CSCI 580 Artifical Intelligence: Final Report

This report documents our model for handwritten digit recognition. Specifically, it will cover various attributes of our model and its overall performance. The people in our team who worked on the project are **Harsh Sharma, Marco U Calderon, Luis Manuel Melgoza, and Kamaldeep Singh**

Here is a link to a GitHub repository including all relevant materials: (https://github.com/HarshTheSharma/Final580). Also make sure to import these modules:

In [4]:
```
import os
import glob
from io import BytesIO

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image, ImageFilter, ImageEnhance

import torch
from torch import nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import Dataset
from sklearn.metrics import classification_report, confusion_matrix
```

Our Model's Framework

A high-level framework for our solution involves using a Multilayer Perceptron (MLP), which is a fully connected feedforward neural network. This framework is implemented using PyTorch and is designed to classify 28 x 28 grayscale handwritten digit images from the MNIST and team-collected datasets.

Regarding details about our Neural Net architecture, we have exactly four linear layers: three hidden layers and one output layer. Our model starts with hundreds of neurons that gradually decrease in number as we progress through each layer. We end up with ten neurons in our output layer, corresponding to the 10 digit classes. The total number of weights excluding Batch Norm parameters is around 566,528. Furthermore, our model also uses the Rectified Linear Unit (ReLU) activation function after each hidden layer's batch normalization, which stabilizes training alongside dropout in the first two hidden layers to avoid overfitting.

In addition to our Neural Net architecture, our model is driven by carefully tuned hyperparameters. The model uses the Adam optimizer to update weights with the standard cross-entropy loss function for multi-class classification. Its initial learning rate is set to 0.0005, with L2 regularization (weight decay) of 0.0001 to reduce overfitting. Additionally, a Reduce Learning Rate on Plateau is included, which halves the learning rate if the validation loss plateaus for consecutive epochs. Speaking of epochs, we are doing 100 epochs on the MNIST dataset, and handwritten digits made by 10 groups with 4 members each.

Detailed Aspects of Our Solution

Part 1: Data & Augmentation Setup

The first major part of our implementation concerns data and augmentation setup. To maximize our model's performance, we apply various transformations to the images, so our model can pick up on varying image types. Here is all of our transformations:

In [5]:
```
class RaiseDarkPoint(object):
    def __init__(self, grayRange=(10, 60)):
        self.grayRange = grayRange

    def __call__(self, img):
        arr = np.array(img).astype(np.float32)
        grayVal = np.random.uniform(*self.grayRange)
        mask = arr < 30
        arr[mask] = arr[mask] + grayVal
        arr = np.clip(arr, 0, 255)
        return Image.fromarray(arr.astype(np.uint8), mode="L")


class LowerWhitePoint(object):
    def __init__(self, factorRange=(0.7, 0.95)):
        self.factorRange = factorRange

    def __call__(self, img):
        arr = np.array(img, dtype=np.float32)
        factor = float(np.random.uniform(self.factorRange[0], self.factorRar
        arr = arr * factor
        arr = np.clip(arr, 0, 255)
        return Image.fromarray(arr.astype(np.uint8), mode="L")


class AddNoise(object):
    def __init__(self, noiseStd=0.05):
        self.noiseStd = noiseStd

    def __call__(self, img):
        arr = np.array(img).astype(np.float32)
```

```
        noise = np.random.normal(0, self.noiseStd * 255, arr.shape)
        arr = arr + noise
        arr = np.clip(arr, 0, 255)
        return Image.fromarray(arr.astype(np.uint8), mode="L")


class BubblyDigits(object):
    def __init__(self, blurRange=(0.4, 1.0), contrastRange=(1.1, 1.6)):
        self.blurRange = blurRange
        self.contrastRange = contrastRange

    def __call__(self, img):
        sigma = float(np.random.uniform(self.blurRange[0], self.blurRange[1]
        img = img.filter(ImageFilter.GaussianBlur(radius=sigma))
        c = float(np.random.uniform(self.contrastRange[0], self.contrastRang
        img = ImageEnhance.Contrast(img).enhance(c)
        return img


class JPEGCompression(object):
    def __init__(self, qualityRange=(40, 80)):
        self.qualityRange = qualityRange

    def __call__(self, img):
        q = int(np.random.randint(self.qualityRange[0], self.qualityRange[1]
        buf = BytesIO()
        img.save(buf, format="JPEG", quality=q)
        buf.seek(0)
        return Image.open(buf).convert("L")
```

And here is how we are applying the transformations to our images:

In [6]:
```
trainTransform = transforms.Compose(
    [
        transforms.RandomApply(
            [
                transforms.Pad(4, fill=0),
                transforms.RandomCrop(28),
            ],
            p=0.5,
        ),
        transforms.RandomApply(
            [
                transforms.RandomAffine(
                    degrees=(-5, 20),
                    translate=(0.25, 0.25),
                    scale=(0.6, 1.1),
                    fill=0,
                )
            ],
            p=0.5,
        ),
        transforms.RandomApply([AddNoise()], p=0.1),
        transforms.RandomApply([BubblyDigits()], p=0.5),
```

```
        transforms.RandomApply([LowerWhitePoint()], p=0.4),
        transforms.RandomApply([RaiseDarkPoint()], p=0.4),
        transforms.RandomApply(
            [transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 1.5))], p=1
        ),
        transforms.RandomApply([JPEGCompression()], p=0.25),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,)),
    ]
)

transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)
```

Why are We Modifying Data?

The primary reason for these applied transformations is that the class data and the MNIST data were significantly different, presenting an issue that needed to be addressed before defining/training a model.

In addition to the immense differences, another issue was the significant variation in class data. This massive variance made it difficult to find a process that could improve all digits and make them similar to the MNIST dataset without applying transformations targeted to files. Essentially, a solution that could work across all semi-properly formatted images, even ones outside the class data, was needed

Following the image transformations, we define our training and test datasets, which will be incredibly useful to train and evaluate our model:

In [7]:
```
trainSet = datasets.MNIST(
    "~/.pytorch/MNIST_data/", download=True, train=True, transform=trainTrar
)

trainLoader = torch.utils.data.DataLoader(trainSet, batch_size=64, shuffle=T

testSet = datasets.MNIST(
    "~/.pytorch/MNIST_data/", download=True, train=False, transform=transfor
)

testLoader = torch.utils.data.DataLoader(testSet, batch_size=64, shuffle=Fal
```

Specific Attributes

To help make it clear the exact details about our model, specifically its framework and hyperparameters, a list of bullet points will be presented:

Model Framework

- 4 Layers: 3 hidden & 1 output
- Weight Decay: .0001
- Number of Weights: 566,528
- Number of Neurons: 784 → 512 → 256 → 128 → 10
- ReLU Activation Function
- Batch Normalization for each hidden layer (size = 64)
- Adam Optimizer Standard Cross-Entropy Loss
- Validation Loss to adjust LR

Hyperparameters

- Initial learning rate of 0.0005
- Reduce Learning Rate on Plateau
- Fifteen epochs
- Patience Value of 5

Part 2: Training Our Model

The next major part of our implementation concerns the main loop for training our model. We go through each image and its label, provide the data to our model, and then record the results. First, we have to define our model:

In [8]:
```
# ------------------------------------------------------------
# Device
# ------------------------------------------------------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

class MNISTMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)
        self.bn3 = nn.BatchNorm1d(128)
        self.fc4 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = F.relu(self.bn3(self.fc3(x)))
        x = self.fc4(x)
        return x
```

```
model = MNISTMLP().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=5e-4, weight_decay=1e-4)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    factor=0.5,
    patience=5,
)

print(model)
```

```
Using device: cpu
MNISTMLP(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (bn2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (bn3): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
  (fc4): Linear(in_features=128, out_features=10, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

And now we can run it for 10 epochs/iterations (should be higher but using a moderate amount to demonstrate results) and record the results:

In [9]:
```
numEpochs = 10
allLosses = []

for epoch in range(numEpochs):
    model.train()
    runningLoss = 0.0
    batchLosses = []

    for images, labels in trainLoader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        val = loss.item()
        runningLoss += val
        batchLosses.append(val)

    epochLoss = runningLoss / len(trainLoader)
```

```
    allLosses.append(batchLosses)
    scheduler.step(epochLoss)

    print(f"Epoch {epoch+1}/{numEpochs}, Loss: {epochLoss:.4f}")

print("Training complete.")

plt.figure(figsize=(10, 4))
plt.plot(allLosses[0], label="Epoch 1")
plt.xlabel("Batch index")
plt.ylabel("Loss")
plt.title("Training Loss per Batch")
plt.legend()
plt.show()
```

