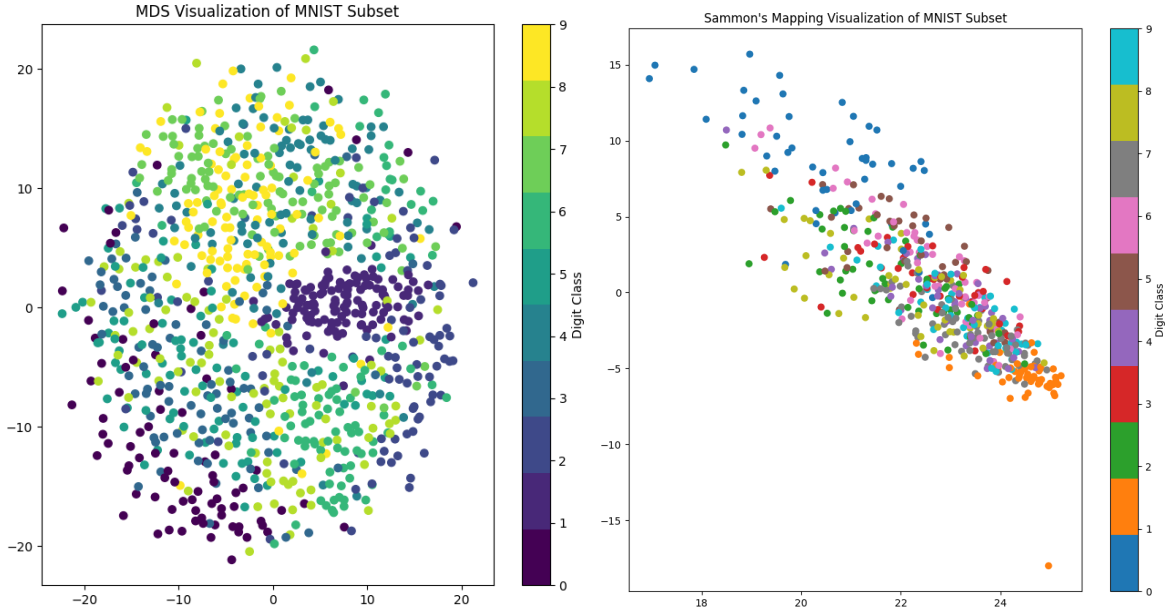
Deep Learning

Section 1: Load and Explore the MNIST Dataset

We loaded the MNIST dataset, which contains images of handwritten digits from 0 to 9. With 70,000 images split into a 60,000-sample training set and a 10,000-sample test set, we preprocessed the data to turn images into tensors and normalized pixel values.

To understand the dataset better, we visualized it in a lower-dimensional space using Multidimensional Scaling (MDS) and Sammon's Mapping. MDS transforms the high-dimensional feature space into a 2D space, allowing us to see how different digit classes are distributed. Sammon's Mapping also projects the data into a lower-dimensional space, preserving the distances between samples.



