MNIST Handwritten Digit Classification Using Convolutional Neural Networks

Preyash Shah
24B2184

Contents

1 Introduction

The objective of this project is to design, train and evaluate a Convolutional Neural Network (CNN) for the classification of handwritten digits using the MNIST dataset. This project tests understanding of data preparation, CNN architecture, and the practical application of deep learning frameworks like PyTorch.

The MNIST dataset is a cornerstone of machine learning and computer vision. It contains 70,000 grayscale images of handwritten digits (0 through 9), split into a training set of 60,000 images and a testing set of 10,000 images. Each image is a small, 28×28 pixel square. Aim of this project is to build a model that can look at one of these images and correctly predict the digit it represents. This project mirrors real-world tasks like optical character recognition (OCR) in postal services and banking.

2 Data Preparation and Augmentation

The MNIST dataset was loaded using PyTorch's `torchvision.datasets`. To ensure that the data has been loaded correctly, a batch of images was visualized with their corresponding labels. Preprocessing included converting images to tensors and normalizing them using the dataset's mean (0.1307) and standard deviation (0.3081). Normalization is a crucial preprocessing step when training neural networks because it brings all input pixel values to a standard scale, which leads to faster convergence, better stability during training, and improved performance. Normalization transforms the data so that the pixel values have a mean of 0 and a standard deviation of 1. This centers the data and ensures that all features (pixels) contribute equally to the learning process. For the MNIST dataset, applied normalization using the dataset's empirical mean and standard deviation values:

- Mean (μ): 0.1307
- Standard Deviation (σ): 0.3081

These values were calculated by iterating through all images(in batches), summing pixel values and their squares, and then applying the standard formulas for mean and standard deviation.

Data augmentation techniques such as random rotation and translation were applied to the training set to artificially increase data diversity and improve model generalization, thereby reducing the risk of overfitting. By seeing different variations of the same digit during training, the model learns to recognize digits regardless of their position, orientation, or scale. This helps the network focus on invariant features of digits and improves its ability. As a result, the model achieves higher accuracy on the test set and introduces controlled randomness to training samples, reducing overfitting and improving generalization.

Augmentation Techniques Used

In this project, the following augmentation techniques were applied to the training dataset:

- **RandomRotation(10):** Rotates the image by a random angle in the range $[-10°, +10°]$ to simulate slanted handwriting.
- **RandomResizedCrop(28, scale=(0.7, 1.0), ratio=(0.9, 1.1)):** Randomly crops and resizes the image to the original size (28×28), introducing variation in the scale and aspect ratio of digits.
- **RandomAffine(0, translate=(0.1, 0.1)):** Applies random translations (up to 10% of image width and height) to mimic different digit placements within the image.

3 CNN Architecture Design

The convolutional neural network (CNN) used in this project is a multi-layer architecture designed to classify 28×28 grayscale handwritten digits from the MNIST dataset. The CNN model comprises two convolutional layers, each followed by ReLU activation and max pooling. The first layer has 32 filters (3×3 kernel), and the second has 64 filters (3×3 kernel). The output is then flattened and passed through a fully connected layer with 128 neurons. Finally, a second fully connected layer outputs class probabilities for the 10 digits using the LogSoftmax activation function.

Layer Type	**Parameters**
Conv2D #1	1→32, 3×3 kernel, stride=1, padding=1
ReLU	Activation function
MaxPool2D #1	2×2 kernel, stride=2
Conv2D #2	32→64, 3×3 kernel, stride=1, padding=1
ReLU	Activation function
MaxPool2D #2	2×2 kernel, stride=2
Flatten	-
Fully Connected	64×7×7→128
Output (FC)	128→10, LogSoftmax

Table 1: CNN Architecture

Activation Function: ReLU (Rectified Linear Unit) is applied after each convolutional layer to introduce non-linearity which helps to learn complex patterns and avoid vanishing gradients compared to sigmoid or tanh. ReLU is used here in MNIST because it enables fast and stable learning of digit features, leading to high accuracy and efficient training.
Pooling: Pooling reduces the spatial dimensions, preserving important features while reducing computation and overfitting risk. MaxPooling selects the most dominant features by taking the maximum value in each window.
Output Layer: The final layer applies LogSoftmax activation to output log-probabilities for 10 digit classes. This is suitable for multi-class classification tasks as each input belongs to exactly one class and also works with the NLLLoss used during training.

4 Model Training and Evaluation

The model was trained using the Adam optimizer (learning rate 0.001) and Negative Log Likelihood Loss(NLLLoss). Training was conducted over 10 epochs with batch size 64. The training loop included forward passes, loss computation, backpropagation, and weight updates.
After training, the model was evaluated on the test set. The final test accuracy achieved was 98.93%.

Loss Function

For this multi-class classification task, **Negative Log Likelihood Loss (NLLLoss)** was selected as the loss function, as it is directly connected to **LogSoftmax** as the activation function in the output layer of the model. LogSoftmax converts the raw network outputs (logits) into log-probabilities for each class. NLLLoss then measures how well the predicted log-probabilities align with the true class labels. This combination is mathematically stable and efficient. NLLLoss expects log-probabilities as input. This pairing avoids numerical instability and ensures correct gradient computation during training. Using CrossEntropyLoss would only be suitable if the output layer produced raw logits, since CrossEntropyLoss internally applies LogSoftmax before computing the loss.

