Generalization of an MNIST Trained MLP to Real Handwriting

Dylan Dorey, Ethan Harris, Halin Gailey, Jake Baartman

December 11, 2025

1 High-Level Framework of Our Solution

In this project we developed a handwritten digit classification model using a fully connected neural network (MLP) trained on the MNIST dataset. We evaluated the model on: the MNIST test set for baseline performance, our class dataset of handwritten digits, and per group digit subsets to evaluate variations in writing styles. The overarching view of our approach is listed as such:

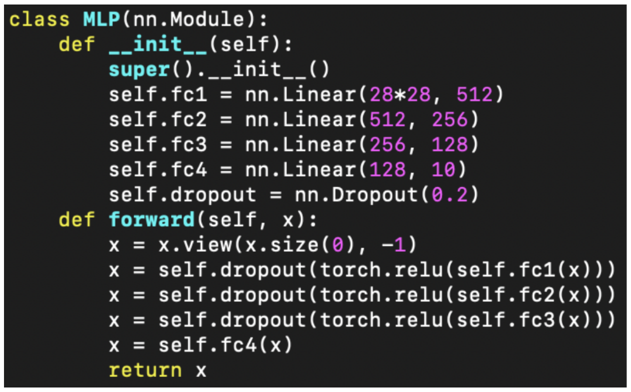
1. Data pre-processing for MNIST and class digits
2. Data augmentation to improve generalization
3. Hold out validation set with early stopping
4. A fully connected MLP classifier
5. Evaluation through confusion matrices, per class loss, and per group accuracy/loss

The goal is to explore how well our model generalizes from MNIST to real handwritten digits that were collected from every student in the class where writing style will have a large variation from those seen in the MNIST dataset.

2 Hyperparameters and Final Neural Network Architecture

2.1 Final Architecture

Our model is a fully connected feed forward neural network (MLP) designed for 28×28 grayscale digit images. Each image is flattened into a 784 dimensional input vector and passed through three hidden layers before a 10 dimensional output layer representing the digit classes 0-9. The final model (Figure 1) has an input layer of dimension 784 (flattened 28×28 image) with three fully connected hidden layers that decrease in dimension from 512 to 256 to 128. We originally started with a smaller model with hidden layers varying form $256 \rightarrow 128 \rightarrow 64$, but found that this slightly under fit the MNIST and performed worse on the class digits. Each hidden layer uses a ReLU activation function because it is standard for image classification, trains very efficiently, avoids the vanishing gradient problem, and introduces nonlinearity so that the MLP can learn complex shapes like our digits. We chose not to implement batch normalization because the model trained stably without it and adding more complexity to it was not necessary for this small problem size. Each hidden layer has a dropout regularization of 0.2 during training which deactivates 20% of the layer's neurons to prevent overfitting and to help generalize. The output layer is fully connected from dimension 128 to 10, outputting raw logits for the 10 digit classes. Its final probabilities are obtained using `CrossEntropyLoss` which combines `LogSoftmax` and negative log likelihood under the hood, telling us how confident our model is in its predictions.



```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.2)
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.dropout(torch.relu(self.fc1(x)))
        x = self.dropout(torch.relu(self.fc2(x)))
        x = self.dropout(torch.relu(self.fc3(x)))
        x = self.fc4(x)
        return x
```

Figure 1: Final Model Architecture: Fully connected feed forward neural network (MLP) designed for 28×28 grayscale digit images.

2.2 Final Hyperparameters

We experimented with a few combinations of learning rate, batch size, and training schedule before settling on a final set of hyperparameters that balanced performance, stability, and runtime since each run took ~ 10 seconds per epoch using GPU and ~ 20 seconds using CPU. This time adds up when cross validating hyperparameters, so we tried to keep it simple.

Learning Rate: We selected a learning rate of 1×10^{-3} after comparing 1×10^{-2}, 1×10^{-3}, and 1×10^{-4}. 1×10^{-2} produced unstable training and fluctuating loss, while 1×10^{-4} converged slowly and occasionally plateaued prematurely. On the other hand, 1×10^{-3} consistently yielded the best validation performance and smoothest training curve (Figure 2).

Optimizer: We used the Adam optimizer, which adaptively adjusts the learning rate during training and it typically converges faster than using SGD while also yielding better results specifically for our MLP model.

