Deep Learning Architecture and Hardware Acceleration

Lab 2 Report – Task 1

Jinay Panchal (jdpancha@iu.edu)

Objective:

Task1: Standardized Implementation of LENET-5 (with SoftMax function)

Task 2: Adjusted LENET-5 Implementation with modifications

This report documents the implementation of the task 1 LeNet-5 architecture for classifying the MNIST dataset. The LeNet-5 model is a convolutional neural network (CNN) designed by Yann LeCun and his colleagues and is a pioneering architecture for image classification tasks. This implementation includes training the model on the MNIST training dataset and evaluating its performance on the MNIST test dataset.

Later on in this report, we have implemented task 2, which comprises of various modifications in the LENET and observations related to the accuracy of the model, with the changes in the different number of convolutional layers, different number of Fully Connected Layers, and a different criterion for the Loss which is MSE Loss.

Task 1:

1. LeNet-5 Architecture

The LeNet-5 architecture is implemented in the LeNet class. It consists of two convolutional layers, followed by two fully connected layers, and an output layer. The architecture is defined as follows:

Convolutional Layer 1: 6 filters, each with a kernel size of 5x5.
Max Pooling 1: Max-pooling operation with a 2x2 kernel.
Convolutional Layer 2: 16 filters, each with a kernel size of 5x5.
Max Pooling 2: Max-pooling operation with a 2x2 kernel.
Fully Connected Layer 1: 120 neurons.
Fully Connected Layer 2: 84 neurons.
Output Layer: 10 neurons (corresponding to the 10 classes of MNIST digits).

2. Data Preprocessing

The MNIST dataset is loaded and preprocessed using the torchvision library. The following preprocessing steps are applied:

Resizing the images to 32x32 pixels to match the input size expected by the LeNet-5 architecture.
Converting the images to tensors.
Creating data loaders for both the training and test datasets with batch sizes of 64.

3. Training and Testing Functions

Two functions, train and test, are defined to facilitate training and testing of the LeNet-5 model.

The train function takes care of training the model. It includes forward and backward passes, optimization, and computes training loss and accuracy.

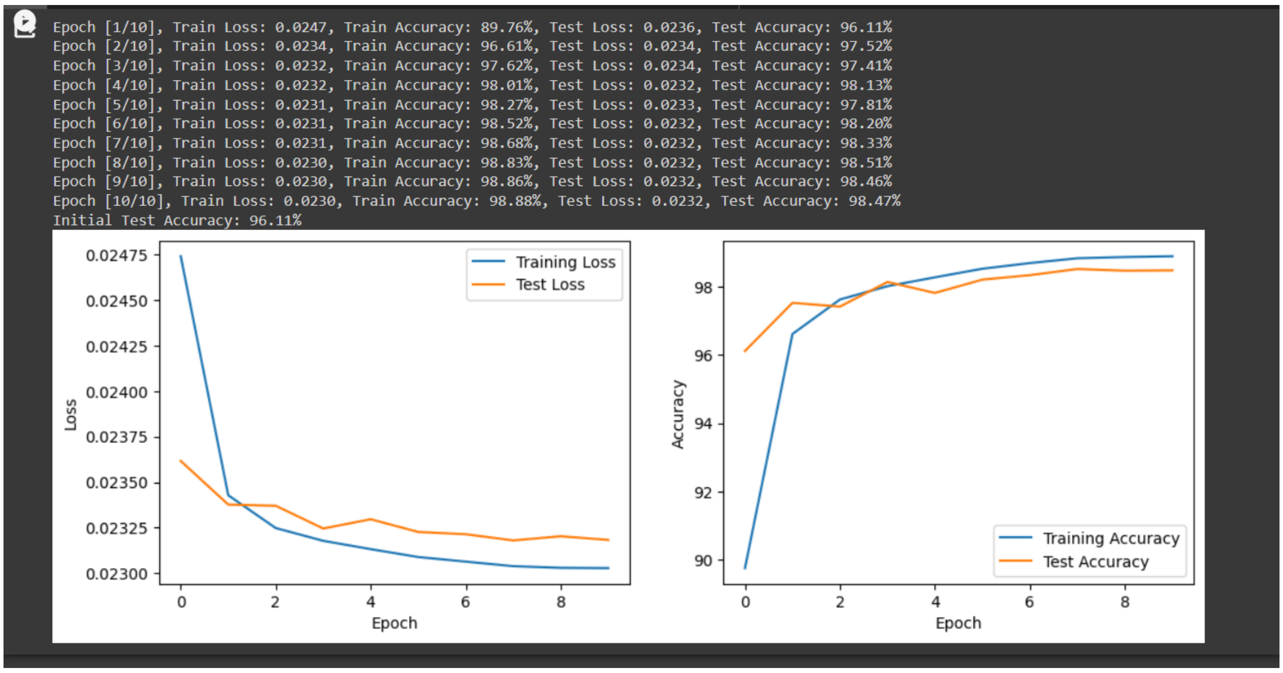
The test function evaluates the model's performance on the test dataset and computes test loss and accuracy.

