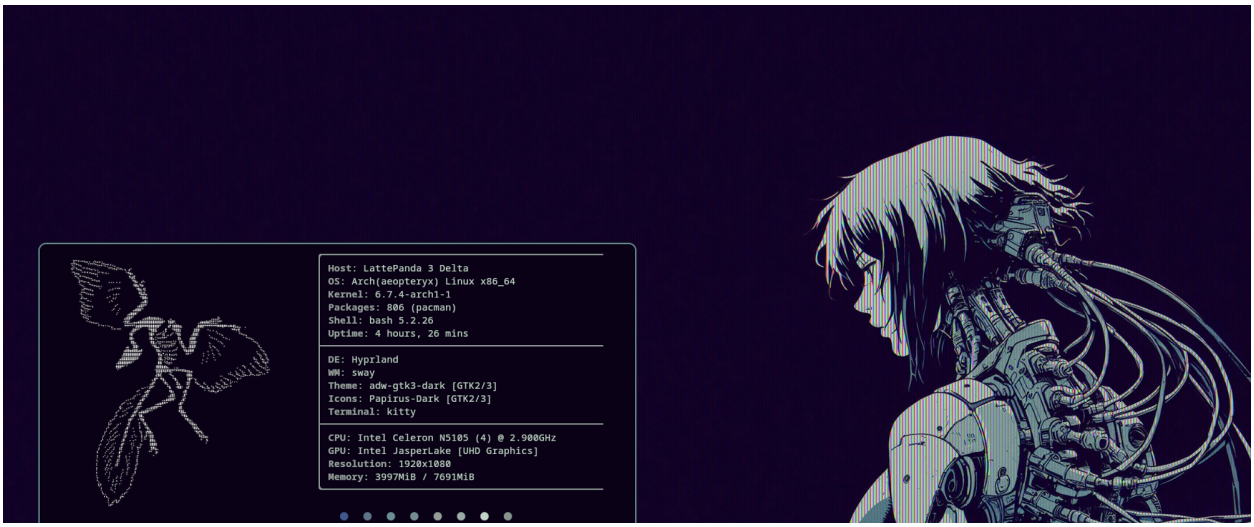


B22EE036_PRML_PA6_REPORT

Author: K. K. N. SHYAM SATHVIK

Roll No: B22EE036

Link to the Colab File:

Google Colaboratory

https://colab.research.google.com/drive/1pQB_17f2PwEEQQVGsVe2Py3nv0wEd1Ss?usp=sharing

Table of Contents

Training Neural Networks

Task-0:

DATASET:

Task-1:

Task-2:

Task-3:

Epoch 1:

Epoch 2:

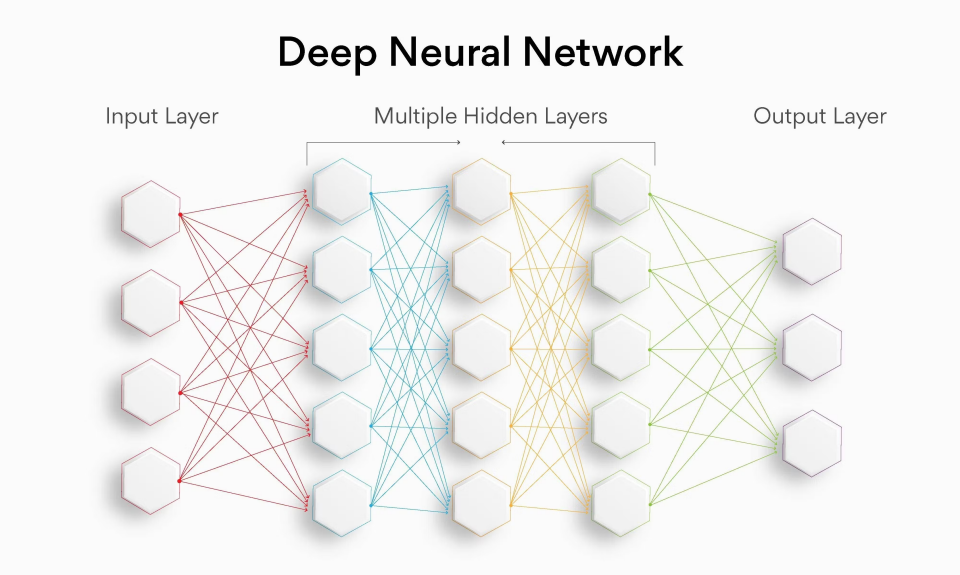
Epoch 3

Epoch 4

Epoch 5

Task-4:

Neural Networks

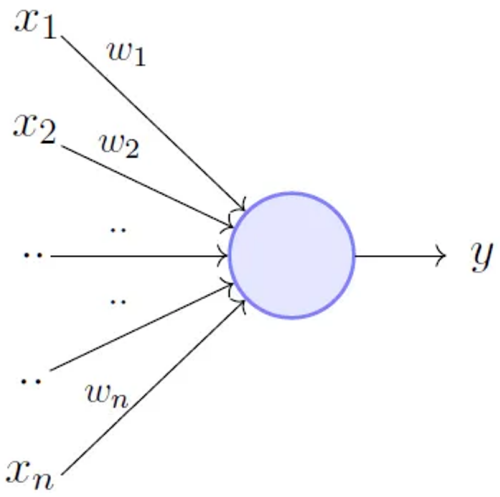


Neural networks consist of layers of neurons, each connected by synapses that transmit signals. These neurons process inputs by

applying weights and biases, adjusting as they learn from data. The typical layers include:

- Input layer: Receives the raw data.
- Hidden layers: Process the inputs through weighted connections.
- Output layer: Produces the final decision or prediction.

Mathematical Model



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

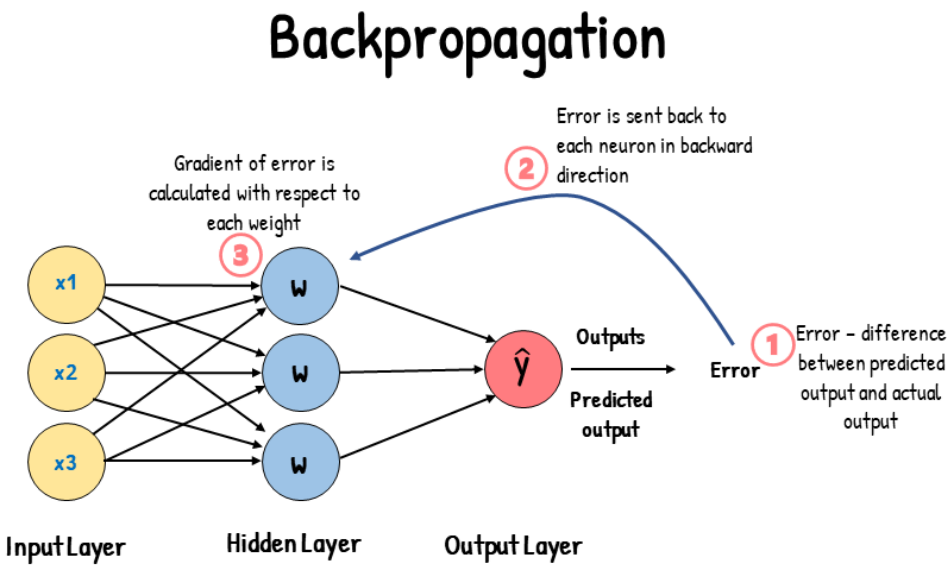
The operation of a neuron in a network can be represented by:

$$y = f\left(\sum_i (w_i \cdot x_i) + b\right)$$

Here,

- y is the output.
- f is the activation function, such as sigmoid or ReLU, that introduces non-linearity.
- w_i represents the weights.
- x_i are the inputs.
- b is the bias term.

Training Neural Networks



Training involves adjusting the weights and biases of the network to minimize the difference between the predicted output and the actual data. This process uses algorithms like:

- **Backpropagation**: A method where errors are propagated backwards through the network to update the weights, improving accuracy gradually.

- **Gradient Descent**: An optimization algorithm that iteratively adjusts parameters to minimize the cost function.

> 💡 Note : Logs and plots are subject to change on reruns.

Task-0:

- Download the MNIST dataset using torchvision. Split data into train, test, and validation. Apply the following augmentations to images:

  RandomRotation, **RandomCrop**, **ToTensor**, and **Normalize**. [2 pts]

DATASET:

The MNIST dataset is an image recognition based dataset, typically used to train classifiers.