Batch size: We used a batch size of 128 because it was a good balance between training stability and efficiency. Using smaller batches like 64 slowed down training while yielding an almost identical loss and accuracy to that of a batch size of 128. Using large batches like 256 sped up training time, but decreased the test accuracy to $\sim 89\%$.

Maximum Epochs: Training was allowed up to 20 epochs, but the actual number was determined by early stopping with a patience of 3 (Section 3.2). This upper limit ensures that the model has enough opportunity to converge while preventing unnecessarily long training runs.

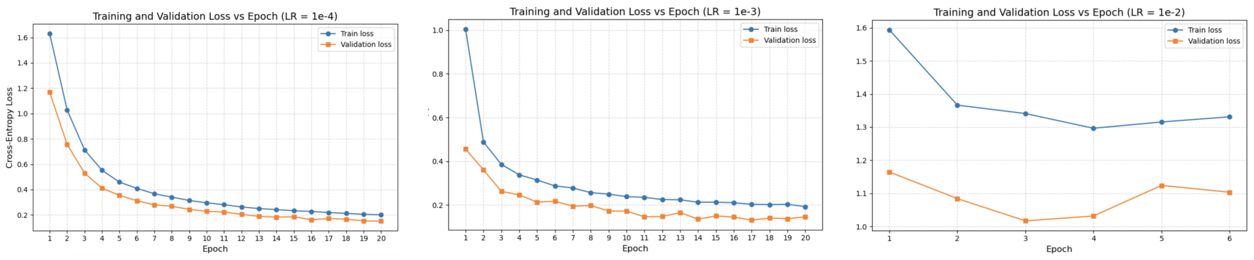


Figure 2: A learning rate of 1×10^{-4} yielded a class digits loss of 0.3164 and an accuracy of 89.72%. LR = 1×10^{-3} yields the lowest loss of 0.2337 and the highest accuracy of 92.83%. LR = 1×10^{-2} yields the highest loss of 1.1868 and the lowest accuracy of 61.99%. Each of the above plots was produced using a batch size of 128.

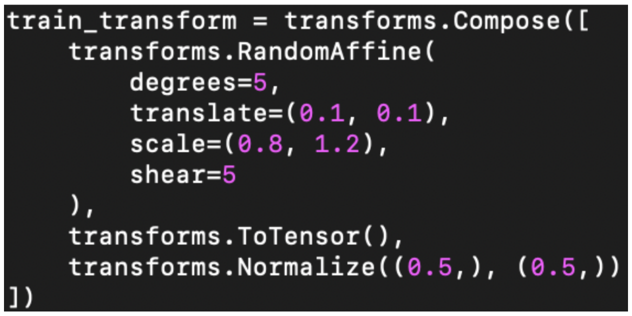
3 Detailed Aspects of the Solution

3.1 Data Augmentation

To help the model generalize beyond the standardized MNIST digits, we applied light geometric data augmentation during training. We did this because the handwritten digits from the class vary in rotation, translation, scale, and stroke shape. MNIST digits are much more uniform and without augmentation, the model tended to overfit MNIST's structure and performed noticeably worse on our class digits. We used PyTorch's `RandomAffine` transform with the following settings:

- Rotation: up to $+/-$ 5 degrees
- Translation: up to 10% in both x and y directions
- Scale variation: between 0.8 and 1.2 zoom
- Shear: up to $+/-$ 5 degrees

