```
Activating project at `~/GITHUB/Scientific-Programming-Handwritten-Digit-Recognition-Data-Augmentation`
```

Digit Recognition - Data Augmentation

1 - Introduction

Digit recognition enables machines to interpret and process handwritten or printed numerical data, bridging human input with digital systems.

Data augmentation helps in training for digit recognition by artificially expanding the training dataset through transformations like rotation or noise addition, allowing the model to generalize better for real-world usage.

2 - Motivation

In this project, we train and test models implemented with a convolutional neural network (CNN) architecture based on 'LeNet-5'.

We first train a model on most[1] of the full MNIST dataset (60.000 images) and test it on the MNIST test set (10.000 images). This represents our 'base-case', and its values serve as guide-values.

In real-world applications, usable data is often limited, and research time and computation power are constrained. Therefore our goal is to apply various augmentation methods to achieve comparable results using only fractions of the training data in a fraction of the time and processing power.

[1]More on this in section 4

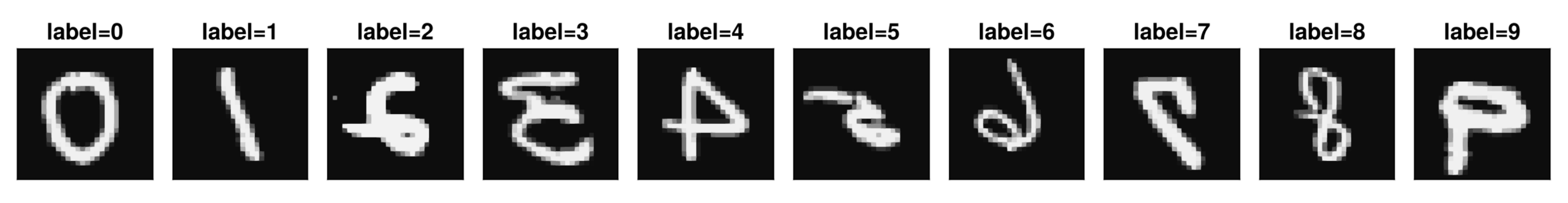
3 - Data Augmentation Methods

3.1 - No Augmentation

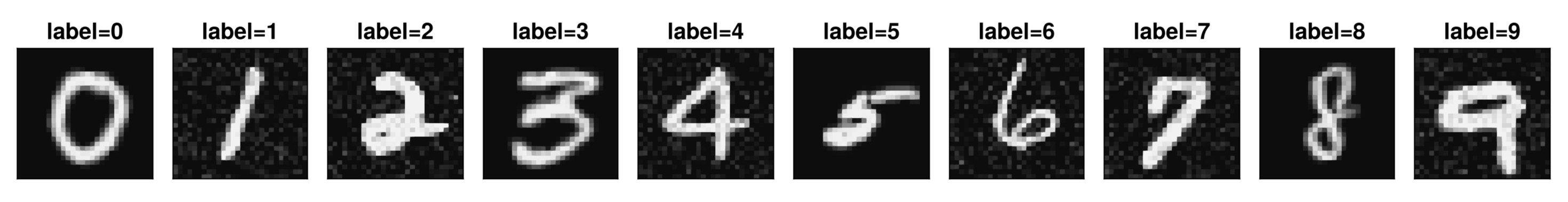
3.2 - Rotation

3.3 - Noise

3.4 - Flip (Mirror)



3.5 - Full Augmentation w/o Flip



4 - Trained Models

We train the base-case model, hereafter referred to as the 'fully trained model', on 5421 images of each digit (54210 images in total) to be able to reliably compare the accuracy per label in testing.

We train the models with fractional input on 542 images per digit (10% of the fully trained model) using the same training function, modifying only the augmentation methods to analyze their impact.

