BIHAR ENGINEERING UNIVERSITY, PATNA

(Under Govt. of Bihar)

Loknayak Jai Prakash Institute of Technology

CHAPRA - 841302

Session: 2021-2025

8th Semester

Major Project Report

On

"SEQUENCE DIGIT RECOGNITION USING DEEP LEARNING"

Submitted in Partial fulfilment of the requirement for the degree of

Bachelor of Technology

In

Computer Science & Engineering

Submitted by

Name	**Registration No.**
MD ASHER	21105117055
MD ABDULLAH	21105117011
FARIYA RAFAT	21105117057
RUPESH KUMAR	21105117023

Under the Supervision of

Prof. Sudhir Kumar Pandey

(Asst. Professor, Department of C.S.E)

DECLARATION BY THE STUDENTS

I the undersigned solemnly declare that the report of the project work entitles "Sequence Digit recognition using Deep Learning" is based on my own work carried out during the course of my study under the guidance of SUDHIR KUMAR PANDEY (Asst. Prof.) Department of computer Science & Engineering, Lok Nayak Jai Prakash Institute of Technology, Chapra.

I further declare the statements made and conclusion drawn are an outcome of my project work.

(Signature of Student)	(Signature of Student)
Md Asher	Md Abdullah
Reg. No. - 21105117055	Reg. No. – 21105117011
(Signature of Student)	(Signature of Student)
Fariya Rafat	Rupesh Kumar
Reg. No. - 21105117057	Reg. No. – 21105117023

CERTIFICATE BY GUIDE

This is to certify that the project entitled "Sequence Digit Recognition Using Deep Learning" is a record of work carried out by **Md Asher** - 21105117055, **Md Abdullah** - 21105117011, **Fariya Rafat** - 21105117057, **Rupesh Kumar** – 21105117023 being Under my guidance and supervision for the award of the Degree of Bachelor of Technology, Lok Nayak Jai Prakash Institute of Technology, Chapra, Bihar, India.

To the best of my knowledge and belief the Project

i. Embodies the work of the candidate him/herself.

ii. Has not been submitted for the award of any degree.

ii. Fulfils the requirement of the Ordinance relating to the B.TECH. degree of the University.

iv. Is up to the desired standard in respect of contents and is being referred to the examiners.

_____

(Signature of the Guide)

SUDHIR KUMAR PANDEY

(Asst. Prof. Dept of CSE)

(L.N.J.P.I.T., Chapra)

RECOMMENDATION

The Project work as mentioned is above here by being recommended and forwarded for examination and evaluation.

(Signature of the Guide)

SUDHIR KUMAR PANDEY

(Asst. Prof. Dept of CSE)

(L.N.J.P.I.T., Chapra)

CERTIFICATE BY THE EXAMINERS

This is to certify that the project work "Sequence Digit Recognition using Deep Learning" is a record of work carried out by **Md Asher** - 21105117055, **Md Abdullah** - 21105117011, **Fariya Rafat** - 21105117057, **Rupesh Kumar** – 21105117023 have been completed under the guidance of SUDHIR KUMAR PANDEY (Asst. Prof.) Department of Computer Science & Engineering, Lok Nayak Jai Prakash Institute of Technology, has been examined by the undersigned as a part of the examination for the award of Bachelor of Technology degree in Computer Science & Engineering branch in Bihar Engineering University, PATNA

" Project Examined and Approved "

_____ _____

Internal Examiner External Examiner

Date: Date:

ACKNOWLEDGEMENT

At every outset we express our gratitude to almighty lord for showering his grace and blessing upon us to complete this project.

Although our name appears on the cover of this project, many people had contributed in some form or the other form to this project development. We could not do this project without the assistance or support of each of the following we thank you all.

We wish to place on our record our deep sense of gratitude to our project guide and project in-charge SUDHIR KUMAR PANDEY (Asst. Prof.) Department of C.S.E, (L.N.J.P.I.T., Chapra) for their constant motivation and valuable help through the project work. For his valuable suggestion and advices throughout the course. We also extend our thanks to other faculties for their cooperation during our course.

Finally, we would like to thank our friends for their cooperation to complete this project.

(Signature of Student)
Md Asher
Reg. No. - 21105117055

(Signature of Student)
Md Abdullah
Reg. No. – 21105117011

(Signature of Student)
Fariya Rafat
Reg. No. - 21105117057

(Signature of Student)
Rupesh Kumar
Reg. No. – 21105117023

ABSTRACT

This project presents the design and implementation of a deep learning-based application for recognizing sequences of handwritten digits using a Convolutional Neural Network (CNN) trained on the MNIST dataset. The primary objective of the system is to enable real-time recognition of handwritten digit sequences, offering a practical solution for digitizing handwritten content found in various domains such as education, banking, postal services, and form processing. The application is divided into two interactive phases: the first allows recognition from uploaded images containing handwritten digits, and the second focuses on recognition from freehand drawings made directly on a digital canvas via a graphical user interface.

The core of the system involves a trained CNN model capable of classifying digits (0–9) with high accuracy. The model is supported by a comprehensive image preprocessing pipeline that includes grayscale conversion, thresholding, contour detection, dilation, and Gaussian blurring. These steps enhance the clarity of handwritten digits, separate individual digit components, and prepare them for consistent input formatting. Each extracted digit is resized to the standard 28×28 pixel format required by the MNIST-trained model, normalized, and then passed through the CNN for prediction.

To ensure accessibility and ease of use, the system is built with a Python-based graphical user interface (GUI) using the Tkinter library. The interface enables users to either draw digit sequences on a canvas or upload digit-containing images. The predicted output is visually annotated on the image using OpenCV, and the final digit sequence is displayed to the user via dialog prompts. This enhances the system's interactivity and makes it suitable for both technical and non-technical users.

Overall, this project demonstrates the effective integration of deep learning with traditional image processing techniques to address the problem of multi-digit handwritten recognition. It serves as both a practical tool and a learning platform for understanding the application of CNNs, image preprocessing, and GUI development in real-world AI solutions. Future extensions of the project may include support for alphanumeric recognition, deployment on mobile devices, and adaptation to multilingual handwritten scripts.

Table of Contents

CHAPTER # 1

Introduction

Contents:

1.1 Introduction to the Project

In the modern era of digital transformation, intelligent systems capable of interpreting and understanding human-generated content are in high demand. One such area that has seen a significant surge in research and applications is handwritten digit recognition. Handwriting is a natural form of communication for humans, and enabling machines to accurately interpret handwritten content has opened doors to numerous applications including postal sorting, bank cheque processing, document digitization, and automated form filling.

This project explores the development of a deep learning-based system capable of recognizing sequences of handwritten digits using a Convolutional Neural Network (CNN). The system is trained on the MNIST dataset, which consists of thousands of labelled handwritten digit images ranging from 0 to 9. Unlike traditional digit classification systems that identify isolated digits, this project focuses on detecting and classifying a series of digits from an input image, whether drawn on-screen or uploaded.

