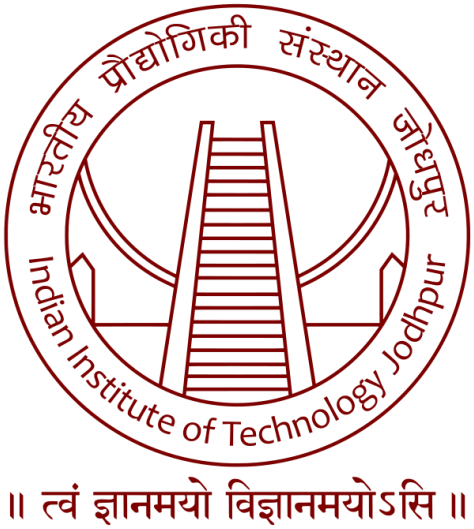
Assignment 1

Deep Learning



Submitted By

Ratnesh Kumar Tiwari
(M23MAC011)

To

Dr. Deepak MIshra

Kindly visit through this code for this assignment:
https://colab.research.google.com/drive/1BwRLNYdIfU2_r2nwyjLB2238pGuJX0wy?usp=sharing

Abstract:

This assignment outlines the implementation of a Convolutional Neural Network (CNN) for the task of handwritten digit recognition using the MNIST dataset. This whole assignment involves data preprocessing, model architecture design, training, and evaluation.

1. Introduction

a. Problem Definition:

Handwritten digit recognition is a classic problem in deep learning, with applications in digitizing postal codes, recognizing bank checks, and more. The goal of this assignment is to develop a CNN-based model that can accurately classify handwritten digits from the MNIST dataset.

b. Objectives:

Design and implement a CNN architecture for digit recognition.
Train the model on the MNIST dataset.
Evaluate the model's performance on a validation set.
Analyze the results and discuss potential improvements.

2. Methodology

a. Data Loading and Preprocessing:

The MNIST dataset, consisting of images of handwritten digits, was loaded and preprocessed. The images were loaded with the help of idxtonumpy package. This package provides us tools to convert files from idx format to numpy array format. Then the pixel values were scaled to the range [0, 1]. This process is called normalization. This is done to make our model less complex.
Then the training dataset was split into training and validation sets in the ratio of **90:10.**

b. Model Architecture

Our model consists of three hidden layers and one fully connected layer. The first layer uses 16 filters, each of size 7X7. In the same way, the second and third layers also use 8 and 4 filters of size 5X5 and 3X3 respectively. All of the three hidden layers use ReLU (Rectified Linear Unit) as activation function to ensure non linearity. There are more activation functions that can be used, for example Leaky ReLU. The output layer uses sigmoid as activation function.

To ensure the complexity of CNN architecture is not high, extra pooling layers are introduced in each hidden layer. The first two layers use max pooling while the last layer uses average pooling.
To handle the corner pixels in the convolutional operation, zero padding is used.
Since it is a classification problem, cross entropy is used as the loss function.

According to the assignment, our model has been implemented in two ways.
The first one has 10 neurons in the output layer for 10 class classification while the second one has 4 neurons in the output layer for four class classification(i.e. combined classes).

Now let us see the calculation of input and output dimension at each layer.

Convolution Layer 1:
Input Dimensions = 28 x 28 x 1 (Single channel grayscale image)
Kernel Size = 7 x 7 x 1
Number of filters = 16 (because output channel is given as 16)
Stride = 1
Padding Size = Floor(Kernel Size/2)
Therefore Padding Size = 3 x 3
Output Size = [(Input Size + 2 x Padding Size - Filter Size)/ Stride] + 1
Therefore, Output Size = 28 x 28 x 16

Now this will be passed to ReLU, it doesn't change the dimension
Now this will be passed to the pooling layer. Now let us calculate the dimensions after the pooling layer.

(**NOTE:** In polling layer, we have selected the size of kernel and stride arbitrarily as per the assignment)

Input Dimension = 28 x 28 x 16
Kernel Size = 2 x 2
Stride = 2
Therefore Output Dimension will be 14 x 14 x 16

Now, Convolution Layer 2:
Input Dimensions = 14 x 14 x 16
Kernel Size = 5 x 5 x 16
Number of filters = 8 (because output channel is given as 8)
Stride = 1
Padding Size = Floor(Kernel Size/2)
Therefore Padding Size = 2 x 2
Output Size = [(Input Size + 2 x Padding Size - Filter Size)/ Stride] + 1

Therefore, Output Size = 14 x 14 x 8

Now this will be passed to ReLU, it doesn't change the dimension
Now this will be passed to the pooling layer. Now let us calculate the dimensions after the pooling layer.

Input Dimension = 14 x 14 x 8
Kernel Size = 2 x 2
Stride = 2
Therefore Output Dimension will be 7 x 7 x 8

Now, Convolution Layer 3:
Input Dimensions = 7 x 7 x 8
Kernel Size = 3 x 3 x 8
Number of filters = 4 (because output channel is given as 4)
Stride = 2
Padding Size = Floor(Kernel Size/2)
Therefore Padding Size = 1 x 1
Output Size = [(Input Size + 2 x Padding Size - Filter Size)/ Stride] + 1
Therefore, Output Size = 4 x 4 x 4

Now this will be passed to ReLU, it doesn't change the dimension
Now this will be passed to the pooling layer. Now let us calculate the dimensions after the pooling layer.