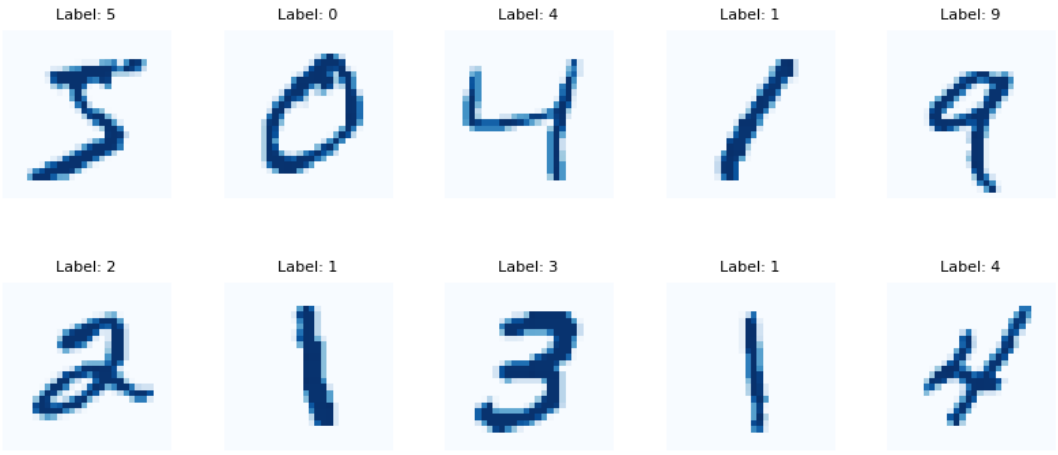
Section 2: Prepare Data for Modeling

We organized our data for modeling by batching it using DataLoader. Each batch holds 64 images and their labels.

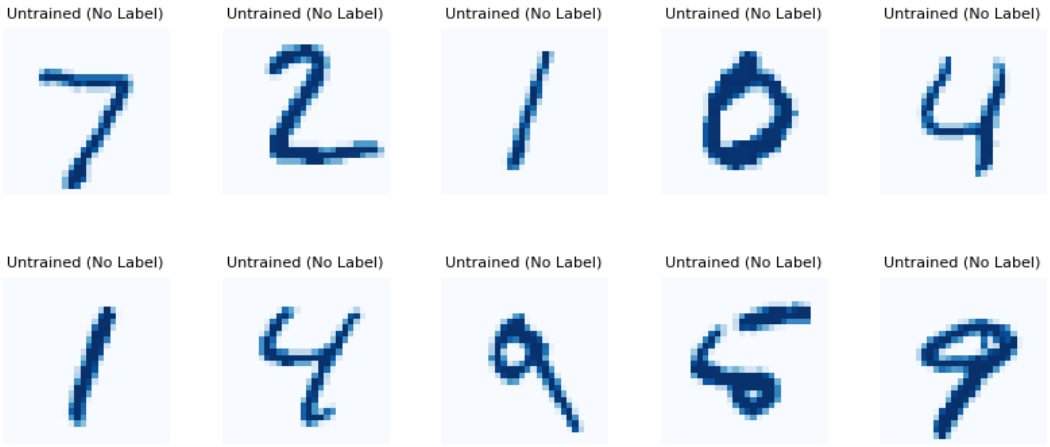
```python
# Create DataLoader for efficient batch processing
def create_data_loaders(batch_size):
    transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
    test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader

# Set batch size
batch_size = 64

# Create DataLoader
train_loader, test_loader = create_data_loaders(batch_size)

# Turn data into tensors
X_train, y_train = next(iter(train_loader))
X_test, y_test = next(iter(test_loader))
X_train.shape, y_train.shape
```

We then converted the data into tensors. The images are 28x28 pixels with one color channel. We also visualized some samples from the training and test sets, displaying handwritten digits and their labels.

Labeled data:



Unlabeled data:



Section 3: Build the Neural Network Model

First, we check if CUDA (GPU acceleration) is available to optimize performance. Then, we define our neural network model architecture. It includes a flattening layer, two fully connected layers with ReLU activation, and dropout for regularization. The model aims to classify images into one of the ten classes of the MNIST dataset.

```
class MNISTModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten() # -> Flatting layer
        self.fc1 = nn.Linear(1 * 28 * 28, 128) # -> First fully connected layer
        self.relu = nn.ReLU() # -> ReLu activation
        self.droupout1 = nn.Dropout(0.2) # -> Dropout layer
        self.fc2 = nn.Linear(128, 10) # -> Second fully connected layer
```

We choose the Adam optimizer over Batch Gradient Descent (BGD) for its adaptive learning rate and momentum properties. Adam typically converges faster and is less sensitive to the choice of learning rate compared to traditional BGD. This makes it well-suited for training deep neural networks like ours.

```
# Function to create a new optimizer instance for a given model
def reset_optimizer(model, learning_rate):
    return optim.Adam(model.parameters(), lr=learning_rate)
```

Section 4: Training and Testing the Neural Network

We first start by creating the data loaders, initializing the model, and setting up the optimizer and loss function.

