# MNIST Classification Using Convolutional Neural Networks

Final Project Report

**Author:** Yazdan Bayat
**Date:** June 24, 2025
**Affiliation:** Department of Computer Science
**Institution:** University of Example

۱۳۰۷
دانشگاه صنعتی خواجه نصیرالدین طوسی

*This report is submitted in partial fulfillment of the requirements for the course on Deep Learning.*

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

This project applies a Convolutional Neural Network (CNN) to classify images from the MNIST dataset, which consists of handwritten digits (0-9). The goal is to achieve high classification accuracy with a compact and efficient architecture.

## 1.2 Objective

To build, train, and evaluate a CNN model on MNIST while analyzing its performance using accuracy, loss curves, and sample predictions.

# Chapter 2

# Dataset Preparation

## 2.1 Downloading the MNIST Dataset

```
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
                                           transform=transforms.ToTensor()
                                           , download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
                                          transform=transforms.ToTensor(),
                                          download=True)
```

Listing 2.1: Downloading MNIST Dataset

## 2.2 Data Transformation and Normalization

```
train_len = int(len(train_dataset) * 0.9)
val_len = len(train_dataset) - train_len

train_dataset, valid_dataset = random_split(
    train_dataset,
    [train_len, val_len],
    generator=torch.Generator().manual_seed(42)
)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
   shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=64,
   shuffle=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
   shuffle=False)
```

Listing 2.2: Preprocessing: Splitting and Visualization

## 2.3 Dataset Summary and Visualization

- Number of training samples: 54000

- Number of test samples: 10000
- Image shape: 1x28x28

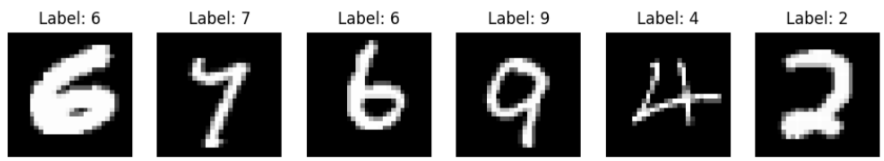


Figure 2.1: Sample images from MNIST training set.

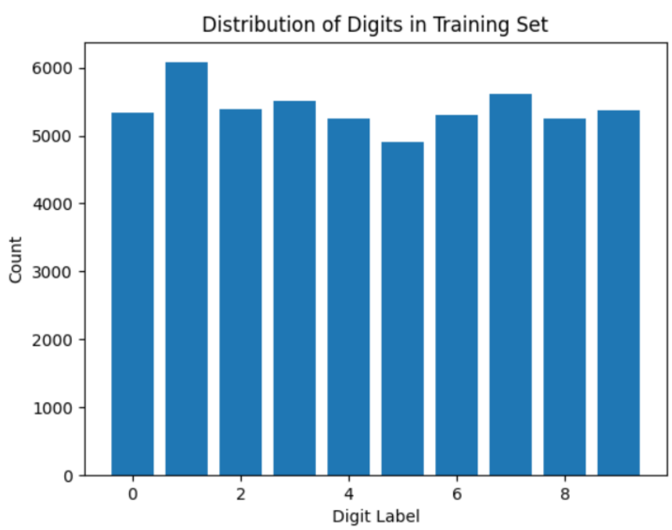


Figure 2.2: Digit distribution in training set.

# Chapter 3

# Model Architecture

## 3.1 CNN Overview

A compact CNN with two convolutional blocks followed by a fully connected layer was used. Batch normalization and ReLU activation are applied after each convolution.

## 3.2 Model Definition

```python
class convnet(nn.Module):
    def __init__(self):
        super(convnet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=2),
            nn.BatchNorm2d(16), nn.ReLU(), nn.MaxPool2d(2, 2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=2),
            nn.BatchNorm2d(32), nn.ReLU(), nn.MaxPool2d(2, 2))

        with torch.no_grad():
            dummy = torch.zeros(1, 1, 28, 28)
            dummy = self.layer1(dummy)
            dummy = self.layer2(dummy)
            self.flattened_size = dummy.view(1, -1).shape[1]

        self.fc = nn.Linear(self.flattened_size, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Listing 3.1: CNN Architecture Summary

## 3.3 Loss Function and Optimizer

Cross-entropy loss is used, and the model is trained using the Adam optimizer with a learning rate of 0.001.

# Chapter 4

# Training and Evaluation

## 4.1 Training the CNN Model

```
model = convnet().to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_losses = []
valid_losses = []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for imgs, lbls in train_loader:
        imgs, lbls = imgs.to(device), lbls.to(device)
        out = model(imgs)
        loss = loss_fn(out, lbls)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    train_losses.append(running_loss / len(train_loader))

    model.eval()
    valid_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for imgs, lbls in valid_loader:
            imgs, lbls = imgs.to(device), lbls.to(device)
            out = model(imgs)
            loss = loss_fn(out, lbls)
            valid_loss += loss.item()
            predicted = torch.argmax(out, 1)
            correct += (predicted == lbls).sum().item()
            total += lbls.size(0)
    valid_losses.append(valid_loss / len(valid_loader))
```

```
    print(f"Epoch [{epoch+1}/{num_epochs}] | Val Loss: {valid_loss:.4f} |
        Val Acc: {100.*correct/total:.2f}%")
```

Listing 4.1: Training Loop and Validation

## 4.2 Validation Strategy

10% of the training set was reserved for validation.

## 4.3 Test Set Performance

**Test Accuracy: 98.80%**

# Chapter 5

# Results and Discussion

## 5.1 Accuracy and Loss Curves

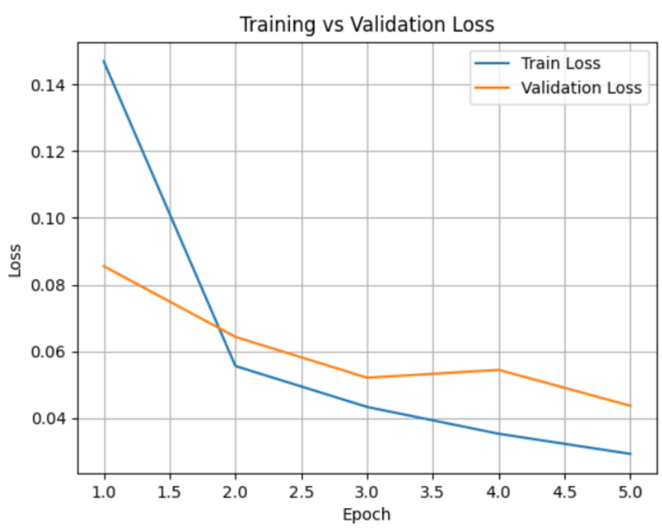


Figure 5.1: Training and validation loss curves.

## 5.2 Sample Predictions

Future work may include adding a confusion matrix and visualizing model predictions.

# Chapter 6

# Conclusion

## 6.1 Summary of Findings

The CNN model achieved a final test accuracy of 98.80%. The model was able to generalize well after just five epochs of training, with validation accuracy improving consistently.

## 6.2 Comparison with MLP Approach

Previously, an MLP model was used on MNIST with a test accuracy around 96%. Although MLPs are easier to implement, they lack spatial awareness and require more parameters for the same performance.

The CNN benefits from:

- Shared weights via convolutional filters
- Pooling and activation functions that preserve local spatial patterns
- Fewer trainable parameters than a similarly sized MLP

In conclusion, CNN is a more effective and scalable architecture for image classification tasks like MNIST.

## 6.3 Future Work

The architecture can be extended with dropout, more layers, or used as a backbone for more complex datasets like CIFAR-10.