**Handwritten Digit Recognition Using Convolutional Neural Networks (CNN)**

**Project Report**

INDUSTRIAL TRAINING (ECS591)

Degree

**Bachelor of Technology (Computer Science & Engineering) Specialization in Artificial Intelligence, Machine Learning & Deep Learning**

|  |  |
| --- | --- |
| PROJECT GUIDE:  **Mr. Umesh Chandra Garjola** | SUBMITTED BY:  **Student Name: Saurabh Ekka (TCA2359189)**  **Student Name: Shuaib(TCA2359201)**  **Section: C**  **Semester: Vth** |

November, 2025



**FACULTY OF ENGINEERING & COMPUTING SCIENCES**

**TEERTHANKER MAHAVEER UNIVERSITY, MORADABAD**

**DECLARATION**

We hereby declare that this Project Report titled **Handwritten Digit Recognition Using Convolutional Neural Networks (CNN)** submitted by us and approved by our project guide, Faculty of Engineering & Computing Sciences. Teerthanker Mahaveer University, Moradabad, is a bonafide work undertaken by us and it is not submitted to any other University or Institution for the award of any degree diploma / certificate or published any time before.

|  |  |  |
| --- | --- | --- |
| **Project ID :** |  | |
| **Student Name:** | **Saurabh Ekka (TCA2359189)** | Signature |
| **Student Name:** | **Shuaib(TCA2359201)** | Signature |
|  |  |  |
| **Project Guide :** | **Mr. Umesh Chandra Garjola** | Signature |

Table of Contents

[1 Project Title 4](#_Toc31392321)

[2 Problem Statement 4](#_Toc31392322)

[3 Project Description 4](#_Toc31392323)

[3.1 Scope of the Work 5](#_Toc31392324)

[3.2 Project Modules 7](#_Toc31392325)

[3.3 Context Diagram (High Level) 9](#_Toc31392326)

[4 Implementation Methodology 11](#_Toc31392327)

[5 Technologies to be used 18](#_Toc31392328)

[5.1 Software Platform 18](#_Toc31392329)

[5.2 Hardware Platform 19](#_Toc31392330)

[5.3 Tools, if any 20](#_Toc31392331)

[6 Advantages of this Project 21](#_Toc31392332)

[7 Assumptions, if any 22](#_Toc31392333)

[8 Future Scope and further enhancement of the Project 23](#_Toc31392334)

[9 Project Repository Location 25](#_Toc31392335)

[10 Definitions, Acronyms, and Abbreviations 25](#_Toc31392336)

[11 Conclusion 27](#_Toc31392337)

[12 References 27](#_Toc31392338)

**Appendix**

**A: Data Flow Diagram (DFD)**

**B: Entity Relationship Diagram (ERD)**

**C: Use Case Diagram (UCD)**

**D: Data Dictionary (DD)**

**E: Screen Shots**

# Project Title

**Handwritten Digit Recognition Web Application using CNN and Flask**

# Problem Statement

Digit recognition is a classic pattern‑recognition problem with applications such as automatic form processing, evaluation of objective answer sheets, bank cheque processing, and digit‑based data entry. In most institutes and offices, reading handwritten digits is still done manually, which introduces:

• Human errors due to fatigue or poor handwriting.

• High time and cost when large volumes of documents must be processed.

• Lack of automation in small organizations that cannot afford commercial OCR systems.

This project proposes a web‑based handwritten digit recognition system built entirely with open‑source tools. Using a browser, the user can:

• Draw a digit on a canvas with mouse or touch.

• Capture an image via webcam.

• Upload an image file from the local machine.

The server processes the image, classifies the digit using a CNN trained on MNIST, and returns the prediction and confidence to the user. This prototype demonstrates how deep learning, image processing, and web technologies can work together to automate a real‑world task.

# Project Description

This project titled “Handwritten Digit Recognition Web Application using CNN and Flask” focuses on designing and implementing a complete software system that can automatically recognize handwritten digits (0–9). The system integrates deep learning, image processing, and web technologies and is intended for educational and laboratory use.

The core idea is to provide an interactive web interface where users can input handwritten digits in different ways and obtain predictions in real time:

• By drawing digits directly on an HTML5 canvas using a mouse, stylus, or finger.

• By capturing images of digits using a webcam.

• By uploading digit images stored on the local machine.

These images are sent from the browser to a Flask‑based backend API. The backend runs an image preprocessing pipeline to convert the input into a normalized 28×28 grayscale image similar to the MNIST dataset. This processed image is passed to an Improved Convolutional Neural Network (ImprovedCNN) implemented in PyTorch, which has been trained offline using the MNIST digit dataset.

The model outputs class scores for digits from 0 to 9. The system applies a softmax function to transform these scores into probabilities, selects the digit with the highest probability as the predicted class, and returns the result along with a confidence score to the frontend as JSON. The frontend then updates the user interface to show the predicted digit, the confidence value, and a graphical confidence bar. For some inputs, the server can also perform heuristic two‑digit recognition by splitting the image into two parts and predicting a digit for each side.

The entire solution is structured into three logical layers:

1. Model Training Layer – scripts and components responsible for training the improved CNN model on the MNIST dataset and saving the best model.

2. Application Layer (Backend API) – the Flask application that loads the trained model, preprocesses images, performs inference, and exposes REST API endpoints.

3. Presentation Layer (Frontend) – the HTML/CSS/JavaScript based web interface where users draw or capture digit images and view predictions.

This layered structure keeps the system modular, maintainable, and easy to extend.

## Scope of the Work

The scope of this work covers the complete flow from model training to interactive usage through a web interface.

In‑Scope Activities

**1. Model Design and Training**

◦ Designing an improved CNN architecture (ImprovedCNN) with three convolutional blocks, batch normalization, and dropout.

◦ Using the MNIST dataset as training and evaluation data.

◦ Applying data augmentation (random rotations, translations, and zoom) to improve generalization.

◦ Implementing training procedures with Adam optimizer, cross‑entropy loss, learning‑rate scheduling, and early stopping.

◦ Saving the best model weights as mnist\_model\_best.pth and a deployment model as mnist\_model.pth.

**2. Image Preprocessing**

◦ Implementing a robust preprocessing routine that:

▪ Decodes base64 encoded images,

▪ Converts images to grayscale,

▪ Automatically inverts colors when the background is lighter than the digit,

▪ Enhances contrast using percentile‑based scaling,

▪ Applies Otsu’s thresholding to separate foreground from background,

▪ Uses morphological operations to remove noise and thicken strokes,

▪ Detects the digit’s bounding box, pads and resizes it, and centers it in a 28×28 canvas,

▪ Normalizes pixel values to match MNIST statistics.

3. Backend Web API

◦ Developing a Flask application with the following endpoints:

▪ /predict\_digit – for single‑digit recognition.

▪ /predict\_two\_digits – for heuristic recognition of two digits in a single drawing.

▪ /health – to provide information about server status and whether the model is loaded.

◦ Integrating PyTorch with Flask so that the model can run inference on preprocessed images and return the results as JSON.

4. Frontend User Interface

◦ Designing HTML pages with:

▪ A Drawing Canvas page for sketching digits.

▪ A Camera Capture page for taking photos and uploading images.

◦ Implementing JavaScript logic to:

▪ Draw and capture strokes on canvas.

▪ Capture camera frames and convert them into images.

▪ Send images to the Flask API using fetch with JSON.

▪ Display predictions, confidence bars, and error messages.

▪ Show model/API status using the /health endpoint.

5. Testing and Validation

◦ Running script‑based sanity tests using debug\_model.py.

◦ Performing manual functional tests for all digits 0–9 in various sizes and positions on the canvas.

◦ Testing camera capture and upload with handwritten digits on paper.

◦ Maintaining a simple defect log summarizing issues and fixes.

**Out of Scope Activities**

• User login, authentication, and role management (e.g., admin vs. normal user).

• Persistent storage of images and results in a relational database.

• Large‑scale deployment on cloud platforms or container orchestration systems.

• Recognition of full sentences, equations, or cursive handwriting with complex sequence models.