	Epochs	Batchsize	λ	η	Training Set Size
Fully Trained	20	32	0	0.0003	54210
Fractional	40	32	0	0.0003	5420

```
Epoch (36/40)...
trained for 0.501s | reached 0.026 loss with 99.613% accuracy
trained for 0.48s | reached 0.017 loss with 31.089% accuracy
trained for 0.523s | reached 0.014 loss with 100.0% accuracy
trained for 0.513s | reached 0.015 loss with 99.963% accuracy
trained for 0.461s | reached 0.014 loss with 99.815% accuracy
--------------------------------------------------------------------------------
Epoch (37/40)...
trained for 0.498s | reached 0.025 loss with 99.779% accuracy
trained for 0.478s | reached 0.016 loss with 34.668% accuracy
trained for 0.521s | reached 0.013 loss with 99.963% accuracy
trained for 0.523s | reached 0.014 loss with 98.838% accuracy
trained for 0.463s | reached 0.013 loss with 99.963% accuracy
--------------------------------------------------------------------------------
Epoch (38/40)...
trained for 0.498s | reached 0.021 loss with 99.373% accuracy
trained for 0.484s | reached 0.014 loss with 31.808% accuracy
trained for 0.524s | reached 0.012 loss with 99.723% accuracy
trained for 0.516s | reached 0.013 loss with 99.963% accuracy
trained for 0.468s | reached 0.011 loss with 99.41% accuracy
--------------------------------------------------------------------------------
Epoch (39/40)...
trained for 0.498s | reached 0.025 loss with 99.797% accuracy
trained for 0.48s | reached 0.016 loss with 30.664% accuracy
trained for 0.518s | reached 0.012 loss with 98.579% accuracy
trained for 0.514s | reached 0.013 loss with 100.0% accuracy
trained for 0.461s | reached 0.011 loss with 99.852% accuracy
--------------------------------------------------------------------------------
Epoch (40/40)...
trained for 0.497s | reached 0.022 loss with 99.557% accuracy
trained for 0.479s | reached 0.016 loss with 32.362% accuracy
trained for 0.522s | reached 0.012 loss with 100.0% accuracy
trained for 0.523s | reached 0.012 loss with 99.871% accuracy
trained for 0.465s | reached 0.011 loss with 99.41% accuracy
```

5 - Evaluation

For evaluation, we measure the model's accuracy (total and per digit) on the test dataset and compare performance across different augmentation methods. This allows us to assess which methods most effectively compensate for the reduced training data and contribute to model accuracy.

After testing with 0.1%, 1%, 10%, 25%, and 50% of the training set, we chose to focus on the 10% subset. It provides a good balance between efficiency and performance, achieving more stable and reliable results (~95% accuracy) compared to 1% (around 60–70%). At the same time, it remains distinguishable from the fully trained model, which reaches about 99% accuracy.

Prediction Table: Shows the total accuracy, accuracy per digit and maximum difference of the 'accuracy per digit'
Confusion Matrix: Shows the counts of true and predicted classifications for each digit, highlighting where the model gets predictions right or wrong

