1)

Final Classification Layer:

```
def forward(self, x):
    # Pass input through initial preprocessing layers
    out = self.pre_layers(x)

    # Block 3 layers
    out = self.a3(out)  # First layer in block 3
    out = self.b3(out)  # Second layer in block 3
    out = self.maxpool(out)  # Downsample with max pooling

    # Block 4 layers (deeper block with more layers)
    out = self.a4(out)  # First layer in block 4
    out = self.b4(out)  # Second layer in block 4
    out = self.c4(out)  # Third layer in block 4
    out = self.d4(out)  # Fourth layer in block 4
    out = self.e4(out)  # Fifth layer in block 4
    out = self.maxpool(out)  # Downsample with max pooling

    # Block 5 layers
    out = self.a5(out)  # First layer in block 5
    out = self.b5(out)  # Second layer in block 5

    # Final pooling and classification
    out = self.avgpool(out)  # Global average pooling
    out = out.view(out.size(0), -1)  # Flatten the features
    out = self.linear(out)  # Final fully-connected layer
        for classification

    return out
```

AvgPool2d(8) reduces the size to 1x1 (with CIFAR-10, the initial 32x32 image → becomes 8x8 after the conv layers).

Linear(1024, num_classes) classifies the 10 classes (CIFAR-10).

TASK 2

Model Initialization

```
model = GoogLeNet(num_classes=10).to(device)
```

Function:

Creates an instance of the GoogLeNet class with num_classes=10 (suitable for CIFAR-10, which has 10 classes).

Moves the model to the computational device (the device defined earlier, which can be 'cuda' or 'cpu').

Parameter Explanation:

num_classes=10: The number of output classes (CIFAR-10 has 10 classes such as airplane, car, bird, etc.).

.to(device): Ensures the model runs on the GPU (if device='cuda') or the CPU.

Display Architecture with Summary

```
summary(model, (3, 32, 32))
```

Function:

Prints a summary table of the model architecture, including:

- Each layer in the network.
- The output shape of each layer.
- The number of trainable parameters.
- The total number of parameters in the model.

Parameter Explanation:

(3, 32, 32): The input size of the model (3 color channels RGB, 32x32 pixel images - the size of CIFAR-10).

Set Up Loss Function and Optimizer

```
criterion = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=1e-3)
```

nn.CrossEntropyLoss(): Loss function suitable for multi-class classification problems (CIFAR-10 has 10 classes).

Adam: Optimizer with a learning rate of 1e-3 (0.001), which generally works well with CNN models.

TASK 2

Evaluate Function for Testing on the Test Set

```python
def evaluate(model, testloader, criterion):
    model.eval()
    test_loss = 0.0
    running_correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
            # Move inputs and labels to the device
            images, labels = images.to(device), labels.to(
                device)

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

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

    accuracy = 100 * running_correct / total
    test_loss = test_loss / len(testloader)
    return test_loss, accuracy
```

Function:

Calculates loss and accuracy on the test set.

model.eval(): Disables dropout/batch normalization in evaluation mode.

torch.no_grad(): Disables gradient calculation → increases speed and saves memory.

Output:

test_loss: Average loss over the entire test set.

accuracy: Accuracy (%).

Model Training

Initial Setup

```python
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []
max_epoch = 150
```

TASK 2

Training Loop

```python
# train
for epoch in range(max_epoch):
    model.train()
    running_loss = 0.0
    running_correct = 0    # to track number of correct predictions
    total = 0              # to track total number of samples
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # Move inputs and labels to the device
        inputs, labels = inputs.to(device), labels.to(device)
        # Zero the parameter gradients
        optimizer.zero_grad()
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss += loss.item()
        # Backward pass and optimization
        loss.backward()
        optimizer.step()
        # Determine class predictions and track accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        running_correct += (predicted == labels).sum().item()

    epoch_accuracy = 100 * running_correct / total
    epoch_loss = running_loss / (i + 1)

    test_loss, test_accuracy = evaluate(model, testloader, criterion)
    print(f"Epoch [{epoch + 1:3}/{max_epoch:3}] \t Loss: {epoch_loss:<11.5f} Accuracy: {epoch_accuracy:.2f}% \t Test Loss: {test_loss:<11.5f} Test Accuracy: {test_accuracy:.2f}%")

    # save for plot
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)
```

TASK 2

Details:

Each epoch iterates through the entire training set (trainloader).

optimizer.zero_grad(): Clears gradients from the previous step to avoid accumulation.

loss.backward(): Calculates gradients.

optimizer.step(): Updates the weights.