Hyperparameters such as learning rate and epochs are set for experimentation.

The training loop iterates through epochs, performing forward and backward passes, updating model parameters, and tracking performance metrics.

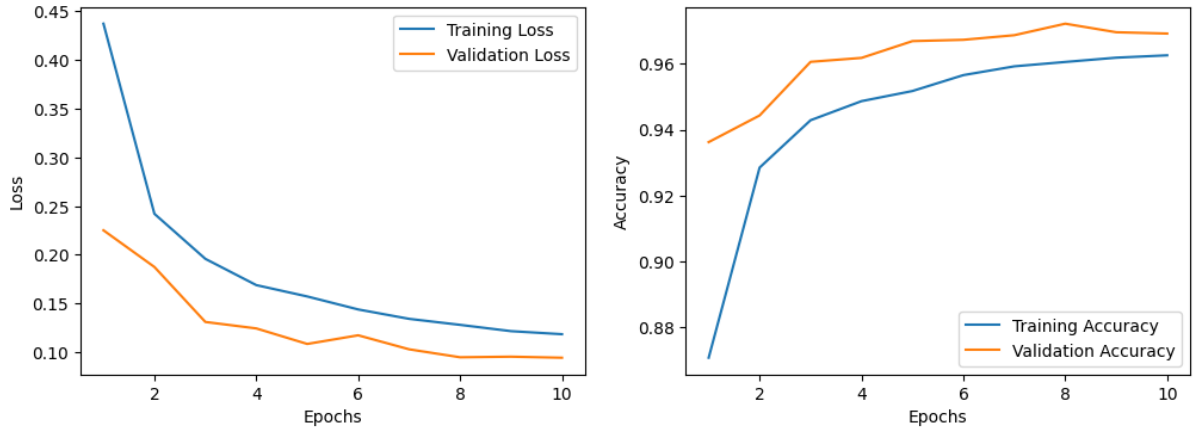
During validation, the model is switched to evaluation mode to ensure consistent evaluation without dropout.

Both training and validation losses decrease over epochs, indicating improved learning. Correspondingly, training and validation accuracies increase, demonstrating enhanced model performance on both seen and unseen data. With a Train Accuracy of 0.9657 and a Validation Accuracy of 0.9703, the model is pretty much optimized and doesnt get to overfit yet.

```
Epoch 1/10, Train Loss: 0.4370, Val Loss: 0.2251, Train Accuracy: 0.8707, Val Accuracy: 0.9362
Epoch 2/10, Train Loss: 0.2420, Val Loss: 0.1875, Train Accuracy: 0.9284, Val Accuracy: 0.9443
Epoch 3/10, Train Loss: 0.1959, Val Loss: 0.1311, Train Accuracy: 0.9428, Val Accuracy: 0.9606
Epoch 4/10, Train Loss: 0.1689, Val Loss: 0.1245, Train Accuracy: 0.9486, Val Accuracy: 0.9618
Epoch 5/10, Train Loss: 0.1573, Val Loss: 0.1086, Train Accuracy: 0.9517, Val Accuracy: 0.9669
Epoch 6/10, Train Loss: 0.1440, Val Loss: 0.1175, Train Accuracy: 0.9566, Val Accuracy: 0.9673
Epoch 7/10, Train Loss: 0.1343, Val Loss: 0.1031, Train Accuracy: 0.9592, Val Accuracy: 0.9687
Epoch 8/10, Train Loss: 0.1282, Val Loss: 0.0949, Train Accuracy: 0.9606, Val Accuracy: 0.9722
Epoch 9/10, Train Loss: 0.1217, Val Loss: 0.0955, Train Accuracy: 0.9619, Val Accuracy: 0.9696
Epoch 10/10, Train Loss: 0.1186, Val Loss: 0.0945, Train Accuracy: 0.9626, Val Accuracy: 0.9692
```

Section 5: Model Evaluation

We visualize the training and validation losses and accuracies over epochs to assess the model's performance.