```
Epoch 1/10, Loss: 1.2615
Epoch 2/10, Loss: 0.8615
Epoch 3/10, Loss: 0.7376
Epoch 4/10, Loss: 0.6716
Epoch 5/10, Loss: 0.6328
Epoch 6/10, Loss: 0.5969
Epoch 7/10, Loss: 0.5680
Epoch 8/10, Loss: 0.5462
Epoch 9/10, Loss: 0.5321
Epoch 10/10, Loss: 0.5117
Training complete.
```

This is the essential framework behind the model/approach utilized. A detailed breakdown of the results using a more finetuned model will be shown

Test Results & Analysis

Part 1: Result Analysis (Model 1)

For some background, there were two models used with significant differences. These differences include an increase in epochs, using augmented data instead of plain data, and using a patience value of five instead of two.

These modifications were implemented to improve the model's overall accuracy and precision when identifying the digits. Now, we will document the results of our first model, which had the same aforementioned framework.

Model 1 Results

We ran our first model with fifteen epochs, which achieved these results:

MNIST

- Accuracy: 98.49%
- Error Rate: 1.21%

Handwritten Digits

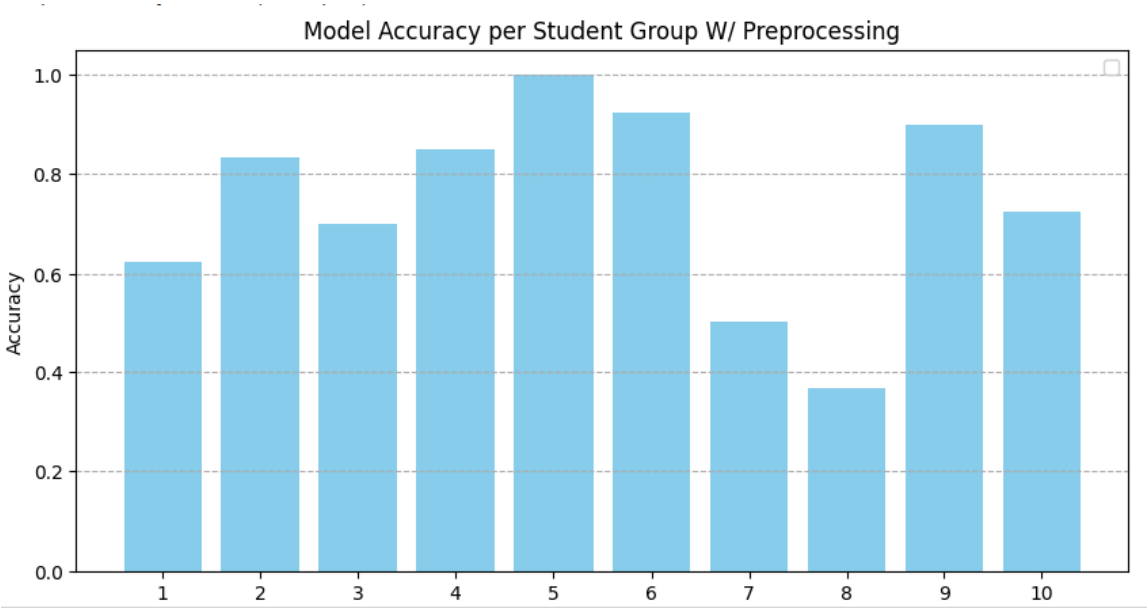
- Accuracy: 75.15%
- Error Rate: 24.85%

We also created many plots to help visualize certain parts of our model results, such as model accuracy per group

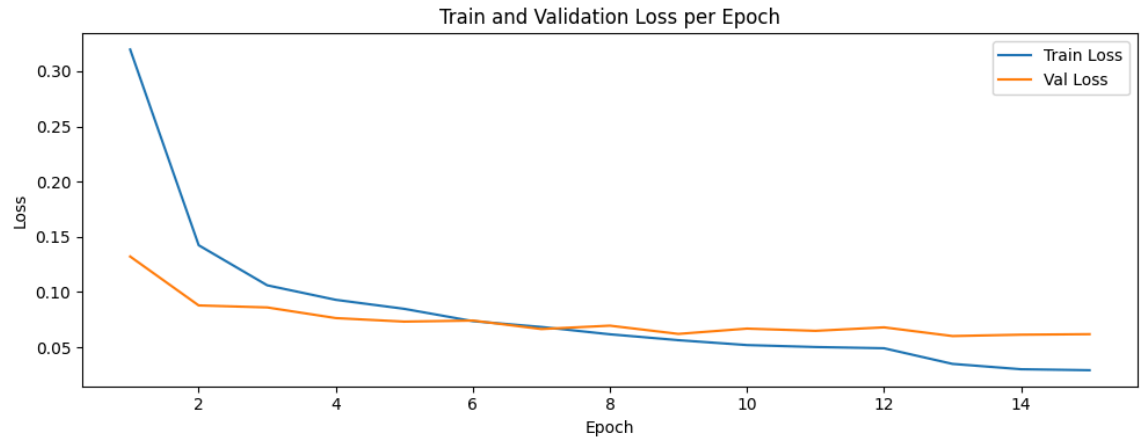
Here are the exact accuracies per group:

- Group 1 Accuracy: 62.50%
- Group 2 Accuracy: 83.33%
- Group 3 Accuracy: 70.00%
- Group 4 Accuracy: 85.00%
- Group 5 Accuracy: 100%
- Group 6 Accuracy: 92.50%
- Group 7 Accuracy: 50.00%
- Group 8 Accuracy: 36.67%
- Group 9 Accuracy: 90%
- Group 10 Accuracy: 72.50%

And here it is visualized as a bar chart graph:



In addition to a per-group accuracy graph, we have plotted the training and validation loss per epoch



And finally, we have various output metrics for our model, such as F1 score, precision, recall, etc.