After the affine transformation, images were converted to tensors and normalized (Figure 3). This augmentation was intentionally mild since our goal was to not distort the digits too heavily, but to just introduce realistic variation that we would see in actual handwritten digits. This augmentation significantly improved the models performance on the class digits set and reduced overfitting on the MNIST training set.



```
train_transform = transforms.Compose([
    transforms.RandomAffine(
        degrees=5,
        translate=(0.1, 0.1),
        scale=(0.8, 1.2),
        shear=5
    ),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Figure 3: MNIST Training and Validation data was transformed with `RandomAffine`, `ToTensor`, and `Normalize`.

3.2 Train–Validation Split and Early Stopping

We created a $85\% - 15\%$ train-validation split from the MNIST training set to monitor generalization during training. Validation loss was evaluated at the end of each epoch, and we applied early stopping with a patience of 3, restoring the model weights from the epoch with the lowest validation loss. This prevented overfitting on MNIST and ensured that the final model generalized better to our class digits. Early stopping also reduced unnecessary training time and made our results more stable across every run.

3.3 Class Digits Loading

Our class digit dataset consisted of PNG files named using the convention: `digit-group-member.png`. Where the first number is the digit label, the second identifies the group, and the third distinguishes samples within a group. To ensure consistency with MNIST, each image underwent the following processing steps:

- Converted to 28×28 grayscale using `PIL`
- Resized to match MNIST's input size
- Converted to NumPy arrays, then to PyTorch tensors
- Filtered based on quality metrics:

  - Maximum pixel intensity below 70
  - Too few foreground pixels with fewer than 10 pixels above intensity 60
  - Very low contrast with max minus min pixel intensity less than 30

These filters removed images that were either unreadable or completely black (Figure 4). The resulting dataset was wrapped in a custom PyTorch `Dataset` class so that it integrates with the same dataloaders and transforms used for the MNIST evaluation. This ensured that the model was receiving inputs formatted identically to the MNIST images.

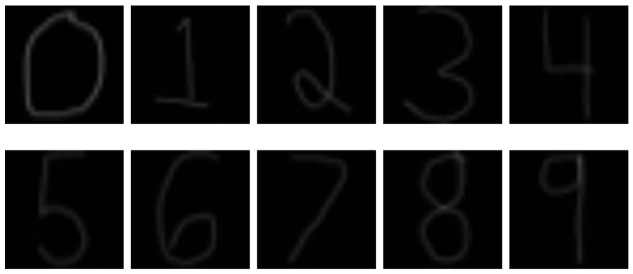


Figure 4: These images were removed during filtering as they make up $\sim 3\%$ of the test data and they are almost completely black.

3.4 Group-Level Evaluation

Using the group ID encoded in each filename, we organized the class digits into separate group datasets. Each group was evaluated individually to compute: average loss, accuracy, and number of samples. This allowed us to compare how writing style differences affected our model's performance. Groups with more MNIST like digits performed better, while groups with unusual or inconsistent handwriting showed higher loss and lower accuracy. While some groups even used MNIST data when they should not have and got near 100% accuracy.

4 Implementation Summary

Our implementation consists of a complete training and evaluation pipeline built in PyTorch and executed inside a Jupyter notebook. The code is structured around several components that handle data loading, pre-processing, model definition, training, and evaluation.

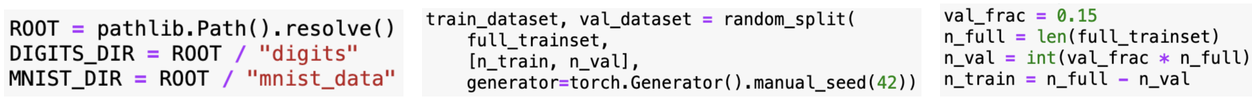
Data Handling: We used PyTorch `Dataset` and `DataLoader` classes to manage both MNIST and the class collected digit dataset. MNIST was downloaded automatically, while the class digits were loaded from PNG files using a custom loader that converts each image to grayscale, resizes it to 28×28, and performs quality filtering (Section 3.3). Group IDs were extracted from filenames to support the per group evaluation utilized at the end of the script.

Model Construction: The neural network was implemented as a simple fully connected MLP with three hidden layers (Section 2.1). The architecture, activation functions, and regularization are encapsulated inside a single model class, making it easy to modify or reuse the model.

Training Pipeline: The training loop performs forward passes, computes losses, performs backpropagation, and updates parameters using the Adam optimizer (Section 2.2). A separate validation loop runs at the end of each epoch, allowing us to use early stopping and track both training and validation loss curves (Section 3.2). Batch size, learning rate, and optimizer settings were configured above the call to the `train_with_early_stopping()` function for easy modification.

Evaluation Tools: To analyze performance, we implemented the helper function `evaluate()` that computes the overall accuracy and loss, confusion matrices, and per class average loss using the final model. The helper function `evaluate_per_group()` uses this same function to extract these same metrics for individual group datasets. The results were visualized directly in the notebook using `matplotlib` and the helper functions: `show_confusion_matrix()`, `show_loss_table()`, and `plot_group_results()`. These functions were used to display confusion matrices, group bar charts, and loss inline rather than saved to files, which keeps all our results contained to the notebook. This was a modification we implemented post presentation after reviewing the project requirements.

Reproducibility and Organization: We used consistent directory paths, random seeds, and deterministic split logic to ensure reproducible results (Figure 5). The notebook workflow is organized into labeled sections (data loading, training, evaluation, visualization), so that it makes it easy for others to follow this entire process.



