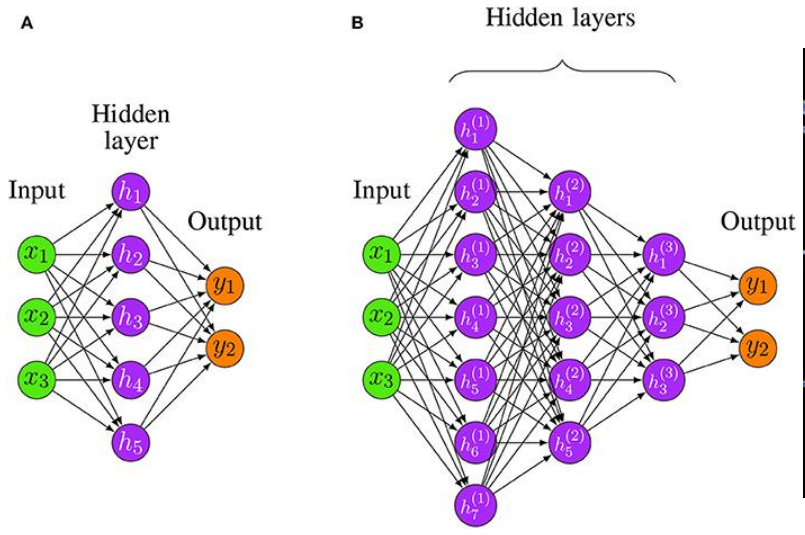
Machine Learning

Assignment 2:

Handwritten Digit Recognition System With
Custom Neural Network


A
Hidden layer
Input
Output
x_1 x_2 x_3
h_1 h_2 h_3 h_4 h_5
y_1 y_2
B
Hidden layers
Input
Output
$h_1^{(1)}$ $h_2^{(1)}$ $h_3^{(1)}$ $h_4^{(1)}$ $h_5^{(1)}$ $h_6^{(1)}$ $h_7^{(1)}$
$h_1^{(2)}$ $h_2^{(2)}$ $h_3^{(2)}$ $h_4^{(2)}$ $h_5^{(2)}$
$h_1^{(3)}$ $h_2^{(3)}$ $h_3^{(3)}$

Under the supervision of:

➔ **Dr. Marwan Torki**

➔ **Eng. Zeyad Ezzat**

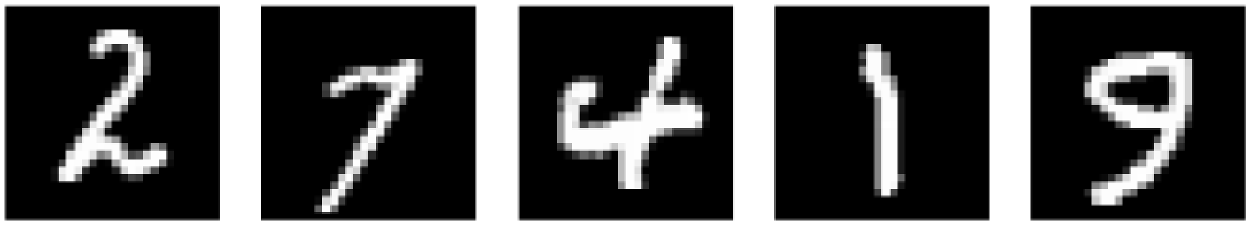
Names:	**IDs:**
Saad El Dine Ahmed Saad	7370
Morougue Mahmoud Ghazal	7524

Handwritten digit recognition system

In the vast landscape of **deep learning** and artificial intelligence, the ability to **recognize handwritten digits** is a fundamental yet challenging task.

This project develops a **system that recognizes handwritten digits**, leveraging the power of deep learning techniques.

The focal point is the creation of a **PyTorch-based neural network model** trained on the widely acclaimed **MNIST dataset**, containing **images** of handwritten digits ranging from **0 to 9**.



➔ In order to accomplish such a task, we went through a systematic process of steps that seamlessly meld theory with hands-on implementation.

Step 1: Data Preprocessing and Handling

1. Loading and Adjusting the Dataset:

- The code starts by loading the training dataset from a CSV file.
- It ensures that the "label" column is moved to the last column for consistency.
- Features (pixels) and labels are then separated for further processing.

2. Custom Dataset Class:

- We then define a custom dataset class (**CustomDataset**) that inherits from PyTorch's **Dataset** class.
- The class is designed to take images, labels, and a transformation as input and provide length and item retrieval functionality.