The application has two key features: the ability to process user-uploaded images and the capability to recognize digits drawn directly onto a digital canvas using a graphical user interface (GUI) built with Tkinter. The core of the system relies on image pre-processing techniques like grayscale conversion, thresholding, contour detection, morphological operations (dilation), Gaussian blurring, and resizing. These techniques prepare digit segments for accurate prediction by the trained CNN model.

To enhance the model's usability, OpenCV is integrated to visually mark and annotate detected digits, making it user-friendly and educational. The program not only predicts individual digits but also outputs a complete sequence, which is displayed via a pop-up message and visually overlaid on the image.

This project lies at the intersection of computer vision, machine learning, and human-computer interaction. By combining these domains, we aim to build an end-to-end intelligent system capable of understanding visual digit information in a real-world context.

1.2 **Problem Definition**

Recognizing handwritten digit sequences is a complex task due to variations in handwriting style, size, spacing, and orientation. The challenge increases when digits appear together in a sequence, as the system must accurately segment and classify each digit, even when they are touching or unevenly spaced. Traditional OCR systems are not well-suited for such inputs, especially when dealing with real-world noise, distortion, and handwritten variability. This project addresses the problem by developing an end-to-end system that combines image preprocessing techniques like contour detection with a deep learning model (CNN) to recognize digit sequences from both uploaded images and user-drawn input. The goal is to enable accurate, real-time recognition suitable for practical applications such as form automation, document digitization, and human-computer interaction.

1.3 **Aim**

The primary aim of this project is to design and implement a deep learning-based system capable of recognizing sequences of handwritten digits from both uploaded image files and drawings made on-screen. This is to be achieved using a trained Convolutional Neural Network (CNN) and OpenCV image processing techniques within a user-friendly graphical interface built with Tkinter.

The project strives to integrate machine learning with traditional image processing to offer a complete solution that can be used in practical applications. Unlike many existing systems that recognize only single digits or require user intervention to segment digits, this project aims to automate the entire pipeline: from input acquisition and preprocessing to digit segmentation and recognition.

Another aim is to create an educational and interactive platform where users can visualize how their input is interpreted by the system. This allows users to understand the functioning of CNNs and the importance of preprocessing steps, thus bridging the gap between theoretical knowledge and practical implementation.

Ultimately, the project aims to contribute to the field of intelligent handwriting recognition systems by demonstrating an approach that balances performance, usability, and explainability.

1.4 Objective

To build and train a Convolutional Neural Network (CNN) model using the MNIST dataset for the recognition of handwritten digits.

To create a Python-based GUI using Tkinter that allows users to either upload an image or draw digits on a canvas.

To implement an efficient image preprocessing pipeline using OpenCV to extract digit contours from noisy or cluttered images.

To sequence detected digits based on their position in the image and feed each one into the CNN model for prediction.

To display the final recognized sequence to the user via a message box and annotate predictions on the image for visual clarity.

To ensure high prediction accuracy across various digit styles, sizes, and input sources (upload and canvas).

To design a user-friendly application suitable for educational demonstrations or extension into larger document processing systems.

1.5 Goal

The long-term goal of this project is to contribute to the development of smarter human-computer interaction systems where machines can understand handwritten content just like humans. By recognizing digit sequences effectively, this system can be integrated into larger pipelines such as:

Automatic number plate recognition systems.

Bank check and form digitization.

Exam paper evaluation tools.

Postal code sorting systems.

Data entry automation solutions.

The project also aims to serve as a learning base for students and professionals interested in combining deep learning with computer vision. It showcases the power of CNNs in practical applications and helps users understand how traditional computer vision techniques (like contour detection) can work synergistically with machine learning.

1.6 Need for Our Project

Recognizing handwritten digits is important for automating tasks like form processing, exam evaluation, and postal sorting. Existing OCR tools often fail with handwritten input and do not support digit sequences or interactive use.

Why This Project is Needed:
Handles handwriting variability better than traditional OCR.

Uses deep learning (CNN) for high accuracy.

Supports both image upload and canvas drawing.

Recognizes full digit sequences, not just single digits.

Works offline on basic hardware.

Useful for real-world tasks and learning applications.

This system meets the need for an accurate, user-friendly, and accessible handwritten digit recognition solution.

CHAPTER # 2

Hardware and Software Requirement

Contents:

2.1 Introduction

This chapter provides a comprehensive overview of the technical environment and resources necessary for the successful development, deployment, and testing of the digit sequence recognition system based on a Convolutional Neural Network (CNN) trained on the MNIST dataset. A project of this nature requires a blend of both hardware and software components working together in synchronization to ensure real-time processing, accurate predictions, and a smooth graphical user interface (GUI) experience. The CNN model used in this project is resource-efficient yet powerful enough to process digit images in sequence with an accuracy of up to 99%. Moreover, the use of the OpenCV library for image preprocessing and the Tkinter library for GUI construction further necessitates the presence of a compatible and optimized development environment.

The tools, libraries, and technologies chosen for this project are open-source and widely adopted in the academic and industrial communities, allowing for future scalability and portability. From a hardware standpoint, the system was designed to run on commonly available personal computing resources without the need for specialized hardware such as GPUs, although the presence of a GPU can speed up the training process. This ensures the solution remains accessible to students, developers, and small organizations who wish to implement or extend the work.

This chapter will first define the system environment, followed by detailed listings of the software and hardware requirements that have been used in the development and testing phases of this project.

2.2 System Environment

The system environment includes the operating system, programming language, libraries, and frameworks that form the backbone of this project. Python has been selected as the primary programming language due to its extensive library support in machine learning and image processing. The environment is configured to support TensorFlow for loading the pre-trained CNN model, OpenCV for image processing tasks such as grayscale conversion and contour detection, and Tkinter for building the user-friendly graphical interface.

Operating System: Windows 10 + / Ubuntu 20.04

Programming Language: Python 3.8 or higher

Deep Learning Framework: TensorFlow (2.x)

Image Processing Library: OpenCV (cv2)

GUI Framework: Tkinter (standard Python GUI library)

Image Manipulation: PIL (Python Imaging Library)

Others: NumPy for numerical operations

2.3 Software Requirement

The software requirements include the necessary tools and libraries for running, training, and testing the CNN model and building the GUI interface. These components were carefully chosen for their efficiency, documentation, and widespread community support.

Software	**Version**	**Purpose**
Python	3.8+	Main programming language
TensorFlow	2.x	For loading and predicting with CNN model
OpenCV	4.x	For image preprocessing and contour detection
Tkinter	Built-in with Python	For GUI development
NumPy	1.20+	For numerical and matrix operations
PIL (Pillow)	8.x	Image drawing and manipulation
Jupyter Notebook / VS Code	Any	Optional – for model training/debugging

2.4 **Hardware Requirement**

Though the trained model is lightweight and optimized for digit recognition, certain minimum hardware specifications are necessary to run the project smoothly, especially when processing larger images or handling real-time GUI interactions. However, the project is designed to work without requiring high-end hardware such as GPUs for inference.