5.1 - Tables

5.1.1 - Predictions

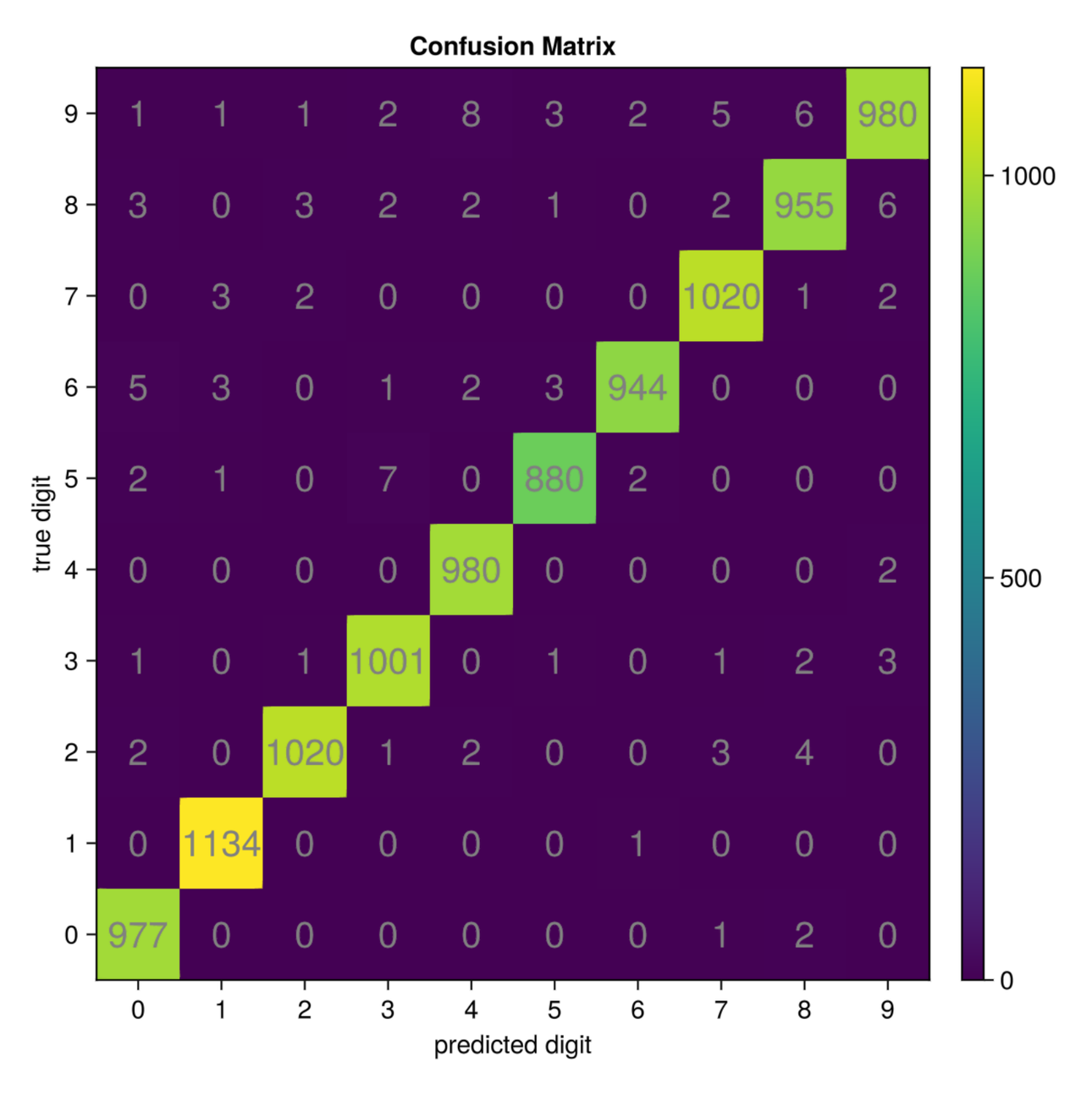
	0s	1s	2s	3s	4s	5s	6s	7s	8s	9s	Total	Δ Max
Fully Trained	99.69%	99.91%	98.84%	99.11%	99.8%	98.65%	98.54%	99.22%	98.05%	97.13%	98.91%	2.79
No Augmentation	98.88%	98.33%	98.84%	98.12%	93.18%	96.52%	96.45%	96.79%	94.25%	98.02%	96.98%	5.7
Full Augmentation	97.04%	99.38%	98.35%	95.64%	98.68%	96.86%	98.43%	96.11%	97.84%	96.33%	97.49%	3.74
Only Rotation	98.78%	99.12%	97.97%	94.75%	98.68%	96.86%	98.96%	96.79%	95.59%	95.04%	97.27%	4.37
Only Noise	98.88%	99.21%	95.83%	97.33%	96.13%	97.87%	96.66%	96.69%	96.1%	97.42%	97.23%	3.37
Only Flip	39.59%	63.79%	11.43%	1.39%	83.3%	2.8%	8.66%	5.64%	83.06%	28.64%	33.26%	81.91