Accuracy is calculated by comparing predictions (predicted) and true labels (labels).

Print Best Result

```
best_epoch = np.argmax(test_accuracies)
print(f"\nBest epoch: {best_epoch + 1} with test accuracy: {
    test_accuracies[best_epoch]:.2f}%")
```

Finds the epoch with the highest accuracy on the test set and prints it.

Plotting Graphs

```
plt.plot(train_losses, label='train_losses')
plt.plot(test_losses, label='test_losses')
plt.legend()

plt.plot(train_accuracies, label='train_accuracy')
plt.plot(test_accuracies, label='test_accuracy')
plt.legend()
```

Graph 1: Loss on the training and test sets.

Graph 2: Accuracy on the training and test sets.

Purpose:

Check for overfitting (if test loss increases while train loss decreases).

Evaluate convergence performance.

TASK 3

Results of MLP

Initial Test Loss: 3.3231, Test Accuracy: 9.99%

lightgray Epoch	Train Loss	Train Accuracy	Test Loss	Test Accuracy
1	2.2700	21.72%	1.8750	32.03%
2	1.9537	28.39%	1.8077	34.79%
3	1.8781	31.25%	1.7499	39.02%
4	1.8272	33.84%	1.7129	39.16%
5	1.7868	35.51%	1.6826	39.22%
6	1.7653	36.16%	1.6345	43.42%
7	1.7448	37.18%	1.6607	39.54%
8	1.7269	38.15%	1.6038	43.29%
9	1.7124	38.40%	1.6328	43.13%
10	1.7019	38.91%	1.5873	43.96%
11	1.6953	39.27%	1.5900	44.00%
12	1.6903	39.46%	1.5912	43.93%
13	1.6830	39.79%	1.5807	43.34%
14	1.6790	39.54%	1.5652	44.69%
15	1.6744	39.58%	1.5771	42.06%
16	1.6657	40.39%	1.5576	45.02%
17	1.6640	40.66%	1.5798	44.97%
18	1.6600	40.68%	1.5775	44.63%
19	1.6597	40.53%	1.5254	46.34%
20	1.6522	40.91%	1.5794	45.53%
21	1.6442	41.18%	1.5609	44.20%
22	1.6439	41.08%	1.5551	44.11%
23	1.6457	41.21%	1.5899	44.07%
24	1.6392	41.21%	1.5279	45.14%
		...		
98	1.3553	51.17%	1.3031	53.89%
99	1.3498	51.53%	1.2884	54.67%
100	1.3490	51.60%	1.2878	54.69%

