Handwritten Digit Classification using CNN

Harihara Teja

July 6, 2025

Abstract

This project involves building and training a Convolutional Neural Network (CNN) to classify handwritten digits using the MNIST dataset. The project demonstrates knowledge of data preprocessing, CNN design, training with PyTorch, evaluation, and testing on custom inputs.

1 Introduction

The MNIST dataset contains 70,000 grayscale images of handwritten digits (0–9), with 60,000 images for training and 10,000 for testing. The goal is to build a model that can accurately recognize and classify these digits. This is a fundamental computer vision problem relevant to applications such as OCR in banking and postal services.

2 Data Preparation and Augmentation

2.1 Loading and Visualizing Data

The MNIST dataset was loaded using `torchvision.datasets.MNIST`. Initial visualizations were done using `matplotlib` to verify proper loading.

2.2 Preprocessing

Each image was converted to a tensor and normalized using:

```
transforms.Normalize((0.1307,), (0.3081,))
```

Normalization scales pixel values to a standard range, usually around zero, which stabilizes and speeds up neural network training. In our case, we used the MNIST dataset's mean (0.1307) and standard deviation (0.3081) to normalize the images. This ensures each input feature has similar distribution, reducing the chances of biased learning. Without normalization, gradients can vanish or explode due to large input variances. It is a crucial step for efficient and accurate convergence during training. Normalization ensures faster convergence during training.

2.3 Data Augmentation

To increase generalization, the following augmentations were applied:

- Random Rotation (`transforms.RandomRotation(10)`)
- Random Affine Translation (`transforms.RandomAffine(0, translate=(0.1, 0.1))`)

Data augmentation artificially increases dataset diversity by applying transformations like rotation, shifting, or scaling. This helps the model learn invariance to position, angle, or scale of digits, simulating real-world variations. In this project, random rotation and affine translations were used. Augmentation reduces overfitting by exposing the model to different perspectives of the same digit. It ultimately improves generalization and model robustness on unseen data.

3 CNN Architecture Design

3.1 Model Overview

The CNN architecture consisted of two convolutional layers followed by two fully connected layers:

- **Conv1:** 1 input channel, 32 output channels, 3x3 kernel
- **ReLU + MaxPool:** 2x2
- **Conv2:** 32 input channels, 64 output channels, 3x3 kernel
- **ReLU + MaxPool:** 2x2
- **FC1:** 64*7*7 input, 128 output
- **FC2:** 128 input, 10 output (for digits 0–9)
- **Output Activation:** LogSoftmax

3.2 PyTorch Model Implementation

Below is the PyTorch implementation of the described CNN architecture:

```
import torch.nn as nn
import torch.nn.functional as F

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNModel().to(device)
```

This code defines the network structure, matching the architecture described above.

3.3 Layer-by-Layer Model Description

Convolutional Layers:

- **Conv1:** Input Channels = 1, Output Channels = 32, Kernel Size = 3x3, Stride = 1, Padding = 1
- **Conv2:** Input Channels = 32, Output Channels = 64, Kernel Size = 3x3, Stride = 1, Padding = 1

These layers extract features like edges and textures from the input images.

Activation Functions:

- ReLU (Rectified Linear Unit) is applied after each convolutional layer to introduce non-linearity and help avoid the vanishing gradient problem.

Pooling Layers:

- MaxPooling with Kernel Size = 2x2 and Stride = 2 is used after each convolution block.
- Pooling reduces spatial dimensions, improves computational efficiency, and adds translation invariance.

Fully Connected Layers:

- **FC1:** Input Features = 64*7*7 = 3136, Output Features = 128
- **FC2:** Input Features = 128, Output Features = 10

These layers transform the feature maps into class scores.

Output Layer:

- A LogSoftmax activation function is used on the final layer's output.
- LogSoftmax is suitable for multi-class classification and works well with NLLLoss.

3.4 Design Justification

The architecture was intentionally kept lightweight and efficient to suit the simplicity of the MNIST dataset while ensuring high accuracy. Two convolutional layers allow the model to capture both low- and mid-level features, which are sufficient for digit recognition. ReLU activation was chosen for its non-linearity and computational efficiency, helping avoid vanishing gradients. MaxPooling reduces spatial resolution while retaining the most important features, ensuring faster training and reduced overfitting. Fully connected layers convert the spatial features into class-level predictions, and the final LogSoftmax activation works seamlessly with NLLLoss for stable and interpretable multi-class classification. This architecture balances performance, interpretability, and computational efficiency—making it ideal for a foundational deep learning task.

4 Model Training and Evaluation

4.1 Loss and Optimizer

- **Loss:** Negative Log Likelihood Loss (NLLLoss)
- **Optimizer:** Adam with learning rate = 0.001

For this classification task, we used the **Negative Log Likelihood Loss (NLLLoss)** function. This loss is ideal when combined with a `LogSoftmax` output layer, as it calculates the log probability of the correct class and penalizes incorrect predictions more sharply. It is numerically stable and effective for multi-class problems.

