Deep Learning Project 2 Task 2: Training on CIFAR-10 Image Classification

1. Task Overview

This task focuses on end-to-end implementation of a deep learning pipeline to train an AlexNet-based convolutional neural network (CNN) for image classification on the CIFAR-10 dataset. It encompasses critical steps from setting up the technical environment to evaluating model performance, ensuring a structured workflow for building and optimizing a CNN.

The task begins with importing essential libraries—including PyTorch for model building, torchvision for dataset handling, and tools for data manipulation and visualization—to lay the foundation for the pipeline. It then moves to preprocessing: calculating dataset mean and standard deviation for proper normalization, applying data augmentation (e.g., random flips, rotations) to enhance generalization, and creating DataLoaders for efficient batch processing of training and test data. Next, the AlexNet architecture is defined, with a feature extractor (convolutional, pooling, and normalization layers) and a classifier (fully connected and dropout layers) adapted for CIFAR-10’ s 10-class task. Finally, the task covers training the model with an Adam optimizer and cross-entropy loss, implementing evaluation routines, and using early stopping with model checkpointing to prevent overfitting and retain the best-performing model. Throughout, performance metrics (loss and accuracy) are tracked and analyzed to assess training dynamics and model generalization.

2. Task Objective

This task aims to guide learners through the end-to-end implementation and training of an AlexNet-based CNN for image classification on the CIFAR-10 dataset, covering technical setup, data preprocessing, model construction, training, and evaluation. The objectives below outline key goals to build practical proficiency in deep learning pipeline development, summarized in Table 1.

Table 1: Task Objectives for Implementing and Training AlexNet on CIFAR-10

Objective Type	Details
Ideological and Political Objectives	Cultivate a rigorous and systematic approach to deep learning implementation, emphasizing the importance of reproducible workflows and

	model optimization—recognizing how structured pipeline design ensures reliable results and lays the groundwork for ethical AI development.
Learning Objectives	1. Understand the technical components of a deep learning pipeline: environment setup (libraries like PyTorch, torchvision), data preprocessing (normalization, augmentation), AlexNet architecture adaptation for CIFAR-10, and training/evaluation mechanisms. 2. Grasp the role of key techniques (data augmentation for generalization, early stopping for overfitting prevention, checkpointing for model preservation) in optimizing CNN performance.
Skill Objectives	1. Implement the full pipeline: import and configure necessary libraries, preprocess CIFAR-10 (calculate normalization stats, apply augmentation, build DataLoaders), define an AlexNet-based model adapted for 10-class classification, and set up training with Adam optimizer and cross-entropy loss. 2. Train and evaluate the model: track loss/accuracy metrics, implement early stopping and checkpointing, analyze training dynamics, and assess generalization performance on test data.
Literacy Objectives	1. Develop the ability to troubleshoot pipeline components (e.g., data preprocessing errors, model architecture mismatches) and optimize training strategies based on performance metrics. 2. Gain proficiency in translating CNN theory into practice, using AlexNet on CIFAR-10 as a template for applying similar workflows to other image classification tasks and datasets.

3. Task Implementation:

3.1 Importing Essential Libraries for Deep Learning and Data Analysis

The following code snippet demonstrates the import of key Python libraries commonly used for deep learning, data preprocessing, and visualization. Each library serves a specific purpose in the machine learning workflow:

Code block

```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import pandas as pd
import numpy as np
from torch import nn
import torch
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

- `torch` and `torch.nn` : Core PyTorch modules, where `torch` provides tensors and basic operations, while `nn` contains neural network layers and loss functions.
- `torchvision.datasets` and `torchvision.transforms` : Used for loading popular datasets and applying transformations (e.g., resizing, normalization) to image data.
- `torch.utils.data.DataLoader` : Facilitates batching, shuffling, and parallel loading of data for training and evaluation.
- `pandas` (`pd`): Provides data structures like DataFrames for structured data manipulation and analysis.
- `numpy` (`np`): Enables fast numerical operations on arrays and matrices, commonly used alongside PyTorch.
- `sklearn.metrics.accuracy_score` : Computes the accuracy of model predictions, a standard metric for classification tasks.
- `matplotlib.pyplot` (`plt`): Used for visualizing data, training progress, or model results through plots and charts.

This set of imports establishes a foundation for building, training, evaluating, and visualizing deep learning models efficiently.

3.2 Calculating Dataset Mean and Standard Deviation for Normalization

This code snippet demonstrates how to load the CIFAR-10 dataset using PyTorch's `torchvision` library and compute the mean and standard deviation of the training data, which is essential for proper normalization before feeding images into a neural network:

Code block