4. Training Loop
The training loop iterates over a specified number of epochs (in this case, 10 epochs). In each epoch, it trains the model using the training data, computes training loss and accuracy, and evaluates the model on the test data, computing test loss and accuracy. The loss and accuracy values for both training and testing are recorded during each epoch.

5. Model Optimization
The Adam optimizer is used to optimize the LeNet-5 model's parameters with a learning rate of 0.001. The loss is computed using the Cross-Entropy Loss function, suitable for multi-class classification tasks.

Results and Visualizations:
1. Initial Test Accuracy:
The initial test accuracy is calculated after the first epoch of training and is recorded. This provides a baseline accuracy to assess model improvement over subsequent epochs which is 96.11%

2. Training Loss and Accuracy Trends:
Training and test loss as well as training and test accuracy are recorded for each epoch and plotted for visual inspection. These trends help to monitor how the model is learning and whether it's overfitting or underfitting. Final accuracy being 98.47%



```
Epoch [1/10], Train Loss: 0.0247, Train Accuracy: 89.76%, Test Loss: 0.0236, Test Accuracy: 96.11%
Epoch [2/10], Train Loss: 0.0234, Train Accuracy: 96.61%, Test Loss: 0.0234, Test Accuracy: 97.52%
Epoch [3/10], Train Loss: 0.0232, Train Accuracy: 97.62%, Test Loss: 0.0234, Test Accuracy: 97.41%
Epoch [4/10], Train Loss: 0.0232, Train Accuracy: 98.01%, Test Loss: 0.0232, Test Accuracy: 98.13%
Epoch [5/10], Train Loss: 0.0231, Train Accuracy: 98.27%, Test Loss: 0.0233, Test Accuracy: 97.81%
Epoch [6/10], Train Loss: 0.0231, Train Accuracy: 98.52%, Test Loss: 0.0232, Test Accuracy: 98.20%
Epoch [7/10], Train Loss: 0.0231, Train Accuracy: 98.68%, Test Loss: 0.0232, Test Accuracy: 98.33%
Epoch [8/10], Train Loss: 0.0230, Train Accuracy: 98.83%, Test Loss: 0.0232, Test Accuracy: 98.51%
Epoch [9/10], Train Loss: 0.0230, Train Accuracy: 98.86%, Test Loss: 0.0232, Test Accuracy: 98.46%
Epoch [10/10], Train Loss: 0.0230, Train Accuracy: 98.88%, Test Loss: 0.0232, Test Accuracy: 98.47%
Initial Test Accuracy: 96.11%
```

Conclusion
In this implementation, we successfully implemented the LeNet-5 architecture for MNIST digit classification. The model was trained for 10 epochs, and its performance was evaluated on the test dataset. The initial test

accuracy was calculated, and training and testing loss and accuracy trends were visualized. This provides insights into the model's performance and training progress.

Task 2:

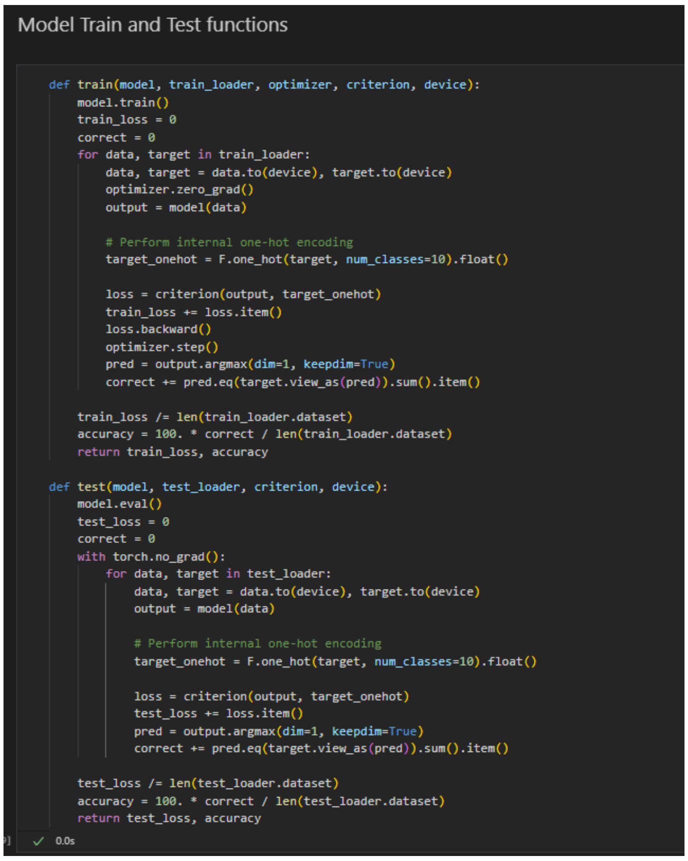
LeNet_Adjusted_4cov_4fc: 4 convolutional layers, 4 fully connected layers, ReLU activation, and Mean Squared Error (MSE) loss.