Optimizer

The network is trained using **Adam optimizer** with a learning rate of `0.001`. The optimizer's role is to repeatedly update the model's parameters (weights and biases) to minimize the loss function. Adam (Adaptive Moment Estimation) achieves this by adaptive learning rate for each parameter based on estimates of moments of the gradients. This allows for faster and stable convergence compared to SGD. Also, 0.001 learning rate is good balance between speed and stability.

Training Loop Description

The model is trained over 10 epochs using the standard PyTorch training loop. Each epoch involves the following steps:

1. **Model Training Mode:** The model is set to training mode using `model.train()` to enable features like batch normalization.
2. **Iterate Over Batches:** For each batch of images and labels from the training set:
    - *Forward Pass:* Pass the batch through the model to obtain the predicted log-probabilities.
    - *Loss Calculation:* Compute the loss between the predicted outputs and the true labels using NLLLoss.
    - *Zero Gradients:* Reset the gradients of all model parameters to zero to prevent accumulation from previous batches.
    - *Backward Pass:* Gradients are computed via backpropagation by calling `loss.backward()`.
    - *Parameter Update:* The optimizer (Adam) updates the model parameters based on the computed gradients.

- *Loss Accumulation:* The loss for each batch is accumulated to compute the average training loss per epoch.

3. **Repeat:** Continue this process for all batches in the training set for the current epoch.

After each epoch, the model is evaluated on the test dataset using `model.eval()` mode, and the validation loss is computed without gradient updates. The training loop enables the model to learn by iteratively adjusting its parameters to minimize the loss, using efficient gradient-based optimization.

5 Analysis and Conclusion

Model Performance

The trained CNN model achieved a final test accuracy of **98.93%**, which is upto my expectations for MNIST using a two-layer convolutional architecture.

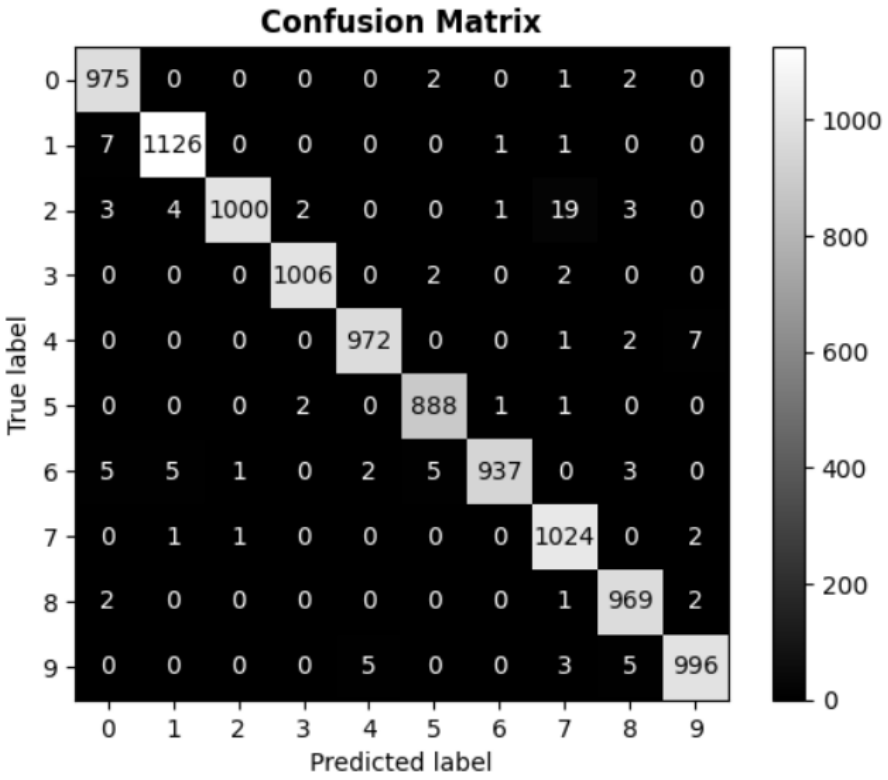

Figure 1: Confusion matrix

The confusion matrix in Figure reveals that the model occasionally confused digits with similar handwritten forms, particularly:

- **2 and 7:** The bottom line in 2 if not done correctly or minimized it looks like 7.
- **4 and 9:** If wick of 4 touches each other it looks like 9
- **6 and 5:** If round in 6 is not correctly done or incomplete it looks like 5.

These suggests that while the model performs well overall, but there should be more particular features to distinguish similar digit pairs.

Challenges Faced

- One challenge was data augmentation as it led to distorting digits excessively. As there were some issues like if I apply verticalflip transformation it would cause confusion between 6 and 9. Also, for some digits like 5 horizontalflip will distort the digit.
- Also, I first decided to include additional layer of dropout in model and it didn't affect accuracy much.
- I also faced some difficulty for writing this report in LATEX.
- Ensuring the model did not become biased towards certain digits.

.

Potential Improvements

To further improve accuracy and robustness, the following enhancements could be explored:

1. Applying a dynamic learning rate can help for better convergence and prevent getting stuck in local minima.
2. Adding a third convolutional layer or increasing filter sizes may help the network capture more complex patterns, potentially boosting accuracy on harder-to-classify digits.
3. Incorporating more advanced augmentations without affecting datasets.