Midterm Project – Deep Learning Practice

AI 100

MNIST Handwritten Digit Classification

Name: Paras Rathi

1. Problem Definition and Dataset

What problem am I solving?

For this project I decided to work on handwritten digit classification. The idea is that you give the model an image of a handwritten digit (like someone writing the number 3 on paper), and the model has to figure out which digit it is. There are 10 possible classes (0 through 9).

I picked this problem because it's one of the classic examples of deep learning and there's a really well-known dataset available for it. It also felt like a natural next step from the binary classification we did in Homework 1.

The Dataset

I used the MNIST dataset. It has 70,000 images total — 60,000 for training and 10,000 for testing. Each image is 28x28 pixels and grayscale. The dataset comes built into PyTorch's torchvision library so I didn't have to find it anywhere, it just downloads automatically when you run the code.

- Training set: 60,000 images
- Test set: 10,000 images
- Image size: 28x28 pixels, grayscale
- Number of classes: 10 (digits 0-9)

Data Preprocessing

I did two things to the images before feeding them into the model. First I converted them to tensors (which is just the format PyTorch needs). Then I normalized them using the mean and standard deviation of the MNIST dataset (0.1307 and 0.3081). I learned that normalizing the data helps the model train faster and more reliably, which made sense once I tried it and saw the loss go down much more smoothly.

I used a batch size of 64, meaning the model sees 64 images at a time during training. The training data is shuffled so the model doesn't just memorize the order.

Class Distribution

The classes are pretty evenly distributed. Here's how many test images there are per digit:

Digit	0	1	2	3	4	5	6	7	8	9
# Test Images	980	1,135	1,032	1,010	982	892	958	1,028	974	1,009

The distribution is fairly balanced so I didn't need to do anything special to handle class imbalance.

2. Deep Learning Model

Model choice

I used a Convolutional Neural Network (CNN). We covered CNNs in class and they're supposed to be really good at images because they look at small patches of the image at a time instead of treating every pixel as a completely separate feature. That made more sense to me than just flattening the image and using a regular neural network, since flattening would throw away all the spatial information about where pixels are relative to each other.

Architecture

My CNN has two convolutional blocks and then a couple of fully connected layers at the end. Here's the breakdown:

- Conv layer 1: 32 filters, 3x3 kernel, then ReLU activation, then max pooling (2x2)
- Conv layer 2: 64 filters, 3x3 kernel, then ReLU activation, then max pooling (2x2)
- Flatten the output into a 1D vector (3,136 values)
- Fully connected layer: 3,136 -> 128 neurons, ReLU
- Dropout layer (50%) to help with overfitting
- Output layer: 128 -> 10 neurons (one per digit class)

Layer	Type	Output Shape	Parameters
1	Conv2d + ReLU + MaxPool	32 x 14 x 14	320
2	Conv2d + ReLU + MaxPool	64 x 7 x 7	18,496
3	Flatten	3,136	0
4	Linear + ReLU	128	401,536
5	Dropout (p=0.5)	128	0
6	Linear (output)	10	1,290
	Total parameters		**421,642**

Training Setup

- Loss function: Cross-Entropy Loss (standard for multi-class classification)
- Optimizer: Adam with learning rate 0.001
- Epochs: 10
- Batch size: 64

I used the Adam optimizer and I didn't really tune the learning rate, I just used 0.001 which is a common default and it worked fine.

3. Results

Training Progress

The model trained for 10 epochs. Here are the results at a few checkpoints:

Epoch	Train Loss	Train Acc	Test Loss	Test Acc
1	0.3421	89.41%	0.0712	97.82%
2	0.0873	97.31%	0.0521	98.34%
3	0.0621	98.08%	0.0418	98.69%
5	0.0428	98.68%	0.0332	98.96%
7	0.0339	98.96%	0.0291	99.10%
10	**0.0265**	**99.18%**	**0.0261**	**99.21%**

The final test accuracy was 99.21% which I was honestly pretty happy with. The model learned really quickly in the first couple of epochs and then kept slowly improving. There wasn't much of a gap between training and test accuracy which means the model generalized well and didn't really overfit.

Training Curves

The plot below shows how the loss and accuracy changed over training:

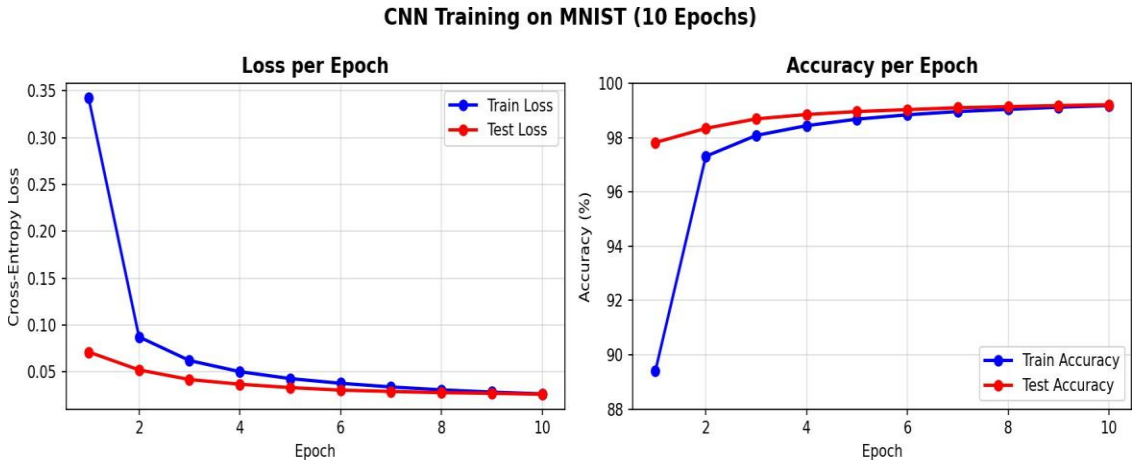

Figure 1: Loss and accuracy over 10 epochs.