```
train_dataset = datasets.CIFAR10(root='./dataset/', train=True, download=True, transform=transforms.ToTensor())
test_dataset = datasets.CIFAR10(root='./dataset/', train=False, download=True, transform=transforms.ToTensor())

train_loader = DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=False)
data = next(iter(train_loader))[0]
mean = data.mean(dim=[0,2,3])
std  = data.std(dim=[0,2,3])

print("Mean:", mean)
print("Std:", std)
```

- `datasets.CIFAR10(...)`: Downloads and loads the CIFAR-10 dataset. `train=True` loads training data, `train=False` loads test data. The `transform=transforms.ToTensor()` converts images to PyTorch tensors with pixel values in the range `[0, 1]`.
- `DataLoader(...)`: Wraps the dataset to enable batch processing. Here, `batch_size=len(train_dataset)` loads all training images in a single batch, and `shuffle=False` maintains the original order.
- `data = next(iter(train_loader))[0]`: Retrieves the entire batch of images from the DataLoader. The `[0]` selects only the image tensors (excluding labels).
- `mean = data.mean(dim=[0,2,3])` and `std = data.std(dim=[0,2,3])`: Computes the mean and standard deviation for each channel (R, G, B) across all images and spatial dimensions (height and width). These statistics are crucial for normalizing images during model training, which often improves convergence.
- `print(...)`: Displays the calculated mean and standard deviation values for reference or use in normalization transforms.

This approach ensures the dataset is standardized, which helps neural networks learn more effectively.

3.3 Applying Data Augmentation and Creating DataLoaders for CIFAR-10

This code snippet illustrates how to apply image augmentation to the CIFAR-10 dataset and prepare `DataLoader` objects for efficient training and testing of a neural network. Data augmentation enhances model generalization by artificially increasing the diversity of training images.

Code block

```
# Augmentation
train_augment = transforms.Compose(
    [transforms.Resize((227, 227)),
     transforms.RandomHorizontalFlip(),
     transforms.RandomRotation(15),
     transforms.ToTensor(),
     transforms.Normalize(mean=mean, std=std)]
)