• Support for letters or non‑numeric symbols.

These limitations keep the project focused on demonstrating single‑digit and simple two‑digit recognition in a lab‑friendly environment.

## Project Modules

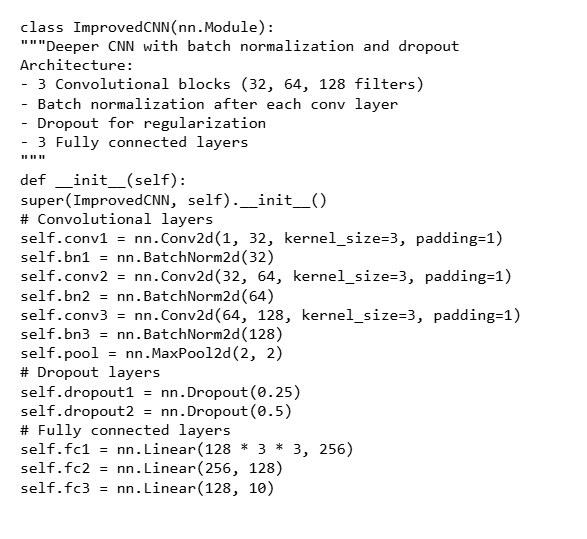
The project is decomposed into the following functional modules.

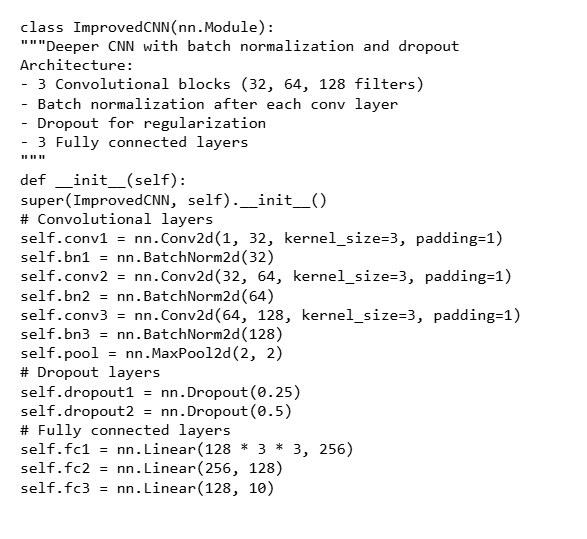
**3.2.1 Data Preparation and Model Training Module**

File: train\_improved\_model.py

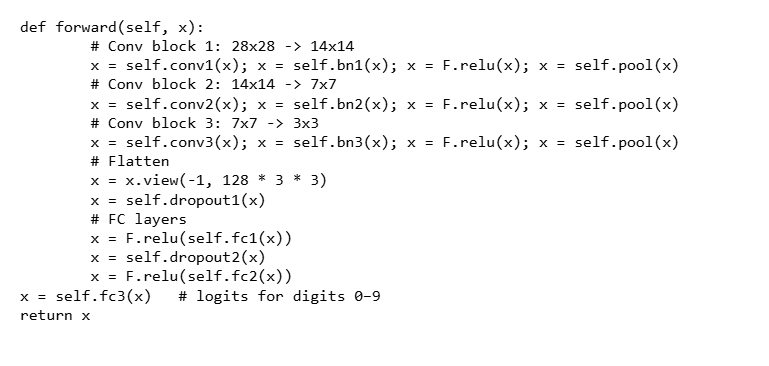
This module defines the CNN model, loads the MNIST dataset, and executes the training loop. It is responsible for producing the files mnist\_model\_best.pth and mnist\_model.pth which are later consumed by the Flask backend.

**ImprovedCNN Architecture**

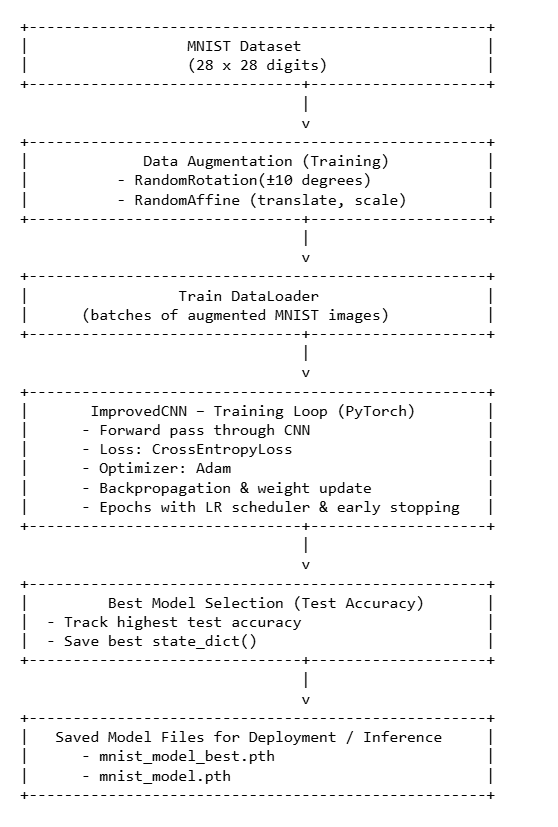
****

****

**Forward propagation**

****

**Data Flow (Training Time)**



## Context Diagram (High Level)

The context diagram represents the Handwritten Digit Recognition Web Application as a single high‑level process and shows how it interacts with the outside world. At Level‑0 there is only one external entity, the User, who communicates with the system through a web browser.

From the User to the system, the primary inputs are:

• **Digit Images** – images generated from the drawing canvas, captured from the webcam, or uploaded as files. These images are sent to the application whenever the user initiates a prediction request.

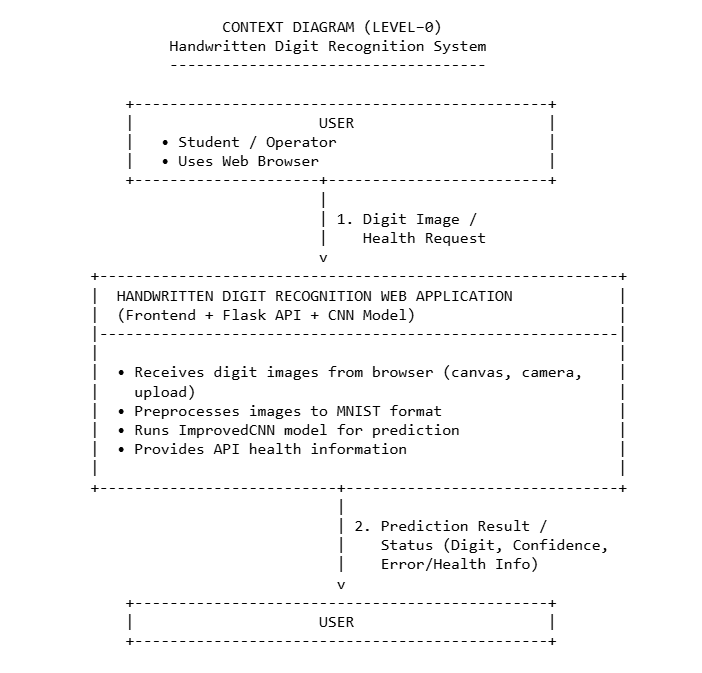
• **Health Requests** – simple requests triggered by the frontend to check whether the Flask API and the trained model are running correctly.

From the system back to the User, the main outputs are:

**• Prediction Results** – the recognized digit (or two‑digit combination) along with its confidence value and, optionally, the full probability distribution over digits 0–9.

**• Status / Error Information** – responses indicating whether the model is loaded, which device (CPU/GPU) is being used, and any error messages if the input is invalid or the server encounters a problem.

In this diagram, all internal details—such as the preprocessing steps, CNN model, and UI components—are abstracted into a single process box labelled “Handwritten Digit Recognition Web Application”. This gives a clear picture of the system boundary: everything inside the box belongs to the project, and everything outside (the User) is considered external. The context diagram therefore provides a simple, top‑level view of how the application fits into its environment and what flows of information cross the system boundary.