3. Applying Transformations:

- As the file loaded is a .csv file, we need to convert Pandas DataFrames to PyTorch tensors for compatibility.
- Then we define a transformation using PyTorch's **Compose** class, which includes normalization.

4. Creating Dataset Instances:

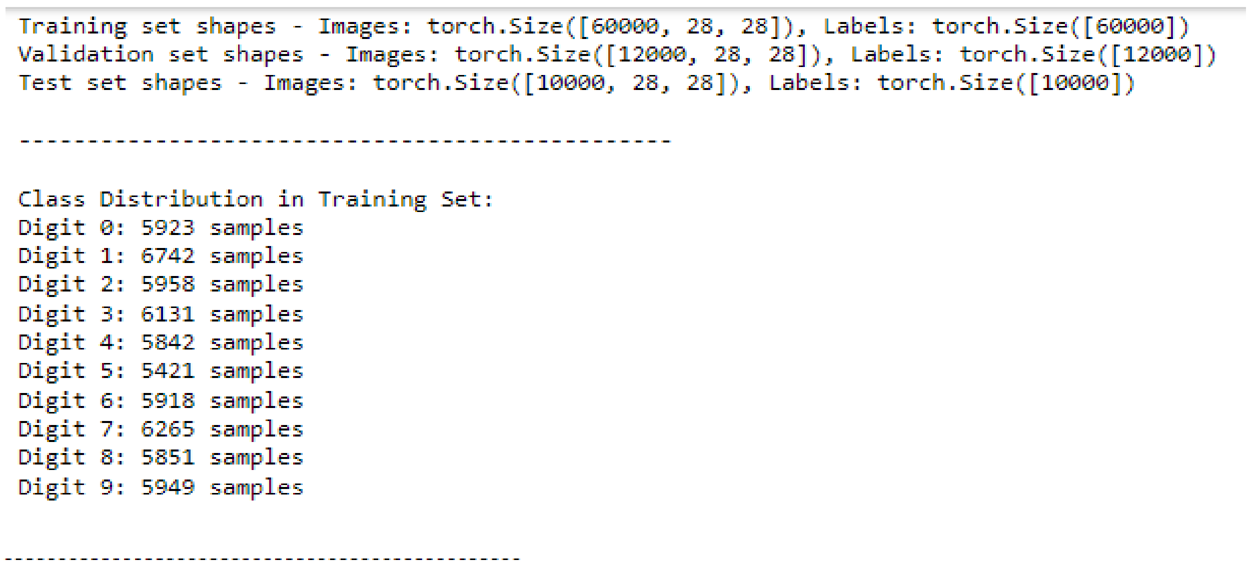
- We Create instances of the **CustomDataset** class for the training dataset.
- Split the training dataset into training and validation sets using **train_test_split** from sklearn.

5. Data Loaders:

- Define the batch size (which we're going to change later) and create data loaders for training and validation sets using PyTorch's **DataLoader** class.
- The **shuffle** parameter is set to **True** for the training loader to enhance training performance.

6. Visualization:

- Define a function (**visualize_samples**) for visualizing sample images and their labels.
- Call the function to visualize a few samples from the training set, then the code prints shapes of the training and validation sets, accesses images and labels from the tuple in **val_set**, explores class distribution in the training set, and finally visualizes a few samples from the training set.

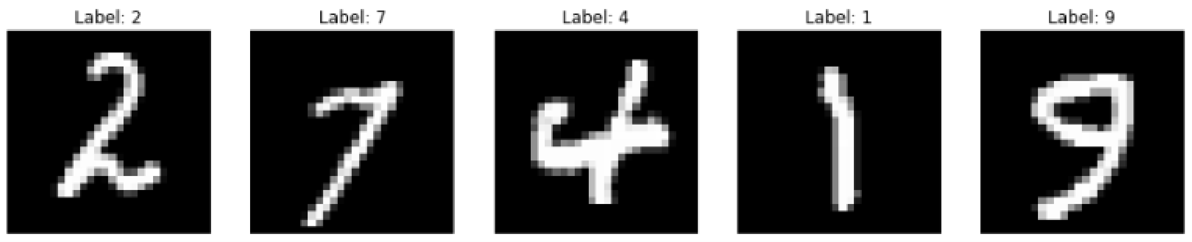


```
Training set shapes - Images: torch.Size([60000, 28, 28]), Labels: torch.Size([60000])
Validation set shapes - Images: torch.Size([12000, 28, 28]), Labels: torch.Size([12000])
Test set shapes - Images: torch.Size([10000, 28, 28]), Labels: torch.Size([10000])

------------------------------------------------

Class Distribution in Training Set:
Digit 0: 5923 samples
Digit 1: 6742 samples
Digit 2: 5958 samples
Digit 3: 6131 samples
Digit 4: 5842 samples
Digit 5: 5421 samples
Digit 6: 5918 samples
Digit 7: 6265 samples
Digit 8: 5851 samples
Digit 9: 5949 samples

------------------------------------------------
```



