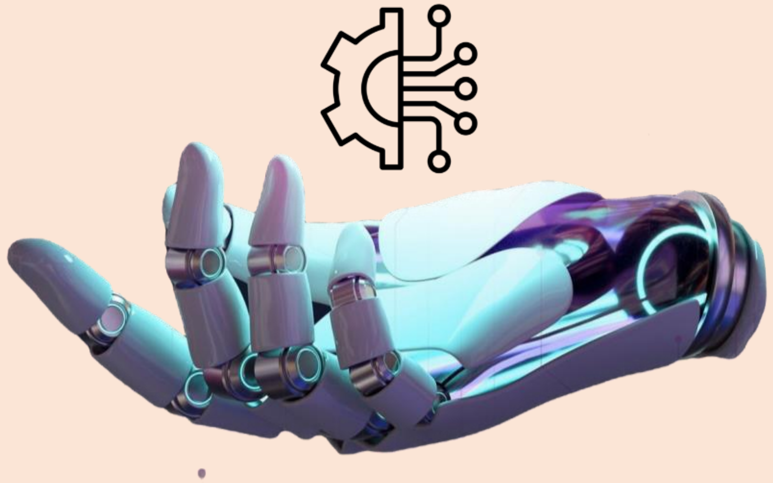
Artificial Intelligence Internship

Handwritten Digit Recognition Using LeNet-5 Model in PyTorch

Exploring LeNet-5 for accurate digit classification on the MNIST dataset.

By J. Meghana
meghasetty1@gmail.com



Introduction

Handwriting recognition is the task of interpreting handwritten characters or digits.

"In this project, we use the MNIST dataset, a well-known benchmark for handwritten digit recognition.“

Our goal is to compare the performance of three models: MLP (Multilayer Perceptron), CNN (Convolutional Neural Networks), and LeNet-5 and choose the best ones out of those.

Handwritten Digit Recognition Problem

Handwritten digit recognition is the task of identifying digits (0-9) in images of handwritten characters. The goal is to develop systems capable of interpreting various handwriting styles, which can be challenging due to the variability in penmanship.

Handwriting varies across individuals, making it difficult for models to generalize across different styles. Recognition systems need to account for factors like slant, size, and spacing of digits.

Automated Postal Systems

Banking

Form Processing

OCR (Optical Character Recognition)

Tools and Dataset

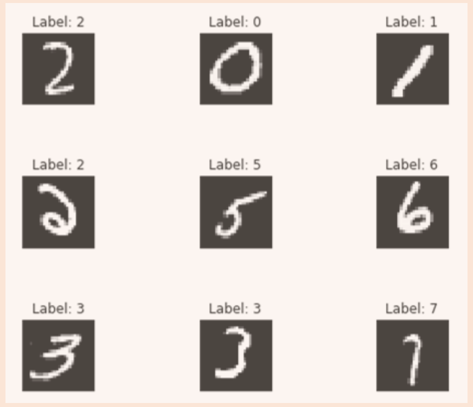
PyTorch

An open-source deep learning framework offering flexibility for building efficient machine learning models. Widely used for its dynamic computational graph and ease of debugging.

Jupyter Notebook

An interactive environment for developing and testing Python code. Great for visualizing data and iteratively building models.

MNIST Dataset



The dataset size (70,000 images).
Distribution (60,000 for training and 10,000 for testing).
Image characteristics (grayscale, 28x28 pixels, digits 0-9)
The MNIST dataset is widely used for benchmarking models in image classification tasks.

Loading the dataset

The MNIST dataset is loaded seamlessly using PyTorch's datasets. MNIST module, which allows easy access for training and testing.

Data preprocessing

Data preprocessing is a crucial step to prepare raw data for model training and improve performance. It includes transforming images into a suitable format and applying augmentations to enhance generalization.

Resize - All images were resized to 28x28 pixels for uniform input dimensions.

Normalization - Pixel values were normalized to the range [-1, 1] for better convergence during training.