Input Dimension = 4 x 4 x 4
Kernel Size = 2 x 2
Stride = 2
Therefore Output Dimension will be 2 x 2 x 4

Now this volume will be flattened for the fully connected output layer.

c. Training:

The model was trained using the Adam optimizer and cross-entropy loss. During training, the average loss per epoch was recorded, and the model's performance was evaluated on the validation set. Training parameters included a learning rate of 0.001 and a batch size of 20.

Adam Optimizer: The Adam optimizer, short for Adaptive Moment Estimation, was chosen as the optimization algorithm for training the convolutional neural network. Adam combines the advantages of two other popular optimizers: RMSprop and momentum.

Cross entropy loss: Cross-entropy loss is a widely used loss function for classification tasks, including handwritten digit recognition. It measures the dissimilarity between the predicted probability distribution and the true distribution of class labels.

$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

Here L_{CE} is Cross Entropy Loss
t_i is the true label of i^{th} class.
p_i is the softmax probability of i^{th} class

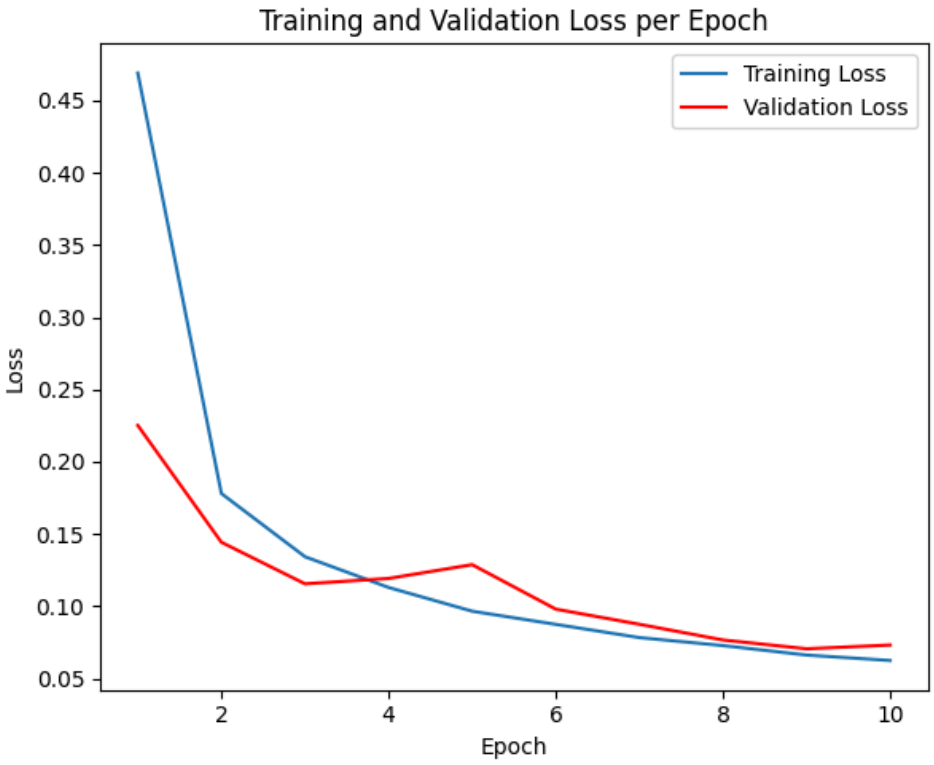
The combination of Adam optimizer and cross-entropy loss proved effective in training the model for handwritten digit recognition. The adaptability of Adam contributed to stable and efficient convergence, while cross-entropy loss provided a suitable measure for the model's performance on the classification task.

3. Results

a. Training Loss:

The training loss and validation loss is plotted against each epoch during the whole training process.