- **Content**: MNIST (Modified National Institute of Standards and Technology) dataset consists of handwritten digits, each of which is a 28x28 pixel grayscale image.
- **Dataset Size**: Contains 70,000 images and labels, split into 60,000 training images and 10,000 test images.
- **Labelling**: Each image is labelled with the digit it represents, from 0 to 9.
- The dataset was created by Yann LeCun, a renowned French-American computer scientist.

> 💡 *For the following tasks, the following data splits have the following defintions:*
> -
> ***Train:*** *The data split used to train the Neural Network.*
> -
> ***Validation:*** *The data split used to test the Neural Network after each epoch to ensure a healthy bias vs variance balance, and no overfitting.*
> -
> ***Test:*** *The data split used to test the final, fully trained Neural Network.*

- The dataset was split into train, test and validation (0.2 % of the train data) set with the sample sizes shown in the data loaders section below.

- The following transformations were applied on all the images before feeding them into the Neural Network to improve the robustness of the model.
  - `transforms.ToTensor()`:
    - Converts images from Python Imaging Library (PIL) format or a NumPy array into PyTorch tensors.
    - Automatically scales the image data to the range [0, 1].
  - `transforms.Normalize((0.1307), (0.3081))`:
    - Normalizes the tensor image with a mean of 0.1307 and a standard deviation of 0.3081.

    - This normalization is typically based on the dataset's global mean and standard deviation, helping to standardize inputs for efficient training.

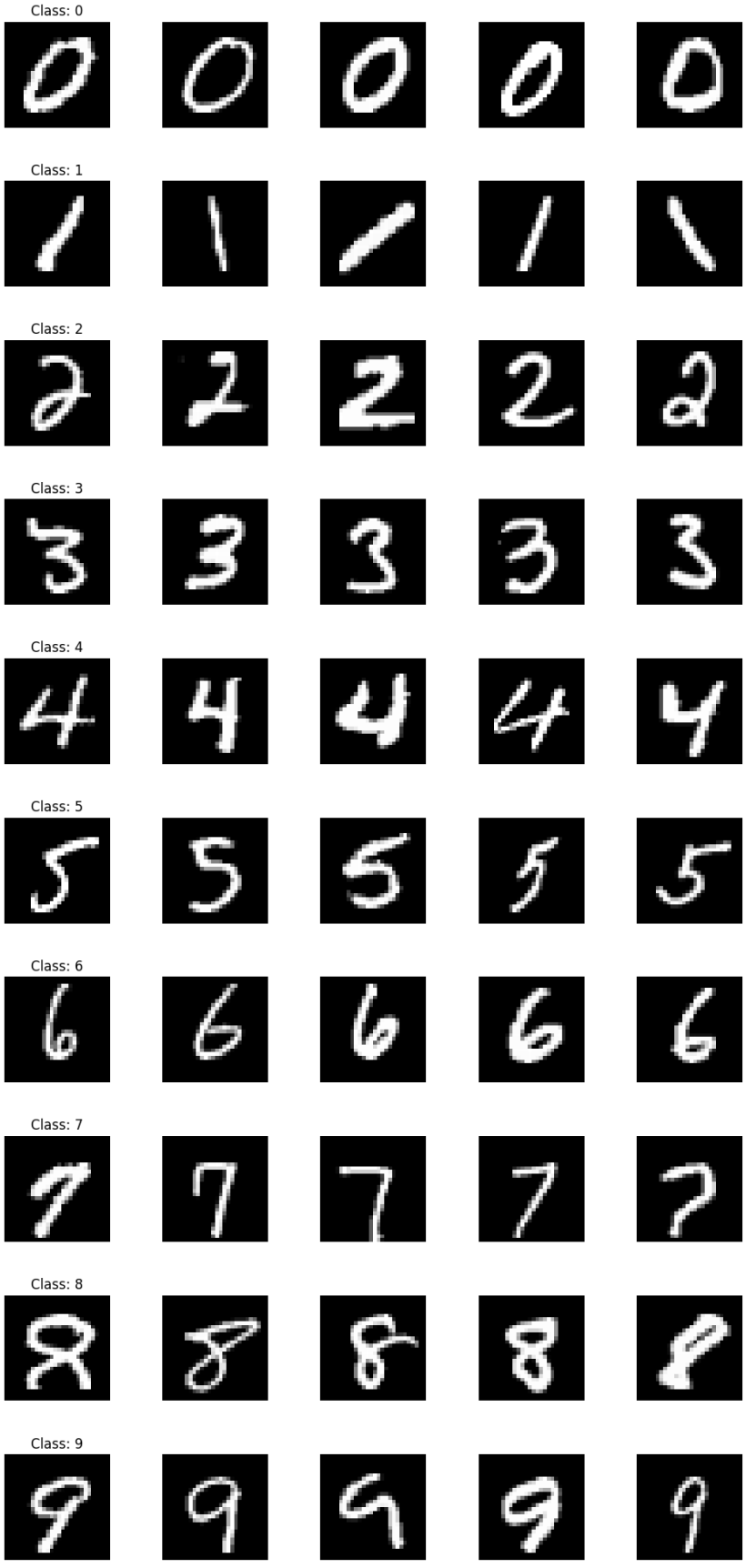
- `transforms.RandomRotation(10)`:
    - Randomly rotates the image by a degree selected between -10 and +10.
    - This augmentation increases model robustness by simulating variations in the dataset.