Step 2: Custom Neural Network Architecture with Bonus (Dropout and Layer Normalization)

1. Model Class Definition:

- We Defined a custom neural network class (**Model**) that inherits from PyTorch's **nn.Module**.
- The constructor initializes layers, including fully connected (linear) layers, dropout layers, and layer normalization.
- It utilizes He for initializing the weights of the neural network layers. This helps promote stable and efficient training, especially when ReLU activation functions are employed, as ReLU tends to perform well with He initialization.
- It specifies the loss function (CrossEntropyLoss) and optimizer (Stochastic Gradient Descent - SGD).

2. Forward Method:

- It implements the forward pass of the neural network using ReLU activation functions and SoftMax for multiclass classification.
- Applies layer normalization and dropout for regularization.

3. Fit and Predict Methods:

- **fit**: Method for training the model. Computes loss, performs backward pass, and updates weights using the optimizer.
- **predict**: Method for making predictions. Returns the index of the maximum value in the output tensor.

4. Model Instantiation and Testing:

- Sets the input size, hidden layer sizes, and output size.
- Creates an instance of the **Model** class with specified parameters.
- Tests the model's layer configuration using **model.parameters**.

5. Training Function:

- The **train** function is responsible for training the neural network.
- It sets the model to training mode using **model.train()**.
- Iterates through batches in the training loader, performs forward pass, computes loss, and updates weights using the **fit** method of the model.
- Tracks the total loss and the number of correctly predicted training samples.
- Calculates average training loss and accuracy.

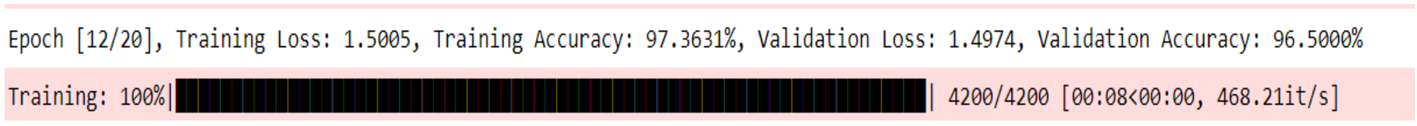
6. Validation Function:

- The **validate** function is responsible for evaluating the model on the validation set.
- Sets the model to evaluation mode using **model.eval()**.
- Iterates through batches in the validation loader and computes validation loss without updating weights.
- Tracks the total validation loss and the number of correctly predicted validation samples.
- Calculates average validation loss and accuracy.

➔ Both **Training** and **Validation** functions play a crucial role in the training and evaluation process, enabling the model to learn from the training data and assess its performance on a separate validation set.

The use of **model.train()** and **model.eval()** in both **Training** and **Validation** functions ensures proper behavior of layers like **dropout** during training and evaluation.

The **"tqdm progress bar"** enhances the visualization of the training progress.



```
Epoch [12/20], Training Loss: 1.5005, Training Accuracy: 97.3631%, Validation Loss: 1.4974, Validation Accuracy: 96.5000%
Training: 100%|████████████████████████████████████████| 4200/4200 [00:08<00:00, 468.21it/s]
```

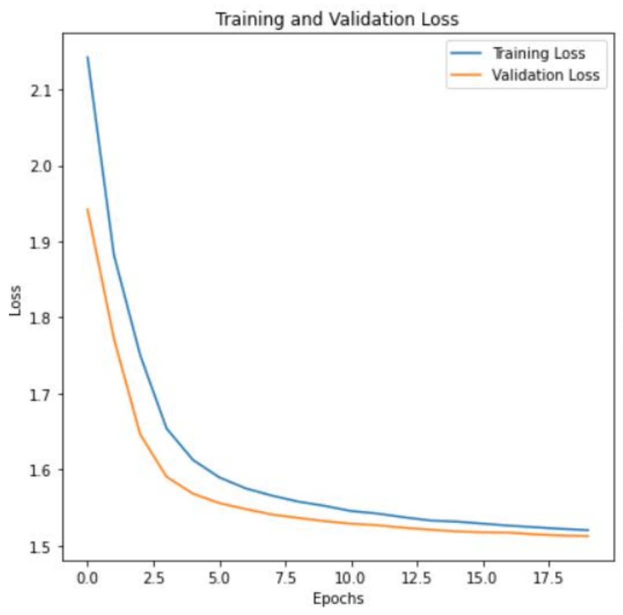
The **accuracy calculations** provide insights into the model's performance on both the training and validation sets.