# Implementation Methodology

The implementation of the Handwritten Digit Recognition Web Application follows a structured process from requirements to deployment. To clearly describe the flow of information, this section uses Data Flow Diagrams (DFDs) and links them to the actual code components in the project.

**4.1 Development Process Overview**

**1. Requirement Analysis** – Identify inputs (digit images from canvas, camera, upload), outputs (digit predictions and confidence), and non‑functional requirements (accuracy, responsiveness, use of open‑source technologies).

**2. High‑Level Design** – Define overall architecture (Model Layer, Backend API Layer, Frontend Layer) and prepare a Level‑0 Context Diagram showing the User and the Handwritten Digit Recognition System.

**3. Detailed Design** – Create Level‑1 and Level‑2 DFDs for the internal processes:

◦ Acquire Image

◦ Preprocess Image

◦ Predict Digit(s)

◦ Display Result & Status

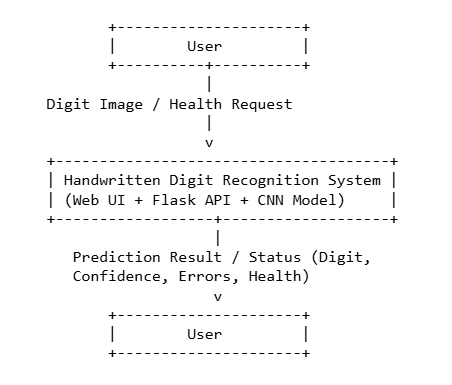
Also define the logical entities involved (input images, preprocessed images, prediction results) and the Flask/PyTorch components that implement these processes.

4. Implementation – Code the CNN model and training script, the Flask API and preprocessing pipeline, and the frontend UI and JavaScript logic.

5. Testing & Defect Log – Test each process individually and then end‑to‑end, recording issues in a defect log and retesting after fixes.

6. Deployment – Train the model, start the Flask server, and run the frontend locally for demonstration.

**4.2 Data Flow Diagrams and Corresponding Code**

**4.2.1 DFD – Level‑0 (Context Diagram)** 

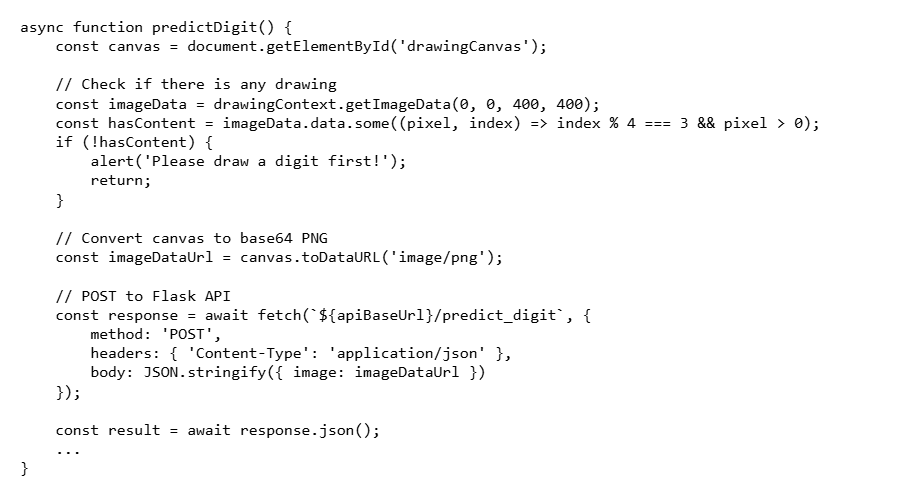
**All internal details (frontend, backend, model) are abstracted into one process, “Handwritten Digit Recognition System”.**

**4.2.2 DFD – Level‑1 (Inside the System) **

Below each process is mapped to the relevant code.

Process 1.0 – Acquire Image

Responsibility: Capture digit images from canvas, camera, or file upload on the frontend and send them to the backend API.

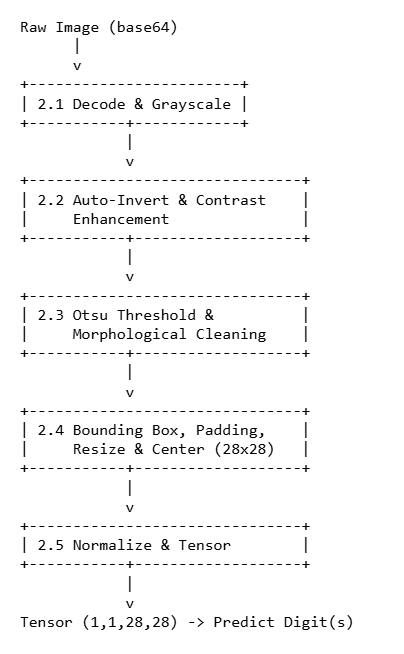
**Main code: script-with-flask.js**

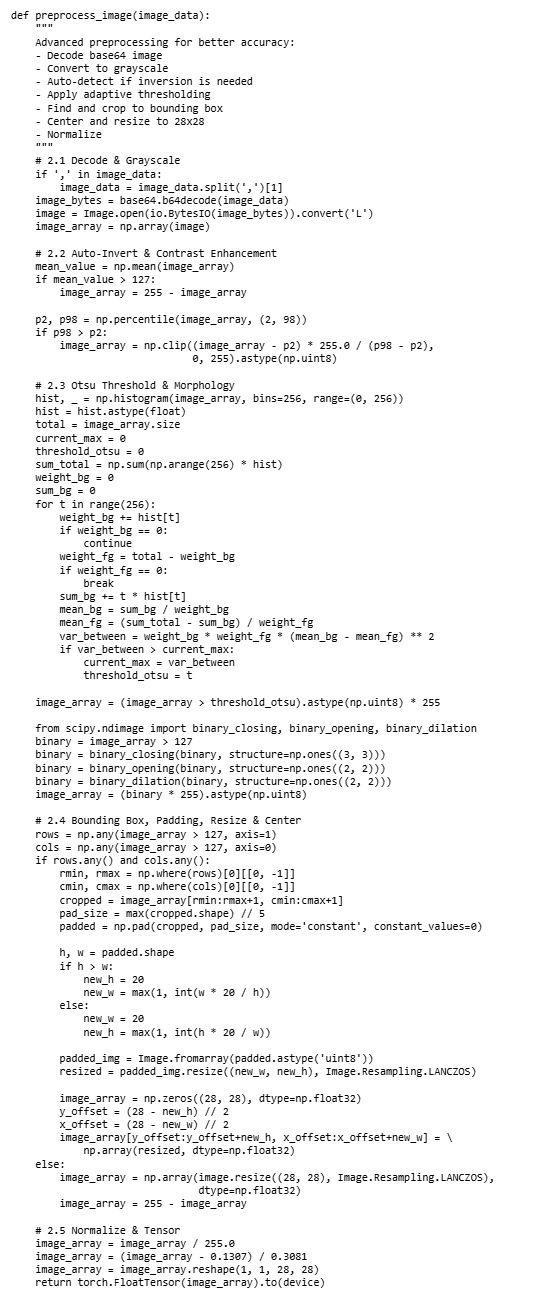
The same approach is used for camera capture and file upload, where the image is drawn onto a hidden canvas, converted to base64, and sent to the server.