- `transforms.RandomCrop(24)`:
    - Randomly crops the image to a size of 24x24 pixels from the original size.
    - This can be a form of data augmentation to help the model generalize better by training on different portions of the images.

Task-1:

Plot a few images from each class. Create a **data loader** for the training dataset as well as the testing (validation) dataset. [2 pts]

- **(a)** Samples from each class.


Class: 0
Class: 1
Class: 2
Class: 3
Class: 4
Class: 5
Class: 6
Class: 7
Class: 8
Class: 9

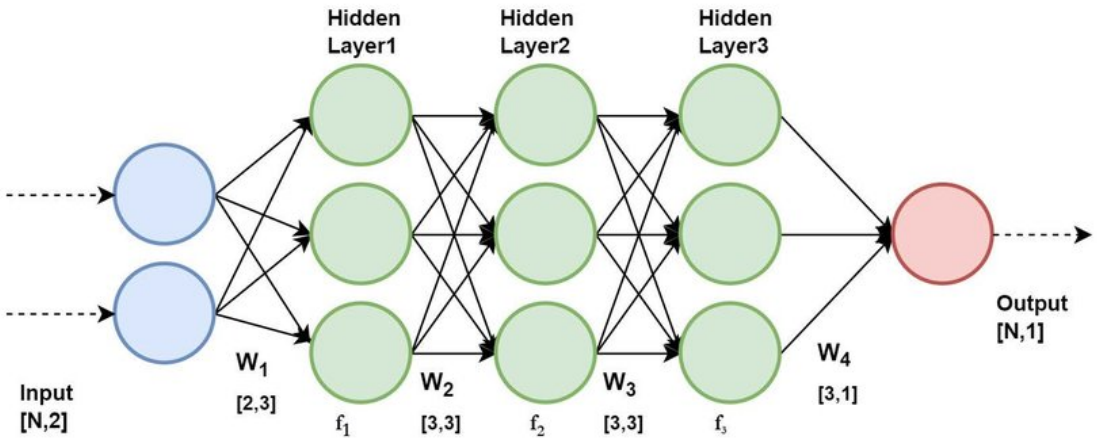
Samples from the MNIST Dataset.

- The data loaders were created with the following sample sizes using `torch.utils.data` after obtaining the data from `torchvision.datasets`.

Data Loader	Number of Samples
Training set	48,000
Validation set	12,000
Test set	10,000

Task-2:

Write a 3-Layer **MLP** using PyTorch all using **Linear layers**. Print the number of **trainable parameters** of the model. [4 pts]