Component	Specification
Processor	Intel Core i3 or equivalent (minimum)
RAM	4 GB (minimum), 8 GB recommended
Storage	500 MB free space for libraries/model
Display	13" or larger, 1366×768 resolution
GPU (Optional)	NVIDIA GPU for faster training
Input Devices	Mouse (for GUI), Keyboard

A system with the above minimum configuration can easily run the trained CNN model to recognize handwritten digit sequences from uploaded images or drawings. During the training phase (if retraining is required), better performance can be achieved with a machine that has GPU support for TensorFlow.

CHAPTER # 3

Literature Review

Contents:

3.1 Literature Review

1. Handwritten Digits Identification Using MNIST Database via Machine Learning Models

Authors: Birjit Gope et al.
Year: 2021
Focus:

- Applied various classical machine learning algorithms for digit classification on the MNIST dataset.

Contribution:

- Compared five ML models: SVM, Decision Tree, Random Forest, Naïve Bayes, and KNN.
- Achieved highest accuracy of 95.88% with SVM.

Techniques/Features Used:

- Manual feature extraction, preprocessing to grayscale/binary formats, classification using traditional ML algorithms.

Dataset:

- MNIST, 28×28 pixels

Best Accuracy:

- SVM – 95.88%

Key Findings / Challenges:

- **Findings:**
  - SVM outperformed others.
- **Challenges:**
  - KNN had low accuracy.
  - Feature extraction is critical for classical models.

2. Classification of MNIST Handwritten Digit Database using Neural Network

Authors: Wan Zhu
Year: 2021
Focus:

- Explored neural network architectures (ANN, CNN, Autoencoder) to classify MNIST digits.

Contribution:

- Designed a neural network and extended it with Autoencoder and CNN.
- Introduced Conv_Autoencoder combining CNN and Autoencoder.

Techniques/Features Used:

- ANN with 1 hidden layer, cross-entropy loss, Adam optimizer.
- Compared ANN, Autoencoder, CNN, Conv_Autoencoder.

Dataset:

- MNIST, 60,000 training, 10,000 testing

Best Accuracy:

- CNN – 98.84%

Key Findings / Challenges:

- **Findings:**
  - CNN was best.

- **Challenges:**
  - Autoencoder reduced accuracy.
  - Conv_Autoencoder needs improvement for classification.

3. Hybrid CNN-SVM Classifier for Handwritten Digit Recognition

Authors: Savita Ahlawat & Amit Choudhary
Year: 2020
Focus:

- Developed a hybrid CNN-SVM model for handwritten digit recognition using MNIST.

Contribution:

- Used CNN for feature extraction and SVM for classification.
- CNN's softmax replaced by SVM.

Techniques/Features Used:

- CNN with 5x5 filters and Sigmoid activation.
- SVM with RBF kernel.
- Evaluated one-vs-one and one-vs-rest.

Dataset:

- MNIST, 28×28 pixels

Best Accuracy:

- Hybrid CNN-SVM – 99.28%

Key Findings / Challenges:

- **Findings:**
    - High accuracy, effective hybrid approach.

- **Challenges:**
    - Sensitive to SVM parameters.
    - Complex integration.

Comparative Analysis of Handwritten Digit Classification Models

Author	Gope et al. (2021)	Wan Zhu (2021)	Ahlawat & Choudhary (2020)
Main Approach	Classical ML	ANN, CNN, Autoencoder	Hybrid CNN + SVM
Best Model	SVM	CNN	Hybrid CNN-SVM
Best Accuracy	95.88%	98.84%	99.28%
Feature Extraction	Manual	CNN Layers, Autoencoder	CNN Feature Maps
Complexity	Low	Moderate	High
Advantages	Simple, fast	Modular, high accuracy	Robust, best accuracy
Challenges	Poor generalization in some models	Autoencoder weak on categories	SVM sensitive to parameters

CHAPTER # 4

Methodology

Contents:

4.1 Introduction to Image Processing

Image processing is crucial for **handwritten digit recognition**. It helps to enhance, analyze, and prepare images for deep learning models. In this project, image processing techniques were used for preprocessing the **MNIST dataset** and handling custom input images for prediction. The main goal was to improve the quality of the input data and ensure accurate classification by the **Convolutional Neural Network (CNN)**.

4.1.1 Purpose of Image Processing

The primary reasons for using image processing in this project was to:

- **Improve Image Quality:** This involved enhancing contrast, removing noise, and normalizing pixel values to ensure consistent input for the model.
- **Standardize Image Format:** All images were **resized to 28x28 pixels** to match the MNIST dataset's requirements.
- **Feature Extraction:** Techniques like edge detection, grayscale conversion, and normalization were applied to make distinguishing features more prominent.
- **Reduce Computational Complexity:** Converting images to grayscale and resizing them significantly reduced the amount of data the neural network had to process, leading to more efficient training.
- **Improve Model Accuracy:** Preprocessing helped the model focus on the actual digit patterns rather than irrelevant background noise.

4.1.2 Types of Images

Images used for handwritten digit classification can be categorized by their format and characteristics:

- **Binary Images:**
    - Consist of only two-pixel values: black (0) and white (1).
    - Commonly used in Optical Character Recognition (OCR) and document digitization.
    - Example: Scanned handwritten forms with clear digits.

- **Grayscale Images:**
    - Contain shades of gray ranging from 0 (black) to 255 (white).
    - These were primarily used in this project, as the MNIST dataset images are in grayscale.
    - Grayscale reduces complexity while still retaining important visual features.

- **RGB (Colored) Images:**
    - Composed of three-color channels: Red, Green, and Blue.
    - More complex, but not required for this project since handwritten digits are typically black and white.

4.1.3 Types of Image Processing Techniques

Different image processing techniques were applied to prepare digit images for both training and prediction:

- **Preprocessing Techniques (Used in this Project):**
    - **Grayscale Conversion:** Converts any colored input images to grayscale to match the MNIST format.
    - **Resizing:** Ensures all images are consistently **28x28 pixels** in dimension.
    - **Normalization:** Scales pixel values to a **0-1 range** to improve training efficiency.

        - **Thresholding:** Converts images to a binary (black & white) format if necessary.
- **Feature Extraction Techniques:**
    - **Edge Detection:** Highlights the outlines of the digits for better classification.
    - **Histogram Equalization:** Enhances contrast by distributing pixel intensity more evenly across the image.
- **Image Augmentation Techniques:**
    - **Rotation & Shifting:** Helps the model generalize well to handwritten digits that might be at different angles or positions.
    - **Zoom & Scaling:** Ensures the model can accurately classify digits written at various sizes.
    - **Adding Noise:** Improves the model's robustness by making it more resistant to slight distortions or imperfections in real-world images.