test_transform = transforms.Compose([
    transforms.Resize((227,227)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

```python

train_dataset = datasets.CIFAR10(root='./dataset/', train=True, download=True, transform=train_augment)
test_dataset = datasets.CIFAR10(root='./dataset/', train=False, download=True, transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

- **Data Augmentation (`train_augment`):**
    - `Resize((227, 227))` : Resizes images to 227×227 pixels, commonly used as input size for models like AlexNet.
    - `RandomHorizontalFlip()` : Randomly flips images horizontally to increase variety.
    - `RandomRotation(15)` : Rotates images by ±15 degrees to simulate different orientations.
    - `ToTensor()` : Converts images to PyTorch tensors with values normalized between 0 and 1.
    - `Normalize(mean=mean, std=std)` : Standardizes image channels using previously computed mean and standard deviation, improving convergence during training.
- **Test Transformation (`test_transform`):**
    - Only resizes, converts to tensor, and normalizes. No augmentation is applied to ensure evaluation consistency.
- **Dataset Loading**:
    - `datasets.CIFAR10(...)` loads the training and testing datasets with the defined transformations.
- **DataLoaders**:
    - `train_loader` provides mini-batches of 64 images shuffled randomly for training.
    - `test_loader` provides larger batches of 1000 images without shuffling for evaluation efficiency.

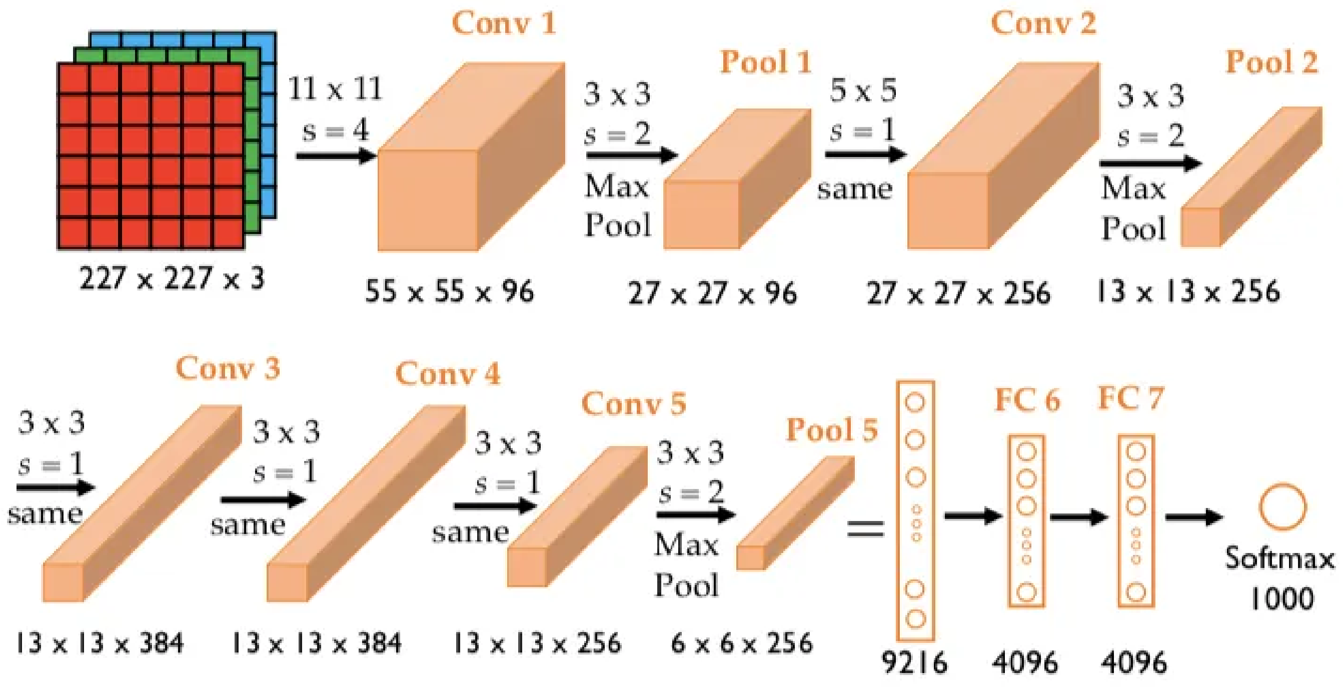
This setup ensures the model sees varied training images while keeping test data consistent, which is a standard practice in deep learning workflows.

3.4 Defining the AlexNet Convolutional Neural Network

This code snippet defines a deep convolutional neural network architecture based on **AlexNet**, which is adapted here for the CIFAR-10 classification task. AlexNet is structured in two main parts: a **feature extractor** and a **classifier**.

Code block

```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Conv2d(64, 192, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(inplace=True),

            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),

            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.AdaptiveAvgPool2d(output_size=(6,6))
        )

        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(9216, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```



Explanation of Key Components:

- **Feature Extractor (`self.features`):**
  - Series of convolutional layers (`nn.Conv2d`) to extract hierarchical features from images.
  - Batch normalization (`nn.BatchNorm2d`) stabilizes and accelerates training.
  - ReLU activations (`nn.ReLU`) introduce non-linearity.
  - Max pooling (`nn.MaxPool2d`) reduces spatial dimensions and controls overfitting.
  - Adaptive average pooling (`nn.AdaptiveAvgPool2d`) ensures a fixed-size output (6×6), independent of input size.
- **Classifier (`self.classifier`):**
  - Fully connected layers (`nn.Linear`) map extracted features to class scores.
  - Dropout layers (`nn.Dropout`) prevent overfitting by randomly deactivating neurons during training.
  - Final linear layer outputs 10 units, corresponding to the CIFAR-10 classes.
- **Forward Pass (`forward`):**
  - Input `x` passes through the feature extractor.
  - `x.view(x.size(0), -1)` flattens the 2D feature maps into a 1D vector for the classifier.