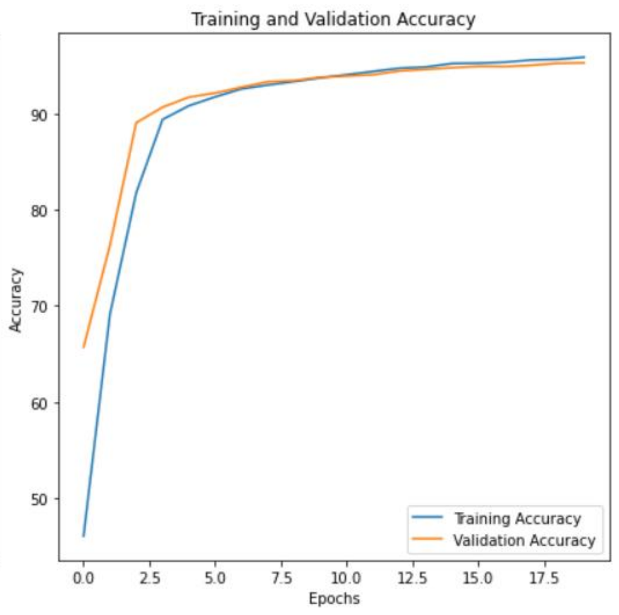
Step 3: Training Loop

1. Default Training Loop:

- This section runs the default training loop for a specified number of epochs.
- It calls the **train** function for training and the **validate** function for validation.
- Tracks training and validation loss, as well as training and validation accuracy.
- Prints and visualizes metrics for each epoch.

```
Training: 100%|██████████████████████████████████████████████████████████████████████| 263/263 [00:01<00:00, 262.97it/s]
Epoch [16/20], Training Loss: 1.5648, Training Accuracy: 92.7708%, Validation Loss: 1.5432, Validation Accuracy: 92.9405%
Training: 100%|██████████████████████████████████████████████████████████████████████| 263/263 [00:00<00:00, 273.99it/s]
Epoch [17/20], Training Loss: 1.5618, Training Accuracy: 92.9464%, Validation Loss: 1.5399, Validation Accuracy: 93.2262%
Training: 100%|██████████████████████████████████████████████████████████████████████| 263/263 [00:01<00:00, 257.96it/s]
Epoch [18/20], Training Loss: 1.5577, Training Accuracy: 93.1756%, Validation Loss: 1.5376, Validation Accuracy: 93.3929%
Training: 100%|██████████████████████████████████████████████████████████████████████| 263/263 [00:00<00:00, 277.66it/s]
Epoch [19/20], Training Loss: 1.5536, Training Accuracy: 93.3065%, Validation Loss: 1.5359, Validation Accuracy: 93.5238%
Training: 100%|██████████████████████████████████████████████████████████████████████| 263/263 [00:00<00:00, 268.61it/s]
Epoch [20/20], Training Loss: 1.5511, Training Accuracy: 93.5744%, Validation Loss: 1.5343, Validation Accuracy: 93.6429%
```