4.2 Training and Model Development

This section outlines the process of building and training the **Convolutional Neural Network (CNN)** to classify handwritten digits. This included data preparation, architectural design, model compilation, and optimization

4.2.1 Data Loading and Preprocessing

Before training the CNN model, the MNIST dataset underwent specific loading and preprocessing steps to ensure optimal learning:

- **Loading the MNIST Dataset:**
    - The MNIST dataset, readily available in TensorFlow/Keras, was loaded.
    - It contains 60,000 training images and 10,000 testing images, each **28x28 pixels** and in grayscale.
- **Preprocessing Steps:**
    - **Normalization:** Pixel values were rescaled from their original 0-255 range down to **0-1**. This helps in faster convergence during training.

- The images were reshaped to a **28x28x1 format** to be compatible with the CNN's expected input dimensions.
- Labels (the actual digit numbers) were converted into a **categorical format** using one-hot encoding.
- **Data augmentation** techniques, such as slight rotations, shifts, and zooms, were applied. This significantly enhances the model's robustness by exposing it to a more diverse set of input variations, which improves its generalization ability to unseen data.

4.2.2 CNN Model Architecture

A **Convolutional Neural Network (CNN)** was designed to automatically extract relevant features and classify handwritten digits. The architecture consisted of multiple specialized layers:

- **Input Layer:**
  - This layer accepted the **28x28 grayscale images**, each having a single channel (representing intensity).
- **Feature Extraction Layers:**
  - The **first convolutional layer** extracted features using **32 filters**, each of size **3x3**. This was immediately followed by a **max-pooling layer**, which reduced the dimensionality of the feature maps, helping to make the model more robust to minor shifts in the input.
  - A **second convolutional layer** with **64 filters** (also 3x3) further refined the feature extraction process, followed by another max-pooling layer for continued dimensionality reduction.
- **Fully Connected Layers:**
  - A **flattening layer** converted the 2D feature maps from the previous layers into a single, one-dimensional vector. This prepares the data for the dense layers.
  - A **fully connected layer** with **128 neurons** and a **ReLU (Rectified Linear Unit) activation function** was used.

- The **output layer** consisted of **10 neurons** (one for each digit from 0 to 9) and used a **softmax activation function**. Softmax is ideal for multi-class classification, outputting probabilities for each digit class.

4.2.3 Model Compilation and Training

After defining the CNN architecture, the model was set up for learning through compilation and then trained:

- **Compiling the Model:**
  - The **categorical crossentropy loss function** was chosen, as it's well-suited for multi-class classification problems like this one.
  - The **Adam optimizer** was selected due to its adaptive learning rate capabilities, which generally leads to faster and more stable training.
  - The **accuracy metric** was used to evaluate the model's performance during training and testing.

- **Training the Model:**
  - The model was trained over **10 epochs**, with a **batch size of 128**. An epoch represents one complete pass through the entire training dataset.
  - The training dataset was augmented (as mentioned in 4.2.1) to increase its diversity, helping the model learn more robust features.
  - A separate **validation dataset** was used to monitor the model's performance during training, helping to detect potential overfitting.

- **Model Performance:**
  - The trained model achieved a high accuracy of **99 percent** on the MNIST test dataset.
  - Analysis of the loss and accuracy curves showed steady improvement throughout training, indicating effective learning and minimal signs of overfitting.

4.2.4 Hyperparameter Tuning

To further enhance the model's accuracy and computational efficiency, **hyperparameter tuning** was performed using **Keras Tuner**. This process involved testing different configurations of parameters such as the number of filters, learning rate, and dense layer units.

- **Defining the Hyper Model:**

A customizable CNN model was created, allowing parameters such as the number of filters in convolutional layers, the learning rate, and the number of units in dense layers to vary within a defined search space. This enabled the exploration of different architectural configurations.

- **Running the Hyperparameter Search:**

A **random search approach** was used to efficiently explore various model configurations. This process involved training multiple models with different hyperparameter combinations.

- **Selecting the Best Model**

After numerous trials, the model that exhibited the **best validation accuracy** was chosen as the optimal one. This selection process helped identify the ideal combination of filters, dense layer units, and learning rate, leading to improved overall classification accuracy and generalization capability.

By diligently utilizing hyperparameter tuning, the CNN model was effectively optimized to achieve superior accuracy while maintaining computational efficiency. The combination of thorough data preprocessing, CNN model development, comprehensive training, and meticulous hyperparameter tuning resulted in a robust and highly accurate digit classification system.

4.3 Predictions and Evaluations

Once the CNN model was successfully trained, the next critical phase involved evaluating its performance using unseen test images from the MNIST dataset and making predictions on new inputs.

4.3.1 Loading the Trained Model

- After the training process was complete and the best-performing model was identified through hyperparameter tuning, it was **saved for future use**.
- This saved model was then **loaded** to make predictions on the dedicated test images and also on any real-world handwritten digit inputs.
- Loading a pre-trained model eliminates the need for retraining, which significantly **reduces computational time** and makes the model readily available for deployment and practical applications.

4.3.2 Processing New Images for Prediction

To ensure that the model could accurately classify new input images, they needed to undergo the exact same preprocessing steps applied to the training dataset. These steps included:

1. **Grayscale Conversion:** If the input image was in RGB (colored) format, it was first converted to grayscale. This ensures consistency, as the model was trained exclusively on grayscale MNIST images.

2. **Resizing to 28x28 Pixels:** The model was specifically trained on images of this size. Therefore, any new input image needed to be resized to **28x28 pixels** to maintain compatibility.

3. **Normalization of Pixel Values:** Pixel values were scaled between **0 and 1**. This is crucial for consistency with the training data and generally leads to better model performance.

4. **Reshaping Image for CNN Input:** The image was reshaped into the specific format (**28x28x1**) that the CNN model expects as input, ensuring smooth compatibility for the prediction phase.

These consistent preprocessing steps guarantee that any handwritten digit image, regardless of its original format, is correctly formatted, allowing the model to classify it accurately and reliably.

4.3.3 Generating Predictions and Confusion Matrix

Once an image was properly pre-processed, it was fed into the trained CNN model for classification:

- The model generated a **probability distribution** across the 10 possible digit classes (0-9). The class with the **highest probability** was then selected as the predicted digit.
- To evaluate the model's overall performance systematically, predictions were made on the **entire MNIST test dataset**.
- The predicted results from the model were then compared against the actual true labels of these test images.
- A **confusion matrix** was generated based on this comparison. The confusion matrix is a powerful visualization tool that clearly shows the accuracy of classifications and highlights specific instances of misclassification (e.g., how many '3's were mistakenly classified as '8's).
- Additionally, a **classification report** would typically be generated alongside the confusion matrix, providing detailed metrics such as precision, recall, and F1-score for each digit class.