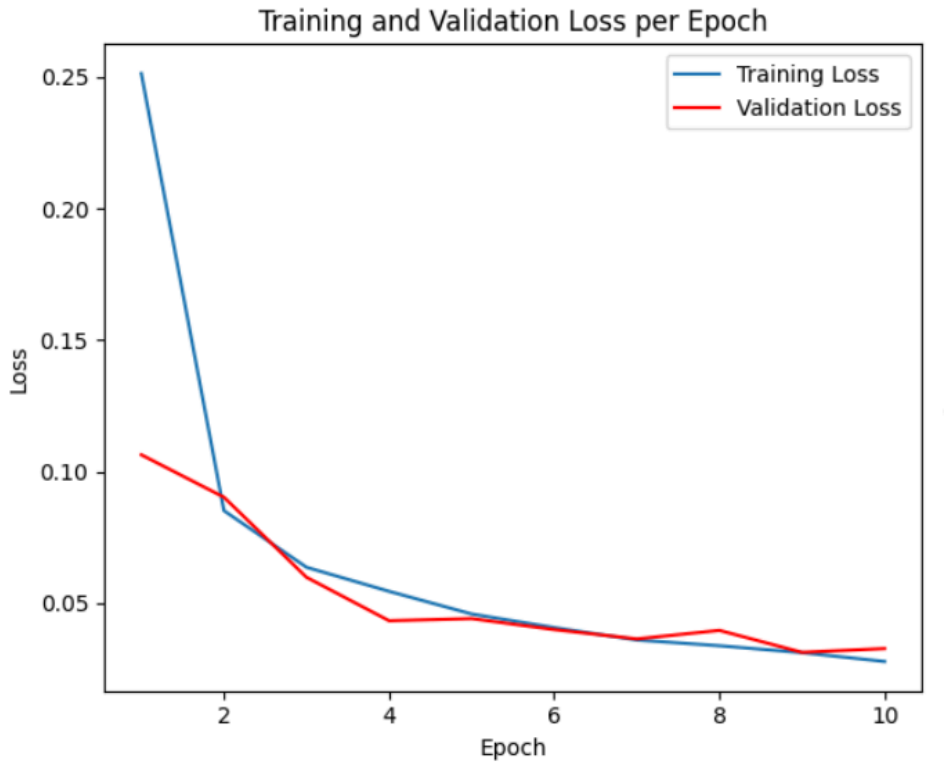
Experiment 1:



Here we can observe that the training and validation loss is being decreased throughout the training process. However we can see the shuttle difference in both the curve at epoch 1. This is because the validation loss calculated at epoch 1 is from randomly initialized weights. While the validation loss is calculated when the model is trained after the first epoch.
However, we can also notice that both curves started to diverge after epoch 9, which may potentially indicate overfitting.

After training for 10 epochs the training loss came out to be **0.0624** and validation loss came out to be **0.0731**.

Experiment 2:



Here also we can observe that the training and validation loss is being decreased throughout the training process.

However, we can also notice that both curves started to diverge after epoch 9, which may potentially indicate overfitting.

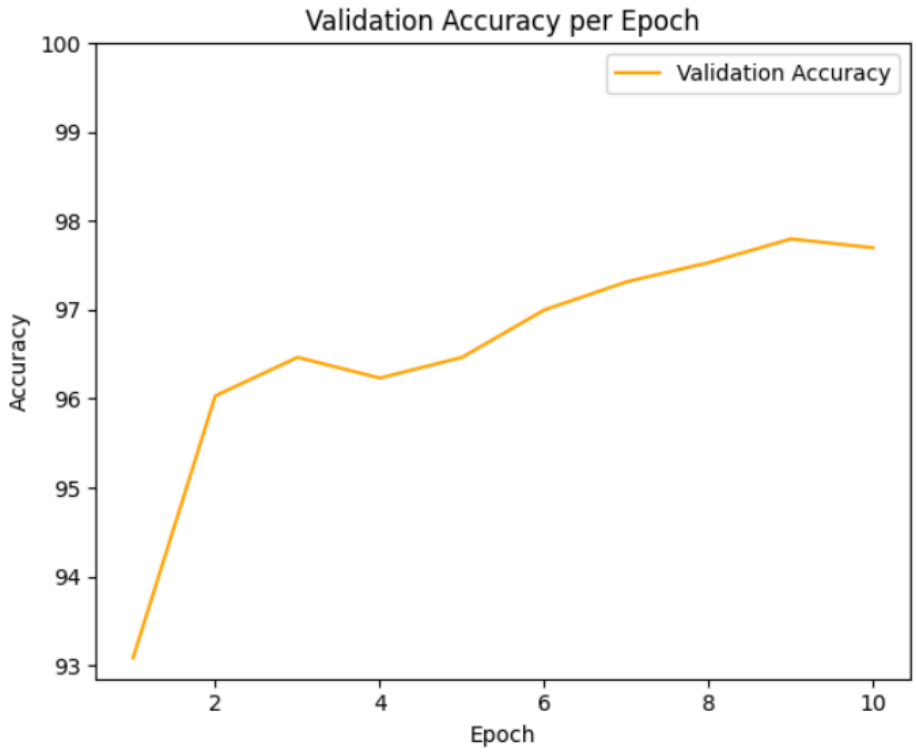
After training for 10 epochs the training loss came out to be **0.0278** and validation loss came out to be **0.0327**.

b. Validation Accuracy:

While the loss trend is a valuable indicator of convergence, evaluating the model's performance on a separate validation set is crucial for assessing its generalization capabilities. The validation accuracy, calculated by comparing the model's predictions to the true labels in the validation set, offers a comprehensive measure of its effectiveness.