Multilayer Perceptron with 3 Hidden Layers

```
class NN(nn.Module):
    def __init__(self, input_size, num_classes):  # 784 (nodes)
        super(NN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, num_classes)

    def forward(self, x):
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- The above Neural Network implementation follows the following structure:
  - **Base Class**: Inherits from `nn.Module`, a base class for all neural network modules in PyTorch.
  - **Initialization**: The constructor `__init__` accepts two parameters:
    - `input_size`: Number of input nodes (784 for 28x28 images).
    - `num_classes`: Number of output classes for classification.
  - **Layers**:
    - `self.flatten`: Flattens the input tensor to a vector, useful for processing image data.
    - `self.fc1`: First fully connected (dense) layer with 64 output nodes.

        - `self.fc2`: Second fully connected layer with 32 output nodes.
        - `self.fc3`: Third fully connected layer that outputs logits for each class.
    - **Activation Function**: Uses ReLU (Rectified Linear Unit) activation function for non-linearity after the first two fully connected layers.
    - **Forward Pass**:
        - `forward`: Defines the computation performed at every call. The method takes an input tensor `x`, applies the flatten layer, then passes it through the three fully connected layers with ReLU activations for the first two layers.
    - **Output**: The output of the network is the raw scores (logits) for each class from the last layer, suitable for classification tasks with a softmax or similar operation applied afterward for probability distribution.

- The Neural Network has the following specifications (shown using `torch.summary`)

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1                  [-1, 784]               0
            Linear-2                   [-1, 64]          50,240
            Linear-3                   [-1, 32]
```

```
2,080
            Linear-4                   [-1, 10]
330
================================================================
=====
Total params: 52,650
Trainable params: 52,650
Non-trainable params: 0
----------------------------------------------------------------
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.20
Estimated Total Size (MB): 0.21
----------------------------------------------------------------
-----
```

- Function calculate the number of trainable parameters in the designed Neural Network:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requi
```

- The number of trainable parameters in the designed Neural Network are: **52650**

Task-3:

Train the model for **5 epochs** using **Adam** as the optimizer and **Cross Entropy Loss** as the Loss Function. Make sure to evaluate the model on the validation set after each epoch and save the best model as well as **log the accuracy and loss of the model** on **training** and **validation data** at the end of each epoch. [4 pts]

- **Adam was set** as the optimizer and **Cross Entropy Loss** as the Loss Function using the following code:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(params = model.parameters(), lr = lear
```

- **The best model between training epochs is saved using the following code, and the following logic.**

```
def save_model(model, filename):
    torch.save(model.state_dict(), filename)
    print(f"Model saved successfully to {filename}")
```

```
    prev_max = 0
    if acc_valid > prev_max:
      save_model(model, "best_val")
    prev_max = max(acc_valid, prev_max)
```

- The model’s performance on both the validation and training set at the end of each epoch is as follows:

```
`Epoch [1/5]: 100%|██████████| 750/750 [00:15<00:00, 49.33it/
`Epoch: 1  Train Loss: 0.4593  Train Accuracy: 0.9351  Valida
`Epoch [2/5]: 100%|██████████| 750/750 [00:15<00:00, 48.23it/
`Epoch: 2  Train Loss: 0.2025  Train Accuracy: 0.9476  Valida
`Epoch [3/5]: 100%|██████████| 750/750 [00:14<00:00, 50.05it/
`Epoch: 3  Train Loss: 0.1536  Train Accuracy: 0.9638  Valida
`Epoch [4/5]: 100%|██████████| 750/750 [00:18<00:00, 40.15it/
`Epoch: 4  Train Loss: 0.1244  Train Accuracy: 0.9696  Valida
`Epoch [5/5]: 100%|██████████| 750/750 [00:14<00:00, 51.41it/
`Epoch: 5  Train Loss: 0.1008  Train Accuracy: 0.9738  Valida
```

Epoch 1:

Train Loss	0.4593
Train Accuracy	0.9351

Validation Loss	0.2728
Validation Accuracy	0.9352

Epoch 2:

Metric	Value

Train Loss	0.2025
Train Accuracy	0.9476
Validation Loss	0.2279
Validation Accuracy	0.9444

Epoch 3

Metric	Value
Train Loss	0.1536
Train Accuracy	0.9638
Validation Loss	0.1770
Validation Accuracy	0.9568

Epoch 4

Metric	Value
Train Loss	0.1244
Train Accuracy	0.9696
Validation Loss	0.1596
Validation Accuracy	0.9597

Epoch 5

Metric	Value
Train Loss	0.1008
Train Accuracy	0.9738
Validation Loss	0.1515
Validation Accuracy	0.9634