Data Augmentation:

Random Rotation - Images were rotated randomly by up to 10 degrees to make the model robust to rotation.
Random Crop - Random cropping ensures diverse image perspectives within the same size."
Color Jitter - Brightness, contrast, saturation, and hue adjustments simulate real-world lighting conditions."
Gaussian Blur - Blurred images help the model focus on essential features.
Random Translation - Shifting images by 10% of their dimensions improves spatial robustness.
Padding - Added 2-pixel padding increases image size to 34x34, allowing cropping for variability.

Before

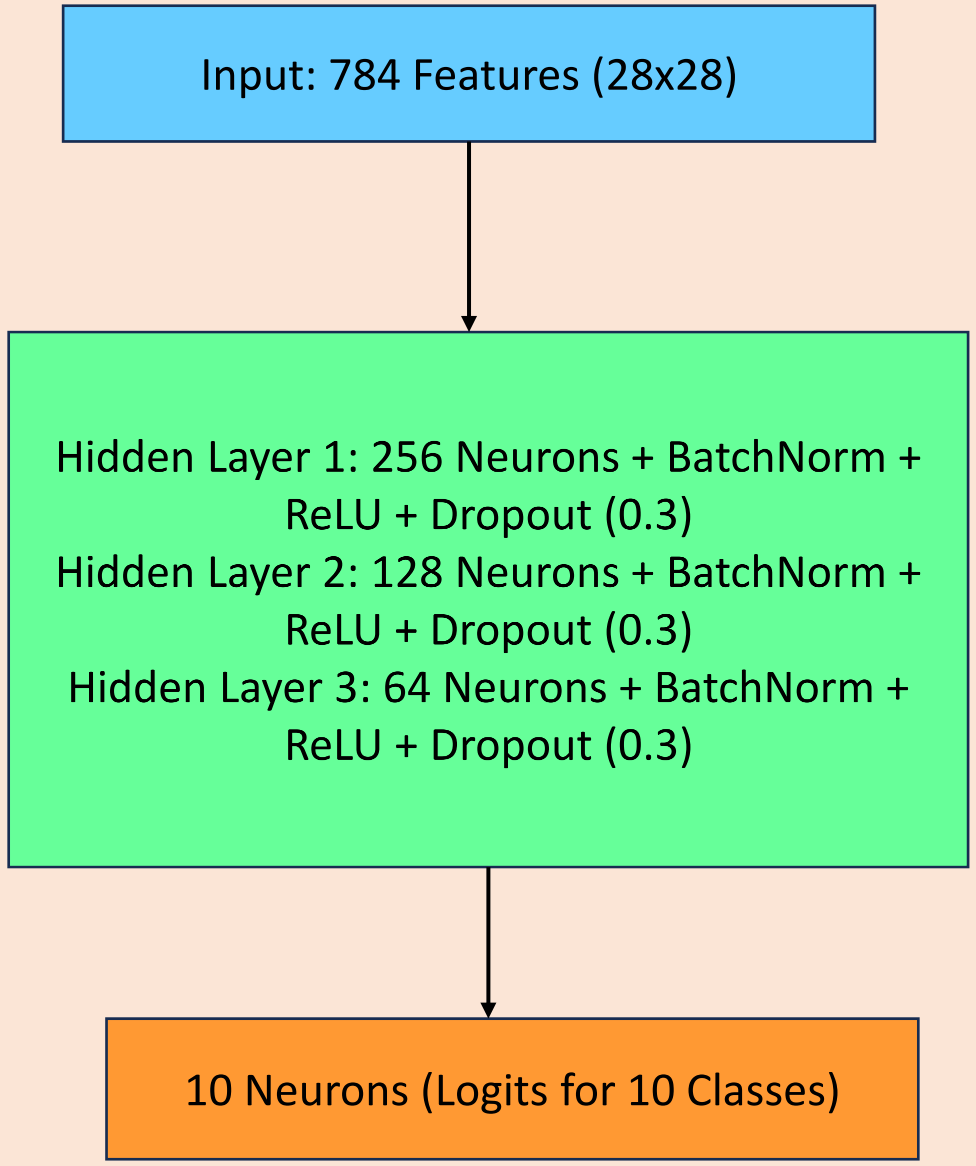
After

Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) is a class of feedforward neural networks that consists of multiple layers of fully connected neurons. It is commonly used for supervised learning tasks such as classification and regression.