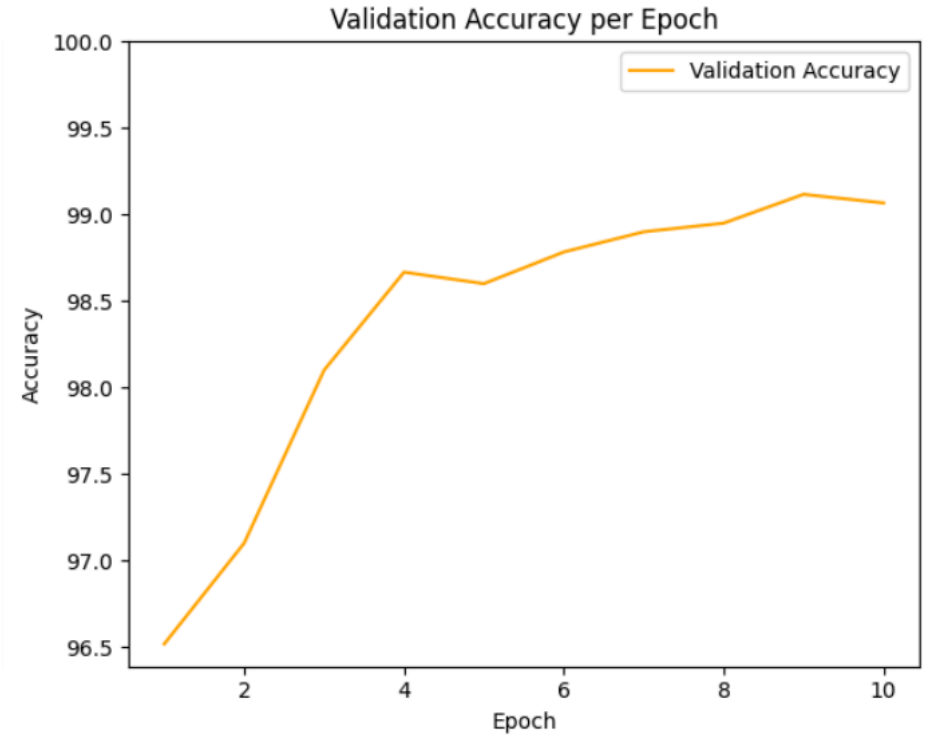
The validation accuracy is plotted against each epoch for the both experiment.

Experiment 1:



Here we can see out model's validation accuracy starts from **93.08%** and after 10 epochs it reached to **97.70**%

Experiment 2:



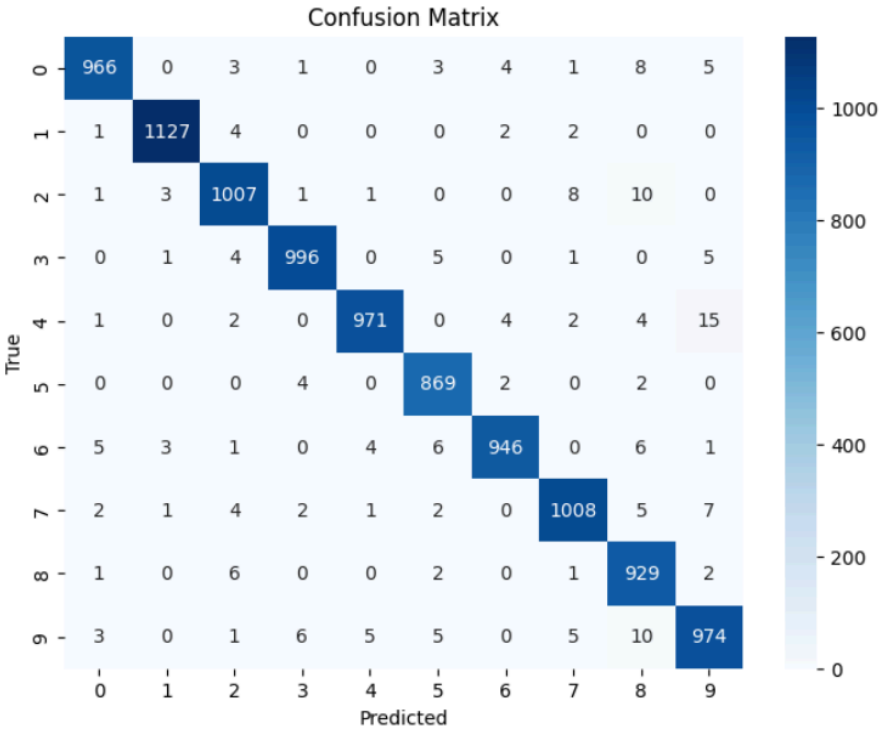
Here we can see out model's validation accuracy starts from **96.52%** and after 10 epochs it reached to **99.00%**

c. Confusion Matrix:

A confusion matrix is a table that summarizes the classification results, comparing the predicted labels with the true labels. In the context of a multi-class classification problem like digit recognition, the confusion matrix is often a square matrix where each row corresponds to the true class, and each column corresponds to the predicted class.

Following is the confusion matrix drawn for each experiment when run on the test dataset.