LeNet_Adjusted_3cov_3fc: 3 convolutional layers, 3 fully connected layers, ReLU activation, and MSE loss.

LeNet_Adjusted_4cov_2fc: 4 convolutional layers, 2 fully connected layers, ReLU activation, and MSE loss.

LeNet_Adjusted_3cov_2fc: 3 convolutional layers, 2 fully connected layers, ReLU activation, and MSE loss.

Code: The code follows extremely simple structure, where the internal parameters of the kernel, and the corresponding convolutional layers, fully connected layers are getting changed, and MSE Loss is calculated, but for all the configurations the data preprocessing, Model Training and Testing functions stays the same.



Model Train and Test functions

```python
def train(model, train_loader, optimizer, criterion, device):
    model.train()
    train_loss = 0
    correct = 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)

        # Perform internal one-hot encoding
        target_onehot = F.one_hot(target, num_classes=10).float()

        loss = criterion(output, target_onehot)
        train_loss += loss.item()
        loss.backward()
        optimizer.step()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

    train_loss /= len(train_loader.dataset)
    accuracy = 100. * correct / len(train_loader.dataset)
    return train_loss, accuracy

def test(model, test_loader, criterion, device):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)

            # Perform internal one-hot encoding
            target_onehot = F.one_hot(target, num_classes=10).float()

            loss = criterion(output, target_onehot)
            test_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    return test_loss, accuracy
```

```
✓ 0.0s
```

Note: I was getting implementation error while executing the code in Modular format, I tried making a list of models which are the objects of the corresponding classes and loop over the list while testing and training each of them, but due to systems constraints and limitations, I was not able to do so, hence I created a separate class and executable code for each of the class.

Model Architectures:

Each of the four model configurations follows a similar structure:

- Convolutional layers with ReLU activation and max-pooling.
- Fully connected layers with ReLU activation.
- Output layer.

Training Parameters:

- Adam optimizer with a learning rate of 0.001.
- MSE loss function (unconventional for classification tasks).
- 10 epochs of training.
-

Results and Observations:

- **LeNet_Adjusted_4cov_4fc**
    - ➢ Initial test accuracy: 98.43%
    - ➢ Final test accuracy: 99.12%
    - ➢ Effect on accuracy: Significant improvement over epochs.

Thoughts:
This model configuration, with deep convolutional and fully connected layers, achieved impressive accuracy on the MNIST dataset.

- **LeNet_Adjusted_3cov_3fc**
    - ➢ Initial test accuracy: 98.92%
    - ➢ Final test accuracy: 99.29%
    - ➢ Effect on accuracy: Steady improvement over epochs.

Thoughts:
Fewer convolutional and fully connected layers, but still a high level of accuracy.

- **LeNet_Adjusted_4cov_2fc**
    - ➢ Initial test accuracy: 98.87%
    - ➢ Final test accuracy: 99.47%
    - ➢ Effect on accuracy: Consistent improvement.