4.4 Sequence Digit Detection Application

In the second phase of the project, the goal is to recognize not just individual digits but a sequence of digits from either an uploaded image or a hand-drawn input using the GUI. This phase integrates image processing with a trained CNN model to detect, segment, and classify multiple digits from a single input. The sequence detection module significantly enhances the practical usability of the digit recognition model, making it capable of reading numbers from addresses, zip codes, and hand-filled forms.

- The implementation involves several steps:
- Detecting digit contours in the image.
- Preprocessing each detected digit.
- Predicting each digit using the trained model.
- Displaying the results with visual annotations and GUI feedback.

This entire functionality is provided in an interactive application built with Tkinter, allowing users to either upload an image or draw directly on a canvas.

4.4.1 Contour Detection for Isolating Digits

One of the fundamental challenges in recognizing a sequence of digits is isolating each digit from the input image. This is where contour detection plays a critical role.

What is a Contour?

In OpenCV, a contour is simply a curve joining all the continuous points (along a boundary) having the same color or intensity. In binary images, contours help in segmenting and identifying distinct objects—in this case, individual digits.

Contours are useful for:

- Object detection
- Shape analysis
- Recognition and segmentation

How Contours Are Used in This Project

The function **process_image(image_path)** handles the core image processing. When a user uploads an image or draws on the screen, this function is called with the path of the image.

Steps:

- **Reading the image**: The image is loaded using cv2.imread(). If the image is invalid or not found, an error is displayed.
- **Grayscale conversion**: The image is converted to grayscale using cv2.cvtColor() to simplify processing.
- **Thresholding**: A binary threshold is applied to create a binary inverse image (cv2.THRESH_BINARY_INV). This helps separate the foreground (digits) from the background.
- **Contour Detection**: Using cv2.findContours(), the outer contours are identified. Each contour ideally represents one digit.

```
contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

- Noise Filtering: To avoid detecting small unwanted dots or artifacts as digits, contours are filtered based on area using a threshold (min_pixel_threshold).
- Sorting: The filtered contours are sorted from left to right based on their x coordinates ensuring the digits are read in order.

4.4.2 Digit Prediction from Image and Canvas

For each filtered contour, the following steps are performed to prepare the digit for prediction:

- **Cropping**: The bounding box of the contour is used to extract the digit region from the grayscale image.
- **Inversion Handling**: If the background is light and the digit is dark, or vice versa, inversion is done to maintain consistency.
- **Resizing**: The digit is resized to a fixed shape of (100×100), then padded into a 150×150 image to standardize size.
- **Image Enhancement**: Dilation is used to thicken strokes, and Gaussian blur is applied to reduce noise.
- **Final Resize**: The processed image is resized to (28×28), which is the expected input shape for the CNN.

```
resized = cv2.resize(object_image, (100, 100))
padded = np.zeros((150, 150), dtype=np.uint8)
padded[25:125, 25:125] = resized
```

- **Normalization**: Pixel values are scaled between 0 and 1.
- **Dimension Expansion**: The image is reshaped to fit the CNN model input shape: (1, 28, 28, 1)

```
input_img = np.expand_dims(np.expand_dims(normalized, axis=0), axis=-1)
```

- **Prediction**: The model predicts the digit and its confidence score using:

```
predictions = model.predict(input_img, verbose=0)
predicted_digit = np.argmax(predictions)
confidence = np.max(predictions)
```

- Each predicted digit is appended to the sequence_digits list, and the digit is annotated on the image using cv2.putText() and cv2.rectangle().

4.4.3 Prediction Using CNN

The pre-processed digit image is passed to the trained CNN model. The model returns a probability distribution over 10 classes (digits 0–9), and the digit with the highest probability is selected as the prediction.

```
predictions = model.predict(input_img, verbose=0)
predicted_digit = np.argmax(predictions)
confidence = np.max(predictions)
```

Each digit is:

- Appended to the result sequence.
- Visually marked on the image using a rectangle and label.