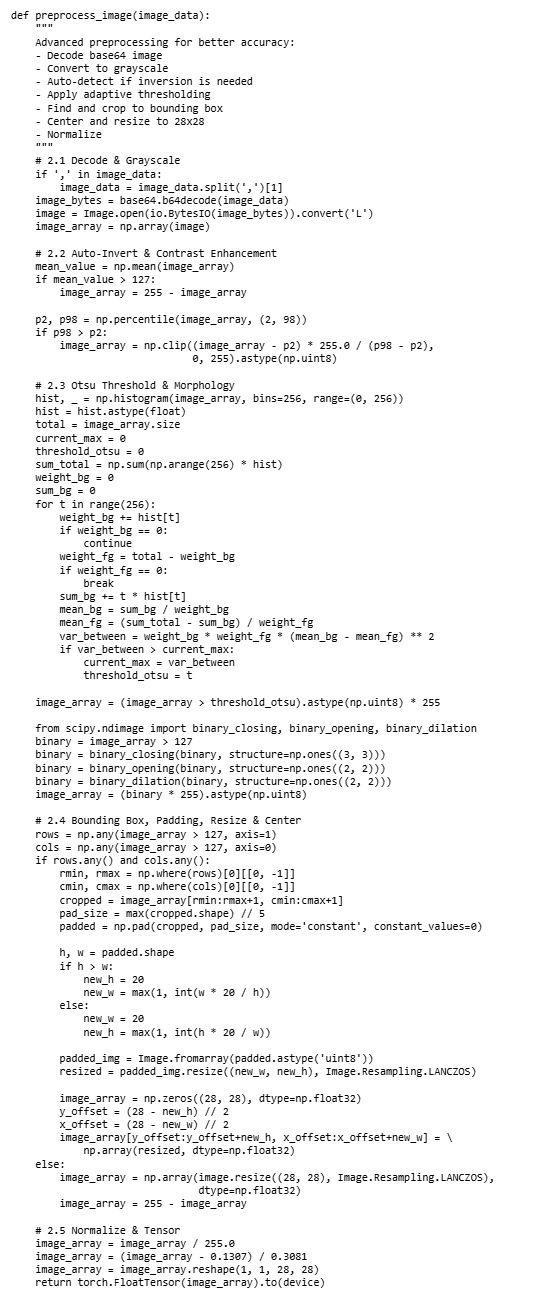
Process 2.0 – Preprocess Image

Responsibility: Convert arbitrary images into MNIST‑style 28×28 grayscale tensors suitable for the CNN.

**DFD – Level‑2 for Preprocessing**

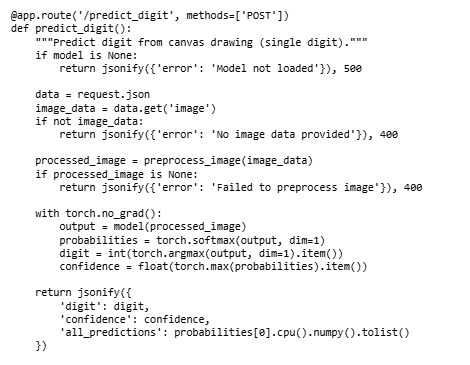
****

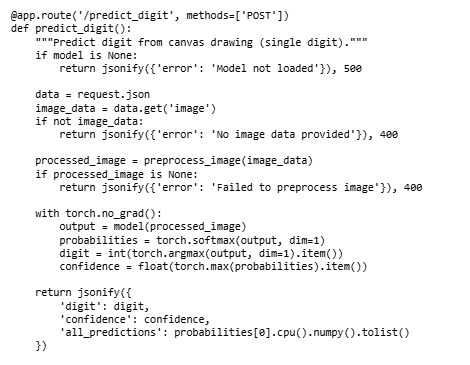
**Main code: preprocess\_image in flask\_app\_improved.py**

****

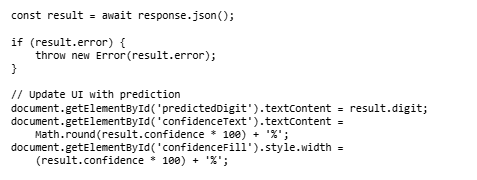
**Process 3.0 – Predict Digit(s)**

**Responsibility: Run the ImprovedCNN model on the preprocessed tensor and produce predictions.**

**Code for single‑digit prediction: flask\_app\_improved.py**

****Two‑digit prediction uses a similar flow but splits the 28×28 image into left and right halves, converts each half back to tensors, and runs the model twice.

Process 4.0 – Display Result & Status

Responsibility: Receive JSON response from backend and show prediction and status to the user in the UI. **

**Code snippet: script-with-flask.js (continuation of predictDigit)**

**The same pattern is used for image prediction from camera/upload with different DOM elements.**

**4.3 Testing and Defect Log in Relation to DFDs**

**Testing is aligned with the DFD processes:**

**• Process 2.0 Preprocess Image:**

**◦ Verified by printing shapes, min/max pixel values and threshold values in flask\_app\_debug.py.**

**◦ Tested with blank canvases, single dots, and extreme lighting images to ensure robust behaviour.**

**• Process 3.0 Predict Digit(s):**

**◦ Tested using debug\_model.py to ensure model weights are correct and predictions are sensible.**

**• End‑to‑End (1.0 → 4.0):**

**◦ Draw/ capture/upload digits and check that predictions display correctly in the UI.**

**Each defect found during testing is mapped to a specific process:**

**• For example, “digits cut off at border” mapped to Process 2.4 Bounding Box & Centering and fixed by increasing padding.**

**• The defect log records the DFD process, description, root cause and resolution, then the scenario is re‑executed to confirm the fix.**

**This completes a detailed Implementation Methodology section that is directly supported by Data Flow Diagrams and real code snippets from your MNIST web application.**

# Technologies to be used

## Software Platform

1. **Front-end**

**• HTML5**

Used to design the structure and content of the web pages (index.html). It defines the layout for the Drawing Canvas page and the Camera Capture page, including canvas, buttons, video elements, and result display areas.

**• CSS3**

Used for styling and responsive design (styles.css). It provides a modern look and feel, layouts for navigation buttons, containers for canvases and videos, confidence bars, and ensures the UI works well on different screen sizes.

**• JavaScript (ES6)**

Client‑side scripting language (script-with-flask.js) used to:

• Handle drawing on the HTML5 canvas (mouse and touch events).

• Access the user’s webcam using navigator.mediaDevices.getUserMedia.

• Convert canvas and camera frames to base64 PNG images using toDataURL.

• Send HTTP requests to the Flask backend via fetch and receive predictions as JSON.

• Update the DOM with predicted digit(s), confidence bar, error messages, and health status.

**• Web Browser**

Any modern browser such as Google Chrome, Mozilla Firefox, or Microsoft Edge is used to run the frontend, execute JavaScript, and render the UI.

1. **Back-end**

**• Python 3.x**

Core programming language used for the training script, preprocessing logic, and Flask backend.

**• Flask**

Lightweight web framework used to build the RESTful API (flask\_app\_improved.py, flask\_app\_debug.py). It defines endpoints /predict\_digit, /predict\_two\_digits, and /health, handles JSON requests, and returns JSON responses.

**• Flask‑CORS**

Used to enable Cross‑Origin Resource Sharing so that the frontend (served as a static file) can communicate with the Flask API without browser CORS issues.

**• PyTorch**

Deep learning framework used to implement and run the ImprovedCNN model (train\_improved\_model.py, flask\_app\_improved.py). It provides tensor operations, neural network layers, optimizers, and GPU support.

**• Torchvision**

Library used for loading the MNIST dataset and applying image transforms (data augmentation, normalization) during training.

**• NumPy**

Numerical library used for array manipulation, computing histograms, means, and percentiles during preprocessing.

**• Pillow (PIL)**

Image processing library used to open images from bytes, convert them to grayscale, resize them, and convert them back and forth between PIL images and NumPy arrays.

• SciPy (ndimage)

Used for morphological operations (closing, opening, dilation) in the preprocessing pipeline to remove noise and thicken digit strokes.

## Hardware Platform

The project is designed to run on a standard desktop or laptop computer. The typical configuration is as follows:

**• Processor (CPU):**

Dual‑core processor or higher.

**• RAM:**

Minimum 4 GB (recommended 8 GB or more, especially for model training).

**• Hard Disk / Storage:**

At least 2 GB free space to install Python, libraries, the MNIST dataset, model files, and project source code.

**• Operating System:**

Any operating system capable of running Python 3, Flask, and modern web browsers (e.g., Windows, Linux, or similar OS).

**• Input Devices:**

◦ Mouse or touchpad (for drawing on the canvas).

◦ Optional touch screen (for finger/stylus input).

◦ Webcam (for camera capture of handwritten digits).

**• Display:**

Monitor with resolution 1366×768 or higher, sufficient to view the canvas and prediction panels comfortably.