```
ROOT = pathlib.Path().resolve()
DIGITS_DIR = ROOT / "digits"
MNIST_DIR = ROOT / "mnist_data"
```

```
train_dataset, val_dataset = random_split(
    full_trainset,
    [n_train, n_val],
    generator=torch.Generator().manual_seed(42))
```

```
val_frac = 0.15
n_full = len(full_trainset)
n_val = int(val_frac * n_full)
n_train = n_full - n_val
```

Figure 5: Directory paths ensure consistency across users, random seed=42 and deterministic train/validation split ensures reproducibility.

5 Progression of the Solution

Our solution was developed in several stages, with each step addressing a limitation observed in the previous version.

Stage 1: Baseline MNIST Training
We began with a simple MLP trained only on MNIST without data augmentation or early stopping. Training loss here decreased very quickly, but the model overfit the MNIST and performed poorly on our class digits.

Stage 2: Adding Data Augmentation
We introduced light affine augmentation with small rotations, shifts, shear, and scale changes (Section 3.1). This greatly reduced overfitting and improved accuracy on the class digits because the augmented MNIST images were more similar to the natural variations we could see in our actual handwritten set.

Stage 3: Introducing a Validation Split and Early Stopping
To prevent the model from over training on MNIST, we created a 85/15 train-validation split and used early stopping based on validation loss (Section 3.2). This stabilized training, avoided unnecessary epochs, and consistently produced a better performing model on both MNIST and the class digits by $\sim 1\%$.

Stage 4: Cleaning and Filtering Class Digits
Some class digit images were nearly blank. So filtering these out improved accuracy by $\sim 3\%$.

Stage 5: Per Group Evaluation
We broke the class dataset into groups and evaluated each group separately (Section 3.4). This revealed a clear performance difference between groups due to their handwriting styles. Some groups had digits shaped very similarly to MNIST (if not copied MNIST) and achieved high accuracy, while others used unusual digit forms most likely not written by hand.

6 Test Results and Analysis

After training the final model with augmentation, validation based early stopping, and the selected hyperparameters, we evaluated its performance on three datasets: the MNIST test set, the combined class digits dataset, and each group within the class dataset. These results show us how well the model is able to generalize and the impact writing style variability has on it.

