Artificial Intelligence Final Report

Team Members:

Aaron Partridge

Vlad Avdeev

Lindsey Lydon

Sean Angrisani

High-Level Framework of the Solution

Our objective in this project was to develop a robust neural network that was capable of highly accurate classification of handwritten digits. We used the MNIST dataset as our benchmark. We trained our model using PyTorch. In order to improve the performance and accuracy, we explored a few strategies such as tuning the hyperparameters, applying normalization, and using data augmentation techniques. We then compared these results over a few configurations in order to see which approach would generalize the best, especially when used to evaluate a custom data set, i.e., the handwritten digits written by our classmates.

<u>Final Neural Network Architecture:</u>

- Input: 28x28 grayscale image
- Flatten()
- Linear(784 → 512) + ReLU
- Linear(512 → 256) + ReLU
- Linear(256 → 64) + ReLU
- Linear(64 → 10) (output classes)

<u>Hyperparameters:</u>

- Optimizer: Adam
- Learning Rate: 0.001
- Loss Function: CrossEntropyLoss
- Epochs: 10

- Batch Size: 64

Code Snippet:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)

# Loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Track metrics
train_loss_history = []
val_acc_history = []

epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0

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

        optimizer.zero_grad()
        outputs = model(images)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    avg_loss = running_loss / len(trainloader)
    train_loss_history.append(avg_loss)
```

```
# --- Validation on handwritten test set ---
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for images, labels in testloader_custom:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