Input, hidden, and output layers.
Non-linear activation functions, like ReLU, allow learning complex patterns.
Optimized through backpropagation using gradient descent or advanced optimizers like Adam.

The MLP was used to classify MNIST digits, converting 28x28 pixel images into class probabilities for 10 digits.


Input: 784 Features (28x28)
Hidden Layer 1: 256 Neurons + BatchNorm + ReLU + Dropout (0.3)
Hidden Layer 2: 128 Neurons + BatchNorm + ReLU + Dropout (0.3)
Hidden Layer 3: 64 Neurons + BatchNorm + ReLU + Dropout (0.3)
10 Neurons (Logits for 10 Classes)

Hyperparameters and Results

Hidden Layers
3 hidden layers with the following no.of neurons
Layer 1: 256 neurons
Layer 2: 128 neurons
Layer 3: 64 neurons

Activation Function
ReLU for all hidden layers.

Dropout
0.3 dropout applied after each hidden layer to reduce overfitting.

Output Layer
10 neurons, with **CrossEntropyLoss** implicitly applying softmax for classification.

Optimizer
Adam optimizer with learning rate = **0.001**.

Batch Size
64 (for both training and testing).

Epochs
10 epochs for training.

Loss Function
CrossEntropyLoss (for multi-class classification tasks).

RESULTS

Training Accuracy
Epoch 1: 90.88%
Epoch 2: 95.11%
Epoch 3: 96.01%
…
Epoch 10: 97.75%