6.1 MNIST Test Set

The MNIST test set is the baseline here since the model was trained on MNIST. At stage 1, it was performing with a test loss of $\sim .07$ and an accuracy of $\sim 97\%$. All the way at stage 5, it was performing with a test loss of $\sim .05$ and an accuracy of $\sim 98\%$. This is not much of an improvement because the model learned the MNIST data style well regardless of data augmentation. The confusion matrix shows these results and how almost all of the digits are classified correctly, with occasional confusion between similar shapes such as 4/9 or 3/5, etc. (Figure 6). The per class loss table is shown showing that the model is very confident with its predictions (Figure 7). Overall, the MNIST results confirm that the model learned the digit structure well and generalizes strongly within the MNIST data.

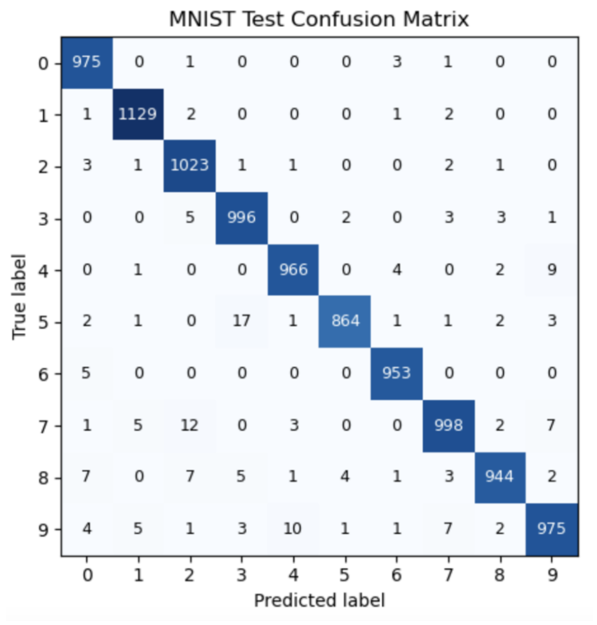
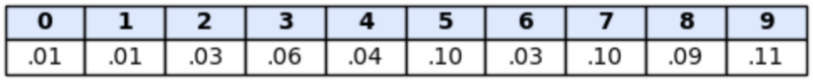


Figure 6: Confusion matrix on the MNIST test set. The model is very accurate in its predictions.

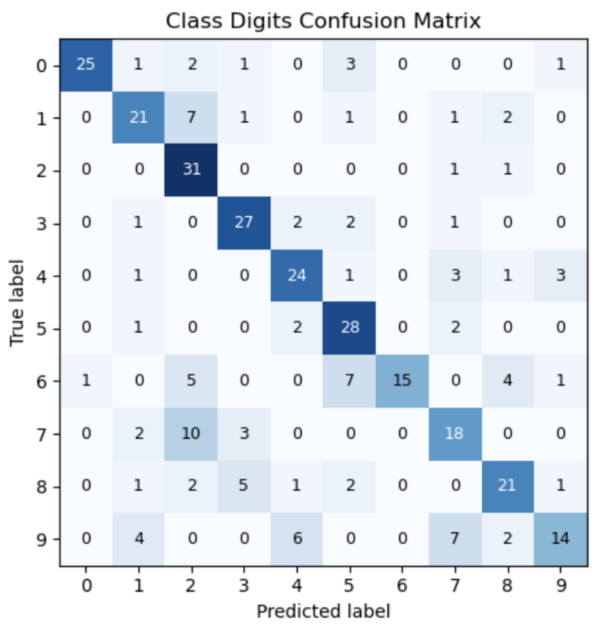


0	1	2	3	4	5	6	7	8	9
.01	.01	.03	.06	.04	.10	.03	.10	.09	.11

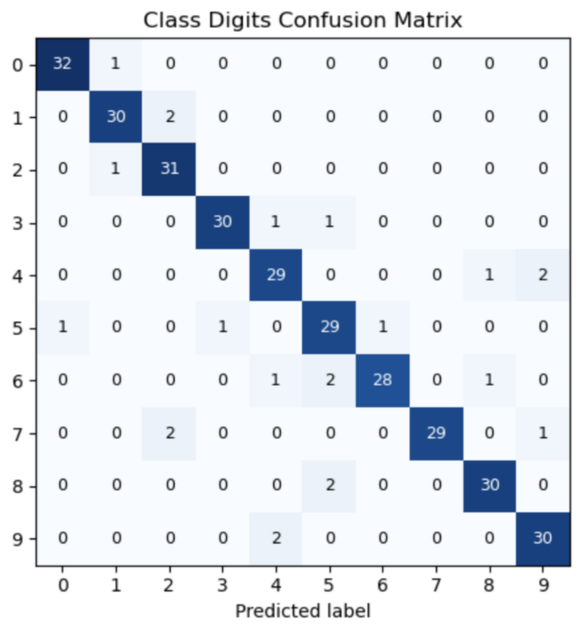
Figure 7: Average per class loss table on the MNIST test set. The model is very confident in its predictions.

6.2 Class Digits Dataset

The next evaluation uses our class collected handwritten digits. This dataset is more challenging due to variation in style, stroke thickness, slant, and spacing. At stage 1, it was performing with a test loss of ~ 1.71 and an accuracy of $\sim 67\%$. All the way at stage 5, it was performing with a test loss of $\sim .23$ and an accuracy of $\sim 93\%$. This is a very large improvement and shows that the model is generalized much better than in stage 1. Although the model is much improved, the accuracy is still lower than that of the MNIST evaluation. This is expected though since the handwriting style differs significantly form the standardized MNIST digits. Some digits resemble MNIST closely and are predicted correctly, while others are often misclassified, but our results are still great. The confusion matrices at stages 1 and 5 show the evolution of these results as well as the average per class loss tables showing a large improvement in confidence for every digit (Figures 8, 9).

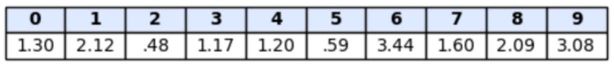


True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	25	1	2	1	0	3	0	0	0	1
1	0	21	7	1	0	1	0	1	2	0
2	0	0	31	0	0	0	0	1	1	0
3	0	1	0	27	2	2	0	1	0	0
4	0	1	0	0	24	1	0	3	1	3