val_acc = 100 * correct / total
val_acc_history.append(val_acc)
```

Implementation Strategy:

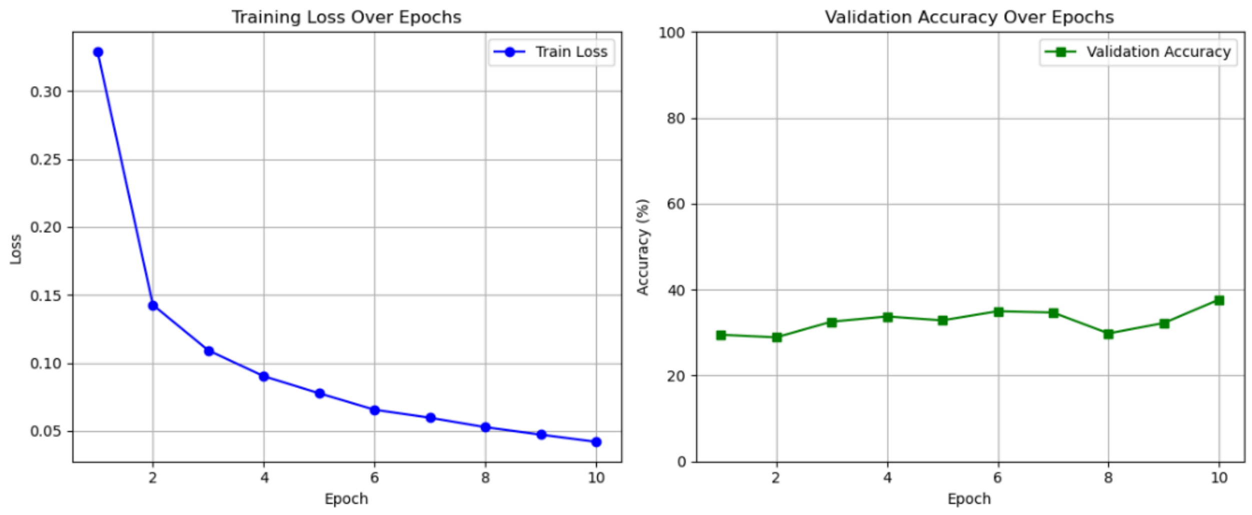
Our strategy is to apply different functions and apply different values to certain values, such as how many cycles the model will need to train. Apply normalization or transformation, seeing if that can improve accuracy.
For us to see and validate the accuracy of methods, we will generate graphs that will help us visualize the loss over epochs. Graphs show us validation data, as well as helping us understand if our methods are accurate and efficient.

Our strategy involved experimentation with a few different functions and tuning certain key parameters, such as the number of training cycles, in order to optimize our model's performance. We also applied normalization as well as data transformations to help evaluate how they impact the accuracy of our model. In order to validate our approach, we will generate graphs that visualize the training and validation loss across our epochs. Our visual tools are helping us track progress and also provide valuable insight into the effectiveness of our methods.

Normalization:

- **Implementation:** In this run, the model was trained solely on a handpicked subset of our group's handwritten digit dataset. The goal was to avoid poorly written or noisy samples, hoping that using only clean examples would improve generalization. Both training and testing data were normalized, but no data augmentation was applied.

- **Accuracy** Training on the handpicked handwritten dataset and testing on the MNIST dataset resulted in a validation accuracy of 37.69%.
- **Graph Evaluation:** Accuracy remained relatively low and inconsistent throughout training, starting at ~29%, fluctuating slightly across epochs, and peaking at ~37% by epoch 10. These fluctuations suggest that even the cleaner samples in our group's data were not diverse enough for the model to generalize to the MNIST test set.