5.1.2 - Training Times

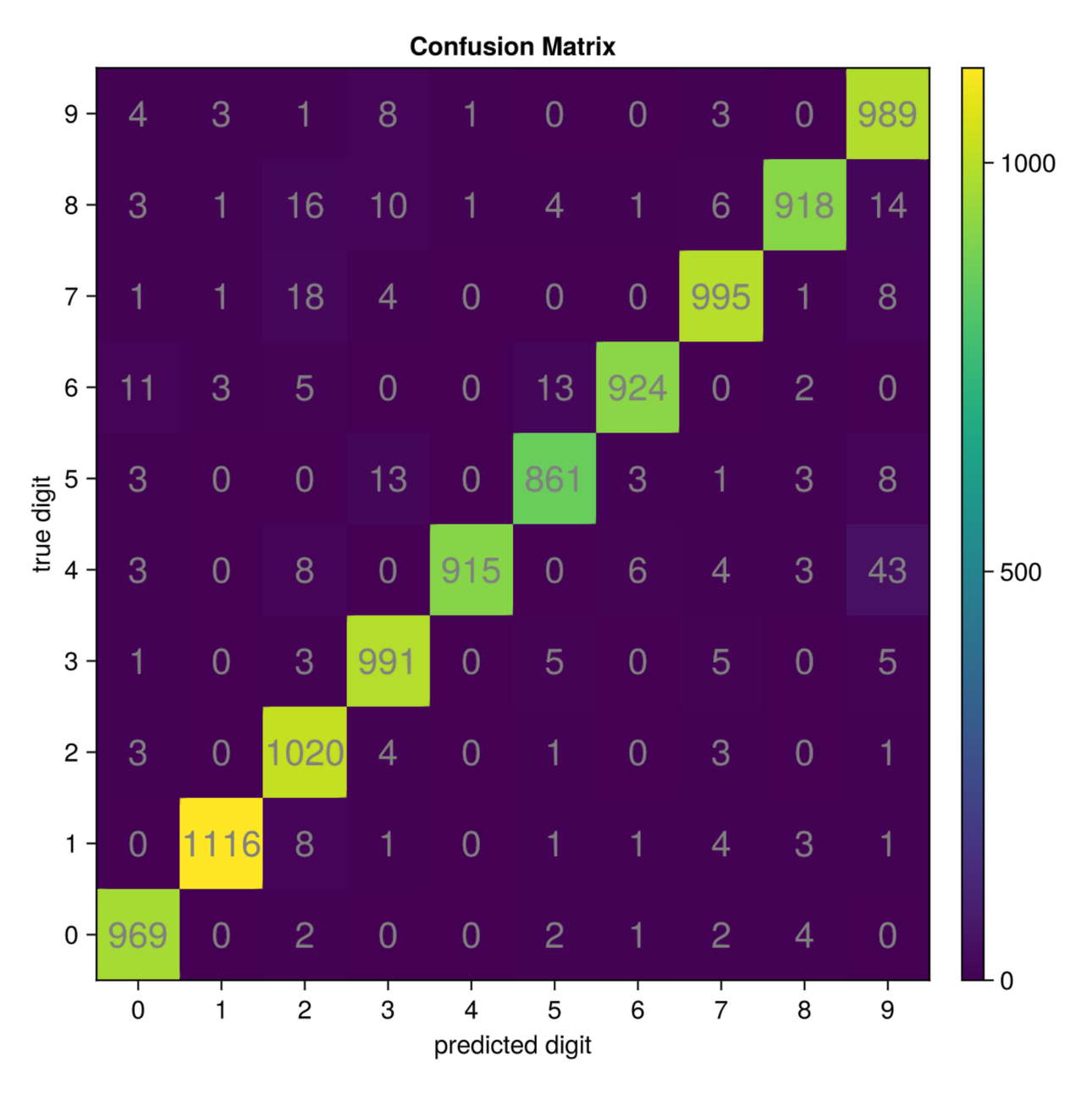
	Time per Epoch	Total Time
Fully Trained	~4.5s	~90s
Fractional (each)	~0.45s	~18s

```
Trained on M1 Pro, 16GB Ram
```