**• Editor / IDE:**

◦ Visual Studio Code / PyCharm or any text editor to edit Python, HTML, CSS, and JavaScript files.

**• Browser:**

◦ Google Chrome, Mozilla Firefox, Microsoft Edge, or equivalent modern browser for running and testing the frontend.

## Tools, if any

During the project, several tools are used to support different phases of the life cycle:

**• Visual Studio Code (Microsoft, latest stable version)**

◦ Purpose: Source‑code editor for Python, HTML, CSS, and JavaScript.

◦ Used for: Writing and editing code, managing project files, running terminals, and using extensions (Python, Git, etc.).

**• Git (Distributed Version Control System)**

◦ Purpose: Optional but recommended for version control.

◦ Used for: Tracking changes in the codebase, creating commits, branching, and collaborating if multiple developers are involved.

**• pip (Python Package Installer)**

◦ Purpose: Package manager for installing required Python libraries listed in requirements.txt.

◦ Used for: Installing Flask, Flask‑CORS, PyTorch, Torchvision, Pillow, NumPy, SciPy, etc.

**• Browser Developer Tools (built into Chrome/Edge/Firefox)**

◦ Purpose: Debugging frontend code.

◦ Used for: Inspecting HTML/CSS, viewing console logs, monitoring network requests to /predict\_digit and /health, and detecting JavaScript errors.

# Advantages of this Project

The Handwritten Digit Recognition Web Application using CNN and Flask offers several technical and practical advantages:

**1. End‑to‑End Machine Learning Integration**

The project demonstrates the complete lifecycle of an AI application—from model design and training to deployment and real‑time inference through a web interface. This helps students understand not only how a CNN works, but also how to expose it as a usable service.

**2. Automation of Manual Tasks**

Manual recognition and entry of handwritten digits (e.g., from forms, answer sheets, small registers) can be slow and error‑prone. This system automates that task, thereby reducing human effort, minimizing mistakes, and speeding up data entry.

**3. User‑Friendly Web Interface**

The use of an HTML5 canvas and camera capture makes the application intuitive and interactive. Users can simply draw digits with a mouse or take a photo, without needing any special hardware or complicated configuration.

**4. Robust Image Preprocessing**

The advanced preprocessing pipeline (auto‑inversion, contrast enhancement, Otsu thresholding, morphological operations, and centering) makes the system robust to variations in handwriting style, digit position, background brightness, and noise. This improves recognition accuracy on real‑world inputs compared to using the raw image directly.

**5. Improved CNN Architecture with High Accuracy Potential**

The ImprovedCNN model uses multiple convolutional layers, batch normalization, dropout, and data augmentation. These design choices help the model achieve high accuracy on the MNIST dataset and generalize better to digits drawn by users on the canvas or captured by the camera.

**6. Modular and Maintainable Design**

The project is divided into clear modules: training script, preprocessing, Flask API, and frontend. Each part can be modified or upgraded independently—for example, replacing the CNN with a more advanced model, or redesigning the UI—without rewriting the entire system.

**7. Open‑Source and Cost‑Effective**

All technologies used—Python, Flask, PyTorch, HTML, CSS, JavaScript—are open‑source. This eliminates licensing costs and allows the system to be installed and modified freely for academic or research purposes.

8. Educational Value for Multiple Subjects

The project links concepts from several subjects: Neural Networks, Image Processing, Web Technologies, and Software Engineering. It can be used as a teaching example or lab exercise to show how theoretical concepts are applied in a practical project.

**9. Easy Extensibility**

The current system recognizes digits, but the same architecture can be extended to:

• Recognize letters or multiple‑character strings.

• Integrate with a database for storing results.

• Deploy on cloud platforms for wider accessibility.

Because of its clean API‑based design, such enhancements can be added incrementally.

**10. Portable and Platform‑Independent**

The application runs on any machine that supports Python and a modern browser. There is no dependency on specialized hardware; even without a GPU, the model can perform inference on CPU fast enough for interactive use.

Overall, this project not only solves a specific problem—handwritten digit recognition—but also serves as a reusable and extensible template for building other AI‑enabled web applications.

# Assumptions, if any

The design and implementation of the Handwritten Digit Recognition Web Application are based on the following assumptions:

**1. Quality of Input Digits**

It is assumed that users will provide reasonably clear digits similar to those in the MNIST dataset:

• Single digits or two digits written side‑by‑side.

• Dark strokes on a relatively lighter background (paper or canvas).

• Digits not heavily overlapped, scribbled, or extremely small.

**2. Single User / Light Load Environment**

The system is primarily intended for lab demonstrations and academic use, where only a small number of users access the application at the same time. High‑concurrency scenarios and strict performance guarantees are not considered.

**3. Local or Trusted Network**

It is assumed that the frontend and Flask backend run on the same machine or a trusted local network. Network latency is low enough that HTTP requests and responses feel real‑time to the user.

**4. Availability of Required Software**

The machine where the backend runs is assumed to have:

• Python 3.x installed,

• Necessary libraries installed via pip (Flask, Flask‑CORS, PyTorch, Torchvision, NumPy, Pillow, SciPy), and

• A compatible browser for running the frontend.

**5. Model Files Present and Valid**

It is assumed that at least one trained model file (mnist\_model\_best.pth or mnist\_model.pth) exists in the project directory and is not corrupted. The application does not handle cases where the model file is missing or incompatible beyond reporting an error.

**6. Webcam Permissions (for Camera Feature)**

For the camera capture functionality, it is assumed that:

• The system has a working webcam, and

• The user grants camera access permissions in the browser when prompted.

**7. No Malicious Inputs**

The application assumes that users do not intentionally send malformed or extremely large images to crash or attack the system. Basic validation is implemented, but full security hardening against adversarial inputs is outside the scope of the project.

# Future Scope and further enhancement of the Project

Although the current system successfully recognizes single handwritten digits (and simple two‑digit combinations) via a web interface, there are many directions in which the project can be extended and enhanced.

**1. Support for Multi‑Digit Numbers and Full Equations**

◦ Extend the current two‑digit splitting heuristic to a more robust multi‑digit segmentation method using connected‑component analysis or sequence models.

◦ Implement models that can read entire handwritten numbers (e.g., “2025”) and full arithmetic expressions.

◦ Replace the mock /predict\_equation endpoint with an end‑to‑end pipeline that detects characters, recognizes them, parses the expression, and computes the result.

**2. Recognition of Alphabets and Special Characters**

◦ Train new models (or use transfer learning) to recognize handwritten letters (A–Z, a–z) and symbols (+, −, ×, ÷, =, etc.).

◦ Turn the system into a general handwriting recognition engine that can be used for forms, notes, and classroom assignments.

**3. Integration with Database and User Management**

◦ Add a database layer (e.g., MySQL, PostgreSQL, or SQLite) to store users, their prediction history, and uploaded images.

◦ Implement user login, roles (student, teacher, admin), and secure access control.

◦ Provide analytics dashboards for teachers or administrators to see statistics such as most common digits, error rates, and model performance over time.

**4. Cloud Deployment and Scaling**

◦ Containerize the application using Docker and deploy it on cloud platforms such as AWS, Azure, or any other cloud provider.

◦ Use a reverse proxy (e.g., Nginx) and auto‑scaling groups or container orchestration (e.g., Kubernetes) to handle higher traffic.

◦ Expose the prediction API as a public service that can be used by other applications.

**5. Model Improvements and Advanced Architectures**

◦ Experiment with more powerful architectures such as ResNets, MobileNets, or Vision Transformers for digit recognition, especially if the system is extended to more complex data.

◦ Use techniques like model quantization or pruning to reduce model size and speed up inference, making the system more suitable for deployment on low‑power devices.

**6. Offline and Mobile Support**

◦ Use technologies like Progressive Web Apps (PWA) or on‑device inference frameworks (e.g., TensorFlow Lite, ONNX Runtime Mobile) to run the model locally on mobile devices without requiring a server.

◦ Provide an Android/iOS app or an installable PWA so that users can draw digits and see predictions even when there is no internet connection.