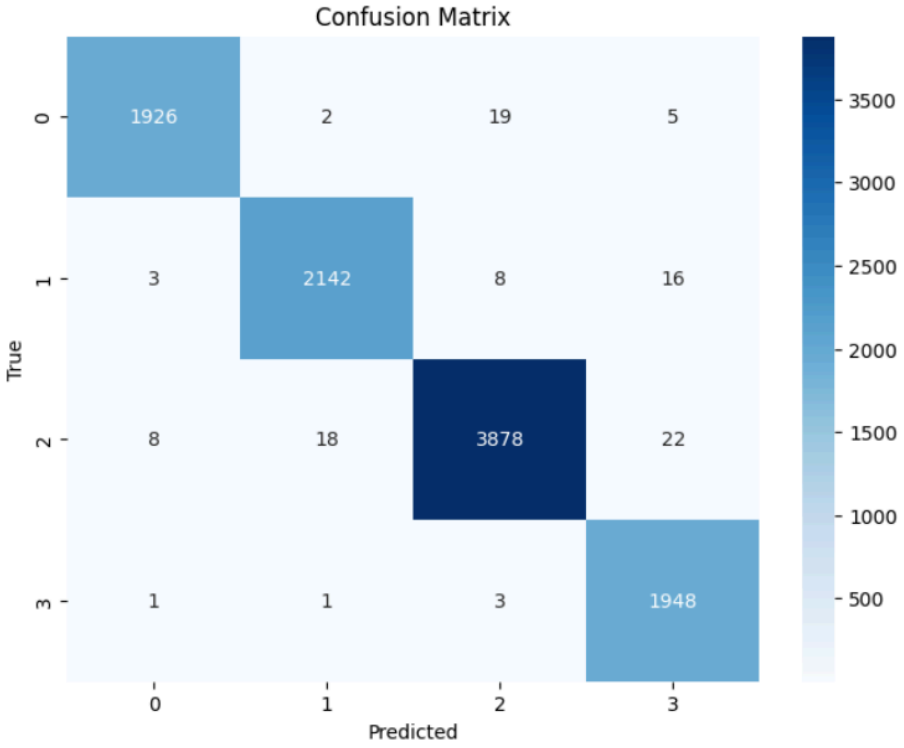
Experiment 1:



Here the diagonal element has very higher values compared to the other elements. It suggests that most of them are correctly classified.

Also the test data accuracy came as **97.93%.**

Experiment 2:



Here also,the diagonal element has very higher values compared to the other elements. It suggests that most of them are correctly classified.

Also the test data accuracy came as **98.94%.**

d. Number of Trainable and Non Trainable Parameters:

Experiment 1:

No. of Trainable Parameters:

CNN Layer 1: Filter Size = 7 x 7 x 1
And no. of filters = 16
Hence, Trainable Features = 7 x 7 x 1 x 16 + 16(bias) = 800

Similarly, for Layer 2, Trainable features = 5 x 5 x 16 x 8 + 8 =3208
For Layer 3, Trainable features = 3 x 3 x 8 x 4 + 4 = 292

And Finally, Output Layer,

Input Volume = 2 x 2 x 4
And Output Class = 10
Hence the no. of tarinable parameter in fully connected layer = 2 x 2 x 4 x 10 + 10 = 170

Hence total number of Trainable parameters = 800 + 3208 + 292 + 170 = **4470**.

No of non- trainable parameters is equal to 0.

Experiment 2:
No. of Trainable Parameters:
CNN Layer 1: Filter Size = 7 x 7 x 1
And no. of filters = 16
Hence, Trainable Features = 7 x 7 x 1 x 16 + 16(bias) = 800

Similarly, for Layer 2, Trainable features = 5 x 5 x 16 x 8 + 8 =3208
For Layer 3, Trainable features = 3 x 3 x 8 x 4 + 4 = 292

And Finally, Output Layer,
Input Volume = 2 x 2 x 4
And Output Class = 4
Hence the no. of tarinable parameter in fully connected layer = 2 x 2 x 4 x 4 + 4 = 68

Hence total number of Trainable parameters = 800 + 3208 + 292 + 68 = **4368**.

No of non- trainable parameters is equal to 0.

4. Discussion

a. Model Performance

The model's performance was rigorously evaluated through the lens of both validation accuracy and the confusion matrix. These metrics provide a nuanced understanding of how well the CNN-based digit recognition model performs across different classes.

Loss Curve: Throughout the training process, the training loss and validation loss per epoch exhibited a consistent decreasing trend. This indicates that the model was learning to make more accurate predictions over time. The decreasing loss trend is a positive signal, suggesting that the optimization process was converging towards a solution.

The speed of convergence, as reflected in the decline of the loss, provides insights into the effectiveness of the chosen optimization algorithm (Adam) and

the model architecture. A rapid convergence suggests that the model quickly adapted to the underlying patterns in the training data.

Validation Accuracy: The validation accuracy in both the experiments shows an increasing trend with each epoch and then saturates down at a particular range. This suggests our model is learning with each epoch.

Confusion Matrix: The diagonal elements of the confusion matrix represent correct predictions (true positives and true negatives). A strong model will have high values along the diagonal. Our model has very high values in the diagonal element compared to the non diagonal elements when evaluated on the test dataset. This indicates that the model is performing well.