- The classifier produces the final predictions.

This modular design allows the network to learn both spatial hierarchies in images and high-level representations for classification.

Here' s a professional explanation for your training and evaluation code with a suitable heading:

3.5 Training and Evaluating the AlexNet Model on CIFAR-10

This code snippet sets up the **AlexNet model**, defines the **loss function and optimizer**, and implements training and evaluation routines with early stopping based on testing accuracy.

Code block

```python
alex_net = AlexNet().to(device)

# Training parameters
epochs = 30
criterian = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(alex_net.parameters(), lr=0.001)
```

- `AlexNet().to(device)` : Instantiates the AlexNet model and moves it to the specified device (CPU or GPU).
- `nn.CrossEntropyLoss()` : Standard loss function for multi-class classification.
- `torch.optim.Adam(...)` : Adam optimizer with learning rate `0.001` for adaptive gradient updates.

Evaluation Function

Code block

```python
def evaluate(loader: DataLoader):
    test_true_label = []
    test_pred_label = []
    test_overall_loss = 0.0
    with torch.no_grad():
        for data, label in loader:
            data, label = data.to(device), label.to(device)
            test_prediction = alex_net(data)
            test_loss = criterian(test_prediction, label)
            _, test_pred = torch.max(test_prediction, 1)
            test_true_label.append(label.cpu().numpy())
            test_pred_label.append(test_pred.cpu().numpy())
            test_overall_loss += test_loss.item()
        average_test_loss = test_overall_loss / len(loader)
        ty_true = np.concatenate(test_true_label)
        ty_pred = np.concatenate(test_pred_label)
        testing_final_accuracy = accuracy_score(ty_true, ty_pred)
        return testing_final_accuracy, average_test_loss
```

- Evaluates the model on a given `DataLoader` without updating weights (`torch.no_grad()`).
- Computes predictions, aggregates true and predicted labels, and calculates **average loss** and **accuracy**.
- `torch.max(..., 1)` extracts the predicted class with the highest probability.

Training Function with Early Stopping

Code block

```python
def train(training_loader, testing_loader, test_eval: bool = True, patience: int = 5):
    benchmark_testing_accuracy = 0.0
    counter = 0
    train_plot_accuracy = []
    test_plot_accuracy = []
    train_plot_loss = []
    test_plot_loss = []

    for epoch in range(epochs):
        alex_net.train()
        train_loss = 0.0
        true_label = []
        pred_label = []

        for data, label in training_loader:
            data, label = data.to(device), label.to(device)
            alex_net.zero_grad()
            train_pred = alex_net(data)
            training_loss = criterian(train_pred, label)
            training_loss.backward()
            optimizer.step()
            _, prediction = torch.max(train_pred, 1)
            true_label.append(label.cpu().numpy())
            pred_label.append(prediction.cpu().detach().numpy())
            train_loss += training_loss.item()

        average_loss = train_loss / len(training_loader)
        y_true = np.concatenate(true_label)
        y_pred = np.concatenate(pred_label)
        train_accuracy = accuracy_score(y_true, y_pred)

        testing_final_accuracy, average_test_loss = evaluate(testing_loader)

        train_plot_accuracy.append(train_accuracy)
        test_plot_accuracy.append(testing_final_accuracy)
        train_plot_loss.append(average_loss)
        test_plot_loss.append(average_test_loss)

        print(f'Epoch {epoch + 1}: Training Loss: {average_loss}, Training Accuracy: {train_accuracy}, Testing Loss: {average_test_loss}, Testing Accuracy: {testing_final_accuracy}')

        # Early stopping and model checkpoint
        if testing_final_accuracy > benchmark_testing_accuracy:
            torch.save(alex_net.state_dict(), './model_checkpoint/alex_net/alex_net.pth')
            benchmark_testing_accuracy = testing_final_accuracy
            counter = 0
            print(f"Model saved at accuracy of: {testing_final_accuracy}")
        else:
            counter += 1

        if counter > patience:
            print("Model stopping due to no improvement in testing accuracy")
            return train_plot_accuracy, test_plot_accuracy, train_plot_loss, test_plot_loss

    return train_plot_accuracy, test_plot_accuracy, train_plot_loss, test_plot_loss
```

Key Points:

1. **Training Loop**:
    - Iterates over `epochs` and `training_loader`.
    - Forward pass → loss computation → backpropagation → optimizer step.
    - Tracks training loss and accuracy.
2. **Evaluation During Training**:
    - Calls `evaluate(testing_loader)` each epoch to monitor testing performance.
3. **Early Stopping & Checkpointing**:
    - Saves model whenever testing accuracy improves.
    - Stops training if testing accuracy does not improve for `patience` consecutive epochs.