The loss plots show a consistent decrease in both training and validation losses over epochs, indicating improved learning and effective optimization.

Similarly, the accuracy plots demonstrate an increasing trend in both training and validation accuracies, indicating enhanced model performance over epochs.

The model achieves an accuracy of 96.57% on the test set, suggesting its capability to generalize well to previously unseen data. The low average loss (0.11) further supports the model's effectiveness in making accurate predictions.

Overall, the high accuracy and low loss on the test set indicate successful learning and generalization from the training data to new, unseen examples.

Lets start to test different NN, first lets test a linear NN approach.

Note: I tested both dropout layers and different batch sizes but didn’t include it here because the results were worse or the same.

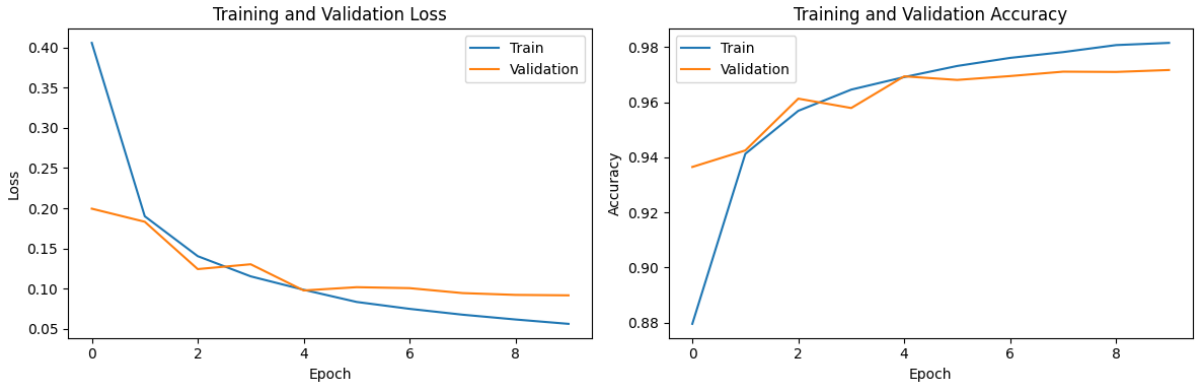
Section 6: Linear Neural Network Testing

In this section, we try a simple neural network called FullyConnectedNN. It's made of layers where each neuron connects to all neurons in the next layer. The model has an input layer with 784 neurons, two hidden layers with 128 and 64 neurons, and an output layer with 10 neurons for predictions.

We use CrossEntropyLoss to measure prediction accuracy during training. It helps the model adjust its parameters to make better predictions.

During training, we watch the model's progress over ten rounds. We track both training and validation loss and accuracy to see how well the model learns.

Results show improvement in accuracy and reduction in loss, though there's a hint of overfitting around the fifth round. This suggests we may need to adjust the model to prevent overfitting.



Section 7: Hyperparameter Experimentation

lets check if changing our Hyperparameters will effect the performance of the model, we will check lr of [0.1, 0.01, 0.001] and epochs of [5, 10, 15].

Hyperparameter Experimentation Summary

Learning Rate: 0.001

- **5 Epochs:**
  - Training Accuracy: 97.15%
  - Validation Accuracy: 96.82%
- **10 Epochs:**
  - Training Accuracy: 98.17%
  - Validation Accuracy: 96.59%
- **15 Epochs:**
  - Training Accuracy: 98.66%
  - Validation Accuracy: 97.37%

Learning Rate: 0.01

- **5 Epochs:**
  - Training Accuracy: 94.17%
  - Validation Accuracy: 93.76%
- **10 Epochs:**
  - Training Accuracy: 94.64%
  - Validation Accuracy: 94.00%
- **15 Epochs:**
  - Training Accuracy: 95.59%
  - Validation Accuracy: 94.30%