```
cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
cv2.putText(image, str(predicted_digit), (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
```

4.4.4 GUI Integration (Tkinter)

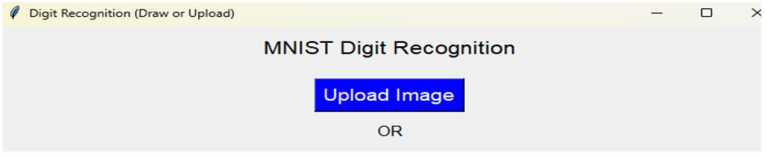
To ensure ease of use, a **Tkinter-based GUI** was developed that allows two modes of input:

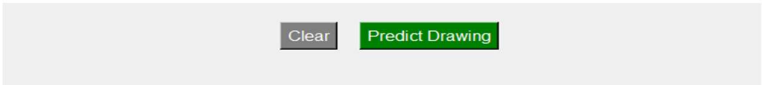
1. **Upload an Image**: Select an existing image file.
2. **Draw on Canvas**: Digit sequences can be drawn with the mouse.

The canvas uses the PIL library to save the drawing to an image file, which is then passed into the same processing pipeline.

The GUI consists of:

- A title and upload button.
- A canvas for drawing digits.
- Buttons for predicting and clearing the canvas.
- A result display dialog box.
- An OpenCV window showing the annotated image.





CHAPTER # 5

Result and Conclusion

Contents:

5.1 Result

The Convolutional Neural Network (CNN) model developed for handwritten digit recognition using the MNIST dataset has achieved high accuracy and robust performance in all stages of evaluation. The model was trained and validated on the MNIST dataset and then tested using unseen test data and user inputs provided via a custom-built Tkinter GUI.

5.1.1 Final Training Metrics Summary

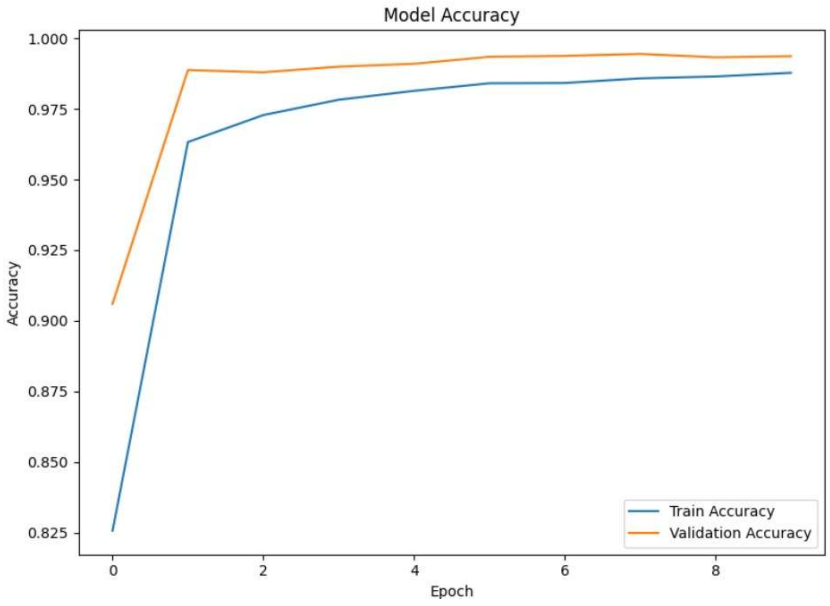
The final training, validation, and test results of the CNN model are summarized below:

Metric	Value
Final Training Accuracy	98.78%
Final Validation Accuracy	99.37%
Final Test Accuracy	99.45%
Final Training Loss	0.0426
Final Validation Loss	0.0202

5.1.2 Accuracy and Loss Curves

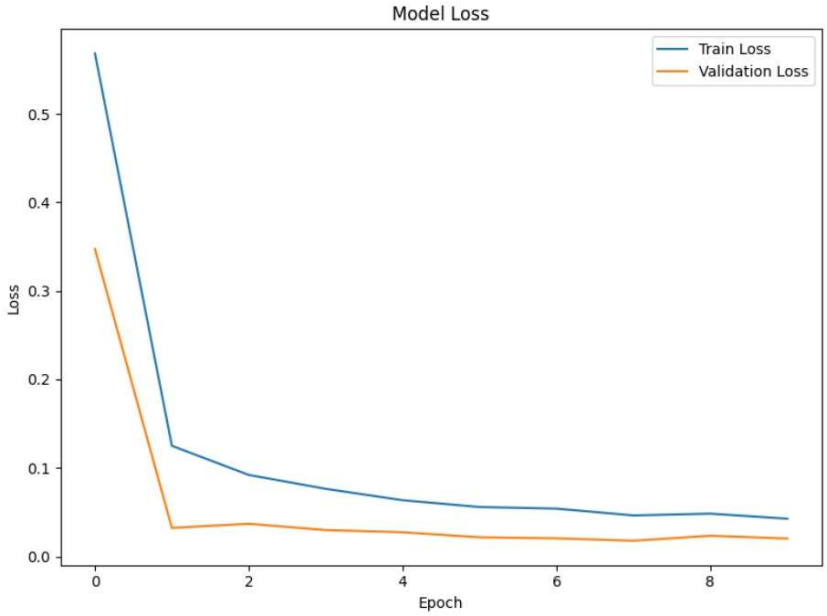
To better understand the model's performance across epochs, training and validation accuracy and loss were plotted.

Training vs Validation Accuracy



The model consistently improved in both training and validation accuracy, eventually converging at around 99.4% validation accuracy. The gap between training and validation curves remained minimal, indicating low overfitting.

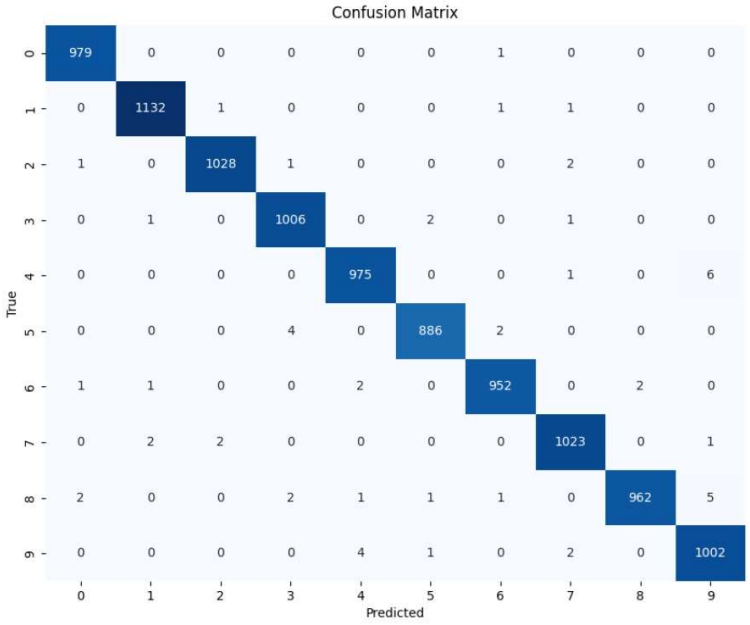
Training vs Validation Loss



5.1.3 Confusion Matrix

The confusion matrix provides insights into how well the model is able to classify each digit (0–9). It highlights misclassifications and helps analyze which digits are most commonly confused.

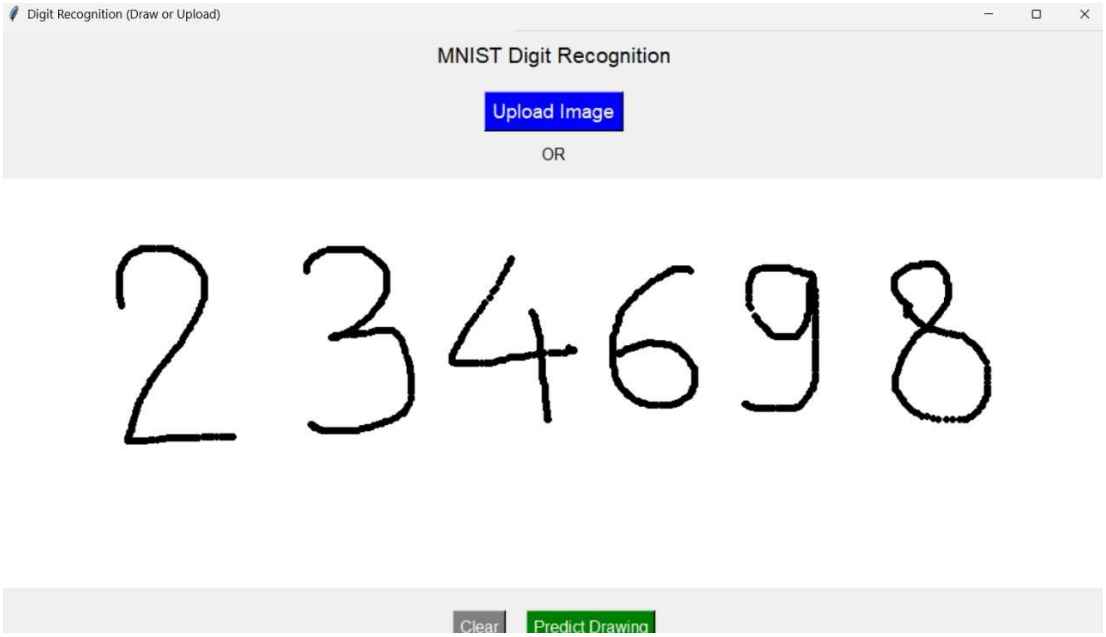
Confusion Matrix Heatmap



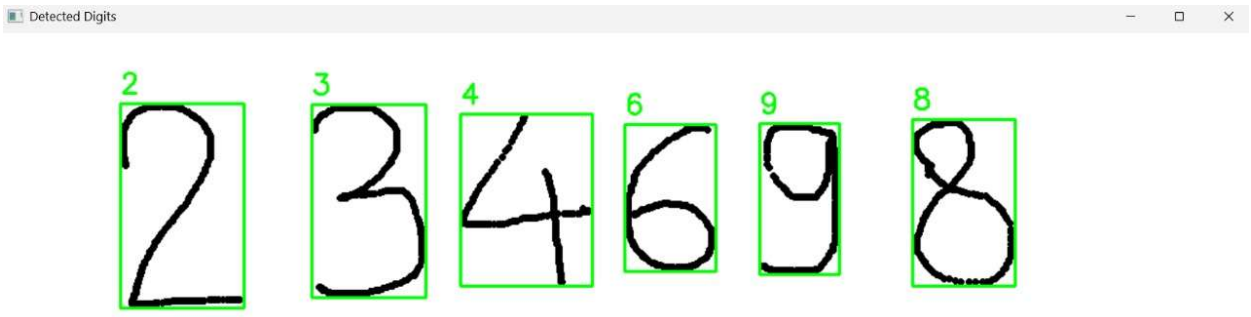
5.1.4 Sample Results from Tkinter Interface

The application allows users to either draw digits on a canvas or upload an image containing a sequence of digits. The model processes these inputs, extracts digit regions using image processing (with contour detection), and then performs classification.

Tkinter Canvas – Drawing Interface & Uploaded Image Interface



Predicted Output



5.2 Conclusion

This project successfully implements a robust, high-accuracy CNN model capable of recognizing handwritten digits from both standalone images and sequences. The following conclusions can be drawn:

The CNN model achieved 99.45% accuracy on the test set, validating its high performance.

The image processing pipeline, especially contour detection, played a crucial role in sequence recognition.

The Tkinter GUI enhances usability, allowing both drawing and image upload for prediction.

The low training and validation loss indicate minimal overfitting and excellent learning efficiency.

The application is capable of real-time digit recognition with high reliability and user-friendly interaction.

This system demonstrates the practical integration of deep learning with image processing and GUI development, paving the way for real-world applications in digit recognition such as automated form reading, zip code extraction, and check processing.

5.3 Future Scope

This project demonstrates effective recognition of handwritten digit sequences using a CNN model with a Tkinter-based interface. While the current system achieves high accuracy, several improvements can enhance its functionality:

1. **Alphanumeric Recognition**: Expanding the system to recognize alphabets along with digits would make it useful for broader applications like license plate or form recognition.
2. **Advanced Models**: Using modern deep learning architectures like ResNet or Vision Transformers can improve accuracy and performance.

Bibliography

1. Gope, B., Biswas, M., & Barman, A. (2021). *Handwritten Digits Identification Using MNIST Database via Machine Learning Models*.

   — Compared traditional machine learning algorithms like SVM, KNN, Decision Tree, and Naïve Bayes for digit classification on the MNIST dataset.

2. Zhu, W. (2021). *Classification of MNIST Handwritten Digit Database Using Neural Network*.

   — Analyzed neural architectures including ANN, CNN, Autoencoder, and introduced a Conv_Autoencoder for digit recognition using MNIST.

3. Ahlawat, S., & Choudhary, A. (2020). *Hybrid CNN-SVM Classifier for Handwritten Digit Recognition*.

   — Proposed a hybrid model using CNN for feature extraction and SVM for classification, achieving high accuracy on MNIST.

4. OpenCV Documentation. https://docs.opencv.org/

   — Official documentation used for image preprocessing, contour detection, and visualization tasks.

5. TensorFlow/Keras Documentation. https://www.tensorflow.org/guide/keras

   — Reference for CNN model building, training, and evaluation.

6. MNIST Handwritten Digit Dataset. http://yann.lecun.com/exdb/mnist/

   — Benchmark dataset used for training and evaluating the digit recognition model.

7. Tkinter GUI Library. https://docs.python.org/3/library/tkinter.html