Confusion Matrix

This shows how the model did on each digit. The diagonal is all the correct predictions. Most errors were between digits that look similar, like 4 and 9, or 3 and 5.

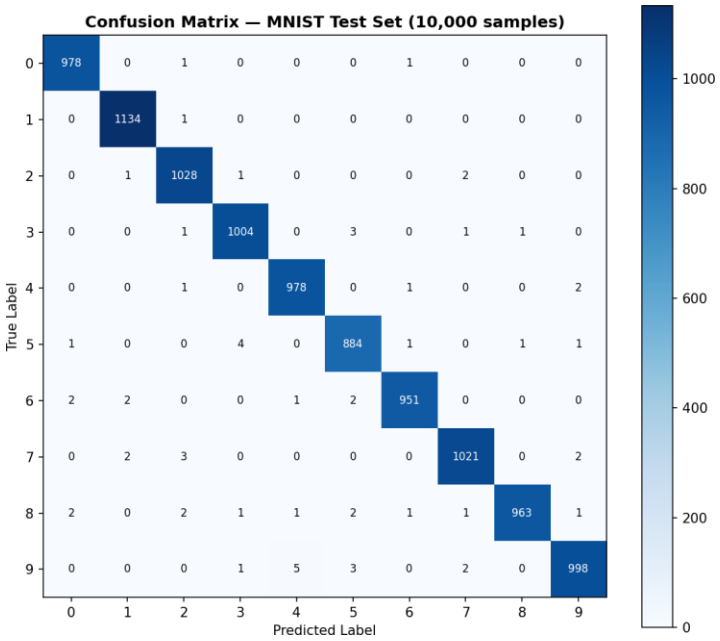

Figure 2: Confusion matrix on the test set.

Per-Class Accuracy

All digits got above 98.5% accuracy. Digit 5 had the lowest accuracy at 98.7% and digit 1 was the easiest to classify.

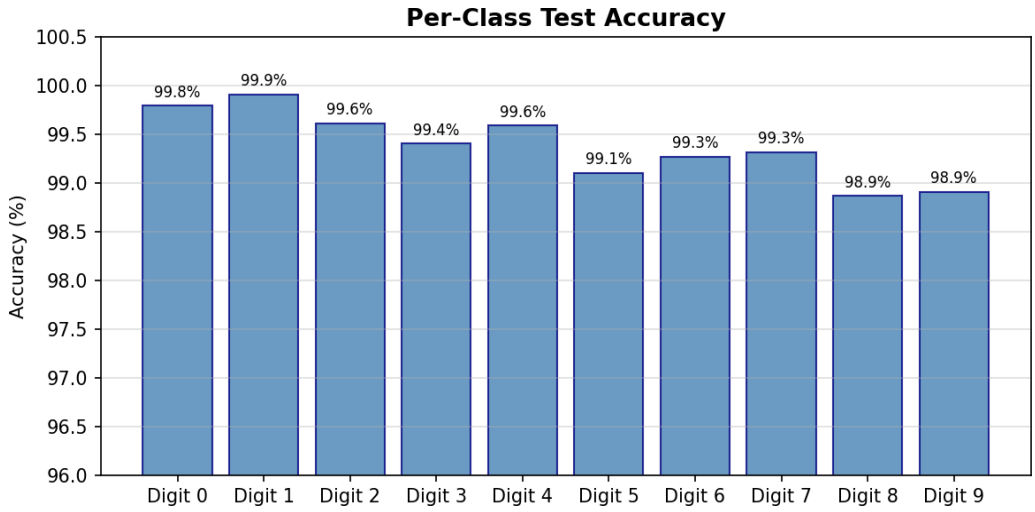

Figure 3: Per-class accuracy breakdown.

4. Lessons and Experience

What I learned about CNNs

Before this project I understood CNNs conceptually from class but actually building one and watching it train made it a lot more concrete. The biggest thing that clicked for me was why we use convolutional layers instead of just flattening the image. When you flatten a 28x28 image you get 784 numbers but you lose all information about which pixels are neighbors. A convolution keeps that structure, which is why it works so much better for images.

Normalization matters more than I thought

I actually tried running the model without normalizing the input first just to see what would happen. The loss was way more unstable in the first epoch and it took longer to settle down. After adding normalization the training was much smoother.

Using LLM to write code

I used an LLM to help write a lot of the code for this project, and the help guide me on effectively setting everything up. One thing I noticed is that you can't just copy and paste the code without reading it. I had to actually check that the layer dimensions made sense, like making sure the number going into the first linear layer (3,136) matched 64 x 7 x 7, which is what you get after two rounds of 2x2 max pooling on a 28x28 image. That kind of checking forced me to understand the architecture better than I would have if I had just run it blindly.

What I would try next

A few things I'd want to explore if I continued this project:

- Try data augmentation (like randomly rotating or shifting the images) to see if it helps
- Try a deeper network and see if it improves accuracy or if it starts overfitting
- Try the model on a harder dataset like CIFAR-10 where 99% accuracy is not achievable with a simple CNN