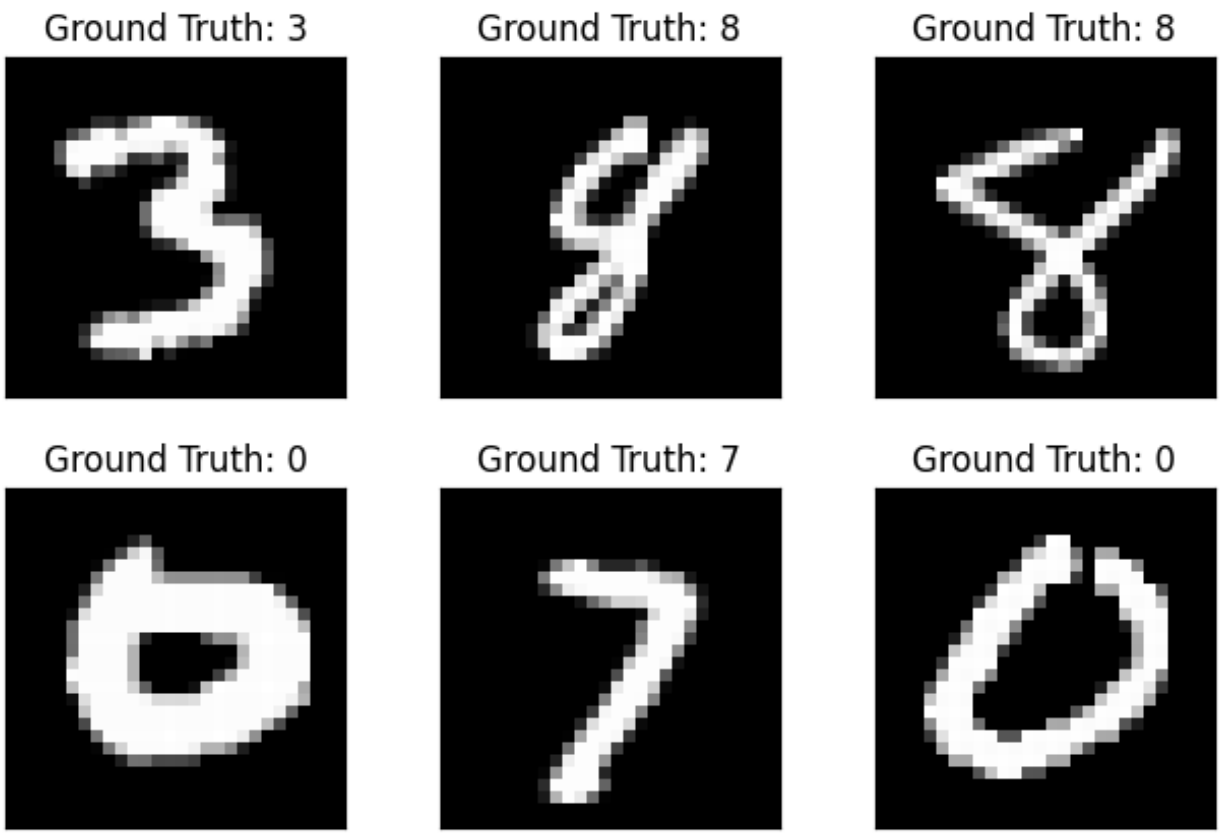
- The final performance of the model on the `test_split` is as follows:

Test Loss	**0.12873058779177013**
Test Accuracy	**0.9614999890327454**

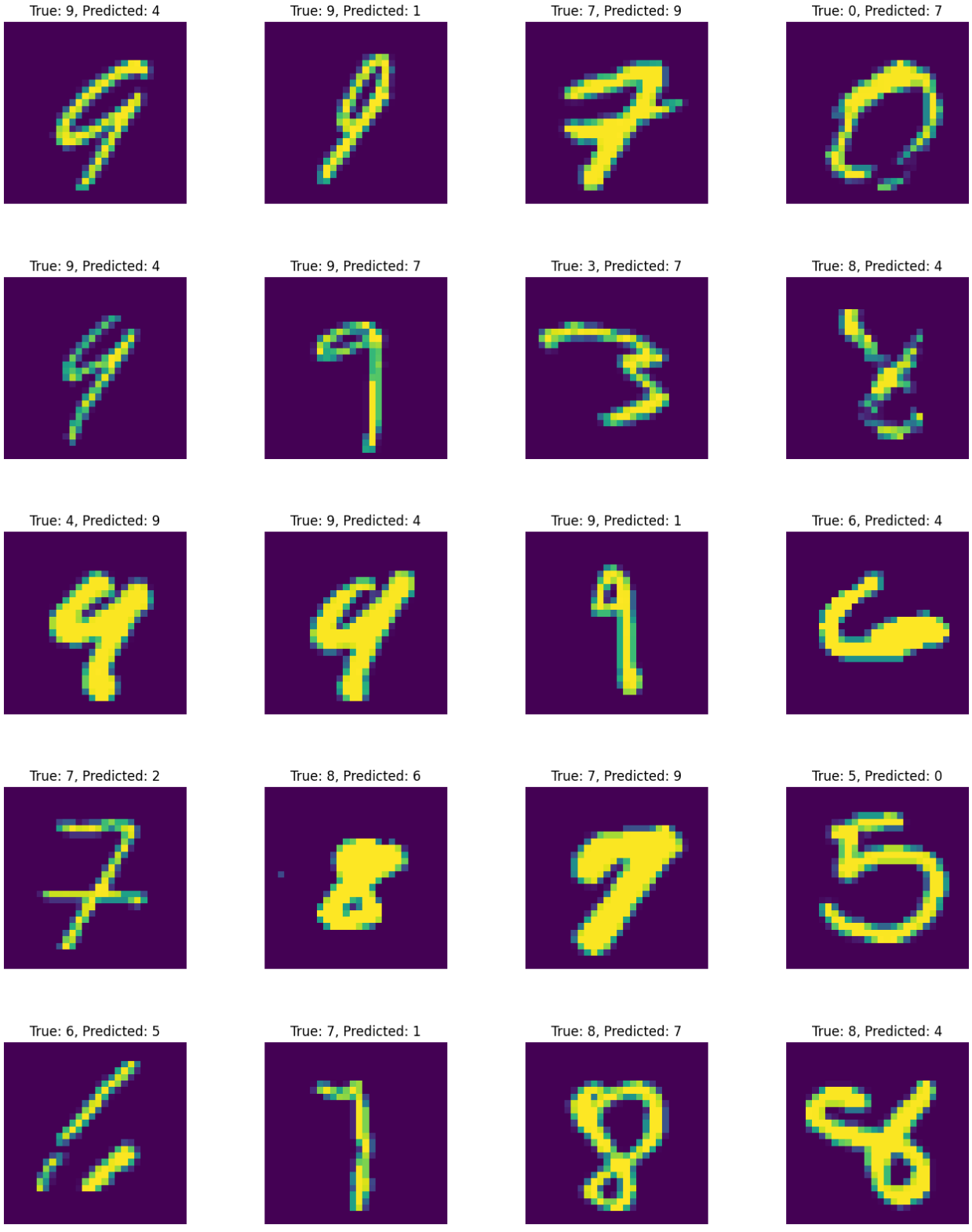
Task-4:

Visualize **correct and Incorrect predictions** along with **Loss-Epoch** and **Accuracy-Epoch graphs** for both training and validation. [3 pts]