   — Used for developing the graphical user interface for image upload and digit drawing functionalities.

Appendices

Appendix A: Model Training Code

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix
import seaborn as sns
import pickle
import os

def load_and_preprocess_data():
    """Load and preprocess MNIST data with enhanced preprocessing"""
    (X_train, y_train), (X_test, y_test) = mnist.load_data()

    # Normalize and resha
    X_train = X_train.astype('float32') / 255.0
              X_test = X_test.astype('float32') / 255.0
              # Add channel dimension
              X_train = np.expand_dims(X_train, axis=-1)
              X_test = np.expand_dims(X_test, axis=-1)

    # One-hot encode labels
    y_train = to_categorical(y_train, 10)
    y_test = to_categorical(y_test, 10)

    return X_train, y_train, X_test, y_test
           def create_enhanced_model():
               model = Sequential([
              # First convolutional block
                  Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same'),
                  BatchNormalization(),
      Conv2D(32, (3, 3), activation='relu', padding='same'),
                   BatchNormalization(),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        # Second convolutional block
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        # Third convolutional block for higher accuracy
        Conv2D(128, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
```

```
Conv2D(128, (3, 3), activation='relu', padding='same'),
BatchNormalization(),
MaxPooling2D((2, 2)),
Dropout(0.25),

# Fully connected layers
Flatten(),
Dense(256, activation='relu'),
BatchNormalization(),
            Dropout(0.5),
            Dense(128, activation='relu'),
            BatchNormalization(),
            Dropout(0.5),
            Dense(10, activation='softmax')
        ])

        # Custom optimizer with learning rate scheduling
        optimizer = Adam(learning_rate=0.001)

        model.compile(optimizer=optimizer,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        return model

    def train_enhanced_model(model, X_train, y_train, X_test, y_test):
        """Train model with enhanced techniques"""
        # Data augmentation
        datagen = ImageDataGenerator(
            rotation_range=10,
            zoom_range=0.1,
            width_shift_range=0.1,
            height_shift_range=0.1)
        datagen.fit(X_train)

        # Callbacks
        early_stopping = EarlyStopping(monitor='val_loss', patience=10,
    restore_best_weights=True)
        reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3,
    min_lr=0.00001)
        model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy',
                                           save_best_only=True, mode='max')

        # Train with data augmentation
        history = model.fit(datagen.flow(X_train, y_train, batch_size=128),
                            epochs=10,
                            validation_data=(X_test, y_test),
                            callbacks=[early_stopping, reduce_lr, model_checkpoint])

        # Save training history
        with open('training_history.pkl', 'wb') as f:
            pickle.dump(history.history, f)
```

```
def save_plots_and_metrics(history, model, X_test, y_test):
    """Save all plots and metrics for later analysis"""
    # Create directory for results
    os.makedirs('training_results', exist_ok=True)

    # 1. Save training history plots
    plt.figure(figsize=(16, 6))

    # Accuracy plot
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.savefig('training_results/training_metrics.png')
    plt.close()

    # 2. Save confusion matrix
    y_pred = model.predict(X_test)
    cm = confusion_matrix(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.savefig('training_results/confusion_matrix.png')
    plt.close()


    # 3. Save metrics summary
    with open('training_results/metrics_summary.txt', 'w') as f:
        f.write("=== Training Metrics Summary ===\n")
        f.write(f"Final Training Accuracy: {history.history['accuracy'][-1]:.4f}\n")
        f.write(f"Final Validation Accuracy: {history.history['val_accuracy'][-1]:.4f}\n")
        f.write(f"Final Test Accuracy: {model.evaluate(X_test, y_test,
verbose=0)[1]:.4f}\n")
        f.write(f"Final Training Loss: {history.history['loss'][-1]:.4f}\n")
        f.write(f"Final Validation Loss: {history.history['val_loss'][-1]:.4f}\n")
```

```
    def load_and_print_history():
    """Load and print saved training history"""
        with open('training_history.pkl', 'rb') as f:
        history = pickle.load(f)

    print("\n=== Saved Training History ===")
    print(f"Final Training Accuracy: {history['accuracy'][-1]:.4f}")
    print(f"Final Validation Accuracy: {history['val_accuracy'][-1]:.4f}")
    print(f"Final Training Loss: {history['loss'][-1]:.4f}")
    print(f"Final Validation Loss: {history['val_loss'][-1]:.4f}")

def main():
    # Load data
    X_train, y_train, X_test, y_test = load_and_preprocess_data()

    # Create enhanced model
    model = create_enhanced_model()
    model.summary()

    # Train with enhanced techniques
    history, test_acc = train_enhanced_model(model, X_train, y_train, X_test, y_test)

    # Save all plots and metrics
    save_plots_and_metrics(history, model, X_test, y_test)

    # Print saved history
    load_and_print_history()

    # Save the final model
    model.save('Trained_Model.h5')
    print("\nModel saved as 'Trained_Model.h5'")
    print("All training results saved in 'training_results' directory")

    return model

if __name__ == "__main__":
    model = main()
```

