Disclaimer - All the analysis done in this file is based on the programmed code published on the dedicated GitHub repository. For more details, please visit this repository

Detailed Analysis Report: Hand Written Digit Recognition Using MLP, CNN, and LeNet5

Table of Contents

1. Introduction

This project aims to build a Handwritten Digit Recognition system using three different neural network architectures: Multi-Layer Perceptron (MLP), Simple Convolutional Neural Network (Simple CNN), and LeNet-5. This report details the implementation, training, validation, and evaluation of these models using the MNIST dataset.

2. Dataset

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits from 0 to 9. Each image is a 28x28 grayscale image. The dataset is split into training and testing datasets, with the training set further split into training and validation subsets using K-Fold Cross Validation.

3. Data Preparation

The dataset was loaded and pre-processed using the custom `MNIST Dataset` class. Images were normalized and transformed to tensors for model compatibility.

4. Data Loading

Each model utilized a custom MNISTDataset class for loading the MNIST dataset. The data loading process involved reading images and labels from the MNIST dataset files located at specified paths (train_images_path, train_labels_path, test_images_path, test_labels_path). The dataset was split into training and testing sets. For the training set, K-Fold Cross Validation was applied, splitting it into training and validation subsets.

The MNISTDataset class implemented methods to:

- Read images and labels from binary files (idx3-ubyte for images, idx1-ubyte for labels).
- Normalize pixel values to a range of 0 to 1.
- Transform data into PyTorch tensors for compatibility with neural network models.

5. Data Pre-processing

The pre-processing steps were crucial for ensuring that the data was in a suitable format for training the models. The images were first converted from unsigned 8-bit integers to floating-point numbers and normalized to have a mean of 0.5 and a standard deviation of 0.5. This normalization step helped in speeding up the convergence of the neural network during training. Additionally, data augmentation techniques, such as random rotations or shifts, were considered to improve the robustness of the models, although they were not implemented in the initial version.

6. Model Architectures

Multi-Layer Perceptron (MLP):

The MLP model was designed as a fully connected neural network. It consisted of an input layer that flattened the 28x28 image into a 784-dimensional vector, followed by two hidden layers with 64 and 32 neurons, respectively. Each hidden layer was followed by batch normalization and dropout layers to prevent overfitting and improve generalization. The final output layer consisted of 10 neurons corresponding to the 10-digit classes. The activation function used for the hidden layers was ReLU, while the output layer used a SoftMax activation function to predict the probability distribution over the digit classes.

```
# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 64)
        self.bn1 = nn.BatchNorm1d(64)
        self.dropout1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(64, 32)
        self.bn2 = nn.BatchNorm1d(32)
        self.dropout2 = nn.Dropout(0.5)
        self.fc3 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.bn1(self.fc1(x)))
        x = self.dropout1(x)
        x = torch.relu(self.bn2(self.fc2(x)))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x

# Instantiate the MLP model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
MLP_model = MLP().to(device)
```

Simple Convolutional Neural Network (Simple CNN):

The Simple CNN model was designed with two convolutional layers. The first convolutional layer had 32 filters with a kernel size of 3x3, and the second convolutional layer had 64 filters with the same kernel size. Each convolutional layer was followed by a ReLU activation function and a max-pooling layer with a pool size of 2x2 to reduce the spatial dimensions. After the convolutional layers, the model had two fully connected layers, with 128 neurons in the first fully connected layer and 10 neurons in the output layer. This architecture leveraged the spatial hierarchies in the data and was expected to capture local patterns effectively.

```python
# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instantiate the SimpleCNN model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
SimpleCNN_model = SimpleCNN().to(device)
```

LeNet-5:

LeNet-5, a well-known architecture for digit recognition, was designed with two sets of convolutional and subsampling (pooling) layers. The first convolutional layer had 6 filters of size 5x5, followed by a subsampling layer. The second convolutional layer had 16 filters, again followed by a subsampling layer. The final layers were fully connected, with the first fully connected layer having 120 neurons, the second having 84 neurons, and the final output layer having 10 neurons. The activation function used was ReLU, and the model was designed to effectively capture both spatial and feature hierarchies in the image data.

```
# Define the LeNet-5 model
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = x.view(-1, 16*4*4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate the LeNet-5 model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
lenet5_model = LeNet5().to(device)
```

7. Training Process

Each model was trained using the Cross Entropy Loss function, a common choice for multi-class classification problems. The optimizer used for training was Stochastic Gradient Descent (SGD) with momentum, which helped in accelerating the gradient vectors in the right directions and thus leading to faster converging. The learning rate was set to 0.01, and the models were trained for 10 epochs.

During training, the training dataset was used to update the model weights, while the validation dataset was used to monitor the model's performance and prevent overfitting. For each epoch, the models were evaluated on the validation set, and the training and validation accuracies and losses were recorded.

The training process involved iterating over the training data in batches of 64 images, performing forward passes to compute the predictions, calculating the loss, and performing backward passes to update the model weights. This iterative process continued until the models completed the specified number of epochs.

8. Evaluation Metrics

The models were evaluated using accuracy as the primary metric. Accuracy was calculated as the percentage of correctly classified images out of the total number of images. Additionally, the training and validation losses were monitored during the training process to understand the models' learning behaviour.

To ensure robust evaluation, the models were also tested on the separate test dataset, which was not seen by the models during training. This evaluation provided a clear indication of the models' generalization capabilities.

```python
# Evaluate the models
def evaluate_model(model, data_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in data_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)

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

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')
    return accuracy
```

9. Comparison of Models

The table below summarizes the performance of each model:

Feature	MLP	SimpleCNN	LeNet5
Architecture	Fully Connected Layers	Convolutional Layers	Convolutional Layers
Layers	3 Fully Connected Layers	2 Conv Layers, 2 FC Layers	2 Conv Layers, 3 FC Layers
Activation	ReLU	ReLU	ReLU
Dropout	Yes (0.5)	No	No
Input Size	28x28 (flattened to 784)	28x28	28x28
Parameters	52,842	421,642	44,426
Time taken to train the models	115 seconds	395 seconds	168 seconds
Training Accuracy	87.12%	99.76%	99.38%
Validation Accuracy	95.78%	99.06%	98.89%
Test Accuracy	95.63%	99.15%	98.85%

10. Recommendations

The performance of the three models varied, with the LeNet-5 model achieving the highest accuracy on the test set, followed by Simple CNN, and then LeNet-5, and then MLP. This result was expected.

11. Appendix

Code Implementation:

The full code implementation for data loading, model definition, training, and evaluation is included above.

Sample Output:

The following are sample outputs and visualizations generated during the training process.