Test Accuracy
98.10%

Precision
0.98

Recall
0.98

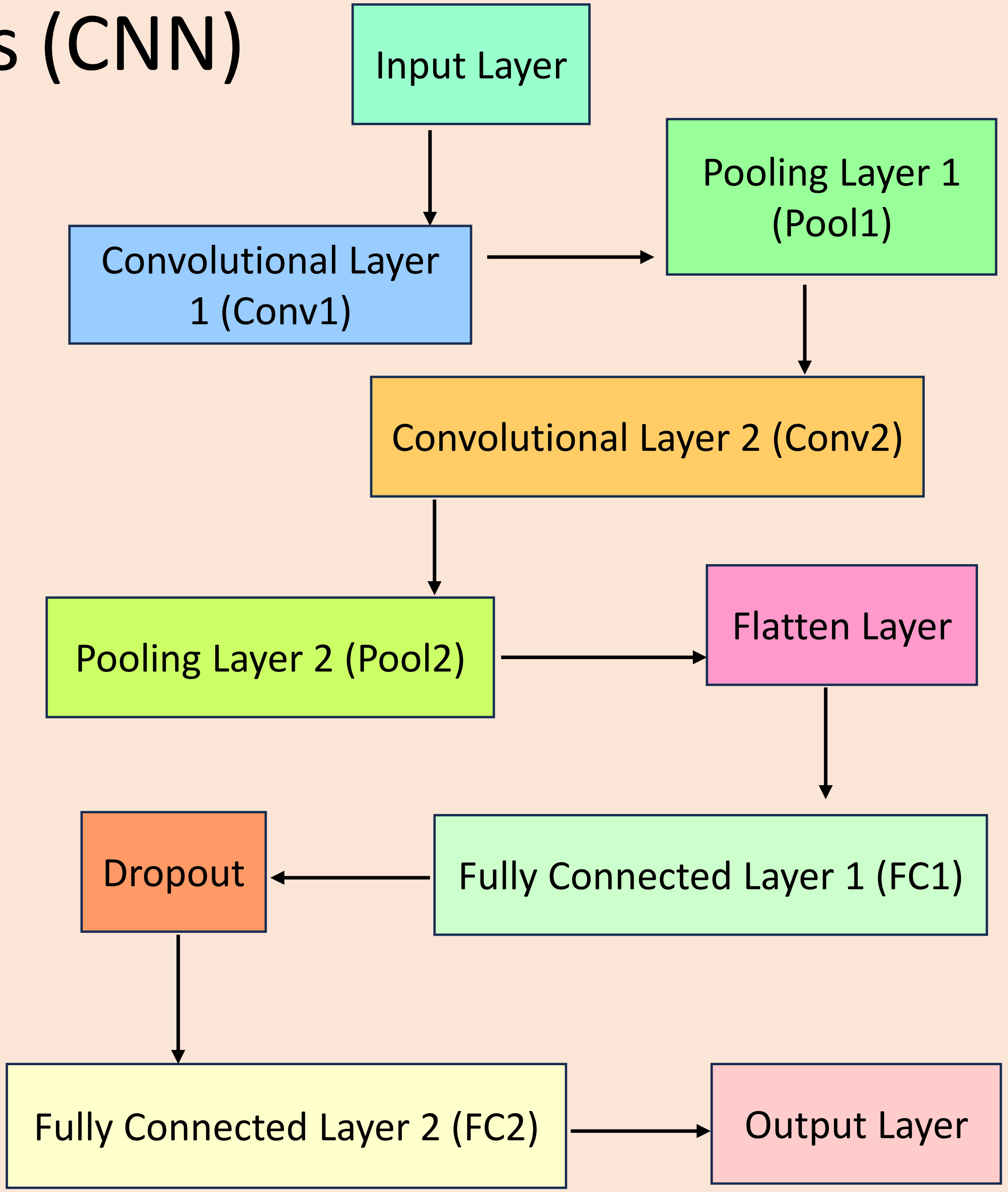
F1 Score
0.98

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are specialized deep learning models designed to process grid-like data such as images. They utilize layers to automatically and adaptively learn spatial hierarchies of features from data.

Convolutional Layers extract features like edges, textures, and patterns.
Pooling Layers reduce spatial dimensions, making computation efficient.
Fully Connected Layers combine extracted features to make predictions.

They capture local patterns (via convolution).
Reduce overfitting and computational complexity (via pooling and dropout).


Input Layer
Convolutional Layer 1 (Conv1)
Pooling Layer 1 (Pool1)
Convolutional Layer 2 (Conv2)
Pooling Layer 2 (Pool2)
Flatten Layer
Fully Connected Layer 1 (FC1)
Dropout
Fully Connected Layer 2 (FC2)
Output Layer

Hyperparameters and Results

Convolutional Layers
Layer 1: 32 filters, 3x3 kernel
Layer 2: 64 filters, 3x3 kernel
MaxPool: 2x2 after each convolutional layer

Activation Function
ReLU non-linearity.

Dropout
0.2 dropout applied prevent overfitting by randomly dropping units

Fully Connected Layer1
Layer 1:128 neurons
Output Layer: 10 neurons

Optimizer
Adam optimizer with learning rate = **0.001**.

RESULTS

Batch Size
100 (for both training and testing).

Epochs
10 epochs for training.

Loss Function
CrossEntropyLoss (for multi-class classification tasks).

Training Accuracy
Epoch 1: 94.05%
Epoch 2: 98.33%
Epoch 3: 98.67%
…
Epoch 10: 98.62%