4. **Performance Tracking**:
    - Keeps lists of training/testing loss and accuracy for plotting or analysis.

3.6 Evaluating Performance

From your training log, we can see a few key patterns and insights about your **AlexNet training on CIFAR-10**:

1. **Training vs. Testing Performance**

- **Training Accuracy:** Starts around 31.6% and steadily climbs to ~91.6%.
- **Testing Accuracy:** Starts at 43.8%, peaks at 82.99%, then fluctuates and eventually stops improving.

This indicates **the model is learning** and overfitting slightly toward the later epochs since training accuracy keeps increasing while testing accuracy plateaus or decreases at times.

2. **Loss Trends**

- **Training Loss:** Consistently decreases from 1.898 → 0.247.
- **Testing Loss:** Generally decreases but shows fluctuations (especially epochs 20–30), which aligns with testing accuracy fluctuations.

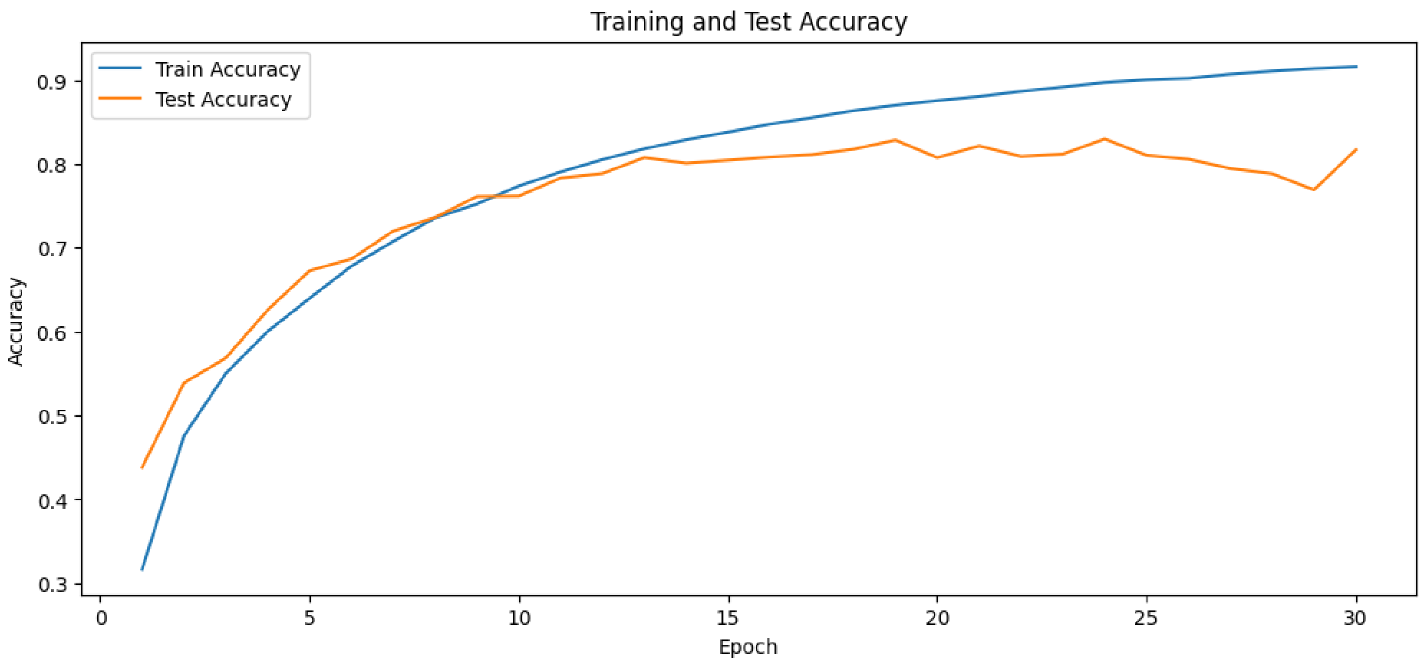
These fluctuations are typical when the learning rate might be a bit high or when the model starts overfitting the training set.