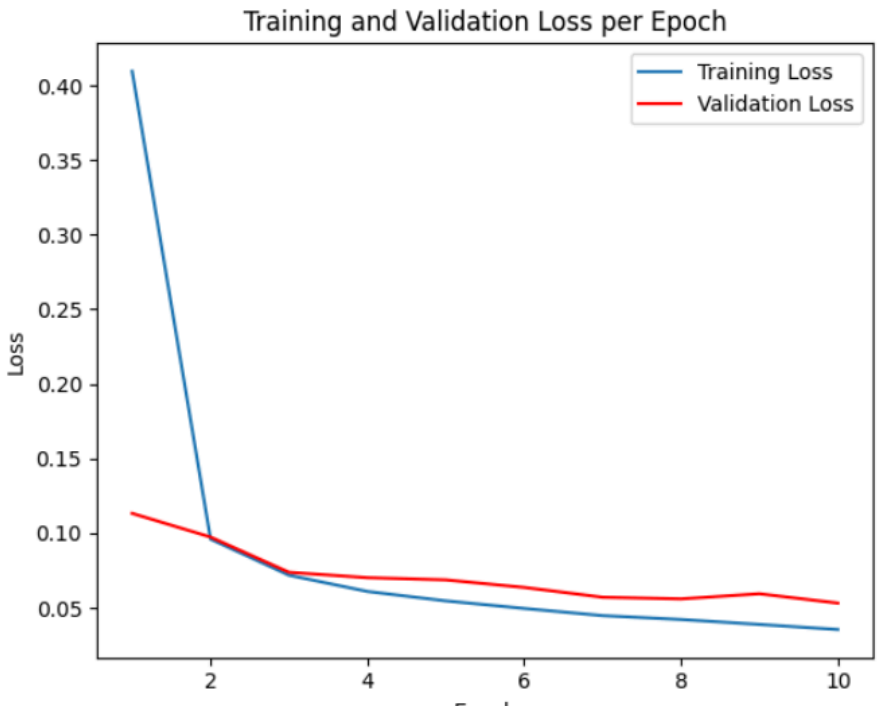
b. Possible Improvement:

Although our model is working well in this dataset, we can improve the model performance and avoid overfitting by using the technique called **Batch Normalization**. It is a technique that helps deep neural networks, like the one we used for digit recognition, train more effectively. It makes sure that the numbers going into each part of the network are at the right scale, which speeds up learning. In our setup, we used BatchNorm after each step of recognizing digits, making the model more stable and better at learning. It also acts like a coach, helping the model generalize well to new examples. Additionally, it deals with common problems in training, like vanishing and exploding gradient. Including BatchNorm turned out to be a smart move in making our model learn better.

Now let us look at the results obtained after incorporating the batch normalization.

Training and Validation Loss:

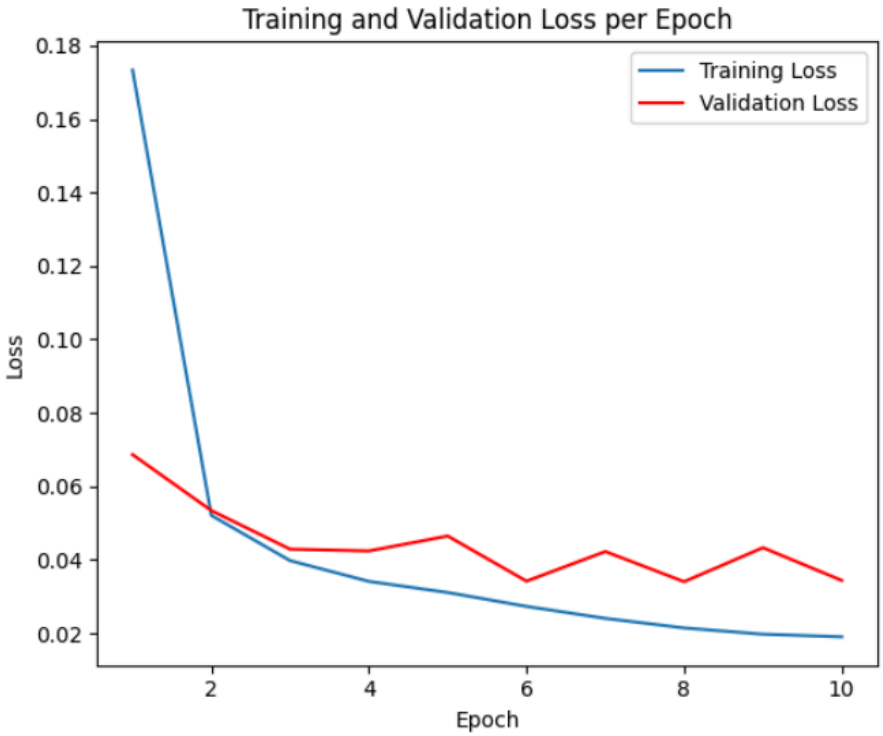
Experiment 1:



Here, we can also notice that both curves started to converge after epoch 9, which may potentially indicate less risk of overfitting.

After training for 10 epochs the training loss came out to be **0.0354** and validation loss came out to be **0.0532**, which is significantly lower than the previous model.