Test Accuracy
98.24%

Precision
0.99

Recall
0.99

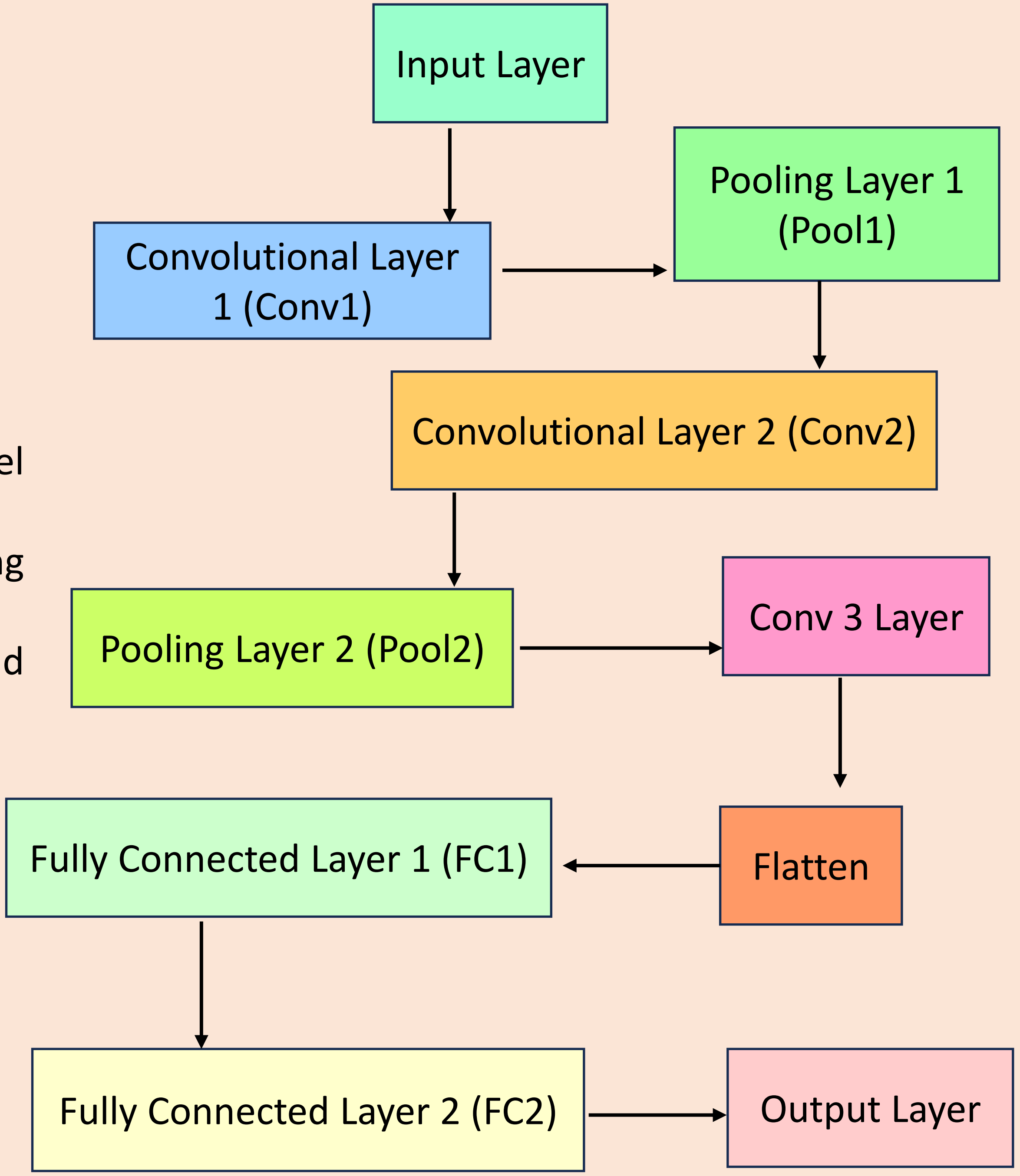
F1 Score
0.99

LeNet - 5

LeNet-5 is a pioneering Convolutional Neural Network (CNN) architecture designed by Yann LeCun for handwritten digit recognition. It consists of a series of layers that automatically learn and extract hierarchical features from input images.

Convolutional Layers detect low-level and high-level features such as edges and complex patterns.
Pooling Layers reduce the spatial dimensions, improving computational efficiency and robustness.
Fully Connected Layers combine extracted features and generate predictions for classification.

LeNet-5 is efficient in learning hierarchical features and performing image classification tasks, laying the foundation for modern CNN architectures.


Input Layer
Convolutional Layer 1 (Conv1)
Pooling Layer 1 (Pool1)
Convolutional Layer 2 (Conv2)
Pooling Layer 2 (Pool2)
Conv 3 Layer
Flatten
Fully Connected Layer 1 (FC1)
Fully Connected Layer 2 (FC2)
Output Layer

Hyperparameters and Results

Convolution Layers
Conv1: 6 output channels
Conv2: 16 output channels
Conv3: 120 output channels
Fully Connected Layers FC1 (256 Neurons), FC2 (128 neurons), FFC3 (64 neurons)

Activation Function
ReLU for all hidden layers.