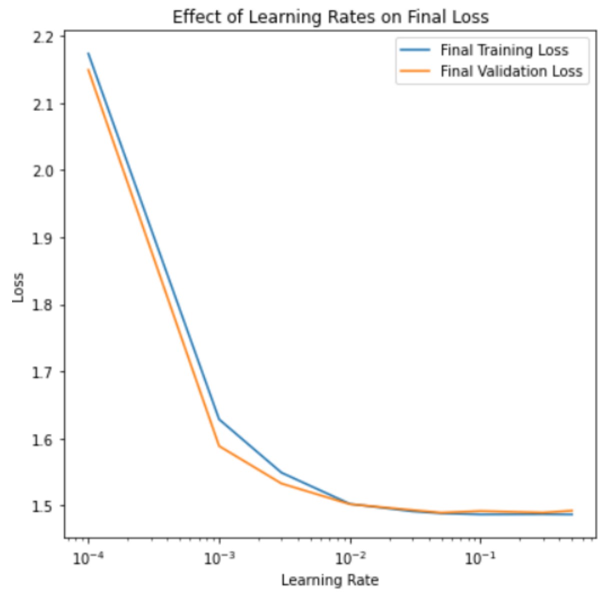


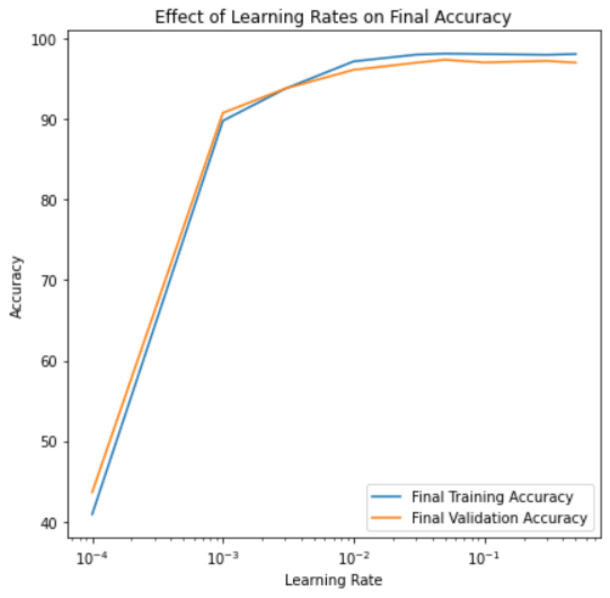
Step 4: Analysis and Iterations

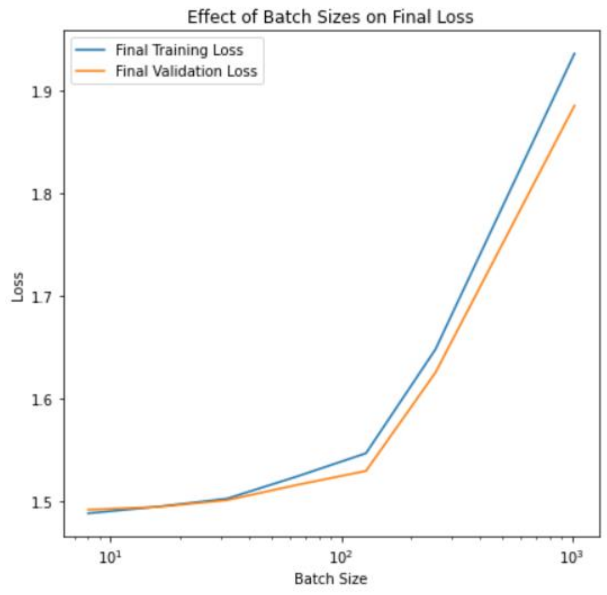
After initializing a default training loop, this section systematically explores different hyperparameters such as **learning rates** and **batch sizes** to understand their **effects** on the model's training and validation **performance.**

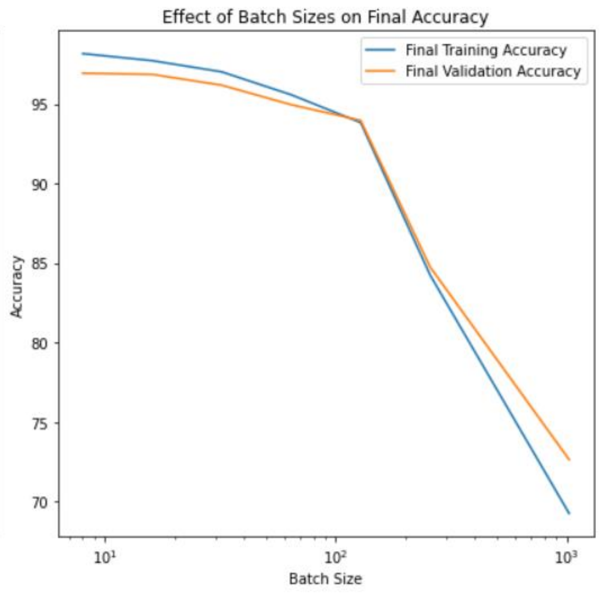
The **iterative** process is crucial for **hyperparameter tuning** and **optimizing** the model for the given task.

In addition to the **visualizations**, that provide insights into the trends and help in making **informed decisions** about hyperparameter choices.