```
==== MNIST Test Set RESULTS ====
Accuracy: 0.9849
Error Rate: 0.0151

Classification Report:
              precision    recall  f1-score   support

           0     0.9858    0.9918    0.9888       980
           1     0.9938    0.9930    0.9934      1135
           2     0.9855    0.9855    0.9855      1032
           3     0.9880    0.9822    0.9851      1010
           4     0.9837    0.9817    0.9827       982
           5     0.9843    0.9843    0.9843       892
           6     0.9874    0.9833    0.9854       958
           7     0.9824    0.9796    0.9810      1028
           8     0.9835    0.9815    0.9825       974
           9     0.9736    0.9851    0.9793      1009

    accuracy                         0.9849     10000
   macro avg     0.9848    0.9848    0.9848     10000
weighted avg     0.9849    0.9849    0.9849     10000
```

```
==== Handwritten Digits RESULTS ====
Accuracy: 0.7515
Error Rate: 0.2485

Classification Report:
              precision    recall  f1-score   support

           0     0.9600    0.7273    0.8276        33
           1     0.9474    0.5455    0.6923        33
           2     0.6667    0.9091    0.7692        33
           3     0.6522    0.9091    0.7595        33
           4     0.8929    0.7576    0.8197        33
           5     0.8108    0.9091    0.8571        33
           6     0.7647    0.7879    0.7761        33
           7     0.5854    0.7273    0.6486        33
           8     0.8148    0.6667    0.7333        33
           9     0.6786    0.5758    0.6230        33