**7. Improved UI/UX and Accessibility**

◦ Add features such as undo/redo strokes on the canvas, zooming, and stroke smoothing.

◦ Provide visual explanations, such as showing the preprocessed 28×28 image that is sent to the model, so that users can understand why some digits are misclassified.

◦ Improve accessibility by adding keyboard shortcuts, high‑contrast themes, and screen‑reader‑friendly labels.

**8. Logging, Monitoring, and Evaluation Tools**

◦ Implement structured logging of all prediction requests (with user consent) to analyze real‑world performance.

◦ Build evaluation tools to compare model predictions against ground truth labels for collected datasets, enabling continuous improvement of the model.

◦ Integrate monitoring tools to track API latency, error rates, and resource usage.

**9. Security and Robustness Enhancements**

◦ Add input validation and rate limiting to protect the API from malformed or excessive requests.

◦ Use HTTPS for secure communication between client and server, especially if deployed over the internet.

◦ Investigate and mitigate adversarial inputs that might cause the model to misbehave.

**10. Educational Extensions**

◦ Provide an admin panel where instructors can upload custom datasets and retrain the model from the web interface.

◦ Add visualizations of training curves and confusion matrices so that students can experiment with network architectures and hyperparameters and immediately see the effect on performance.

By pursuing these enhancements, the current prototype can evolve into a powerful, flexible, and production‑ready handwriting recognition platform that supports a wide range of academic and real‑world applications.

# Project Repository Location

| **S#** | **Project Artifacts (softcopy)** | **Location** (Mention Lab-ID, Server ID, Folder Name etc.) | **Verified by Project Guide** | **Verified by Lab In-Charge** |
| --- | --- | --- | --- | --- |
|  | Project Synopsis Report (Final Version) |  |  |  |
|  | Project Progress updates |  |  |  |
|  | Project Requirement specifications |  |  |  |
|  | Project Report (Final Version) |  |  |  |
|  | Test Repository |  |  |  |
|  | Project Source Code (final version) with executable |  |  |  |
|  | Any other document |  |  |  |

# Definitions, Acronyms, and Abbreviations

|  |  |
| --- | --- |
| Abbreviation | Description |
| **AI** | Artificial Intelligence – a branch of computer science concerned with building systems capable of performing tasks that typically require human intelligence. |
| **API** | Application Programming Interface – a defined set of HTTP endpoints and rules enabling communication between software components; in this project, includes /predict\_digit, /predict\_two\_digits, and /health. |
| **CNN** | Convolutional Neural Network – a deep neural network architecture widely used for image recognition tasks such as handwritten digit classification. |
| **DFD** | Data Flow Diagram – a graphical model depicting how data moves through a system, showing processes, data stores, and external entities. |
| **ERD** | Entity–Relationship Diagram – a conceptual model illustrating relationships among data entities within a database or system. |
| **GUI** | Graphical User Interface – the visual component of the application, including web pages, buttons, and canvas elements used for user interaction. |
| **HTML** | HyperText Markup Language – the foundational markup language for creating the structure of web pages (e.g., index.html). |
| **HTTP** | HyperText Transfer Protocol – the protocol governing communication between the browser and the Flask server via requests and responses. |
| **JSON** | JavaScript Object Notation – a lightweight data format used for exchanging requests and responses between the frontend and backend. |
| **MNIST** | Modified National Institute of Standards and Technology database – a benchmark dataset containing 28×28 grayscale images of handwritten digits (0–9) used to train and evaluate models. |
| **PWA** | Progressive Web App – a type of web application that behaves like a native app, offering features such as offline access and installation. |
| **REST** | Representational State Transfer – an architectural style for designing web services; in this project, implemented via Flask using HTTP methods and JSON. |
| SRS | Software Requirements Specification – a comprehensive document detailing the system’s functional and non-functional requirements. |
| **UI / UX** | User Interface / User Experience – refers to the application’s visual design (UI) and the overall usability and satisfaction of user interactions (UX). |
| URL | Uniform Resource Locator – the address used to access web resources (e.g., http://localhost:5000/predict\_digit). |
| CPU | Central Processing Unit – the primary processor executing most of the system’s computational tasks, including model inference in Flask. |
| **GPU** | Graphics Processing Unit – a specialized processor that accelerates deep learning model training and inference tasks, particularly for CNNs. |

# Conclusion

The Handwritten Digit Recognition Web Application using CNN and Flask successfully demonstrates how modern machine learning techniques can be integrated into a practical, user‑friendly software system. Starting from a clear problem statement—automating the recognition of handwritten digits—the project has covered the complete pipeline: data preparation, model training, backend API development, frontend user interface, and end‑to‑end testing.

On the model side, an improved Convolutional Neural Network (ImprovedCNN) was designed and trained on the MNIST dataset with data augmentation, batch normalization, dropout, and learning‑rate scheduling. These design choices enabled the model to achieve high accuracy on test data and to generalize reasonably well to digits drawn by users on the canvas or captured via camera. The model was exported as reusable weight files (mnist\_model\_best.pth, mnist\_model.pth) and loaded at runtime by the Flask backend.

On the backend side, a lightweight Flask API was implemented to accept images from the client, run a robust preprocessing pipeline, execute the CNN model, and return predictions as JSON responses. The preprocessing stage—covering grayscale conversion, auto‑inversion, contrast enhancement, Otsu thresholding, morphological cleaning, bounding‑box detection, and centering—proved crucial in bridging the gap between noisy real‑world images and the standardized MNIST format.

On the frontend side, an interactive web interface was developed using HTML, CSS, and JavaScript. The Drawing Canvas and Camera Capture pages provide multiple ways for users to supply handwritten digits, while the integration with the Flask API via fetch allows predictions and confidence values to be displayed in real time. Visual feedback, such as confidence bars and status messages, makes the system easy to use and understand even for non‑technical users.

From a software engineering perspective, the project illustrates modular design, clear separation of concerns, and the use of diagrams (Context Diagram, DFDs, ERD, and use‑case descriptions) to document the system. The implementation methodology included iterative development, systematic testing at both module and integration levels, and simple defect‑log maintenance to track issues and fixes.

In conclusion, the project meets its objectives by delivering a working prototype that can accurately recognize handwritten digits via a web interface and can be demonstrated in an academic environment. At the same time, the architecture and codebase have been kept flexible enough to support future enhancements—such as multi‑digit recognition, alphabet recognition, database integration, and cloud deployment—making this application a solid foundation for more advanced handwriting recognition and AI‑enabled systems.

# References

| **S#** | **Reference Details** | **Owner / Project Group ID** |
| --- | --- | --- |
| 1 | MNIST Database of Handwritten Digits, Yann LeCun et al., AT&T Labs / NYU. Available at:http://yann.lecun.com/exdb/mnist/ | Public Dataset |
| 2 | PyTorch Documentation, “PyTorch: An Open Source Machine Learning Framework”. Available at:https://pytorch.org/docs | PyTorch Team / Meta |
| 3 | Flask Documentation, “Flask Web Development Framework”. Available at:https://flask.palletsprojects.com | Pallets Projects |
| 4 | Torchvision Documentation – Datasets and Transforms (MNIST). Available at:https://pytorch.org/vision/stable/index.html | PyTorch Team / Meta |
| 5 | MDN Web Docs – HTML Canvas and Web APIs, articles on<canvas>,getUserMedia, and Fetch API. Available at:https://developer.mozilla.org | Mozilla |

**Annexure A**

**Data Flow Diagram (DFD)**

**(Mandatory)**

**A.1 Level–0 DFD (Context Diagram)**

****

Description

The Level–0 DFD, also called the Context Diagram, shows the Handwritten Digit Recognition Web Application as a single process and its interaction with the external entity User.

• External Entity: User (student / operator using the browser)

• System: Handwritten Digit Recognition Web Application

• Input Data Flows: Digit Image, Health Request

• Output Data Flows: Prediction Result, Status / Error Information

**A.2 Level–1 DFD (Internal Processes)**

The Level–1 DFD decomposes the main system into four key processes and one logical data store.