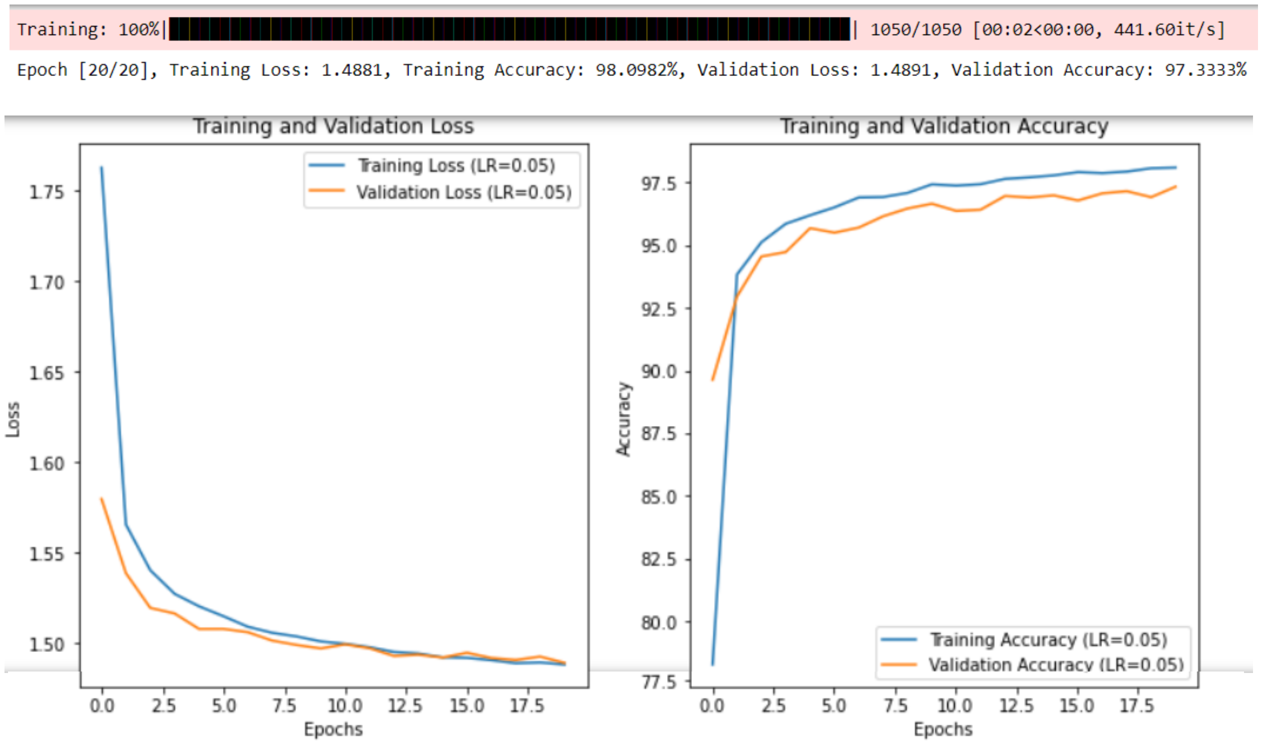






1. Customized Training Loop for Different Learning Rates:

- Explores the impact of different learning rates on training and validation performance.
- Instantiates the model with each learning rate and evaluates its performance.
- Plots training and validation loss and accuracy for each learning rate.



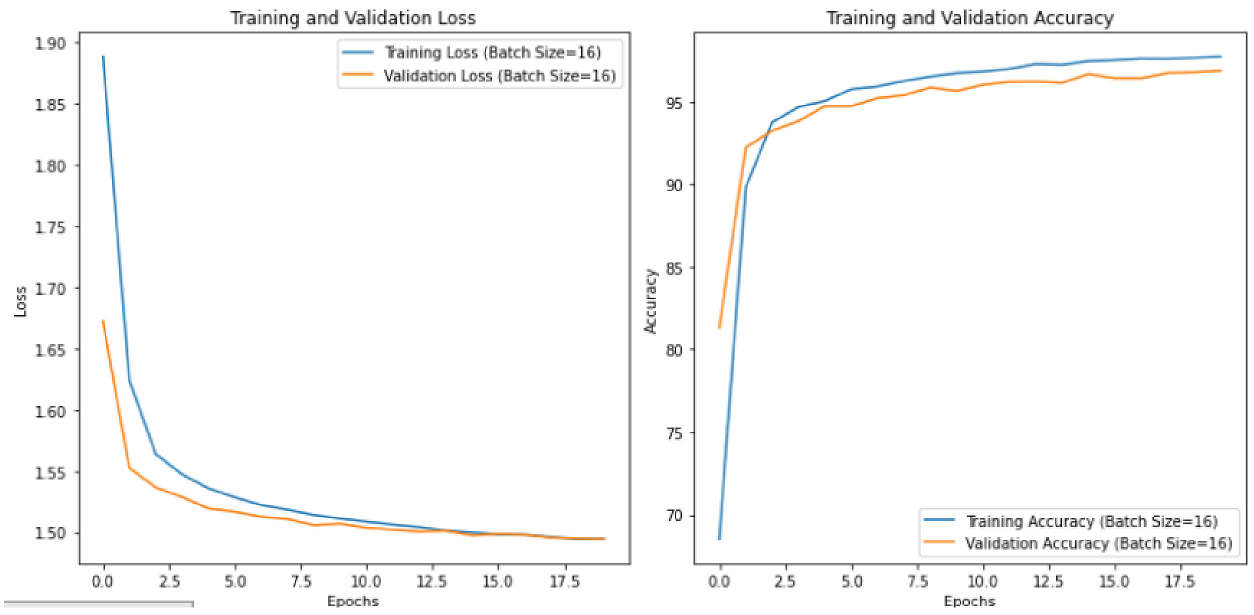
```
Training: 100%|████████████████████████████████████████| 1050/1050 [00:02<00:00, 441.60it/s]
Epoch [20/20], Training Loss: 1.4881, Training Accuracy: 98.0982%, Validation Loss: 1.4891, Validation Accuracy: 97.3333%
```