Momentum
0.9 momentum helps accelerate gradient vectors.

Output Layer
10 neurons, with **CrossEntropyLoss** implicitly applying softmax for classification.

Optimizer
SGD optimizer with learning rate **0.01**

RESULTS

Batch Size
100 (for both training and testing).

Epochs
10 epochs for training.

Loss Function
CrossEntropyLoss (for multi-class classification tasks).

Training Accuracy
Epoch 1: 76.23%
Epoch 2: 95.89%
Epoch 3: 97.33%
…
Epoch 10: 99.13%

Test Accuracy
98.55%

Precision
0.99

Recall
0.99

F1 Score
0.99

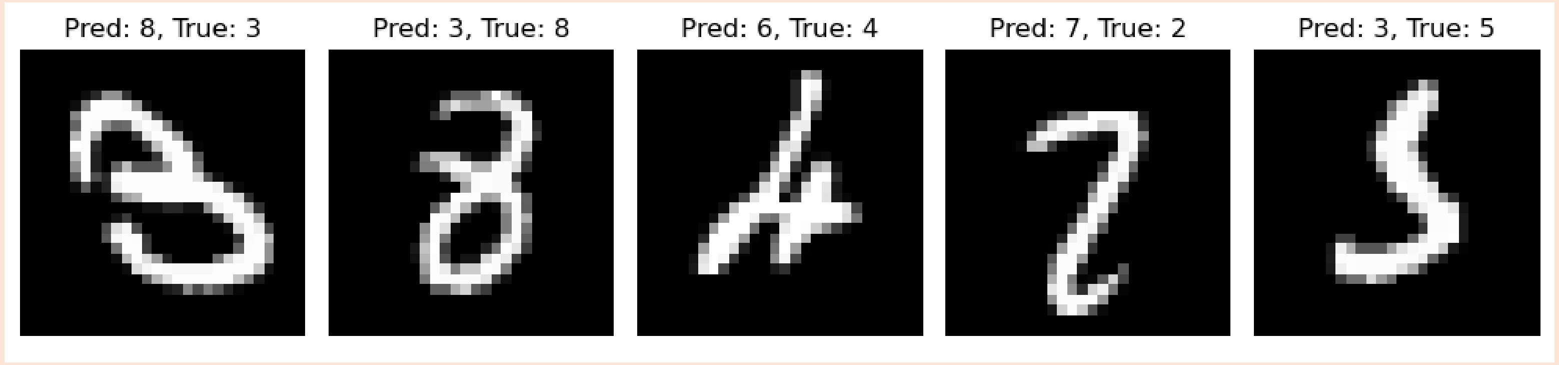
Comparison of MLP, CNN, and LeNet-5

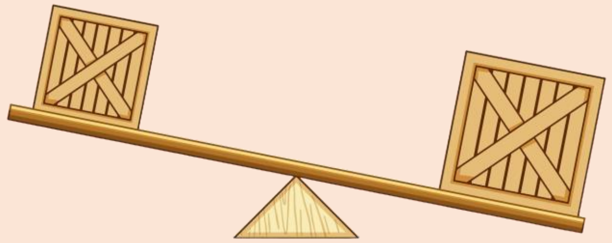
LeNet-5 outperformed both MLP and CNN in terms of accuracy.
MLP struggled with image data due to its fully connected layers.
CNN provided better results than MLP but was outperformed by LeNet-5, optimized for image data.