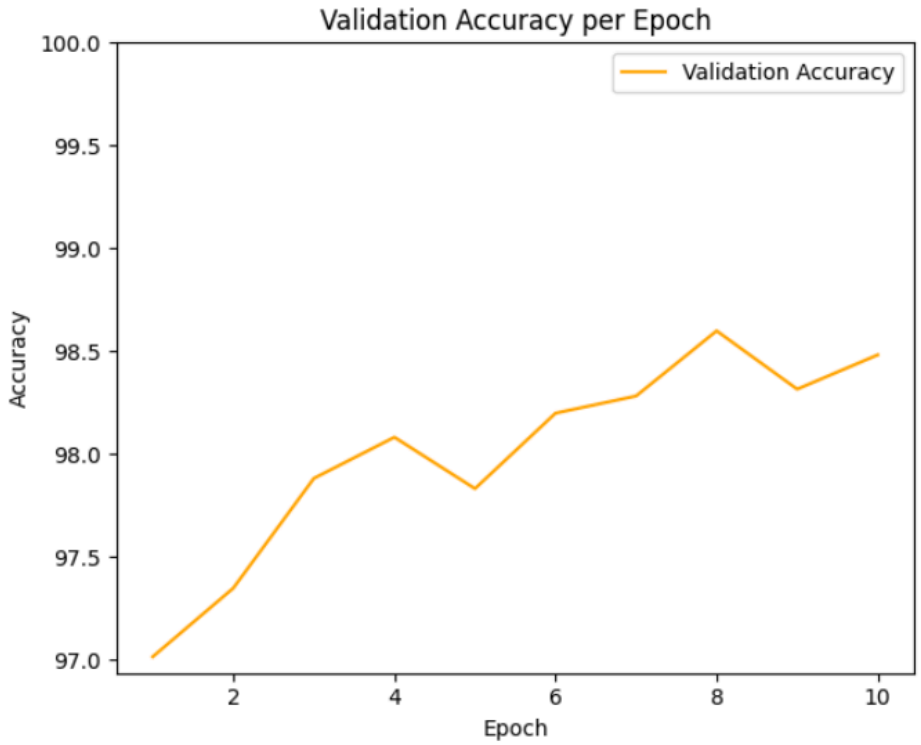
Experiment 2:



After training for 10 epochs the training loss came out to be **0.0191** and validation loss came out to be **0.0344**, which is significantly lower than the previous model.

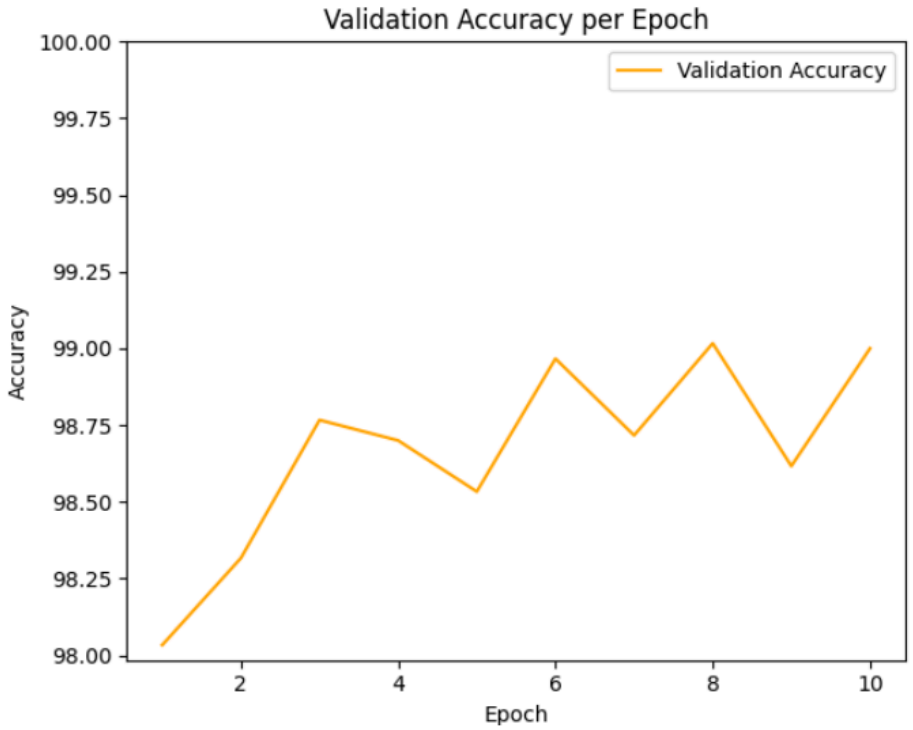
Accuracy:

Experiment 1:



Here accuracy started with **97.2%** and went up to **98.48 %** . It is a humongous difference from our previous model.