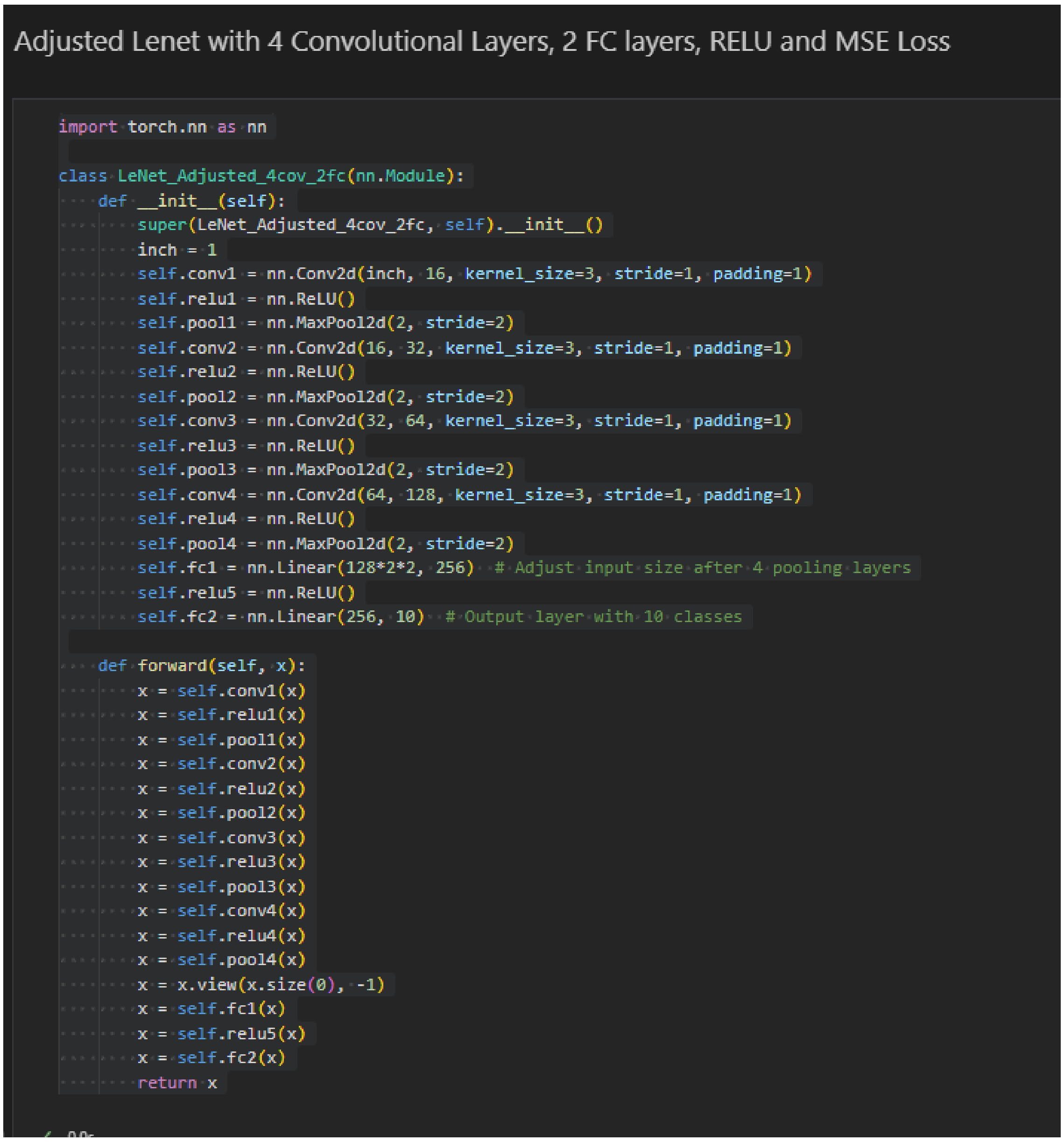
Thoughts:
Reducing the number of fully connected layers while retaining deep convolutional layers still leads to high accuracy.

- **LeNet_Adjusted_3cov_2fc**
    - ➢ Initial test accuracy: 98.82%

- Final test accuracy: 99.46%
- Effect on accuracy: Gradual improvement.