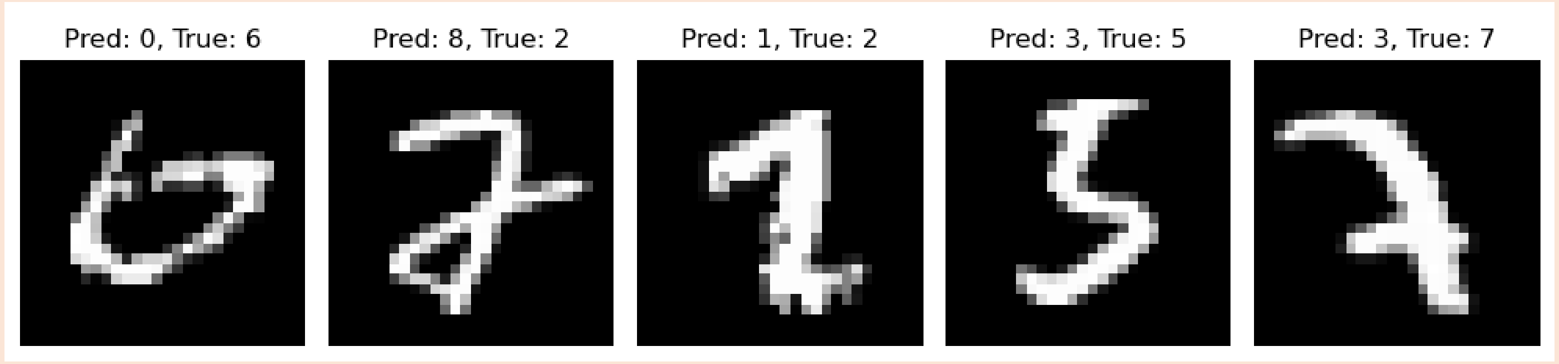
MODEL	ACCURACY	PRECISION	RECALL	F1 - SCORE	Training Loss	Epochs to Convergence
LeNet-5	98.55%	0.99	0.99	0.99	0.0118	10
CNN	98.24%	0.99	0.99	0.99	0.0269	10
MLP	98.10%	0.98	0.98	0.98	0.0699	10

Undetected Images and Model Analysis

Some digits were misclassified by the models, especially for ambiguous or poorly written samples.
Analysis shows that LeNet-5 still handles such cases better than MLP and CNN.
The misclassifications highlight the challenges faced by deep learning models when working with noisy or inconsistent data.
While all models struggle with ambiguous samples, LeNet-5 demonstrated fewer misclassifications compared to MLP and CNN models.