  While the training loss consistently decreased to ~0.04, the validation accuracy remained low, indicating the model was overfitting to the small handpicked training set. Despite better-quality input data, the **limited volume and lack of variation** ultimately restricted performance. This highlights the importance of both data quality and quantity for building generalizable models.

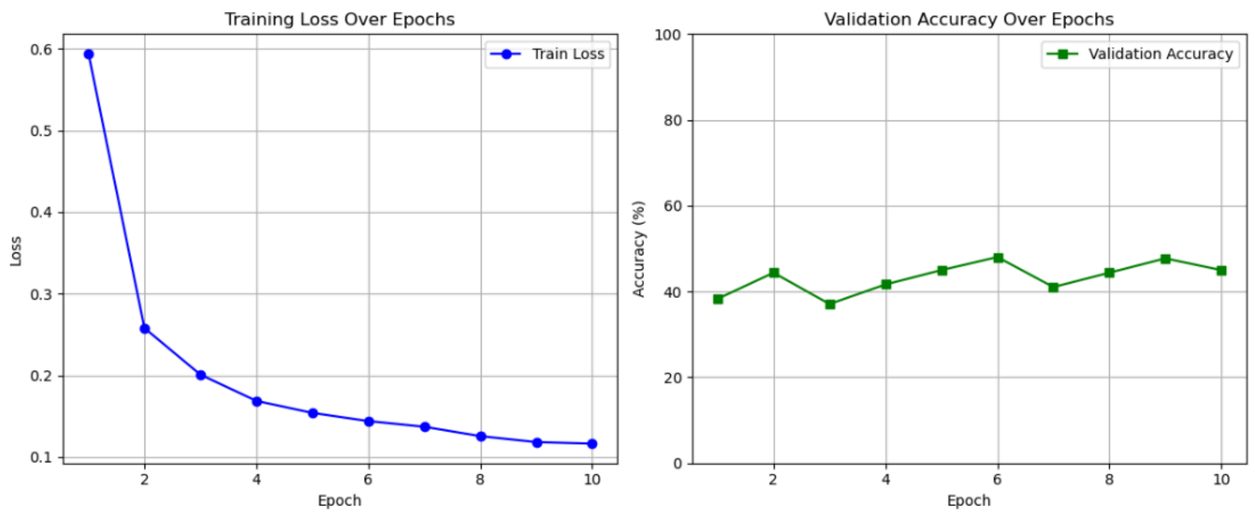


- **Code Snippet**

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

Normalization + Transformation:

- **Implementation:** In this experiment, we trained the model using a handpicked subset of our group's handwritten digit dataset, but this time we applied data augmentation transformations (RandomRotation, RandomAffine) along with normalization. The model was evaluated against the MNIST test dataset to assess generalization beyond our group's samples.
- **Accuracy:** Training with transformations on the handpicked handwritten data resulted in a validation accuracy of 60.00%, showing a significant improvement over the non-augmented version of the same dataset (which only reached ~37.69%).
- **Validation accuracy:** Accuracy began at ~49% and steadily climbed to 60% by epoch 10. While there were some fluctuations across epochs, the overall trend was upward, demonstrating that data augmentation helped the model generalize better to the MNIST test set by exposing it to more variation during training.
  This run clearly shows that data augmentation improved generalization, even with a smaller, handpicked dataset. The model still doesn't reach MNIST-level performance, but it performs substantially better than when trained on raw group data alone. This confirms that augmentation compensates for dataset size and diversity limitations, making it a valuable technique when working with limited data.



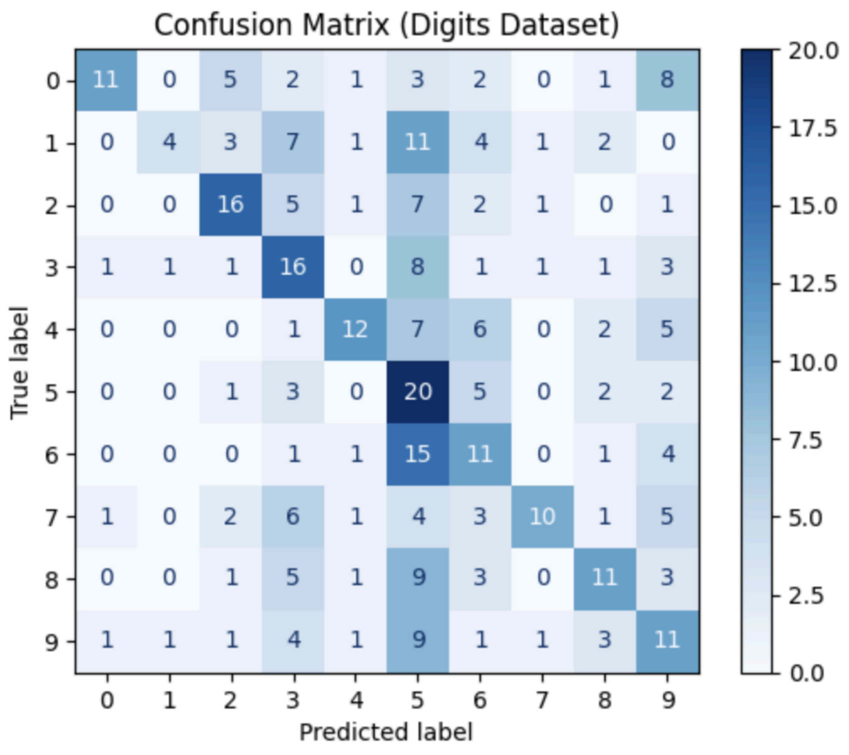
- **Code Snippet:**