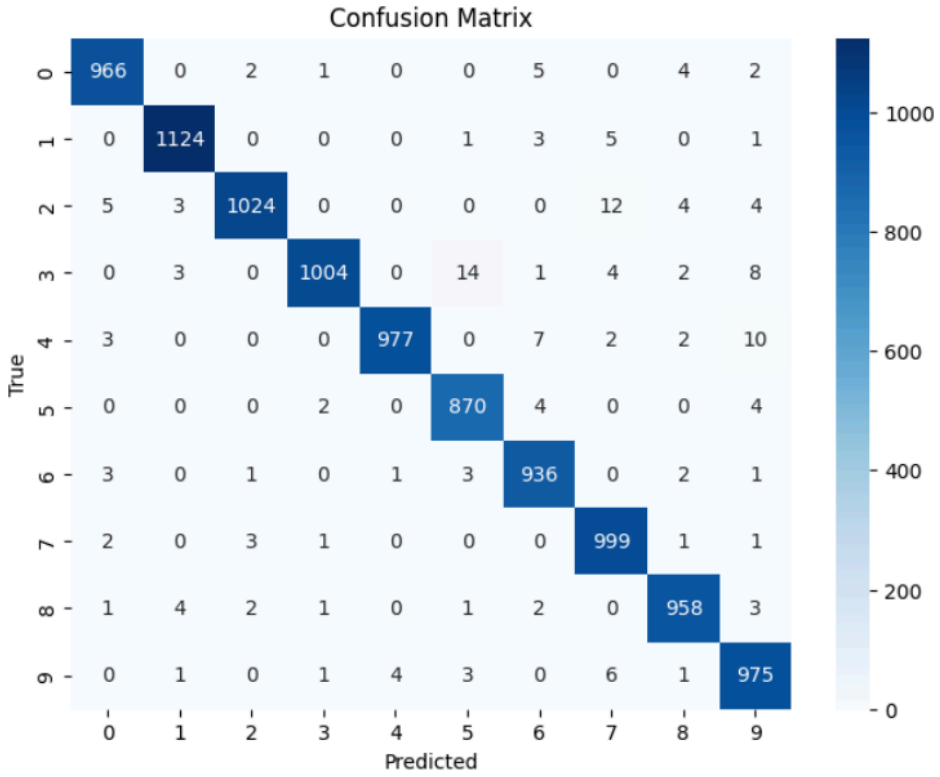
Experiment 2



Here accuracy started with **98.03%** and went up to **99.00%** . It is a humongous difference from our previous model.

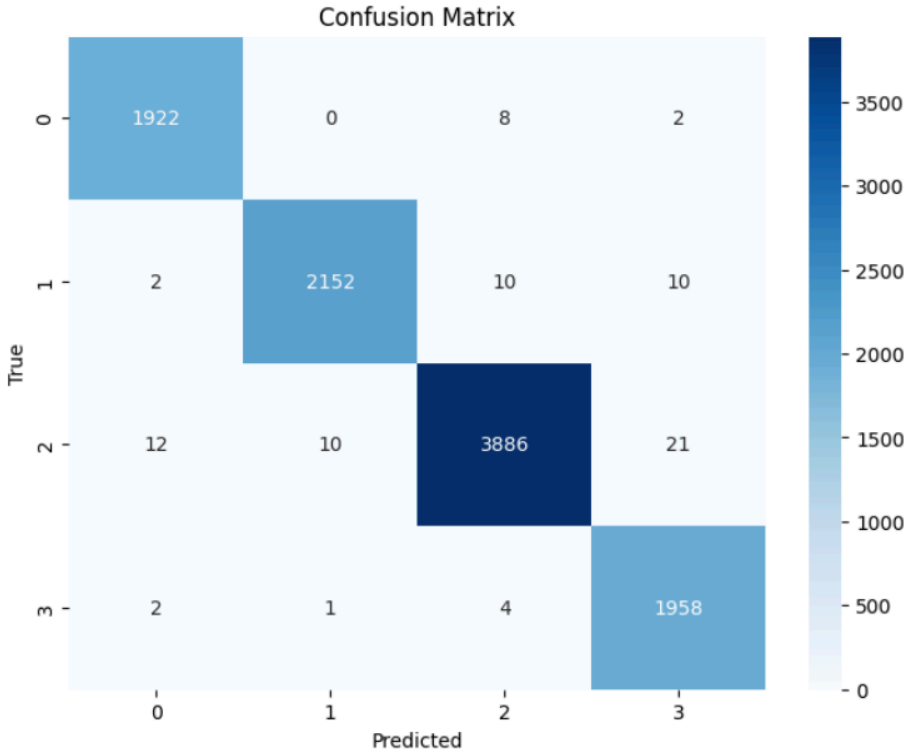
Confusion Matrix of Test Dataset:

Experiment 1:



Test accuracy is **98.33%**

Experiment 2:



Test accuracy is **99.18%**

5. Conclusion:

In this assignment, we have been on a journey of building a Convolutional Neural Network (CNN) for digit recognition using the MNIST dataset. Initially, the CNN was developed without the integration of Batch Normalization, and its performance was evaluated. The model demonstrated a reasonable capability to recognize digits, but there was room for improvement in terms of training stability and overfitting.

To address these challenges, we introduced Batch Normalization into the CNN architecture. **Batch Normalization** played a pivotal role in combating internal covariate shift, stabilizing the distribution of inputs during training, and facilitating faster convergence. The normalization of inputs enabled the use of higher learning rates, accelerating training without the risk of divergence.