5	0	1	0	0	2	28	0	2	0	0
6	1	0	5	0	0	7	15	0	4	1
7	0	2	10	3	0	0	0	18	0	0
8	0	1	2	5	1	2	0	0	21	1
9	0	4	0	0	6	0	0	7	2	14

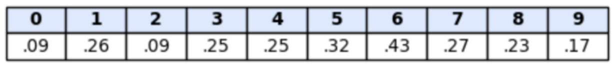


True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	32	1	0	0	0	0	0	0	0	0
1	0	30	2	0	0	0	0	0	0	0
2	0	1	31	0	0	0	0	0	0	0
3	0	0	0	30	1	1	0	0	0	0
4	0	0	0	0	29	0	0	0	1	2
5	1	0	0	1	0	29	1	0	0	0
6	0	0	0	0	1	2	28	0	1	0
7	0	0	2	0	0	0	0	29	0	1
8	0	0	0	0	0	2	0	0	30	0
9	0	0	0	0	2	0	0	0	0	30

Figure 8: Left: Confusion matrix at stage 1 without data augmentation or filtering with a loss of ~ 1.71 and an accuracy of $\sim 67\%$. Right: Confusion matrix at stage 5 with data augmentation and filtering with a loss of $\sim .23$ and an accuracy of $\sim 93\%$. The improvement is noticeably drastic.



0	1	2	3	4	5	6	7	8	9
1.30	2.12	.48	1.17	1.20	.59	3.44	1.60	2.09	3.08



0	1	2	3	4	5	6	7	8	9
.09	.26	.09	.25	.25	.32	.43	.27	.23	.17

Figure 9: Left: per class loss at stage 1 without data augmentation or filtering. Right: per class loss at stage 5 with data augmentation and filtering. The improvements are noticeably drastic with the same digit to digit ratio nearly preserved.

6.3 Per-Group Results

We evaluated each group separately to understand how handwriting style affects performance. For each group, we measured: accuracy, average loss, and number of images (Figures 10, 11). These results are quite varied from group to group, with group 8 only having 3 submissions and group 9 only having 1 usable submission. Groups like 5 and 6, whose handwriting closely resembles MNIST (if not is actually MNIST copies) achieve the highest accuracy and the lowest loss at both stage 1 and stage 5. This leads us to believe they are actually from MNIST which can be confirmed just by looking at some of them. Groups that have unusual patterns, inconsistencies, or faint writing show much higher loss and lower accuracy, but reflect real handwriting well. This variation across the group shows that diversity in handwriting is a major factor in the model's performance.

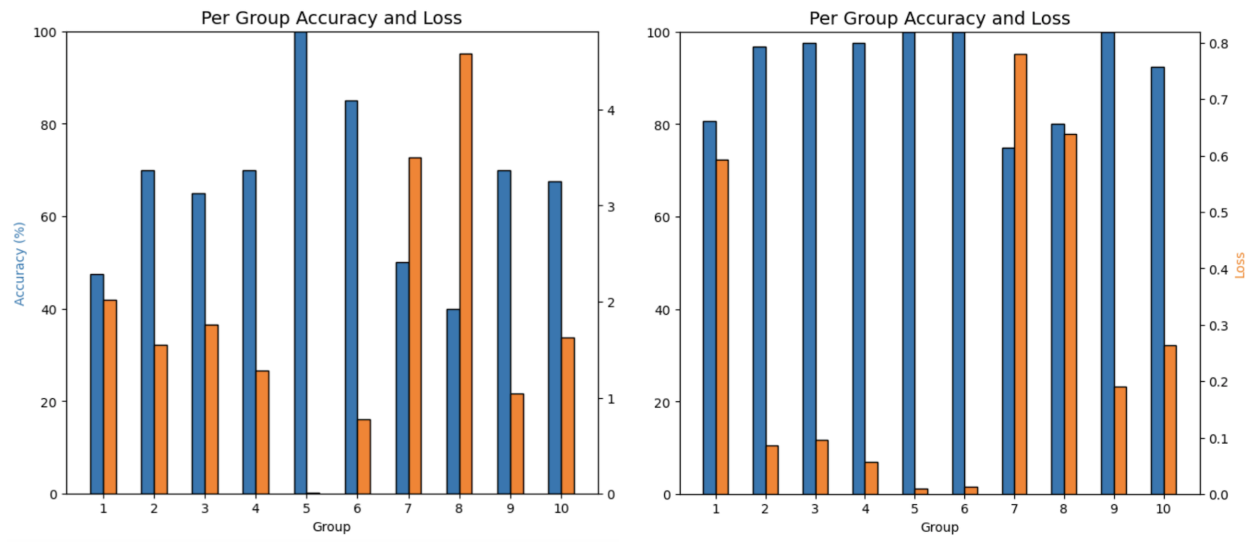
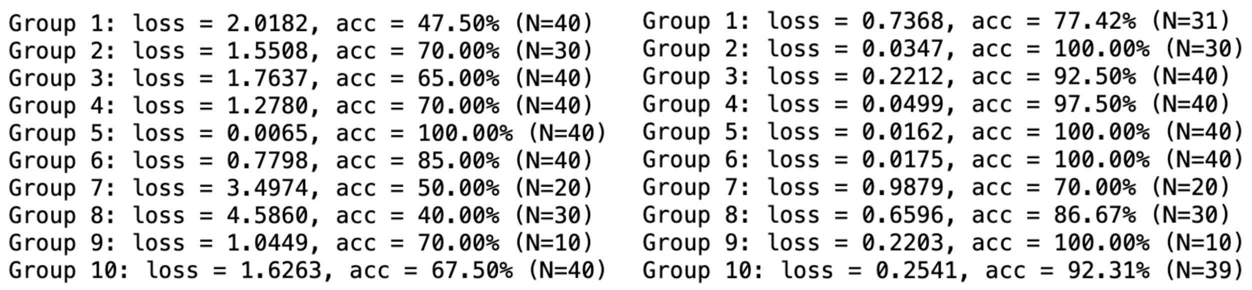


Figure 10: Left: per group results at stage 1 without data augmentation or filtering. Right: per group results at stage 5 with data augmentation and filtering. The improvements are noticeable especially when you see the difference in loss scales, showing just how much more confident the model became. Variations across group results are clear and show how they can significantly affect the models overall performance.