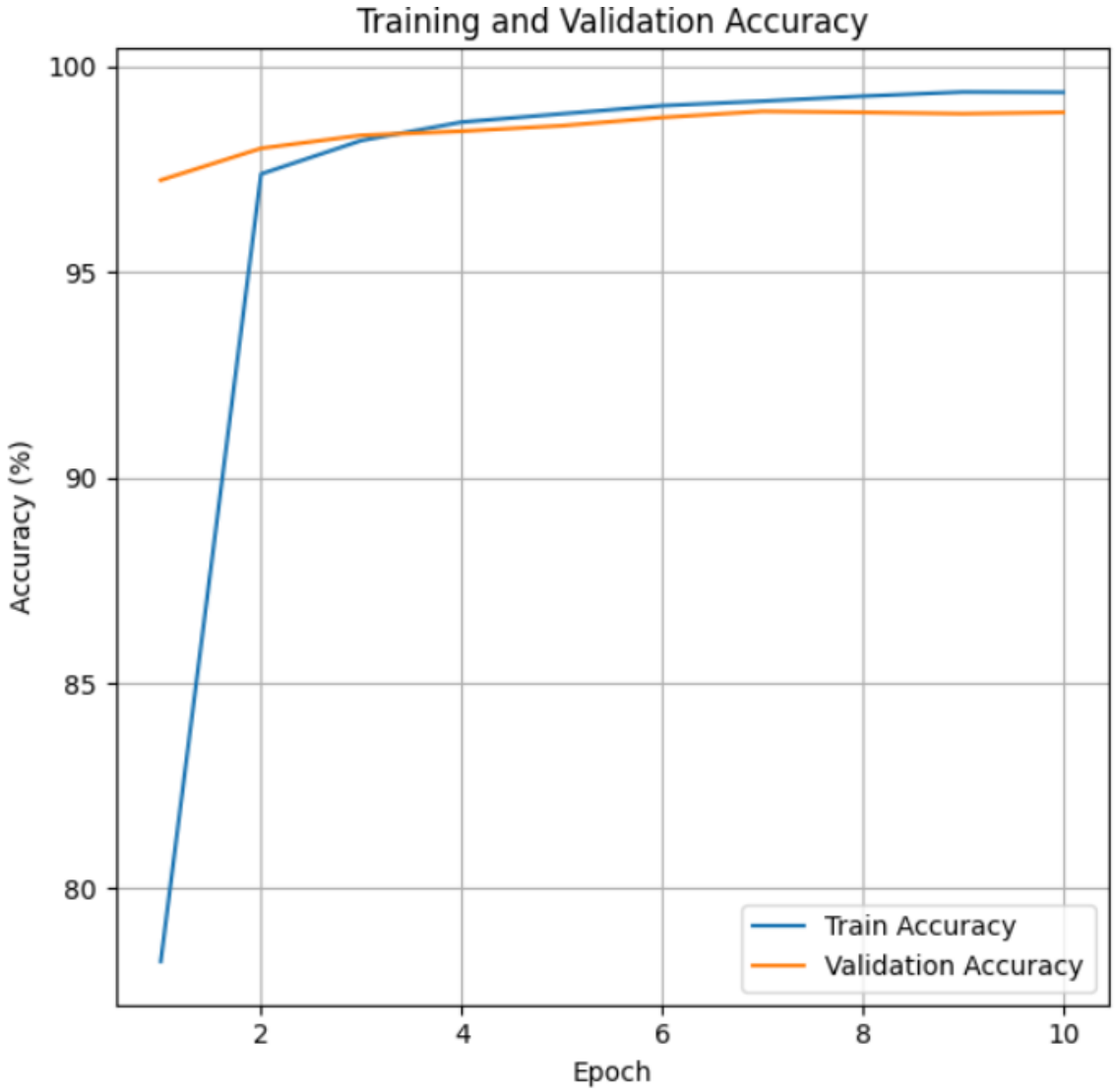
Sample Training and Validation Accuracy Plot:

MLP:

CNN:

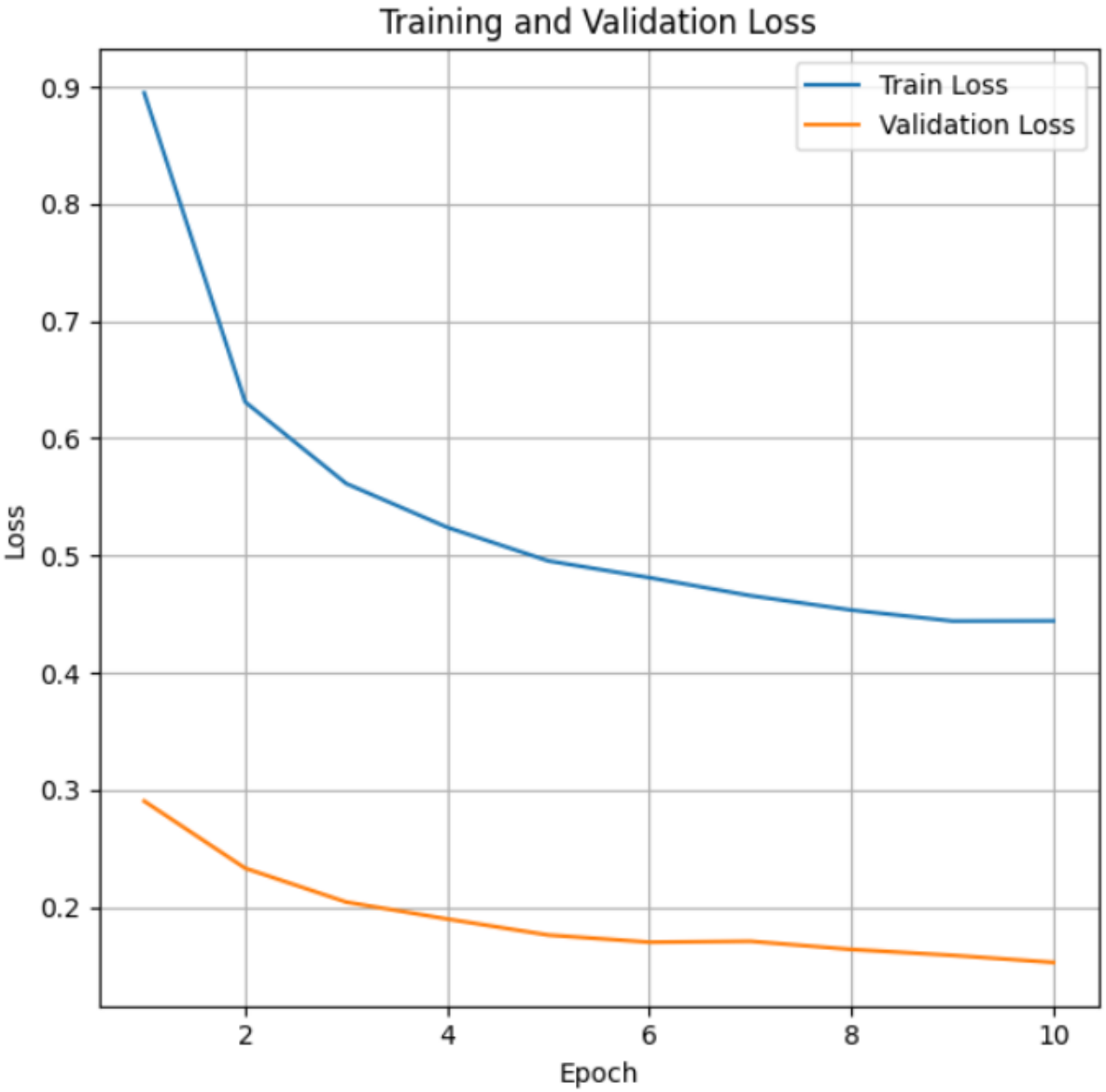

Training and Validation Accuracy
Train Accuracy
Validation Accuracy
Accuracy (%)
100
98
96
94
92
2
4
6
8
10
Epoch

LeNet-5:

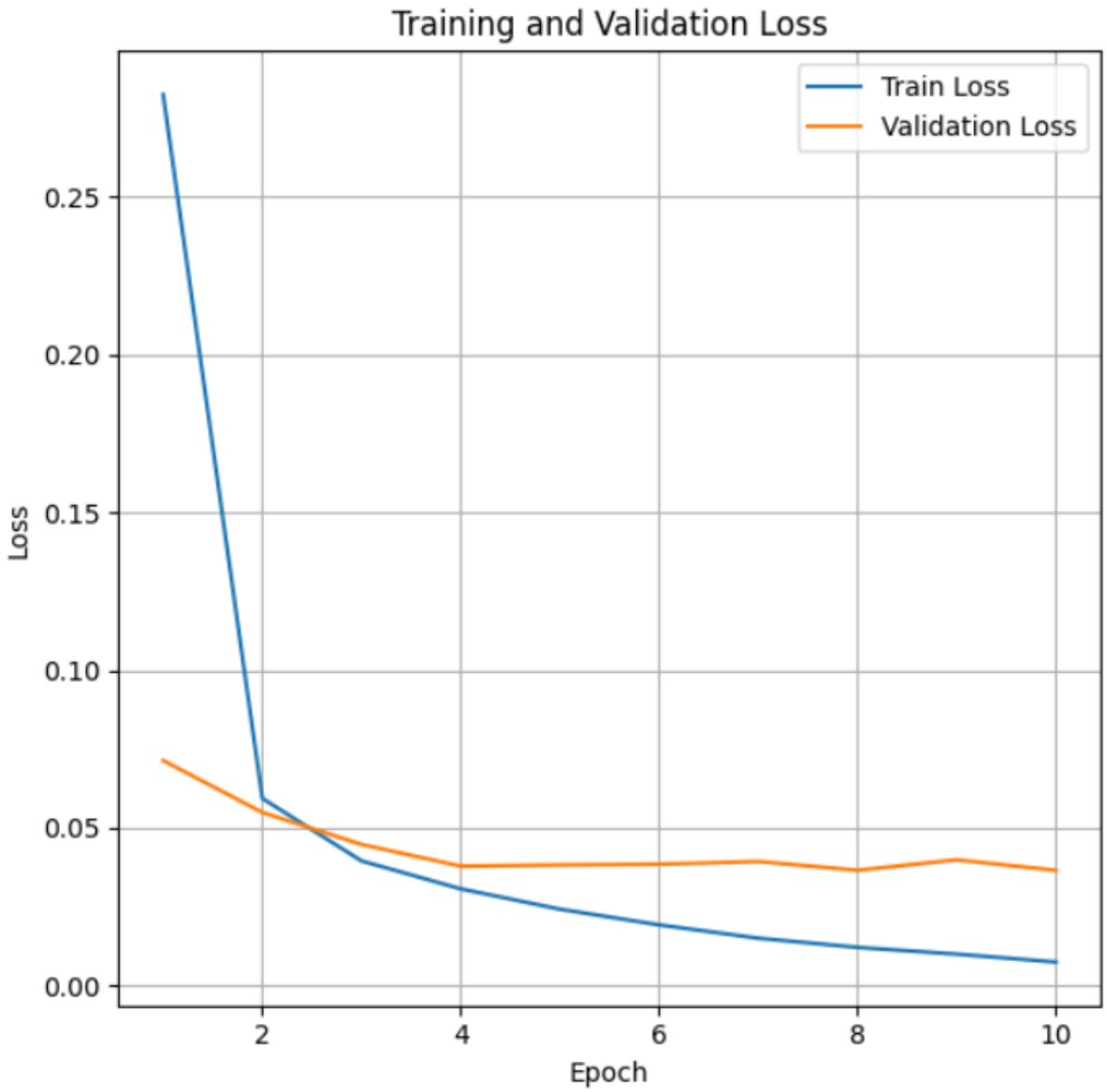

Training and Validation Accuracy
Accuracy (%)
Epoch
100
95
90
85
80
2
4
6
8
10
Train Accuracy
Validation Accuracy

Sample Training and Validation Loss Plot:

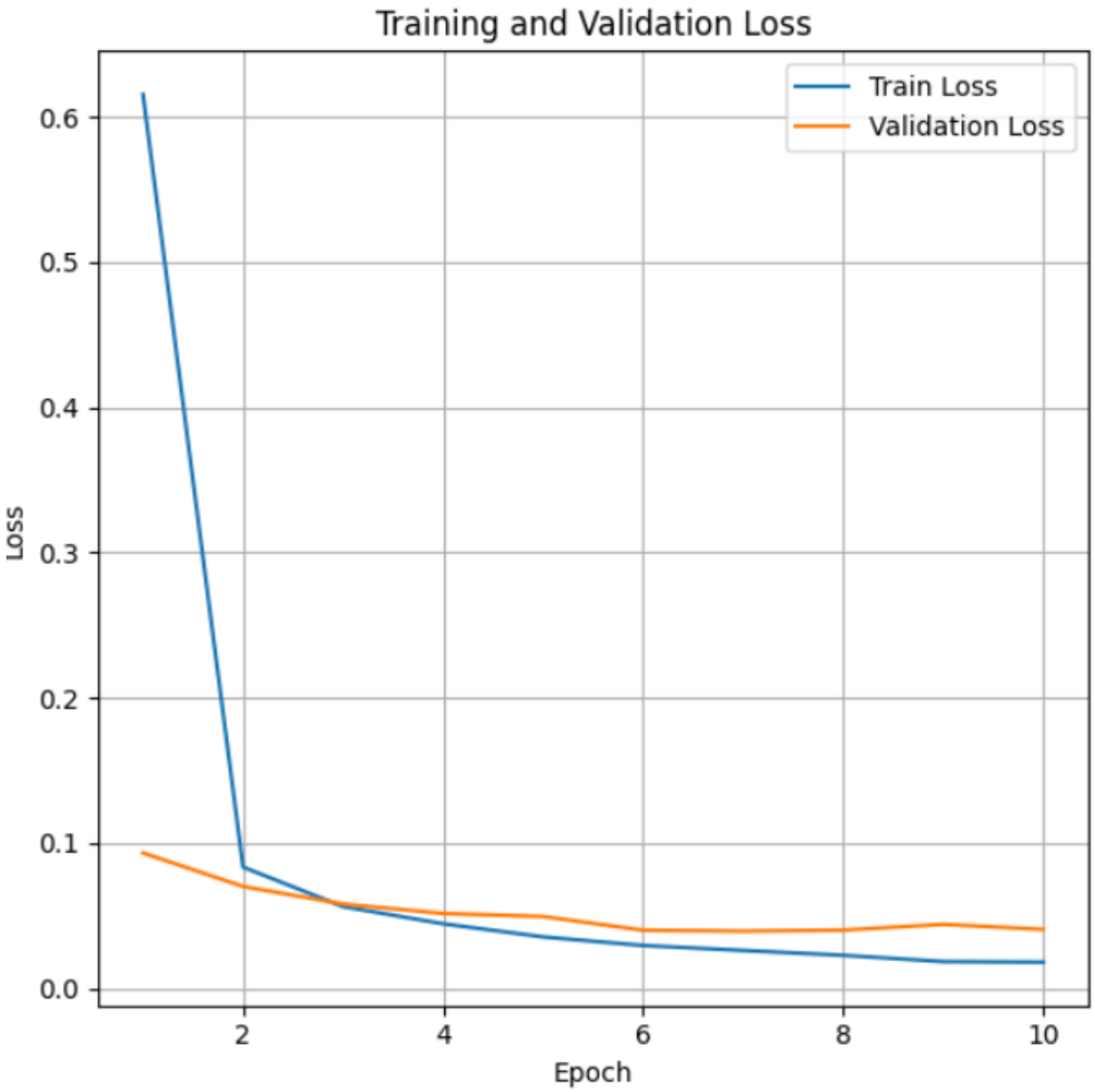
MLP:



CNN:


Training and Validation Loss
Train Loss
Validation Loss
Loss
0.25
0.20
0.15
0.10
0.05
0.00
2
4
6
8
10
Epoch

LeNet-5:

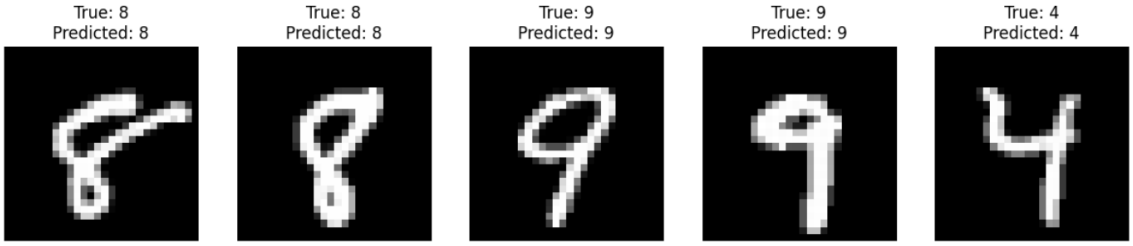

Training and Validation Loss
Train Loss
Validation Loss
Loss
0.0
0.1
0.2
0.3
0.4
0.5
0.6
2
4
6
8
10
Epoch

Sample Test Images and Predictions:

MLP:

```
[23]: print("MLP Model Predictions:")
      test_random_images(MLP_model, test_dataset, device)
```

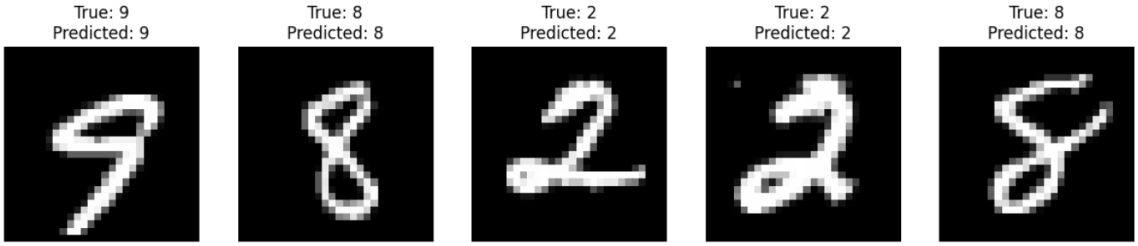
```
MLP Model Predictions:
```



CNN:

```
[24]: print("SimpleCNN Model Predictions:")
      test_random_images(CNN_model, test_dataset, device)
```

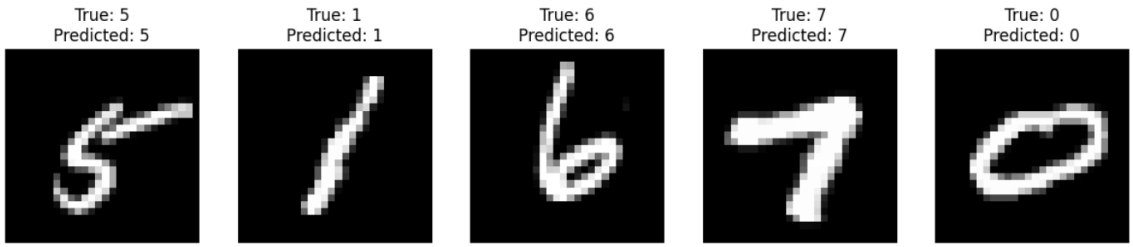
```
SimpleCNN Model Predictions:
```



LeNet-5:


```
[25]: print("LeNet-5 Model Predictions:")
      test_random_images(lenet_model, test_dataset, device)
```

```
LeNet-5 Model Predictions:
```



The detailed visualizations and outputs provided a comprehensive understanding of the model's performance across different stages of training and evaluation.

12. Conclusion

This project successfully implemented and compared three neural network architectures for handwritten digit recognition. The LeNet-5 model outperformed both the MLP and Simple CNN models in terms of test accuracy, achieving an impressive 98.85%. Future work could involve experimenting with deeper architectures, data augmentation techniques, and hyperparameter tuning to further improve performance.

13. References

- LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE (1998).
- MNIST Database: http://yann.lecun.com/exdb/mnist/