Training and Validation Loss
Training Loss (LR=0.05)
Validation Loss (LR=0.05)
Loss
Epochs
1.75
1.70
1.65
1.60
1.55
1.50
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Training and Validation Accuracy
Training Accuracy (LR=0.05)
Validation Accuracy (LR=0.05)
Accuracy
Epochs
97.5
95.0
92.5
90.0
87.5
85.0
82.5
80.0
77.5

2. Customized Training Loop for Different Batch Sizes:

- Explores the impact of different batch sizes on training and validation performance.
- Creates new data loaders with each batch size and evaluates the model's performance.
- Plots training and validation loss and accuracy for each batch size.

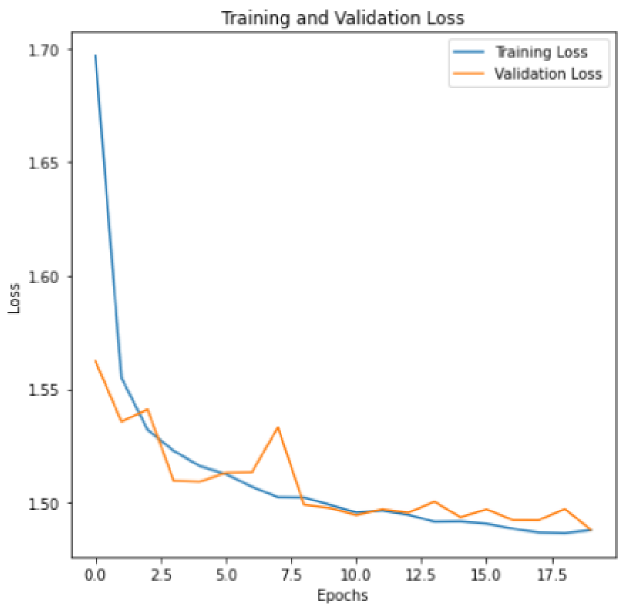
```
Training: 100%|████████████████████████████████████████████████████████████████| 210
0/2100 [00:04<00:00, 465.77it/s]
Epoch [20/20], Training Loss: 1.4948, Training Accuracy: 97.7202%, Validation Loss: 1.49
45, Validation Accuracy: 96.8571%
```

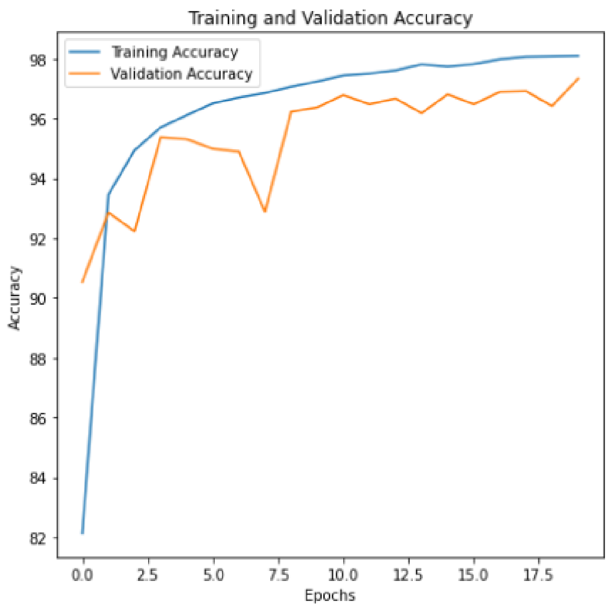


Best Model Selection:

This section utilizes a **grid search over specified hyperparameters** to **find** the **best combination**, **iterates** through all combinations, training and validating a model for each.
It **Selects** the model with the **highest validation accuracy** as the best model, **prints** and **visualizes** the **best** hyperparameters, then **plots** training and validation **loss** as well as training and validation **accuracy** for the **best model**.

```
Best Hyperparameters: {'lr': 0.1, 'batch_size': 16}
Validation Accuracy: 97.3333%
```



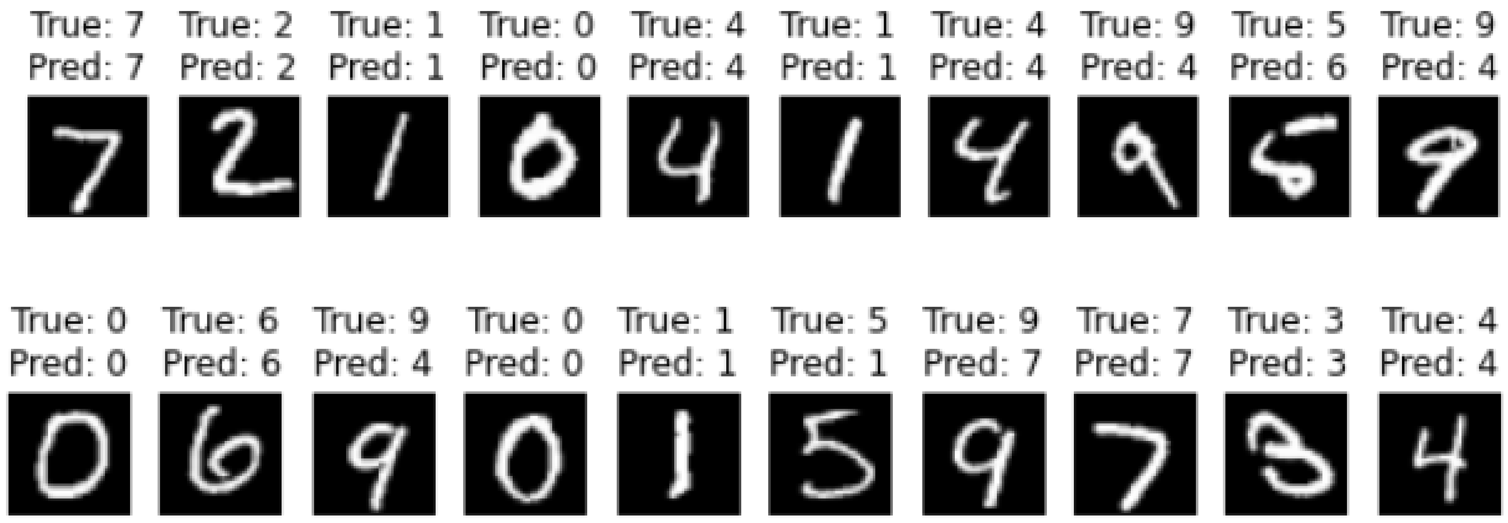


Step 5: Test

This section **loads and prepares** the **test data** to ensure a **proper preparation and evaluation** of the model on the test dataset. It utilizes the **“evaluate”** function to **assess** the **model's performance** on the test set.

We added a **visualization** for 5 samples from the test set without shuffling, in addition to providing a function (**visualize_samples_with_predictions**) to visualize model **predictions** for a subset of test samples. This helps in understanding how well the model performs on individual samples, offering insights into its **predictive capabilities**.

It **evaluates and prints** the test **loss** and **accuracy** for the trained model. These results provide **a comprehensive** summary of the model's **performance** on **unseen** data.



In **conclusion**, the exploration into the realm of deep learning unfolded with a focus on **handwritten digit recognition**. The creation of a **custom PyTorch model**, the systematic experimentation with **hyperparameters**, and the investigation into the subtleties of training dynamics offered valuable insights. What enriched the model's **robustness** is the incorporation of **dropout layers** and **layer normalization**.

This project not only resulted in the development of an effective digit recognition system but also highlighted the **iterative and dynamic nature of machine learning endeavors**.