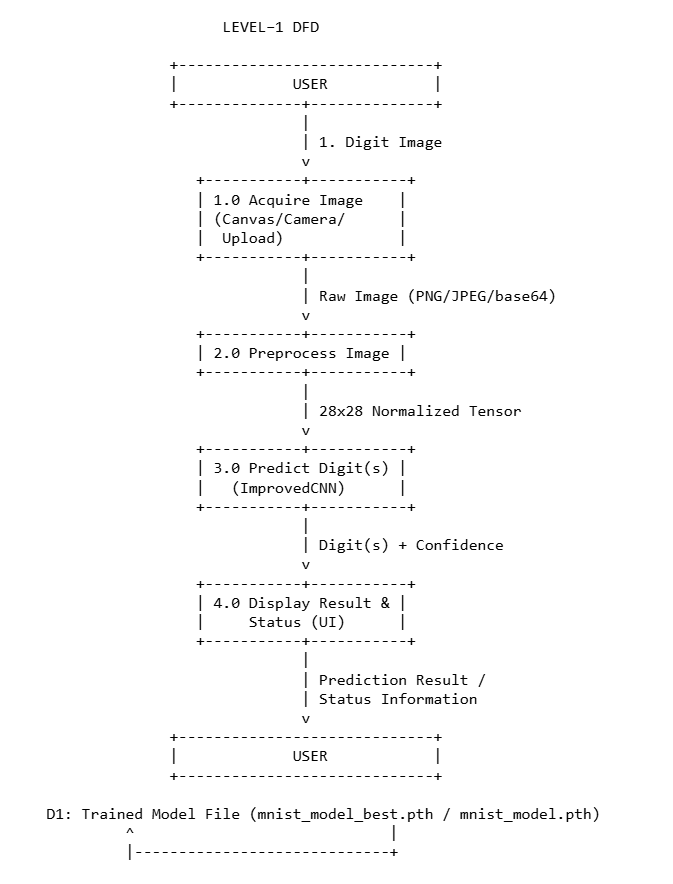
1.0 Acquire Image

2.0 Preprocess Image

3.0 Predict Digit(s)

4.0 Display Result & Status

**• Data Store:**

D1 Trained Model File (mnist\_model\_best.pth, mnist\_model.pth) ****

**A.3 Level–2 DFD for Process 2.0 – Preprocess Image**Process 2.0 Preprocess Image is critical because it converts arbitrary input images into a MNIST‑compatible format. It is refined into five sub‑processes:

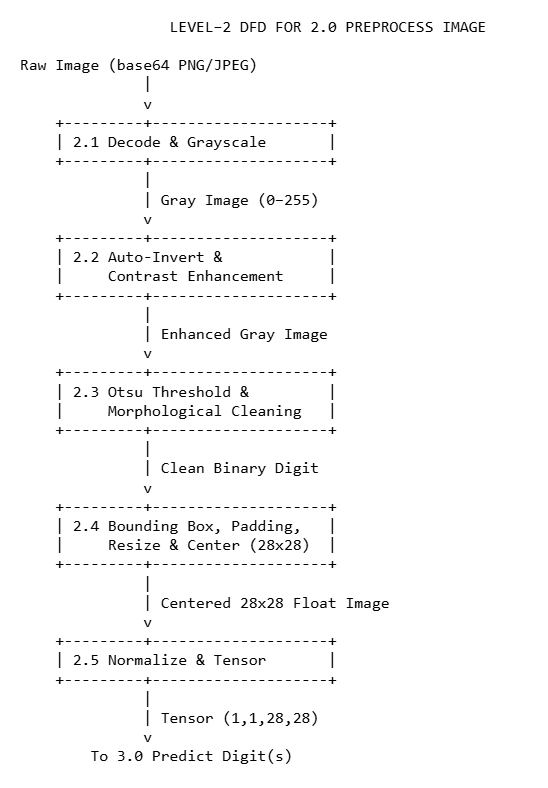
• 2.1 Decode & Grayscale

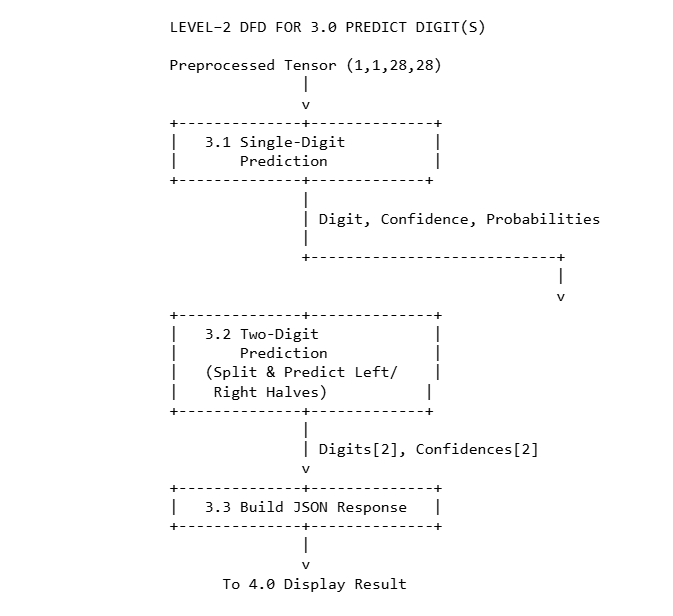
• 2.2 Auto‑Invert & Contrast Enhancement

• 2.3 Otsu Threshold & Morphological Cleaning

• 2.4 Bounding Box, Padding, Resize & Center

• 2.5 Normalize & Tensor Formation

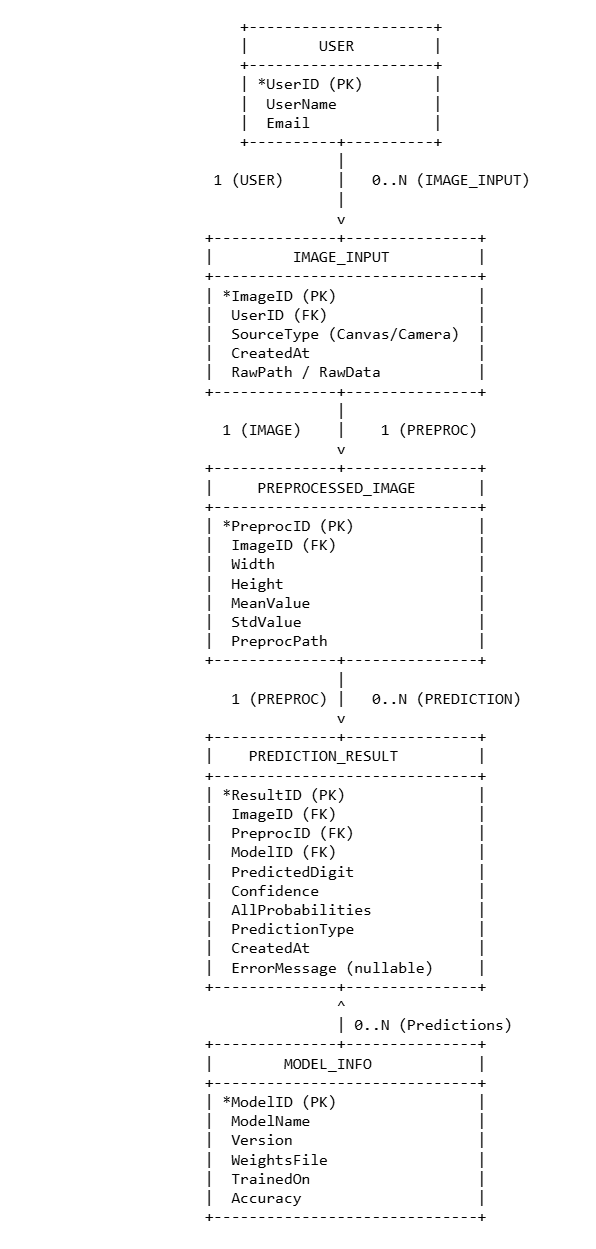
****

A.4 Level–2 DFD for Process 3.0 – Predict Digit(s) 

**Annexure B**

**Entity-Relationship Diagram (ERD)**

**(Mandatory)**

****

**Annexure C**

**Use-Case Diagram (UCD)**

**(Optional)**

**C.1 Overview**

The Use‑Case Diagram describes how external actors interact with the Handwritten Digit Recognition Web Application. For this project, the primary actor is the User (student / operator) who accesses the system via a web browser. Optionally, an Admin actor can be considered for model management and monitoring in future enhancements.