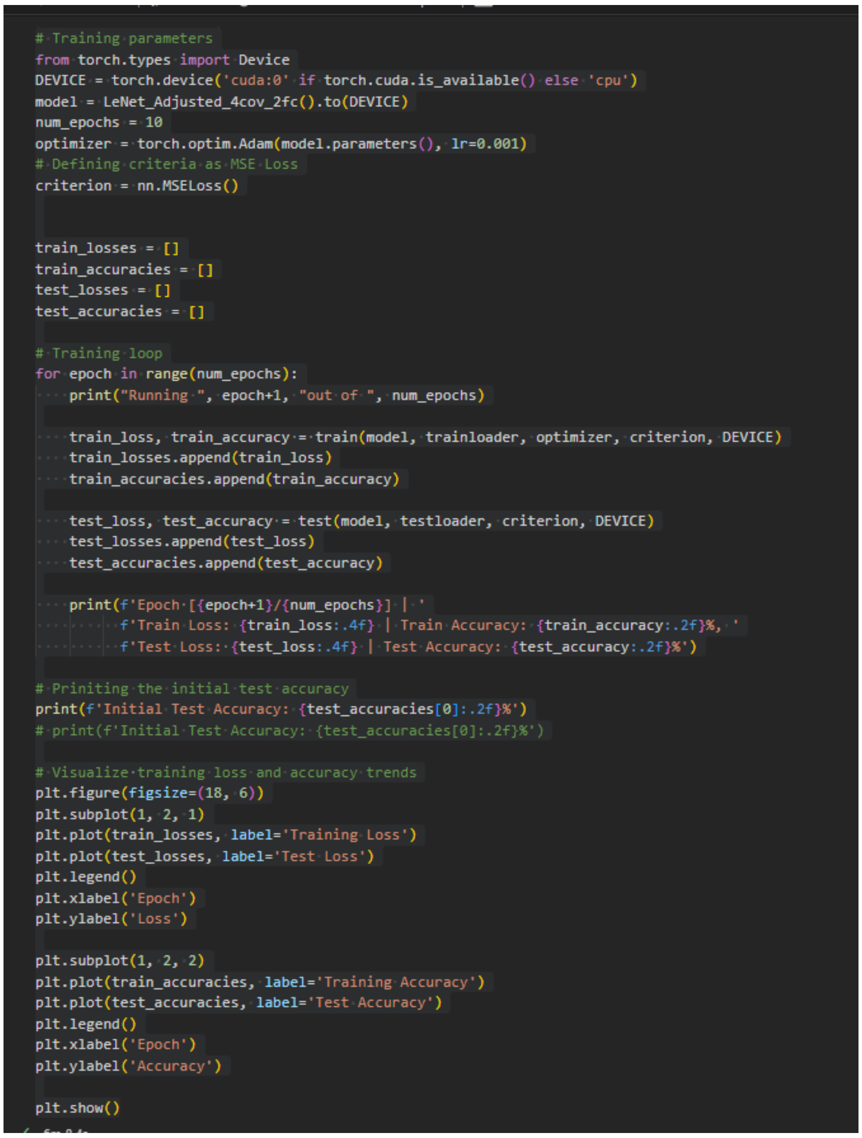
Thoughts:
Despite fewer convolutional and fully connected layers, this configuration performs exceptionally well.

LeNet_Adjusted_4cov_2fc achieved high accuracy of **99.47%** on the MNIST dataset due to a balanced architecture that effectively learned features and prevented overfitting. The model's simplicity and faster training make it a suitable choice for this specific task. However, achieving high accuracy also depends on factors like dataset size and the optimization algorithm used. Further fine-tuning and experimentation may lead to even better results.