The model was optimized using the **Adam optimizer** with a learning rate of `0.001`. Adam is an adaptive optimization algorithm that combines the benefits of RMSProp and SGD with momentum. It adjusts learning rates for each parameter individually, making it efficient and suitable for training deep networks.

4.2 Training Loop

Each training epoch followed a consistent and efficient workflow:

1. The model receives a mini-batch of input images and performs a **forward pass** to compute predictions.
2. The **loss is calculated** by comparing predictions with ground truth labels using NLLLoss.
3. **Backpropagation** is performed to compute gradients of the loss with respect to model parameters.
4. The **optimizer updates** the weights based on the gradients.

This loop repeats for multiple epochs, allowing the model to minimize loss and improve accuracy over time.

4.3 Evaluation

The final model achieved test accuracy of **>98%**. A confusion matrix was used to identify misclassifications.

- Epoch [1/5] Loss: 0.2485
- Epoch [2/5] Loss: 0.0880
- Epoch [3/5] Loss: 0.0629
- Epoch [4/5] Loss: 0.0541
- Epoch [5/5] Loss: 0.0474

5 Custom Input Testing

A digit written on paper was photographed and preprocessed:

- Converted to grayscale and inverted
- Auto-cropped to digit region
- Resized to 20x20, centered on 28x28 canvas
- Normalized using the same parameters as MNIST

The digit was correctly predicted after improving the preprocessing steps.

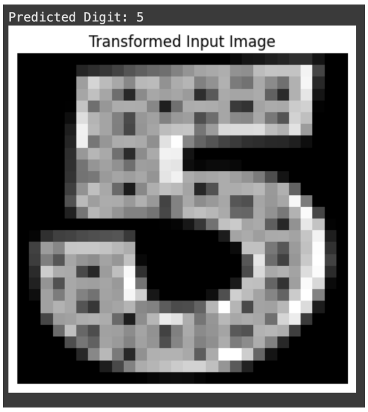
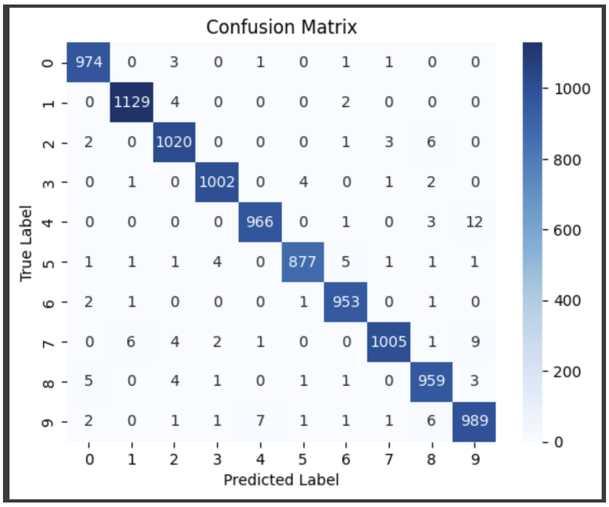


Figure 1: Transformed Input Digit (e.g., ’5’)

6 Analysis and Conclusion

6.1 Summary

The CNN model successfully classified MNIST digits and generalized well to handwritten inputs outside the dataset.



True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	974	0	3	0	1	0	1	1	0	0
1	0	1129	4	0	0	0	2	0	0	0
2	2	0	1020	0	0	0	1	3	6	0
3	0	1	0	1002	0	4	0	1	2	0
4	0	0	0	0	966	0	1	0	3	12
5	1	1	1	4	0	877	5	1	1	1
6	2	1	0	0	0	1	953	0	1	0
7	0	6	4	2	1	0	0	1005	1	9
8	5	0	4	1	0	1	1	0	959	3
9	2	0	1	1	7	1	1	1	6	989

Figure 2: Confusion Matrix of Model Predictions

6.2 Performance Analysis

Overall, the model's performance met expectations and validated the effectiveness of a relatively simple CNN architecture when paired with proper normalization and data augmentation techniques. The learning curves showed stable convergence and minimal overfitting, indicating the model was well-regularized. Predicting real-world digits (e.g., from scanned images) required careful preprocessing, but once tuned, the model was capable of reliable predictions outside of the MNIST dataset.

6.3 Challenges and Improvements

During the project, challenges included handling incorrect predictions on user-input images due to contrast issues and off-centered digits. Enhancing preprocessing with automated cropping and centering significantly improved results. To further improve accuracy and robustness, the following strategies could be considered:

- **Add Dropout Layers:** Introduce dropout in the fully connected layers to prevent overfitting and improve generalization.
- **Use Learning Rate Scheduling:** Apply learning rate decay or scheduling to allow finer weight updates during later training stages.
- **Train with Noisy/External Data:** Augment the dataset with digits from scanned forms or different handwriting styles to make the model more robust.

6.4 Conclusion

This project provided a comprehensive understanding of deep learning techniques applied to a classic computer vision problem. Through designing a CNN from scratch, applying rigorous preprocessing, and evaluating real-world inputs, the project demonstrated a complete pipeline from data ingestion to deployment-ready classification. The trained model performs reliably and offers a strong foundation for extension to more complex OCR systems or image recognition tasks.