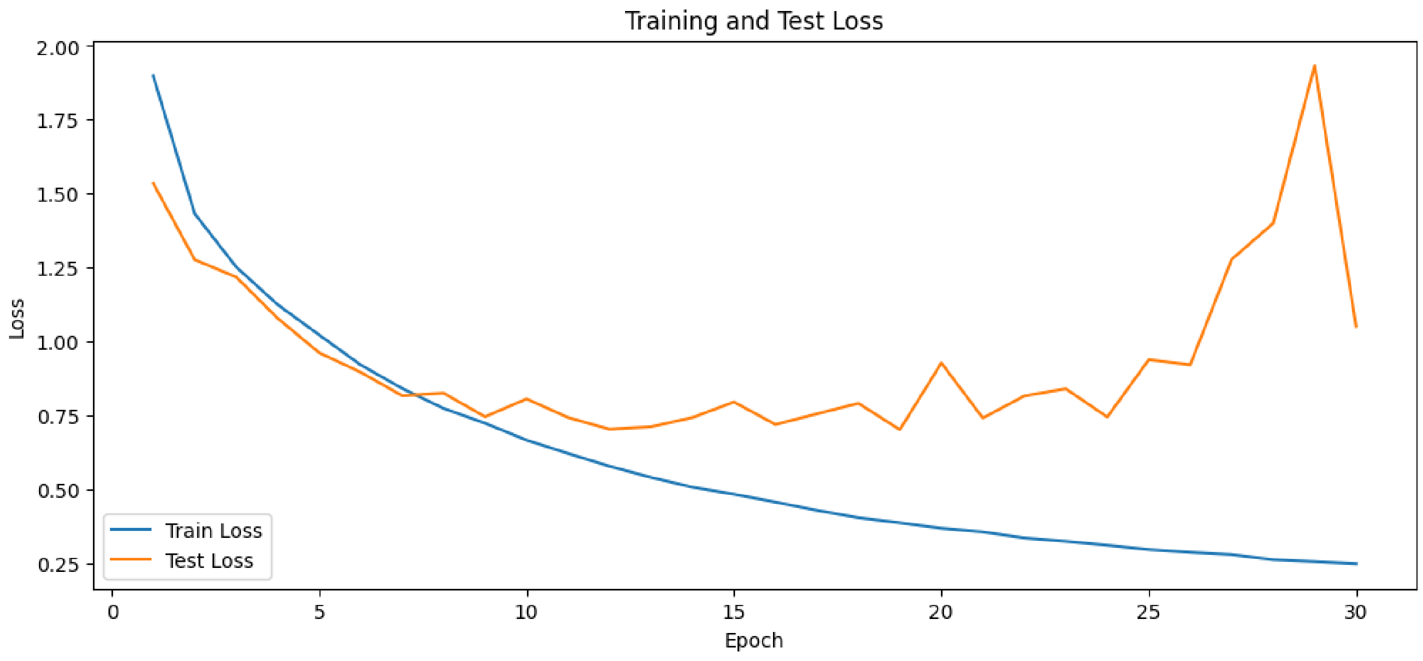
3. **Model Checkpointing**

- The model is **saved whenever testing accuracy improves**, which happened multiple times: Epochs 1, 2, 3⋯ up to 24.
- Final saved model accuracy: **82.99% testing accuracy** at Epoch 24.
- Training was stopped early at Epoch 30 due to **no improvement for 5 consecutive epochs** (patience = 5).

4. **Overfitting Observation**

- Training accuracy continues to increase after Epoch 24, while testing accuracy fluctuates or drops (e.g., Epoch 25–29).
- This suggests **mild overfitting**—the model is memorizing training data patterns more than generalizing.


Training and Test Accuracy
Train Accuracy
Test Accuracy
Accuracy
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
5
10
15
20
25
30
Epoch


Training and Test Loss
Train Loss
Test Loss
Loss
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
5
10
15
20
25
30
Epoch