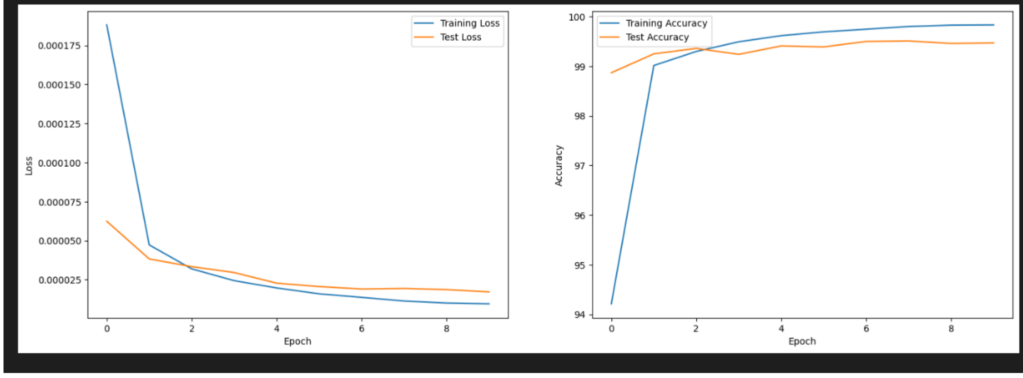
Adjusted Lenet with 4 Convolutional Layers, 2 FC layers, RELU and MSE Loss

```python
import torch.nn as nn

class LeNet_Adjusted_4cov_2fc(nn.Module):
    def __init__(self):
        super(LeNet_Adjusted_4cov_2fc, self).__init__()
        inch = 1
        self.conv1 = nn.Conv2d(inch, 16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, stride=2)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, stride=2)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu4 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(2, stride=2)
        self.fc1 = nn.Linear(128*2*2, 256)  # Adjust input size after 4 pooling layers
        self.relu5 = nn.ReLU()
        self.fc2 = nn.Linear(256, 10)  # Output layer with 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.relu3(x)
        x = self.pool3(x)
        x = self.conv4(x)
        x = self.relu4(x)
        x = self.pool4(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu5(x)
        x = self.fc2(x)
        return x
```



```python
# Training parameters
from torch.types import Device
DEVICE = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model = LeNet_Adjusted_4cov_2fc().to(DEVICE)
num_epochs = 10
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# Defining criteria as MSE Loss
criterion = nn.MSELoss()


train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

# Training loop
for epoch in range(num_epochs):
    print("Running ", epoch+1, "out of ", num_epochs)

    train_loss, train_accuracy = train(model, trainloader, optimizer, criterion, DEVICE)
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    test_loss, test_accuracy = test(model, testloader, criterion, DEVICE)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    print(f'Epoch [{epoch+1}/{num_epochs}] | '
            f'Train Loss: {train_loss:.4f} | Train Accuracy: {train_accuracy:.2f}%, '
            f'Test Loss: {test_loss:.4f} | Test Accuracy: {test_accuracy:.2f}%')

# Priniting the initial test accuracy
print(f'Initial Test Accuracy: {test_accuracies[0]:.2f}%')
# print(f'Initial Test Accuracy: {test_accuracies[0]:.2f}%')

# Visualize training loss and accuracy trends
plt.figure(figsize=(18, 6))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')

plt.show()
```

```
Running  1 out of  10
Epoch [1/10] | Train Loss: 0.0002 | Train Accuracy: 94.21%, Test Loss: 0.0001 | Test Accuracy: 98.87%
Running  2 out of  10
Epoch [2/10] | Train Loss: 0.0000 | Train Accuracy: 99.02%, Test Loss: 0.0000 | Test Accuracy: 99.25%
Running  3 out of  10
Epoch [3/10] | Train Loss: 0.0000 | Train Accuracy: 99.30%, Test Loss: 0.0000 | Test Accuracy: 99.36%
Running  4 out of  10
Epoch [4/10] | Train Loss: 0.0000 | Train Accuracy: 99.49%, Test Loss: 0.0000 | Test Accuracy: 99.24%
Running  5 out of  10
Epoch [5/10] | Train Loss: 0.0000 | Train Accuracy: 99.62%, Test Loss: 0.0000 | Test Accuracy: 99.41%
Running  6 out of  10
Epoch [6/10] | Train Loss: 0.0000 | Train Accuracy: 99.69%, Test Loss: 0.0000 | Test Accuracy: 99.39%
Running  7 out of  10
Epoch [7/10] | Train Loss: 0.0000 | Train Accuracy: 99.75%, Test Loss: 0.0000 | Test Accuracy: 99.50%
Running  8 out of  10
Epoch [8/10] | Train Loss: 0.0000 | Train Accuracy: 99.80%, Test Loss: 0.0000 | Test Accuracy: 99.51%
Running  9 out of  10
Epoch [9/10] | Train Loss: 0.0000 | Train Accuracy: 99.83%, Test Loss: 0.0000 | Test Accuracy: 99.46%
Running  10 out of  10
Epoch [10/10] | Train Loss: 0.0000 | Train Accuracy: 99.83%, Test Loss: 0.0000 | Test Accuracy: 99.47%
Initial Test Accuracy: 98.87%
```



Tabular Comparison:

Adjustments	Initial Test Accuracy	Final Test Accuracy
4 Conv Layers, 4 FC layers, RELU, MSE Loss	**98.43%**	**99.12%**
3 Conv Layers, 3 FC layers, RELU, MSE Loss	**98.92%**	**99.29%**
4 Conv Layers, 2 FC layers, RELU, MSE Loss	**98.87%**	**99.47%**
3 Conv Layers, 2 FC layers, RELU, MSE Loss	**98.82%**	**99.47%**

Conclusion:

In this exploration of LeNet model variations, we observed that even with different layer configurations, all models achieved remarkable accuracy on the MNIST dataset. **The choice of MSE loss, although unconventional for classification tasks,** did not hinder the models' ability to learn and classify digits effectively. Reducing the number of layers did not significantly impact accuracy, demonstrating the robustness of the LeNet architecture.