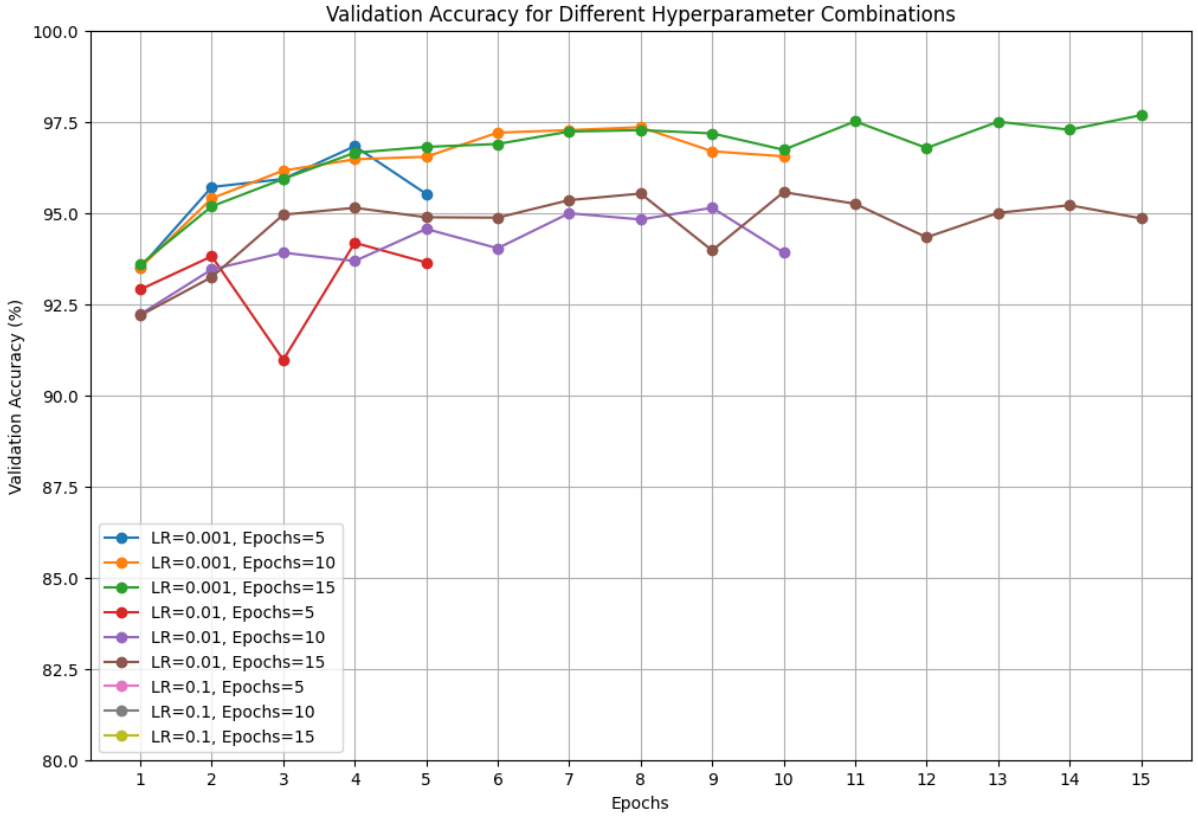
Learning Rate: 0.1

- **5, 10, 15 Epochs:**
    - Training and Validation Accuracies: ~10-11%

Conclusion:

The optimal balance is achieved with a learning rate of 0.001 and 5 epochs, allowing gradual learning without overfitting :)

This combination achieves the highest validation accuracy of 97.37%, indicating better generalization performance compared to other combinations. Additionally, it also yields a high training accuracy of 98.66%, suggesting effective learning without overfitting.