```
Group 1: loss = 2.0182, acc = 47.50% (N=40)     Group 1: loss = 0.7368, acc = 77.42% (N=31)
Group 2: loss = 1.5508, acc = 70.00% (N=30)     Group 2: loss = 0.0347, acc = 100.00% (N=30)
Group 3: loss = 1.7637, acc = 65.00% (N=40)     Group 3: loss = 0.2212, acc = 92.50% (N=40)
Group 4: loss = 1.2780, acc = 70.00% (N=40)     Group 4: loss = 0.0499, acc = 97.50% (N=40)
Group 5: loss = 0.0065, acc = 100.00% (N=40)    Group 5: loss = 0.0162, acc = 100.00% (N=40)
Group 6: loss = 0.7798, acc = 85.00% (N=40)     Group 6: loss = 0.0175, acc = 100.00% (N=40)
Group 7: loss = 3.4974, acc = 50.00% (N=20)     Group 7: loss = 0.9879, acc = 70.00% (N=20)
Group 8: loss = 4.5860, acc = 40.00% (N=30)     Group 8: loss = 0.6596, acc = 86.67% (N=30)
Group 9: loss = 1.0449, acc = 70.00% (N=10)     Group 9: loss = 0.2203, acc = 100.00% (N=10)
Group 10: loss = 1.6263, acc = 67.50% (N=40)    Group 10: loss = 0.2541, acc = 92.31% (N=39)
```

Figure 11: Left: per group results at stage 1 without data augmentation or filtering. Right: per group results at stage 5 with data augmentation and filtering. The accuracy improves for every group if it isn't already 100% and the loss significantly decreases for every group.

7 Detailed Analysis: Why Some Groups Work Better

The differences in performance across groups are mainly due to handwriting style and image quality. Because the model was trained on MNIST, groups whose digits resemble MNIST perform better, while groups with unusual or low quality digits perform worse.

Similarity to MNIST: Groups whose digits look clean, centered, and similar to MNIST achieve higher accuracy. The model generalizes best to digit shapes it has effectively seen before during training.

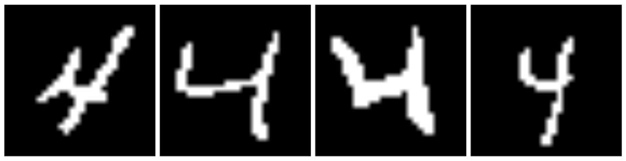


Figure 12: Digit = 4 all from group 5. These are undoubtedly copied MNIST data and is why they achieve 100% accuracy before and after data augmentation and have a lower loss before data augmentation of 0.006 compared to .02 after.

Stroke Thickness and Contrast: Faint or low contrast digits are harder for the model to interpret, especially after resizing. These samples lost important edge information, leading to more misclassifications.

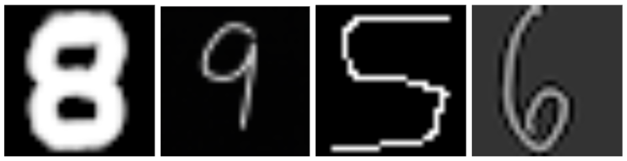


Figure 13: The images of 8 and 9 on the left show contrasting stroke thicknesses. The 5 shows how close the digit details are to the edge of the image, potentially resulting in some loss of information after resizing since some images weren't correctly 28×28. The 6 shows how some images are low contrast, but did not get filtered out and still potentially skewed the results.

Stylized or Unconventional Digit Shapes: Digits written in unusual forms such as alternative versions of 1, 2, 4, 7, or 9 fall outside the models training distribution. Even small stylistic changes seem to significantly impact it even though its robust hyperparameters should be able to pick up on them, it seems to be sensitive.

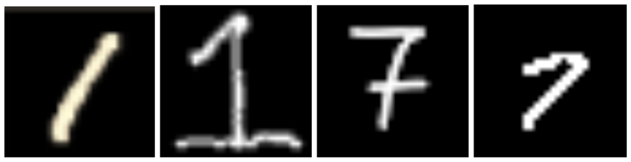


Figure 14: These images show how different some numbers can be written. The two images on the left are two very different versions of the number 1 and the two images on the right are two very different versions of the number 7.