- **Correct and Incorrect Predictions**



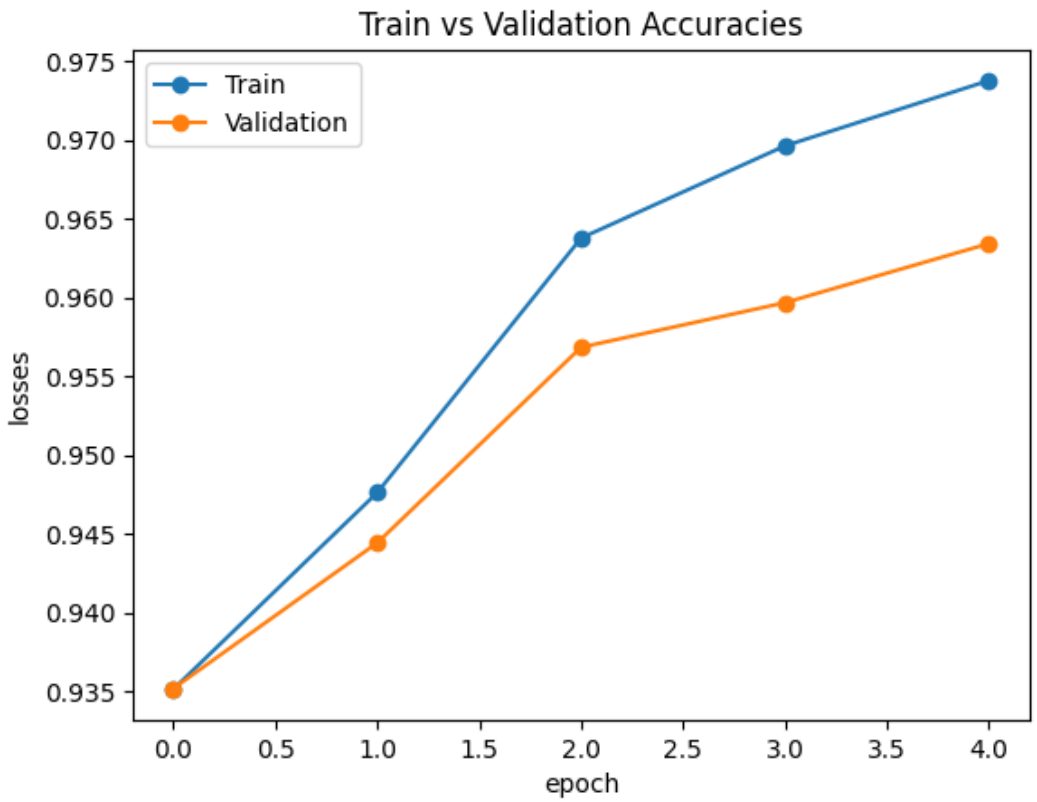
Correct Predictions



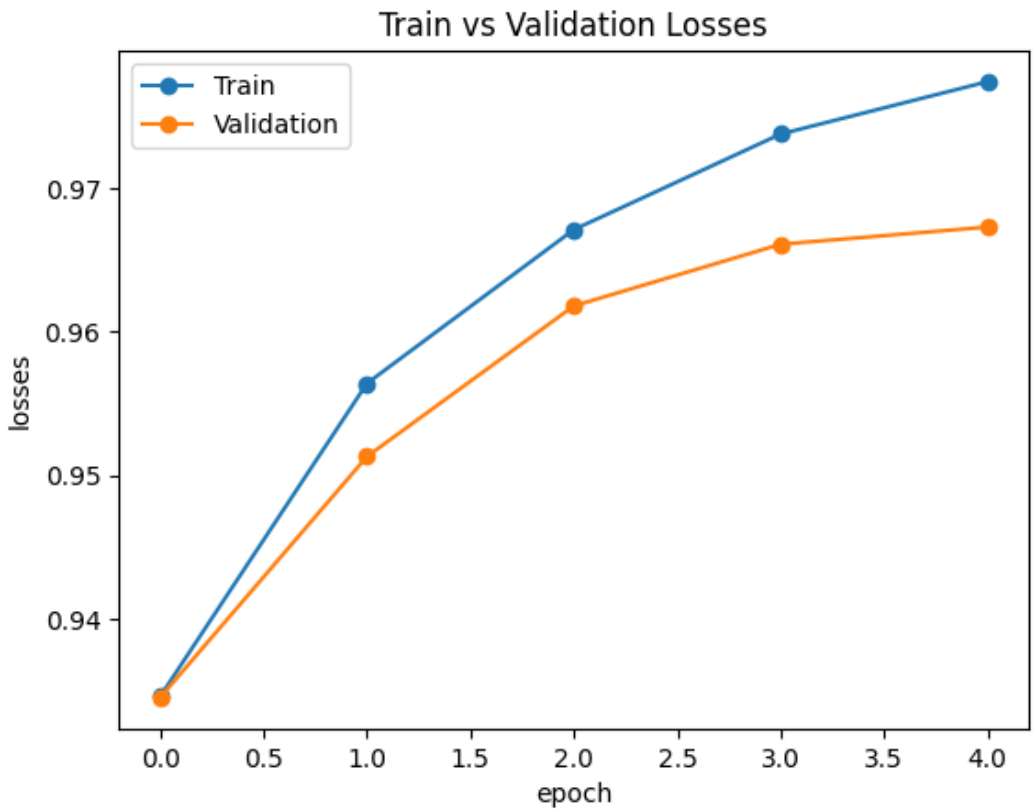
Incorrect Predictions

- **We can see that there's room for improvement in the model's performance by either training it for longer epochs or by allotting the whole data for training, or by efficient hyperparameter tuning. Whereas, some samples seem genuinely ambiguous even to the human eye and can possibly be ignored.**

- **Visualized Training Logs for both the train and validation dataset.**



Accuracies for each Epoch.



Losses for each Epoch.

---

x —THE END— x