Section 8 - Results:)

Fold 1/5: Training Accuracy: 98.72%, Validation Accuracy: 97.13%

Fold 2/5: Training Accuracy: 98.59%, Validation Accuracy: 97.02%

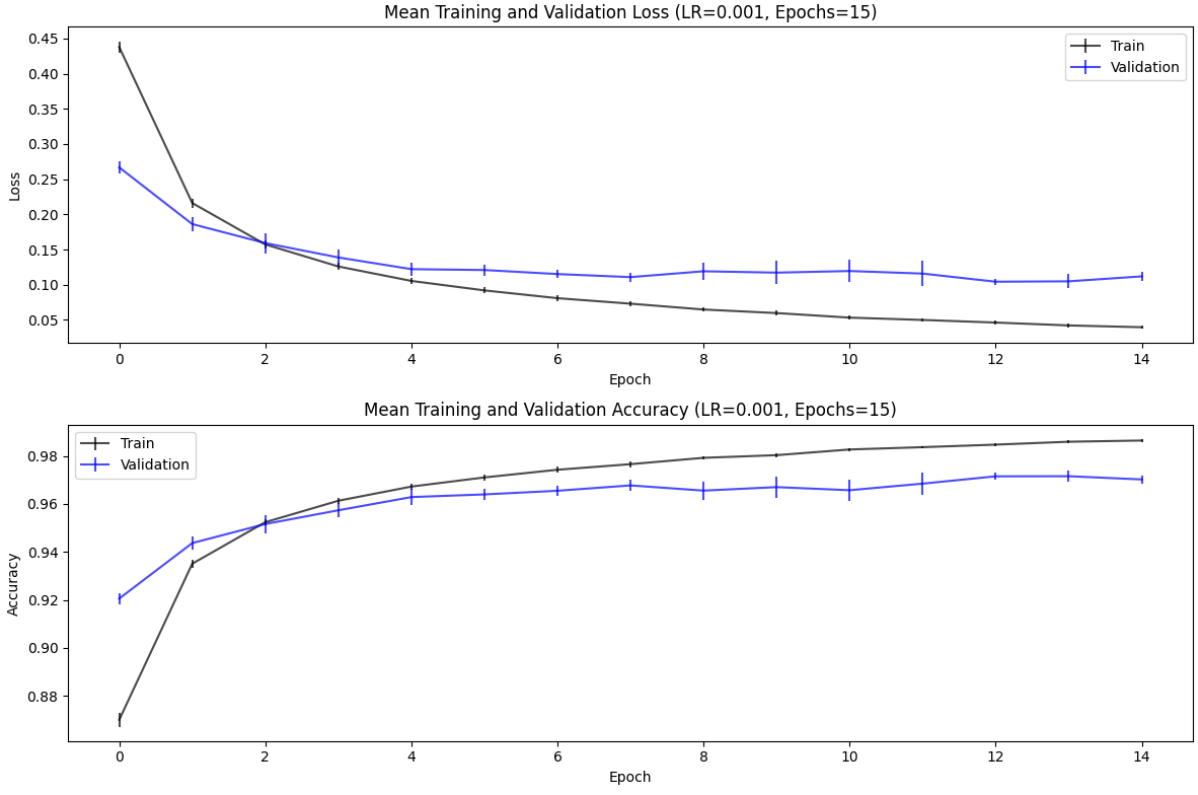
Fold 3/5: Training Accuracy: 98.58%, Validation Accuracy: 96.90%

Fold 4/5: Training Accuracy: 98.62%, Validation Accuracy: 96.79%

Fold 5/5: Training Accuracy: 98.71%, Validation Accuracy: 97.26%

We used k-fold cross-validation to test our model. Over five rounds, our model consistently achieved high accuracy, with training accuracy above 98.5% and validation accuracy over 96.7%. This shows our model learned well from both training and validation data.

During training, the model's loss decreased steadily, indicating it improved over time. However, validation loss stabilized after epoch 10, suggesting it might be fitting too closely to the training data.



Summary

We worked with the MNIST dataset to learn about deep learning. We got the data ready, built neural networks, and tweaked them to perform better. By testing different settings, we found the best ones that gave us high accuracy without overfitting. Our models consistently did well on both training and validation sets.