Final Test Loss: 1.2948, Final Test Accuracy: **54.74%**

TASK 3

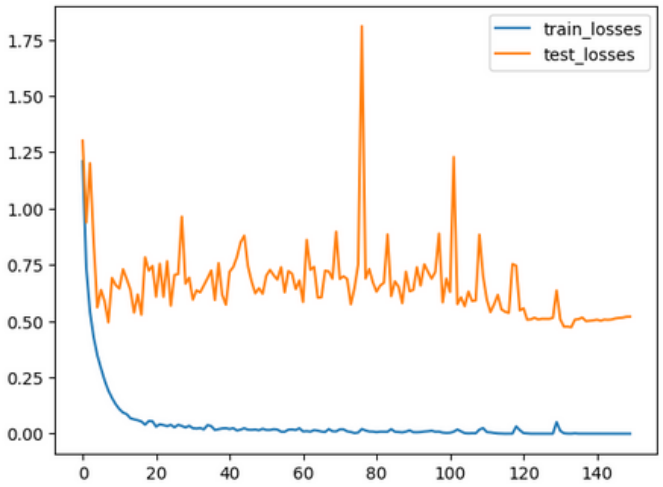
Results of CNN

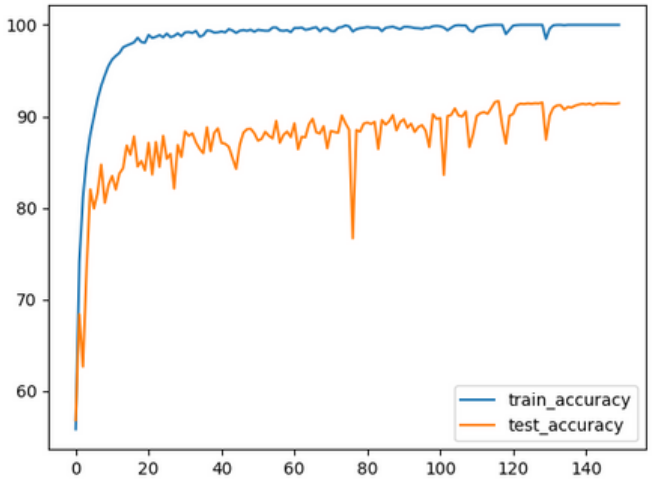
lightgray Epoch	Train Loss	Train Accuracy	Test Loss	Test Accuracy
7	0.23573	91.87%	0.58675	81.62%
8	0.19170	93.35%	0.49480	84.72%
9	0.15858	94.49%	0.69196	80.55%
10	0.13076	95.53%	0.65997	82.50%
11	0.10924	96.21%	0.64479	83.48%
		...		
110	0.02496	99.26%	0.69995	88.02%
111	0.00748	99.74%	0.59695	90.04%
112	0.00591	99.82%	0.54000	90.35%
113	0.00320	99.90%	0.57468	90.46%
114	0.00138	99.96%	0.61672	90.28%
115	0.00069	99.99%	0.55195	90.90%
116	0.00016	100.00%	0.54168	91.56%
117	0.00007	100.00%	0.53675	91.66%
118	0.00040	99.99%	0.75361	89.00%
119	0.03244	98.99%	0.74622	87.02%
120	0.01562	99.47%	0.54752	90.07%
121	0.00272	99.92%	0.55756	90.26%
122	0.00141	99.97%	0.50625	91.20%
123	0.00023	100.00%	0.50788	91.39%
124	0.00020	100.00%	0.51515	91.35%
125	0.00008	100.00%	0.50717	91.42%
126	0.00007	100.00%	0.51035	91.38%
127	0.00005	100.00%	0.50997	91.43%
128	0.00003	100.00%	0.51016	91.41%
129	0.00005	100.00%	0.51442	91.51%
130	0.05158	98.45%	0.63629	87.45%
131	0.01285	99.59%	0.50834	90.06%
132	0.00211	99.95%	0.47548	90.93%
133	0.00058	99.99%	0.47564	91.19%
134	0.00029	100.00%	0.47306	91.22%
135	0.00175	99.96%	0.50686	90.74%
136	0.00038	100.00%	0.50882	91.03%
137	0.00031	99.99%	0.51620	90.96%
138	0.00016	100.00%	0.50006	91.17%
139	0.00008	100.00%	0.50151	91.29%
140	0.00005	100.00%	0.50328	91.37%
141	0.00003	100.00%	0.50639	91.31%
142	0.00002	100.00%	0.50168	91.40%
143	0.00002	100.00%	0.50686	91.22%
144	0.00002	100.00%	0.50574	91.42%
145	0.00002	100.00%	0.50728	91.39%
146	0.00001	100.00%	0.51196	91.41%
147	0.00001	100.00%	0.51439	91.39%
148	0.00001	100.00%	0.51570	91.36%
149	0.00001	100.00%	0.51937	91.36%
150	0.00001	100.00%	0.51986	91.44%