Intra-Group Variation: Groups with consistent handwriting styles seem to perform well while groups having very different writing habits between members show larger drops in accuracy. This is mainly due to unnaturally written digits written on a pre-pixelated screen with their mouse, not actually post-processed handwritten digits.

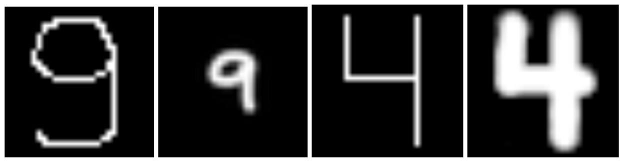


Figure 15: These are digits all from group 10, a couple clearly drawn with a computer mouse on a pre-pixelated 28×28 template, and the other normal, but shown to demonstrate contrast within a group.

Sample Count: Smaller groups are more affected by individual misclassifications, while larger groups produce more stable and representative accuracy measurements. For example, as stated previously, group 9 only has 10 usable digits, while almost every other group has 30-40.

8 Conclusion and Future Work

In this project we trained a fully connected neural network on MNIST and evaluated how well it generalizes to handwritten digits collected from our class. This model performed very well on MNIST with a loss of $\sim .05$ and an accuracy of $\sim 98\%$, confirming that the architecture and training procedure were effective. Performance dropped slightly on the class digits with a loss of $\sim .23$ and an accuracy of $\sim 93\%$. This drop shows us how much our class dataset varies in real handwriting compared to the MNIST dataset. Data augmentation, validation based early stopping, and filtering low quality samples all helped improve generalization. The per group analysis showed that handwriting style plays a major role in model accuracy. Groups whose digits resemble MNIST were classified correctly most of the time, while groups with faint or inconsistent digits performed worse. This reinforces the idea that models trained only on MNIST do not fully capture the diversity of our handwriting.

Future Work: We could use a convolutional neural network (CNN) even though that's what we were explicitly not doing in this project. CNNs can handle spatial patterns better than our simple MLP and would likely be able to generalize more effectively for our set of varied handwriting. We could augment more aggressively by adding different distortions, noise, thickness changes, or even something like random erasing some pixels could help the model learn an even wider variety of digit styles. We chose not to do this in this project just to keep it simple and see the true lesson there is to learn here which is noticing the effect variability has on the model. We could also collect more samples and have a larger and more diverse handwritten dataset which could reduce sensitivity to individual writing habits. But overall, this project is able to demonstrate the strength and limitations of a MNIST trained MLP and shows us clearly why testing hyperparameters and generalizing the model is important for real world data evaluation.

9 References

[1] MNIST Dataset: LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324. https://doi.org/10.1109/5.726791

[2] Data Augmentation (Affine): Simard, P. Y., Steinkraus, D., & Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR). https://doi.org/10.1109/ICDAR.2003.1227801

[3] PyTorch Documentation: Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). PyTorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems, 32. Retrieved from https://pytorch.org/docs/stable/

[4] Machine Learning Fundamentals (used in CSCI581): Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow (2nd ed.). O'Reilly Media.