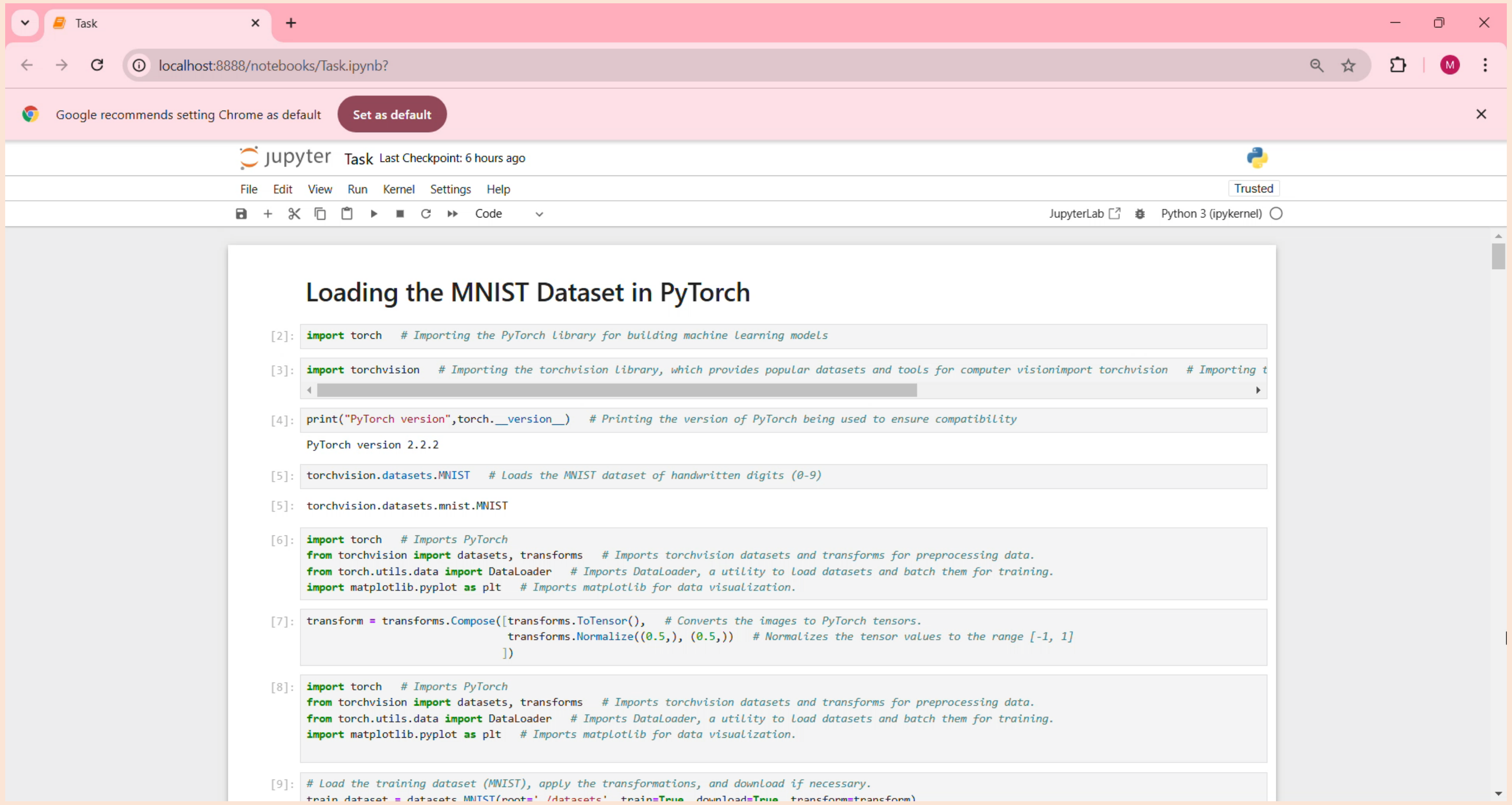






Working with Unseen Data

To evaluate the generalization ability of LeNet-5, we tested the model on previously unseen data samples. The model demonstrated strong performance, correctly classifying the majority of unseen samples. These results highlight the robustness of LeNet-5 in adapting to new, untrained data inputs, which is critical for real-world applications.


Task
localhost:8888/notebooks/Task.ipynb?
Google recommends setting Chrome as default
Set as default
jupyter Task Last Checkpoint: 6 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

Loading the MNIST Dataset in PyTorch

```
[2]: import torch   # Importing the PyTorch library for building machine learning models

[3]: import torchvision   # Importing the torchvision library, which provides popular datasets and tools for computer visionimport torchvision   # Importing t

[4]: print("PyTorch version",torch.__version__)   # Printing the version of PyTorch being used to ensure compatibility

     PyTorch version 2.2.2

[5]: torchvision.datasets.MNIST   # Loads the MNIST dataset of handwritten digits (0-9)

[5]: torchvision.datasets.mnist.MNIST

[6]: import torch   # Imports PyTorch
     from torchvision import datasets, transforms   # Imports torchvision datasets and transforms for preprocessing data.
     from torch.utils.data import DataLoader   # Imports DataLoader, a utility to load datasets and batch them for training.
     import matplotlib.pyplot as plt   # Imports matplotlib for data visualization.

[7]: transform = transforms.Compose([transforms.ToTensor(),   # Converts the images to PyTorch tensors.
                                     transforms.Normalize((0.5,), (0.5,))   # Normalizes the tensor values to the range [-1, 1]
                                    ])

[8]: import torch   # Imports PyTorch
     from torchvision import datasets, transforms   # Imports torchvision datasets and transforms for preprocessing data.
     from torch.utils.data import DataLoader   # Imports DataLoader, a utility to load datasets and batch them for training.
     import matplotlib.pyplot as plt   # Imports matplotlib for data visualization.

[9]: # Load the training dataset (MNIST), apply the transformations, and download if necessary.
     train_dataset = datasets.MNIST(root='./datasets', train=True, download=True, transform=transform)
```

Any Questions Please?

THANK YOU