Best epoch: 117 with test accuracy: **91.66%**

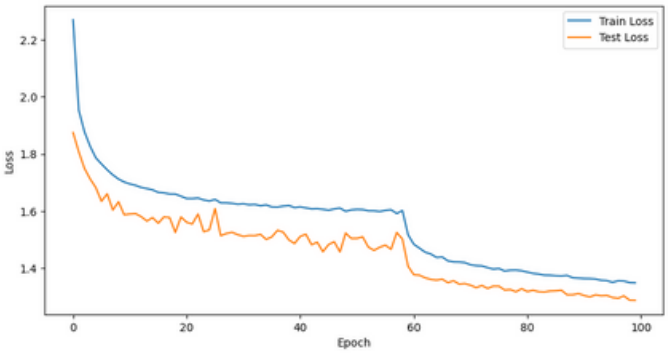
TASK 4

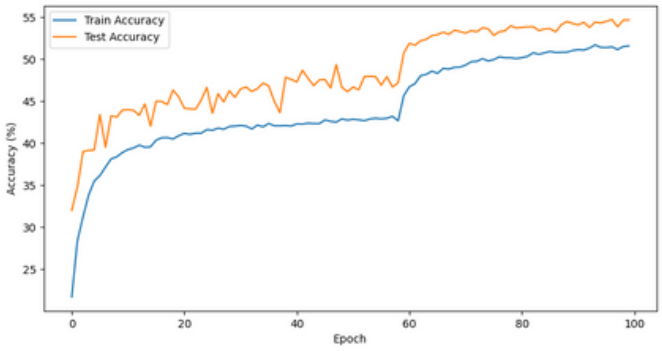
Visualized Results of CNN





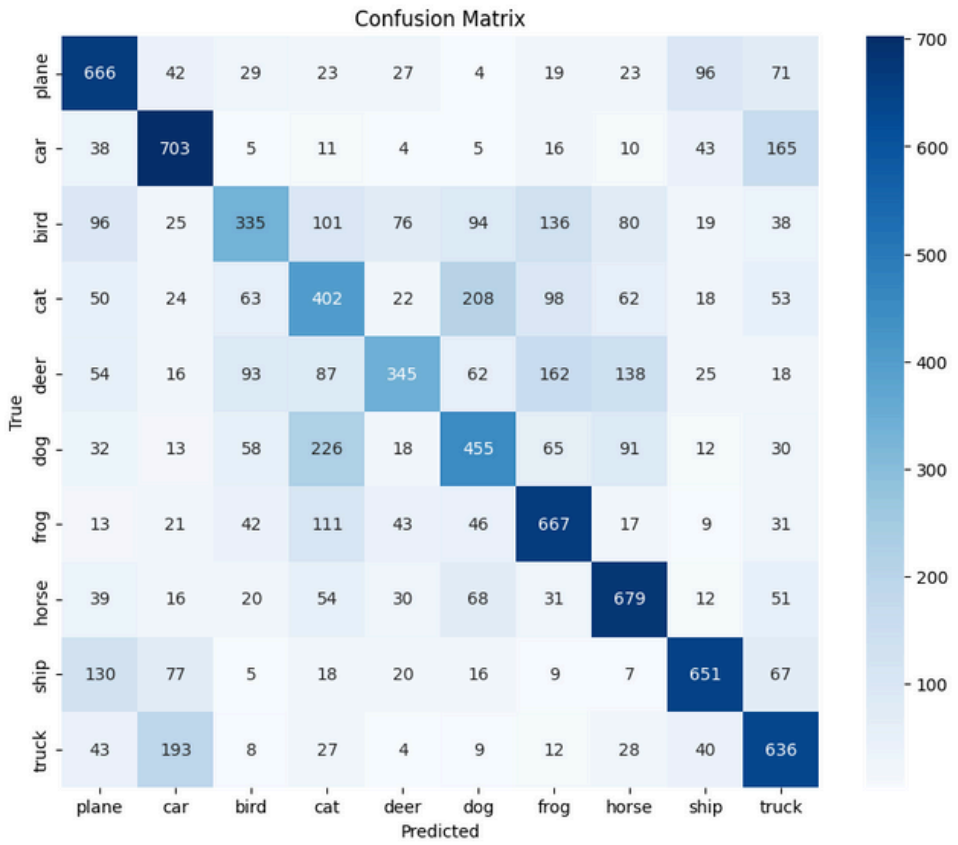
Visualized Results of MLP



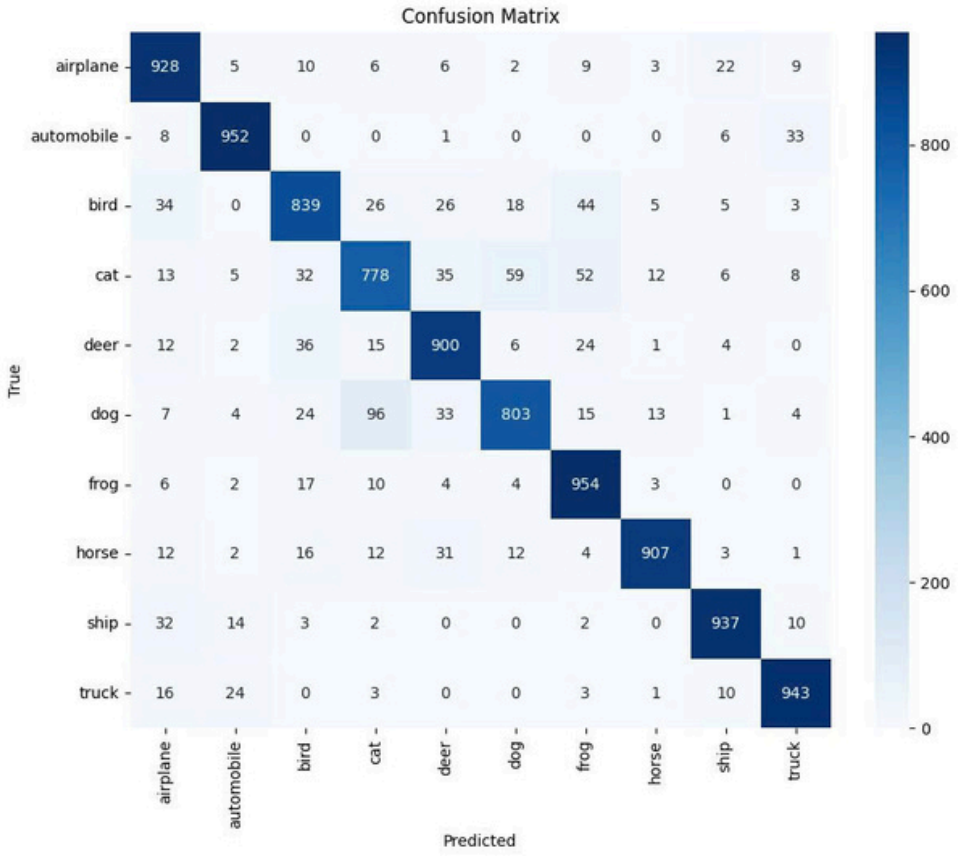


TASK 5

Plot Confusion Matrix MLP



Plot Confusion Matrix CNN



TASK 6

1. Overall Results

Metric	MLP (100 epochs)	CNN (150 epochs)
Test Accuracy	**54.74%**	**91.66%**
Training Accuracy	51.60%	100.00%
Final Test Loss	1.2948	0.5199
Best Epoch	100	117

2. Detailed Analysis

Overall Performance

The CNN significantly outperforms the MLP with a test accuracy ~37% higher.

MLP performs poorly due to:

- Loss of spatial information when flattening the image.
- Failure to capture local features.

CNN leverages advantages:

- Preserves the spatial structure of the image.
- Extracts hierarchical features effectively.

Learning Curves

MLP:

- Slow convergence, accuracy increases gradually.
- Small train-test gap (~3%) indicates little overfitting.

CNN:

- Fast convergence (reaches >80% test accuracy after 10 epochs).
- Severe overfitting when:
  - Training accuracy reaches 100%.
  - Test accuracy plateaus at ~91%.
  - The train-test gap reaches ~9%.

Confusion Matrix

MLP:

- Confuses similar classes frequently (dog-cat, car-truck).
- Per-class accuracy varies widely (36-68%).

CNN:

- Accurately classifies most classes (>90% for 7/10 classes).
- Still struggles with complex classes:
  - Bird (83.9%): mistaken for airplane, cat, deer.
  - Cat (77.8%): mistaken for dog.
  - Dog (80.3%): frequently mistaken for cat.

3.Reasons for Differences

Network Architecture

MLP:

- Processes images as 1D vectors → loses spatial information.
- Lacks weight sharing mechanisms.

TASK 6

CNN:

- Uses convolution to preserve 2D structure.
- Weight sharing helps detect location-invariant features.
- Pooling layers increase invariance.

Feature Extraction Capabilities

MLPs only learn global features.

CNNs learn a hierarchical feature system:

- Low-level: edges, textures
- Mid-level: simple patterns
- High-level: semantic features

Overfitting Phenomenon

MLPs overfit less due to low capacity.

CNNs are more prone to overfitting, but it can be improved with:

- Data augmentation
- Dropout layers
- Regularization (L2, early stopping)

4. Conclusions

CNNs are overwhelmingly more suitable for computer vision tasks.

MLPs should only be used when:

- Computational resources are limited.
- Input images are simple (MNIST).
- High accuracy is not required.