    accuracy                         0.7515       330
   macro avg     0.7773    0.7515    0.7506       330
weighted avg     0.7773    0.7515    0.7506       330
```

Overall, our first model, which used only plain data with no augmentations, performed extremely well on the MNIST dataset, as evidenced by the output metrics like the high F1 scores. However, when it comes to the handwritten digits provided by the class, the model's accuracy and performance heavily decrease, especially with some digits usually hard to perceive.

The drop in performance most likely comes from the huge variance described earlier in the report. Hopefully, the transformations/preprocessing we applied to the images will help. We can now move on to describing our second model's results

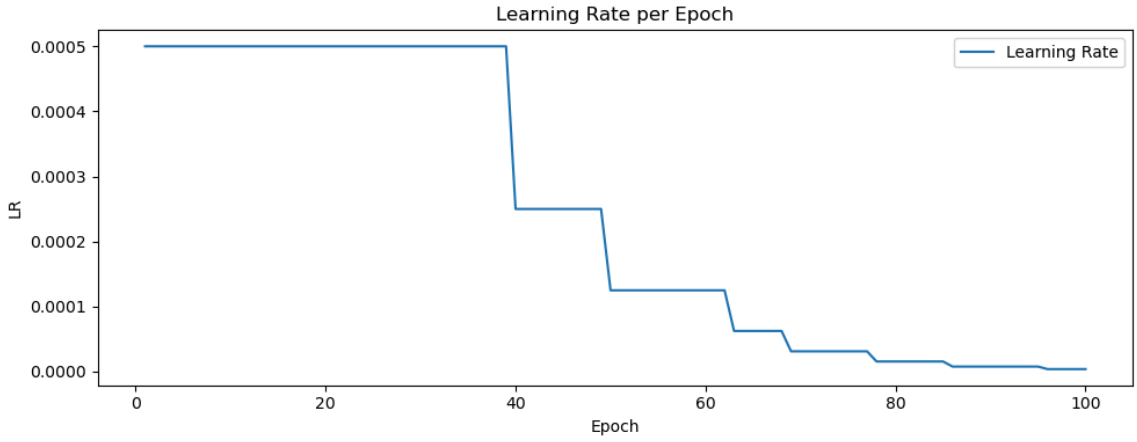
Part 2: Result Analysis (Model 2)

Model 2 Results

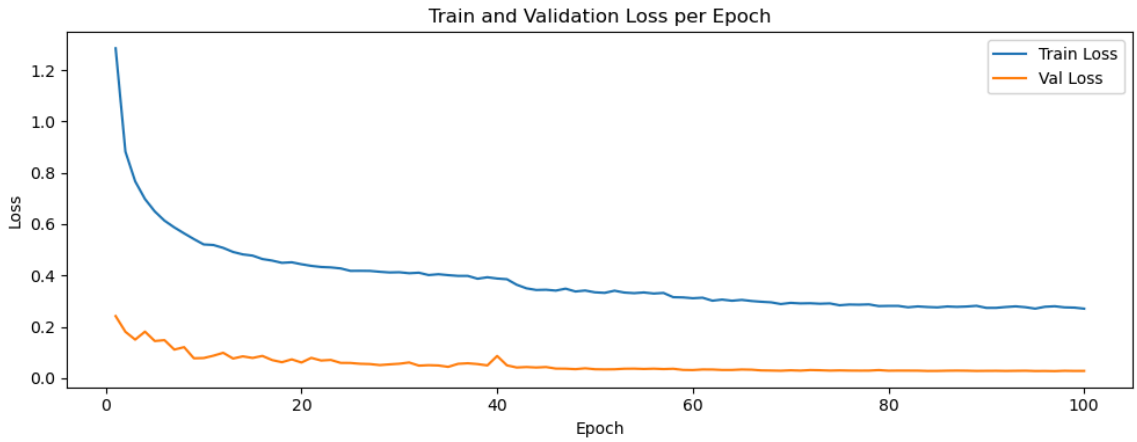
After seeing our first model results, we knew we had to modify certain attributes to attempt to improve the model. Here is a list of the main changes we implemented:

- Epochs: 15 → 100 – With these augmentations, training for 15 epochs was not enough. With 100 epochs, the model got more practice and could better learn from the chaotic data we feed it.