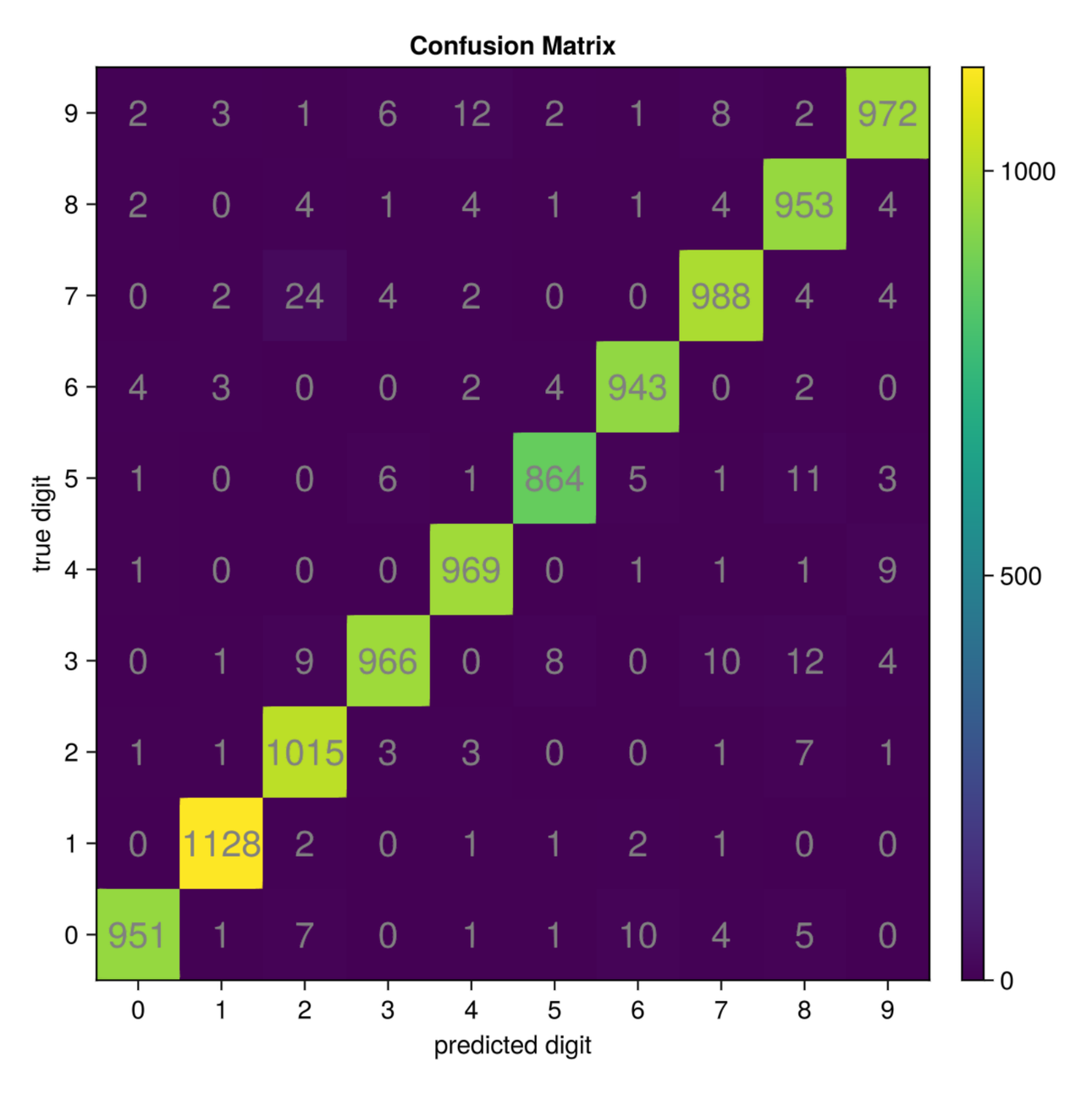
5.2 - Fully Trained Model



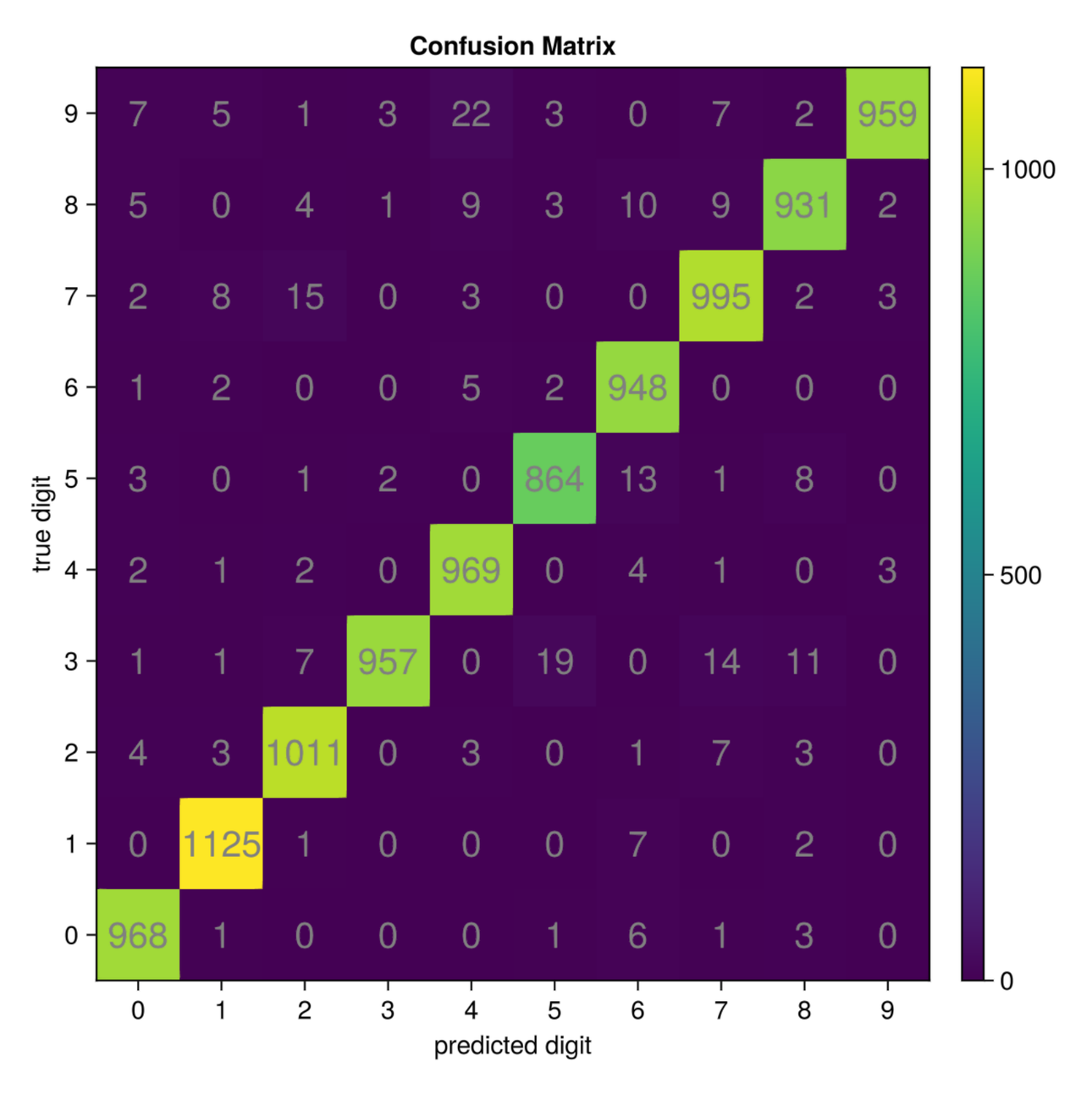
5.3 - Non-Augmented Model



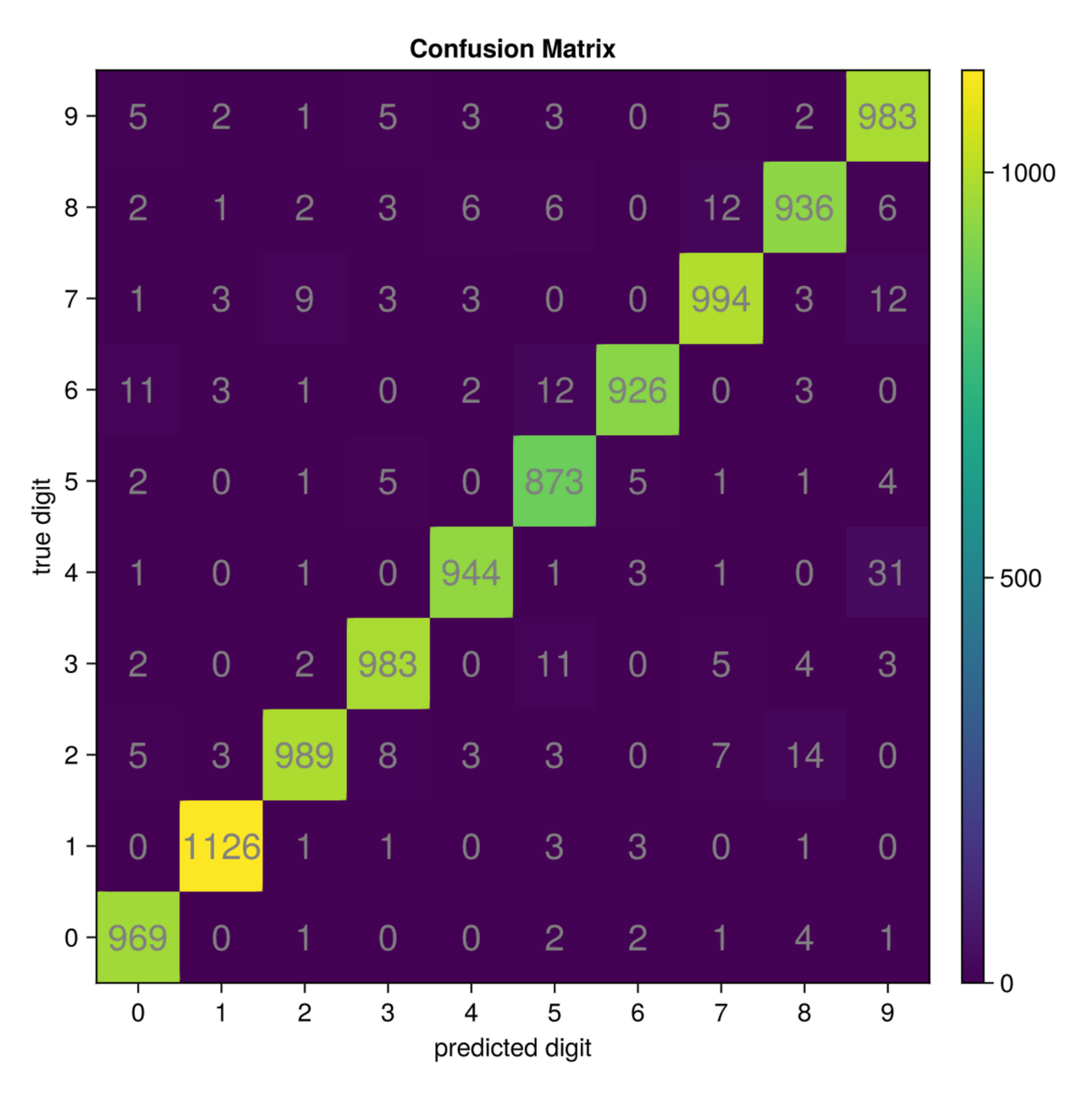
5.4 - Fully-Augmented Model



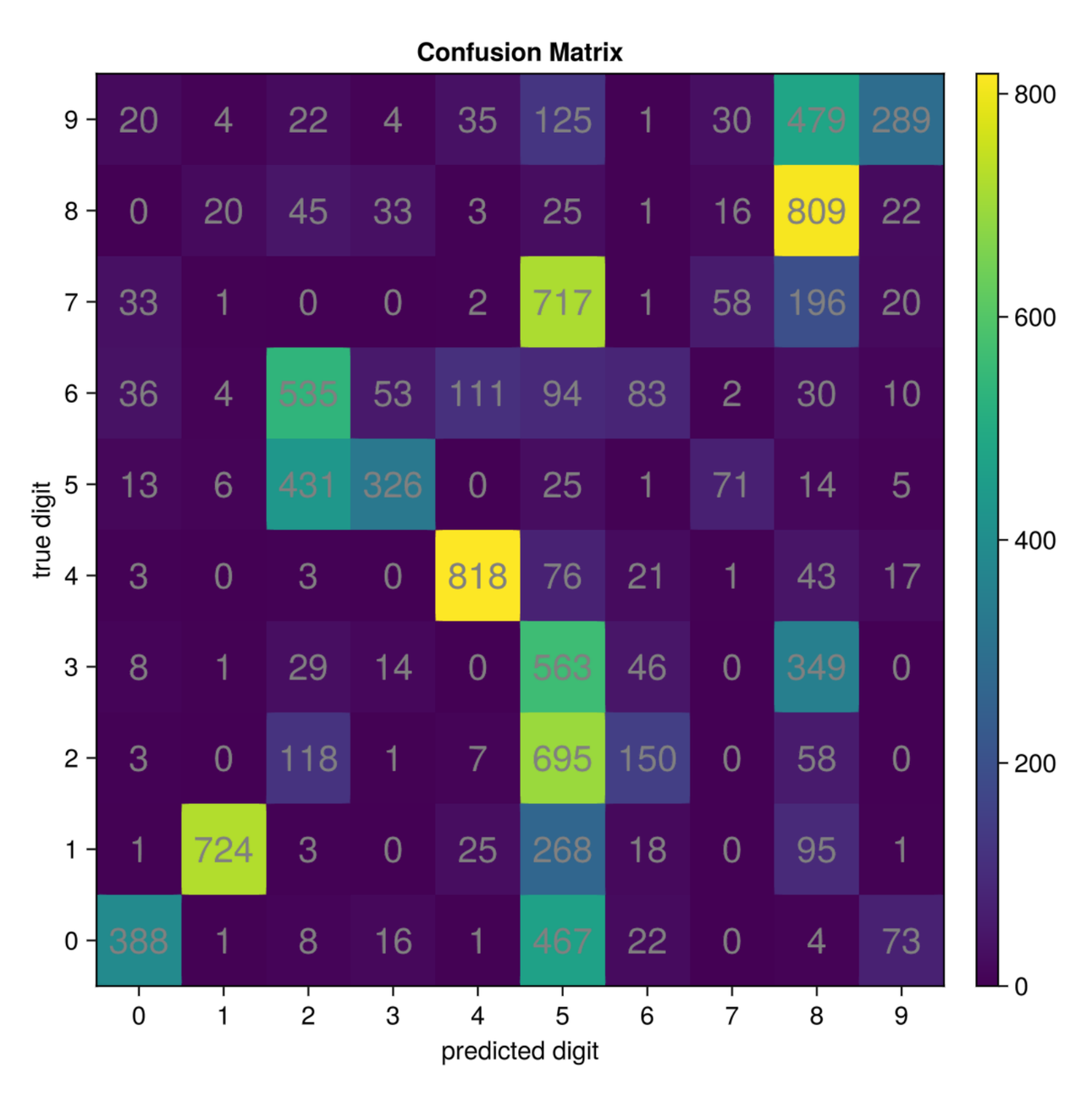
5.5 - Only Rotation Model



5.6 - Only Noise Model



5.7 - Only Flip (Mirror) Model



6 - Conclusion

We found that augmenting the dataset with rotation and noise, individually and in conjunction, helps stabilize the results and reduce variance between runs on average. Still, with only 10% of the training data, the model's performance is more sensitive to randomness. This leads to greater variability in accuracy across runs. On some runs the non augmented model achieved higher accuracy and less variance than the augmented models. This highlights the importance of consistent evaluation and multiple repetitions when working with limited data.

These augmentations should improve the model's ability to generalize from a smaller dataset, making performance more consistent and closer to that of models trained on the full dataset.