```
# --- TRANSFORMS ---
transform_train = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.05)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

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

With no Transformation or Normalization

Without the use of transformation for normalization applied to the MNIST data set, the results that we got for prediction off of the 330 images from the class. To map the predictions, we used the scikit-learn library to analyze the data. From the matrix, we can see that this version of the model did best with predictions on digit five and the worst with digit one. We saw this reflected in our first models, where we were training with the class digits data set and testing with the MNIST data set. Even after the correction, we can see that one is still the hardest number for our model to correctly predict. This may be due to the fact that one can be a straight line, a line on a slant, or made of three lines. It may also be the hardest number due to its similarity to numbers like seven or nine. We even had one misclassified as a three. Sixes also had a high misclassification as fives, this may be due to the similar shape in general. Across all the digits, it appears that the model misclassified digits as a five the most.



<u>Conclusion:</u>

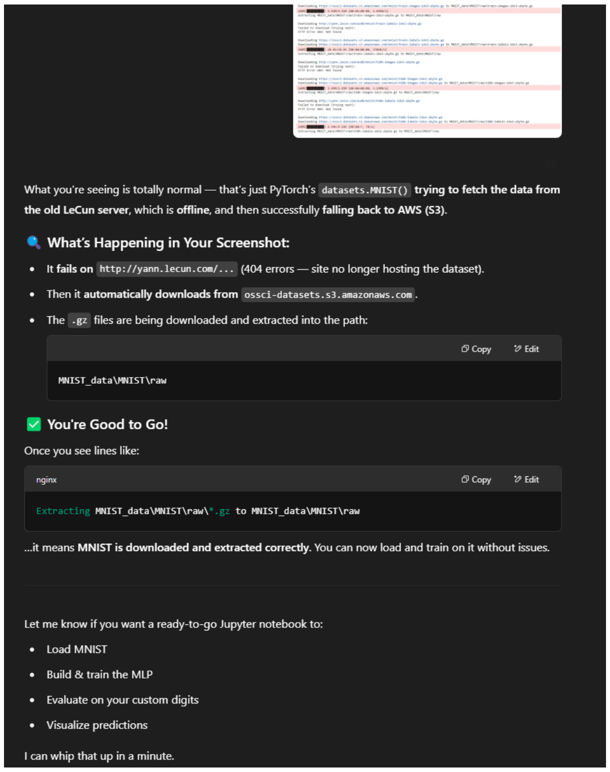
Our best results were transformation and normalization applied to the MNIST data set and testing on the handpicked class data set. The handpicked data set is the images we gathered as a class, but we went through all the values, removing the values that we presumed would give us inaccurate test results.

We believe the reason why this was our best result was that we were training the model to have similar characteristics to our test set. The reason why transformation plays a big role is that a lot of the images that we were testing against were not centered, like the train set. So, being able to augment the location where the number was placed in the train set allowed the model to learn some variation in the positioning of the value. I don't believe normalization on the train set is very optimal, in my opinion, because the numbers already had a pretty big contrast between the actual value and background, while the test set needed it the most, where some examples didn't have the best variation between the background and the value.

Overall, I think where we could try and improve our accuracy is maybe adjusting the neural net, the learning rate, and possibly other parameters in building our model. Where I believe we would see the biggest accuracy would possibly be finding a way to center the test set values, so there would not be such a huge discrepancy between the train and test sets.

● Link to GitHub repo: https://github.com/vlad907/CSCI580_Spring25_Group1

● References:

- Chatgpt 3.5
    - Utilized ChatGPT mainly to assist me in writing my Jupyter notebook file. Writing code in Python is not our strong suit, and it is our first time writing functional code in Python. We used ChatGPT to help us resolve syntax errors and build functions.
    - 

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to MNIST_data\MNIST\raw\train-images-idx3-ubyte.gz
100%|██████████| [illegible]
Extracting MNIST_data\MNIST\raw\train-images-idx3-ubyte.gz to MNIST_data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to MNIST_data\MNIST\raw\train-labels-idx1-ubyte.gz
100%|██████████| [illegible]
Extracting MNIST_data\MNIST\raw\train-labels-idx1-ubyte.gz to MNIST_data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to MNIST_data\MNIST\raw\t10k-images-idx3-ubyte.gz
100%|██████████| [illegible]
Extracting MNIST_data\MNIST\raw\t10k-images-idx3-ubyte.gz to MNIST_data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to MNIST_data\MNIST\raw\t10k-labels-idx1-ubyte.gz
100%|██████████| [illegible]
Extracting MNIST_data\MNIST\raw\t10k-labels-idx1-ubyte.gz to MNIST_data\MNIST\raw
```

What you're seeing is totally normal — that's just PyTorch's `datasets.MNIST()` **trying to fetch the data from the old LeCun server**, which is **offline**, and then successfully **falling back to AWS (S3).**

🔍 **What's Happening in Your Screenshot:**

- It **fails on** `http://yann.lecun.com/...` (404 errors — site no longer hosting the dataset).
- Then it **automatically downloads from** `ossci-datasets.s3.amazonaws.com`.
- The `.gz` files are being downloaded and extracted into the path:

```
MNIST_data\MNIST\raw
```

✅ **You're Good to Go!**

Once you see lines like:

```nginx
Extracting MNIST_data\MNIST\raw\*.gz to MNIST_data\MNIST\raw
```

...it means **MNIST is downloaded and extracted correctly.** You can now load and train on it without issues.

Let me know if you want a ready-to-go Jupyter notebook to:

- Load MNIST
- Build & train the MLP
- Evaluate on your custom digits
- Visualize predictions

I can whip that up in a minute.

- https://matplotlib.org/stable/gallery/index.html
- https://docs.pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://docs.pytorch.org/vision/stable/index.html
- https://docs.pytorch.org/tutorials/
- https://docs.pytorch.org/docs/stable/index.html