- Scheduler patience: 2 → 5 – Because our data is more chaotic, we need to give our model more time to learn. With a patience of 5, we wait longer before changing LR, so the model can keep learning with bigger steps until we need fine-tuning steps.
- Plain data → Strong augmentation – Before, the model only saw clean digits and overfitted to “nice” images. With heavy augmentation (noise, blur, warps, brightness), it learns to handle messy, real-world digits and becomes more robust, finding large trends instead of small characteristics.

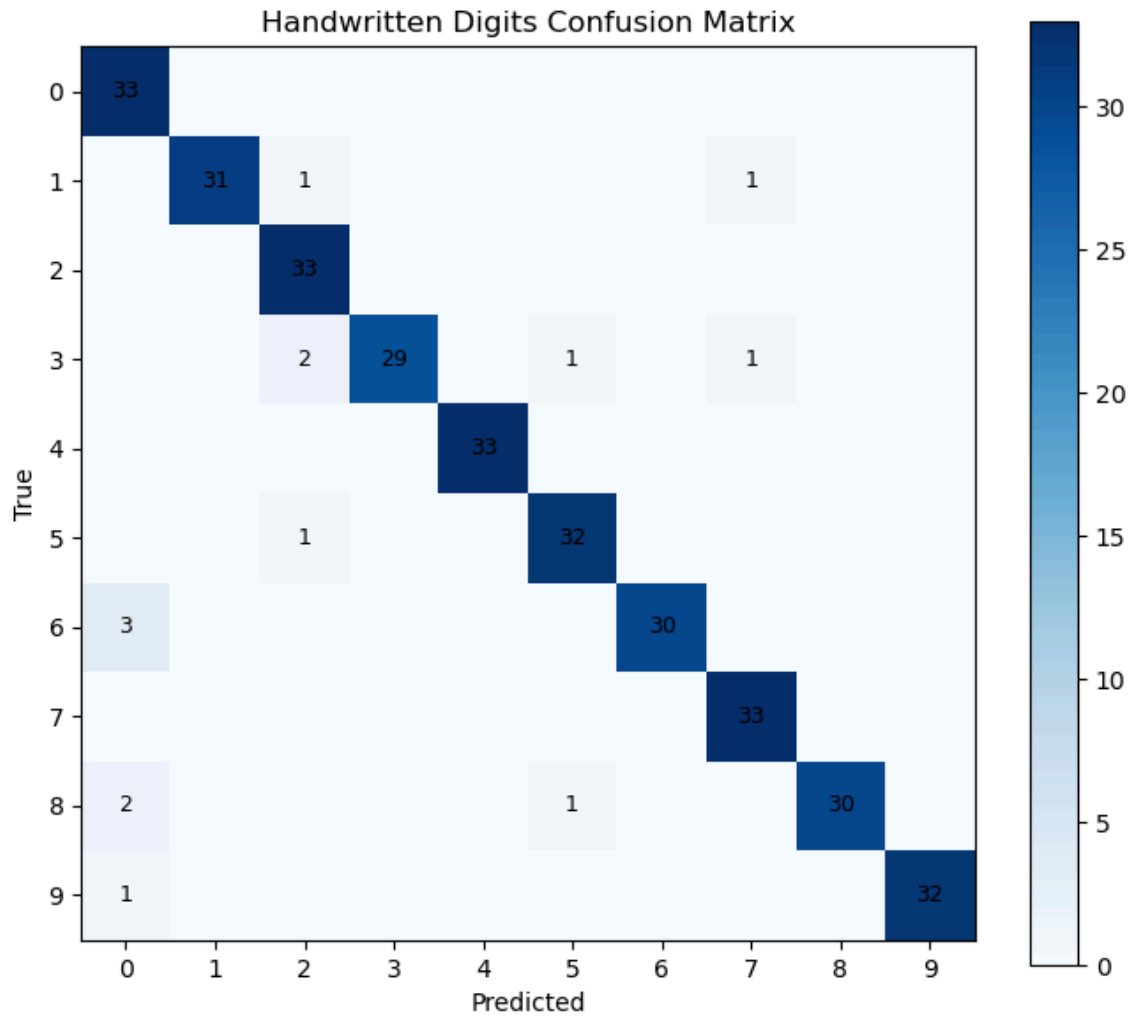
Due to our new modifications, particularly adjusting the scheduler patience value, our model's learning rate changes less frequently, as seen here:



Here is also our training and validation loss per epoch plot:



Moving on, here are some graphs illustrating the results of our second model after running it for 100 epochs. First off is the confusion matrix:



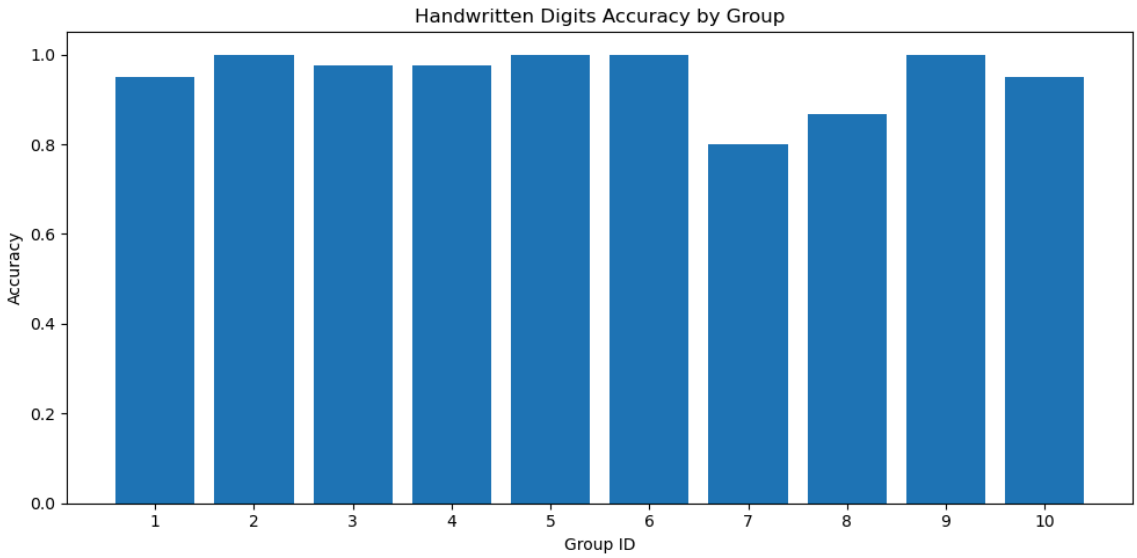
In the confusion matrix, the high concentration of values on the diagonal indicates that the model is accurately predicting the great majority of samples for each class. The low off-diagonal values indicate minimal misclassification. Basically, the model clearly distinguishes between different digits without swapping them or losing track of which is

which. Now, here is the output detailing the results of running our model against the MNIST dataset and the class' handwritten dataset:

```
==== MNIST Test Set RESULTS ====
Accuracy: 0.9915
Error Rate: 0.0085

Classification Report:
              precision    recall  f1-score   support

           0     0.9929    0.9990    0.9959       980
           1     0.9930    0.9974    0.9952      1135
           2     0.9894    0.9932    0.9913      1032
           3     0.9970    0.9881    0.9925      1010
           4     0.9878    0.9919    0.9898       982
           5     0.9911    0.9955    0.9933       892
           6     0.9958    0.9896    0.9927       958
           7     0.9826    0.9912    0.9869      1028
           8     0.9969    0.9887    0.9928       974
           9     0.9890    0.9802    0.9846      1009
```

```
==== Handwritten Digits RESULTS ====
Accuracy: 0.9576
Error Rate: 0.0424

Classification Report:
              precision    recall  f1-score   support

           0     0.8462    1.0000    0.9167        33
           1     1.0000    0.9394    0.9688        33
           2     0.8919    1.0000    0.9429        33
           3     1.0000    0.8788    0.9355        33
           4     1.0000    1.0000    1.0000        33
           5     0.9412    0.9697    0.9552        33
           6     1.0000    0.9091    0.9524        33
           7     0.9429    1.0000    0.9706        33
           8     1.0000    0.9091    0.9524        33
           9     1.0000    0.9697    0.9846        33
```

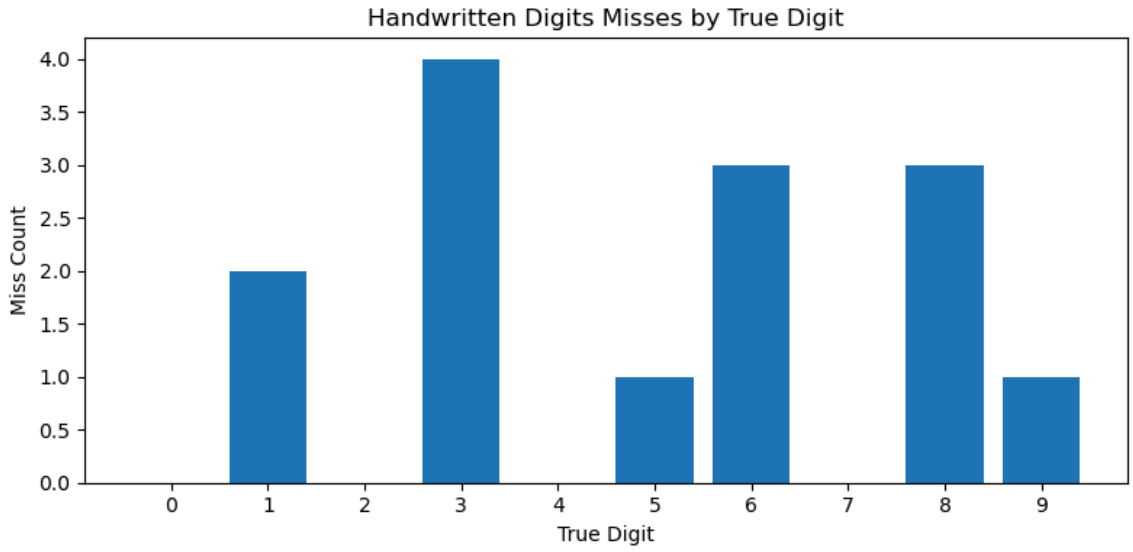
Finally, here is a graph showing our second model's per-group accuracy:



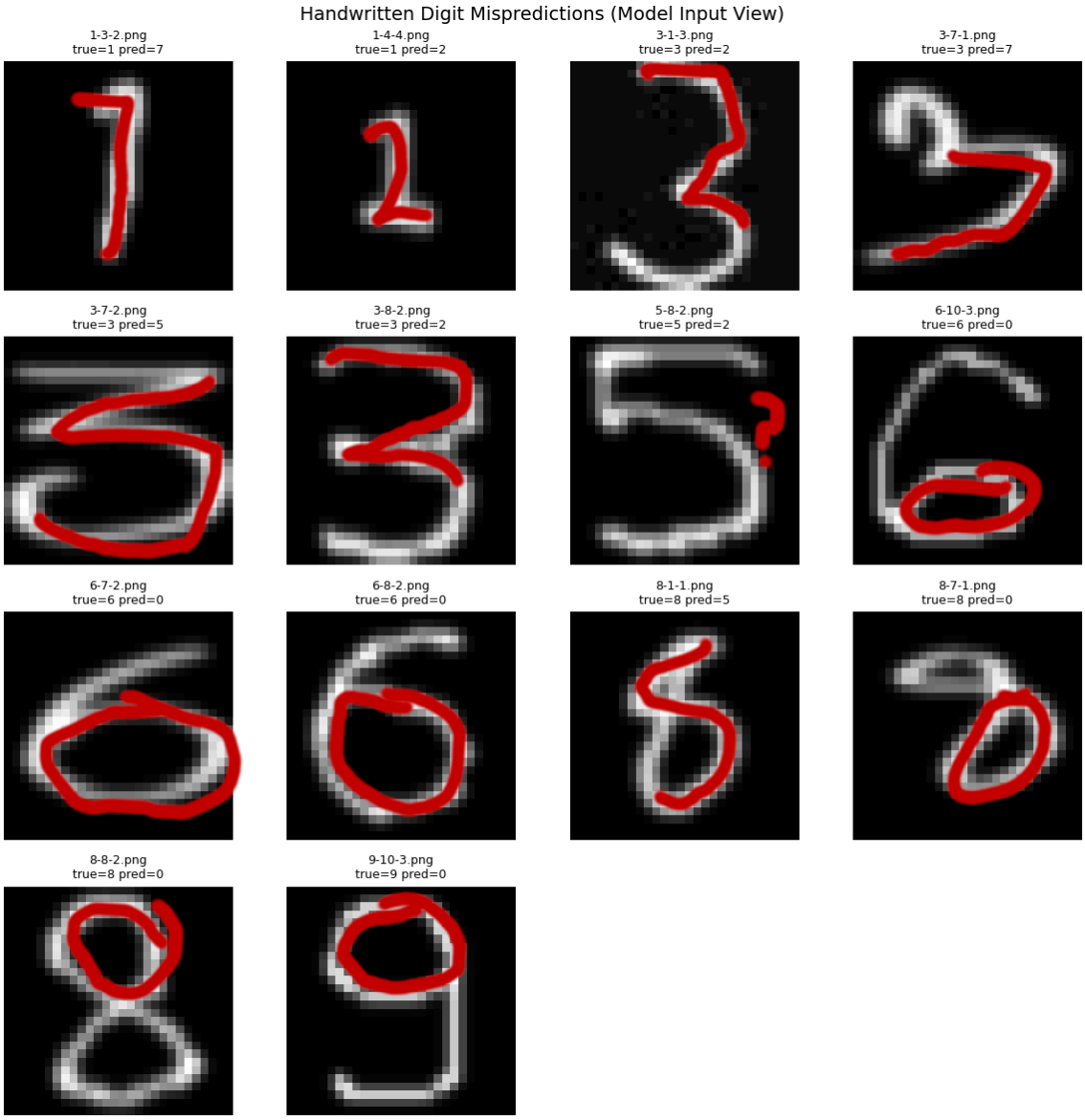
In summary, the second model performed exceedingly well for both the MNIST dataset and class dataset, unlike the first model. The modifications implemented, especially the data augmentations, enhanced the model's ability to predict the handwritten digit. This improvement can be clearly seen when examining the difference between each model's per-group accuracies.

Part 4: Conclusion & Future Work

All in all, our model performed exceptionally well with the attributes we chose to implement. It was able to correctly identify a high majority of the MNIST dataset and class dataset, but still failed on some particular digits. To be specific, our model failed on some 1's, 3's, 5's, 6's, 8's, and 9's



Examining the inaccuracies revealed a possible cause for why our model failed on certain digits. In particular, most of the images/handwritten digits our model incorrectly predicted were "frame filling digits" and faint:



The red-colored areas in the image above represent our group's estimate of what the model might have predicted based on the image. For many malformities, our model could accurately predict the digit; however, it had trouble with images where the digit filled the entire frame. There is a clear chance to improve the model's accuracy by concentrating on these edge cases. Ultimately, even though the model is already very successful, future development should focus on these full-frame malformities to guarantee the required consistency for even the most unusual data inputs.