We only tested on the MNIST dataset, which limits our ability to assess generalization, as it does not reflect the diversity and complexity of real-world digit recognition tasks. Future work could invole testing on more challenging data sets with custom handwritten digits, to better evaluate model robustness.

The results of our models could improve if we limit specific augmentation methods to specific digits (e.g. to only use '0' and '8' in our flip (mirror) method), because these digits remain visually similar when mirrored, whereas others (like '2' or '5') can resemble different digits, potentially confusing the model.
Additionally, even more augmentation methods, training for multiple runs to test the stability of the accuracies, and futher improving the models and training algorithms may improve these results.

7 - Further Information

7.1 - Data Handling

MNIST dataset: A collection of 28×28 grayscale images of handwritten digits (0–9). Using the Julia package MLDatasets.jl, the data is preprocessed to match the input format expected by the LeNet architecture.

Each image is stored as a tensor of shape (28, 28, 1) — the third dimension representing the grayscale feature map. The dataset is also batched, so the full training set has the shape (28, 28, 1, N), where N is the number of images.

The labels are represented using one-hot encoding: a binary vector of length 10, where the position of the 1 indicates the corresponding digit class (e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] represents the digit 3). This encoding is a standard format used for classification tasks in machine learning.

7.2 - LeNet5

LeNet was originally designed to process 32×32 grayscale images; we adapt it here to handle MNIST's 28×28 images.

Our implementation consists of two convolutional-pooling blocks, followed by three fully connected layers. Each hidden layer uses the ReLU (Rectified Linear Unit) activation function, which introduces non-linearity and helps the network learn complex patterns by zeroing out negative values. The final output layer uses the softmax function to convert the network's raw outputs into a probability distribution over the 10 digit classes.

$$f : \mathbb{R}^{28\times28\times1} \to \mathbb{R}^{10}, \text{ with the whole function being:}$$

$$\mathbb{R}^{28\times28\times1} \xrightarrow{\text{Conv 1}} \mathbb{R}^{24\times24\times6} \xrightarrow{\text{Pool 1}} \mathbb{R}^{12\times12\times6} \xrightarrow{\text{Conv 2}} \mathbb{R}^{8\times8\times16} \xrightarrow{\text{Pool 2}} \mathbb{R}^{4\times4\times16} \xrightarrow{\text{Dense}} \mathbb{R}^{256} \xrightarrow{\text{Dense}} \mathbb{R}^{120} \xrightarrow{\text{Dense}} \mathbb{R}^{84} \xrightarrow{\text{Softmax}} \mathbb{R}^{10}$$

where the input matrix $\mathbb{R}^{28\times28\times1}$ is the MNIST image (28 × 28), and the output vector $\mathbb{R}^{10}$ represents the class probabilities for digits 0 through 9 calculated from the last softmax step.

7.3 - Training

Cross-entropy loss function: Commonly used for multi-class classification problems. Let $\hat{\mathbf{y}}_i$ denote the predicted probability distribution for the i-th sample, and $\mathbf{y}_i$ the corresponding one-hot encoded ground truth label. The cross-entropy loss over a batch $\mathbb{N}$ of samples is defined as:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$$

To minimize this loss, we use the ADAMW optimizer, a variant of the ADAM algorithm that incorporates weight decay to regularize the model and reduce overfitting. In our implementation, we also apply a weight decay lambda (L2 regularization parameter) to encourage smaller weights, which can improve generalization.