Appendix B. Prediction and Interface Code

```
import cv2
import numpy as np
from tensorflow.keras.models import load_model
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageDraw

# Load model globally
model = load_model('Trained_Model.h5')

# Prediction pipeline (used by both image upload and canvas draw)
def process_image(image_path):
image = cv2.imread(image_path)
if image is None:
messagebox.showerror("Error", "Could not load image.")
return

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
_, binary = cv2.threshold(gray, 230, 255, cv2.THRESH_BINARY_INV)
contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

if len(contours) == 0:
messagebox.showinfo("Result", "No digits detected.")
return

areas = [cv2.contourArea(c) for c in contours]
mean_area = np.mean(areas)
min_pixel_threshold = mean_area /3

filtered_contours = [c for c in contours if cv2.contourArea(c) >= min_pixel_threshold]
filtered_contours = sorted(filtered_contours, key=lambda c: cv2.boundingRect(c)[0])

sequence_digits = []

for i, contour in enumerate(filtered_contours):
x, y, w, h = cv2.boundingRect(contour)
object_image = gray[y:y + h, x:x + w]

if np.mean(object_image) > 100:
object_image = 255 - object_image

resized = cv2.resize(object_image, (100, 100))
padded = np.zeros((150, 150), dtype=np.uint8)
padded[25:125, 25:125] = resized

dilated = cv2.dilate(padded, np.ones((7, 7), np.uint8), iterations=1)
blurred = cv2.GaussianBlur(dilated, (9, 9), sigmaX=0)
processed = cv2.resize(blurred, (28, 28))
normalized = processed / 255.0
input_img = np.expand_dims(np.expand_dims(normalized, axis=0), axis=-1)
predictions = model.predict(input_img, verbose=0)
predicted_digit = np.argmax(predictions)
```

```
confidence = np.max(predictions

cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
cv2.putText(image, str(predicted_digit), (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

print(f"Object {i + 1}: Predicted = {predicted_digit}, Confidence = {confidence:.2f}")

print("Recognized Sequence:", ''.join(sequence_digits))
messagebox.showinfo("Result", f"Recognized Sequence: {''.join(sequence_digits)}")

cv2.imshow("Detected Digits", image)
cv2.waitKey(0)
cv2.destroyAllWindows()

def upload_and_process():
file_path = filedialog.askopenfilename(filetypes=[("Image files", ".png;.jpg;.jpeg;.bmp")])
if file_path:
process_image(file_path)

# Drawing functionality
canvas_width = 1200
canvas_height = 400
drawn_image_path = "canvas_draw.png"

def clear_canvas():
canvas.delete("all")
draw.rectangle([0, 0, canvas_width, canvas_height], fill="white")
def save_and_predict():
img.save(drawn_image_path)
process_image(drawn_image_path)
# Initialize previous position
prev_x, prev_y = None, None
def paint(event):
global prev_x, prev_y
if prev_x is not None and prev_y is not None:
canvas.create_line(prev_x, prev_y, event.x, event.y,
width=8, fill='black', capstyle=tk.ROUND, smooth=True, splinesteps=36)
draw.line([prev_x, prev_y, event.x, event.y], fill='black', width=8)
prev_x, prev_y = event.x, event.y
def reset(event):
global prev_x, prev_y
prev_x, prev_y = None, None
# GUI Setup
root = tk.Tk()
root.title("Digit Recognition (Draw or Upload)")
root.geometry("500x600")
title_label = tk.Label(root, text="MNIST Digit Recognition", font=("Helvetica", 16))
title_label.pack(pady=10)
upload_button = tk.Button(root, text="Upload Image", command=upload_and_process,
font=("Helvetica", 14), bg="blue", fg="white")
upload_button.pack(pady=10)
separator = tk.Label(root, text="OR", font=("Helvetica", 12))
separator.pack()
```

```
# Canvas for drawing
canvas_frame = tk.Frame(root)
canvas_frame.pack(pady=10)



draw = ImageDraw.Draw(img)
canvas.bind("<B1-Motion>", paint)
canvas.bind("<ButtonRelease-1>", reset)

button_frame = tk.Frame(root)
button_frame.pack(pady=10)

clear_button = tk.Button(button_frame, text="Clear", command=clear_canvas,
font=("Helvetica", 12), bg="gray", fg="white")
clear_button.grid(row=0, column=0, padx=10)

predict_button = tk.Button(button_frame, text="Predict Drawing", command=save_and_predict,
font=("Helvetica", 12), bg="green", fg="white")
predict_button.grid(row=0, column=1, padx=10)

root.mainloop()
```