**C.2 Actors**

**1. User**

◦ Any person using the application to draw or capture digits and view predictions.

◦ Interacts only through the web interface.

2. Admin (optional, future scope)

◦ Person responsible for maintaining the system.

◦ May retrain models, view logs and statistics, and configure system parameters.

**C.3 Main Use Cases**

**For User**:

**1. UC1: Draw Digit on Canvas**

◦ User opens the Drawing Canvas page and draws a handwritten digit (or two digits) using mouse, touchpad, or touch screen.

**2. UC2: Capture Digit Image via Camera**

◦ User opens the Camera Capture page, starts the webcam, and captures an image of handwritten digits on a sheet of paper.

**3. UC3: Upload Digit Image**

◦ User selects an existing image file (e.g., PNG/JPEG) containing handwritten digits and uploads it to the system.

**4. UC4: Request Single‑Digit Prediction**

◦ After drawing or capturing/uploading an image, the user clicks “Predict 1 Digit”.

◦ The system sends the image to the backend, runs preprocessing and the CNN, and returns a single digit and confidence.

**5. UC5: Request Two‑Digit Prediction**

◦ After drawing two digits side‑by‑side, the user clicks “Predict 2 Digits”.

◦ The system splits the image into left and right halves, runs the CNN on each half, and returns a combined two‑digit string (e.g., “12”).

**6. UC6: View Prediction Result**

◦ User views the predicted digit(s), confidence percentage, and confidence bar in the UI.

◦ If an error occurs, an error message is displayed instead.

**7. UC7: Check System Health**

◦ When the application loads, the frontend automatically calls the /health endpoint.

◦ The user sees a small status notification indicating whether the model is loaded and the API is available (e.g., “Model loaded successfully” or “Flask API not running”)

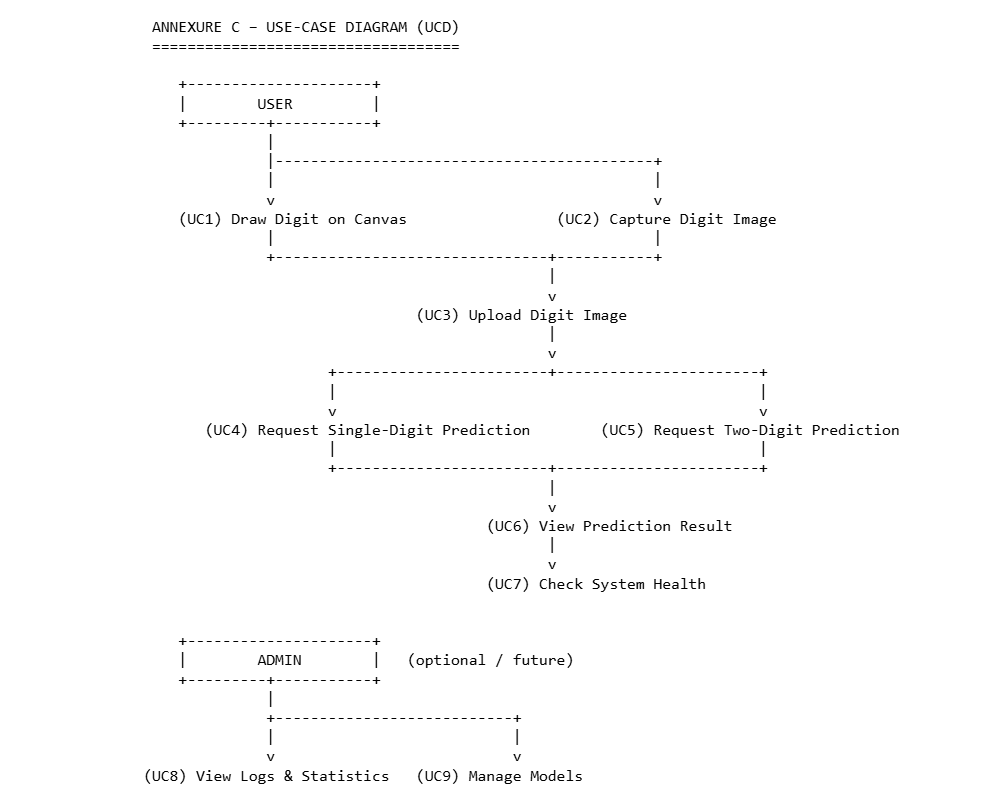
**For Admin (optional / future):**

**8. UC8: View Logs and Statistics**

◦ Admin views summary statistics such as number of predictions, error rate, and model performance.

**9. UC9: Manage Models**

◦ Admin uploads new model files, switches between model versions, or triggers retraining (future enhancement)

**C.4 Use‑Case Diagram **

**Annexure D**

**Data Dictionary (DD)**

**(Mandatory)**

# USER (USR)

| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **USR\_ID** | **Number** | **Unique ID of the user (primary key)** |
| **USR\_Name** | **Text** | **Name of the user** |
| **USR\_Email** | **Text** | **Email address of the user** |
| **USR\_Role** | **Text** | **Role of the user (Student/Teacher/Admin)** |

**2. IMAGE\_INPUT (IMG)**

| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **IMG\_ID** | **Number** | **Unique identifier for each input image (Primary Key)** |
| **IMG\_UserID** | **Number** | **Identifier of the user who submitted the image (Foreign Key to USR\_ID); optional, useful for traceability** |
| **IMG\_Source** | **Text** | **Origin of the image: whether captured from Canvas, Camera, or uploaded file** |
| **IMG\_Timestamp** | **Date/Time** | **When the image was created or captured** |
| **IMG\_RawPath** | **Text** | **Path or reference (file path or base64 string) to the raw image storage location** |

**3. PREPROCESSED\_IMAGE (PRC)**

| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **PRC\_ID** | **Number** | **Unique identifier of the preprocessed image (Primary Key)** |
| **PRC\_IMG\_ID** | **Number** | **Foreign key referring to the original raw image (IMG\_ID)** |
| **PRC\_Width** | **Number** | **Width in pixels of the processed image (usually 28 for MNIST-like input)** |
| **PRC\_Height** | **Number** | **Height in pixels (typically 28)** |
| **PRC\_Mean** | **Number** | **Mean pixel intensity after preprocessing, useful for analysis/debugging** |
| **PRC\_Std** | **Number** | **Standard deviation of pixel intensities after preprocessing** |
| **PRC\_Path** | **Text** | **Reference/path to the stored preprocessed image, if persisted** |

**4. PREDICTION\_RESULT (PRD)**

| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **PRD\_ID** | **Number** | **Unique identifier for each prediction record (Primary Key)** |
| **PRD\_IMG\_ID** | **Number** | **Foreign key linking to the original raw image (IMG\_ID)** |
| **PRD\_PRC\_ID** | **Number** | **Foreign key linking to the preprocessed image (PRC\_ID)** |
| **PRD\_ModelID** | **Number** | **Foreign key linking to the model used for this prediction (MDL\_ID)** |
| **PRD\_Prediction** | **Text** | **The predicted output digit or two-digit result (e.g., "7" or "12")** |
| **PRD\_Confidence** | **Number** | **Confidence score between 0 and 1 for the main predicted value** |
| **PRD\_AllProbs** | **Text** | **Serialized JSON string capturing confidences/probabilities for all classes (digits 0–9)** |
| **PRD\_Type** | **Text** | **Indicates the prediction type: either SingleDigit or TwoDigits** |
| **PRD\_Timestamp** | **Date/Time** | **Timestamp when the prediction was generated** |
| **PRD\_ErrorMessage** | **Text** | **Explanation of any error encountered during prediction; NULL if no error** |

**5. MODEL\_INFO (MDL)**

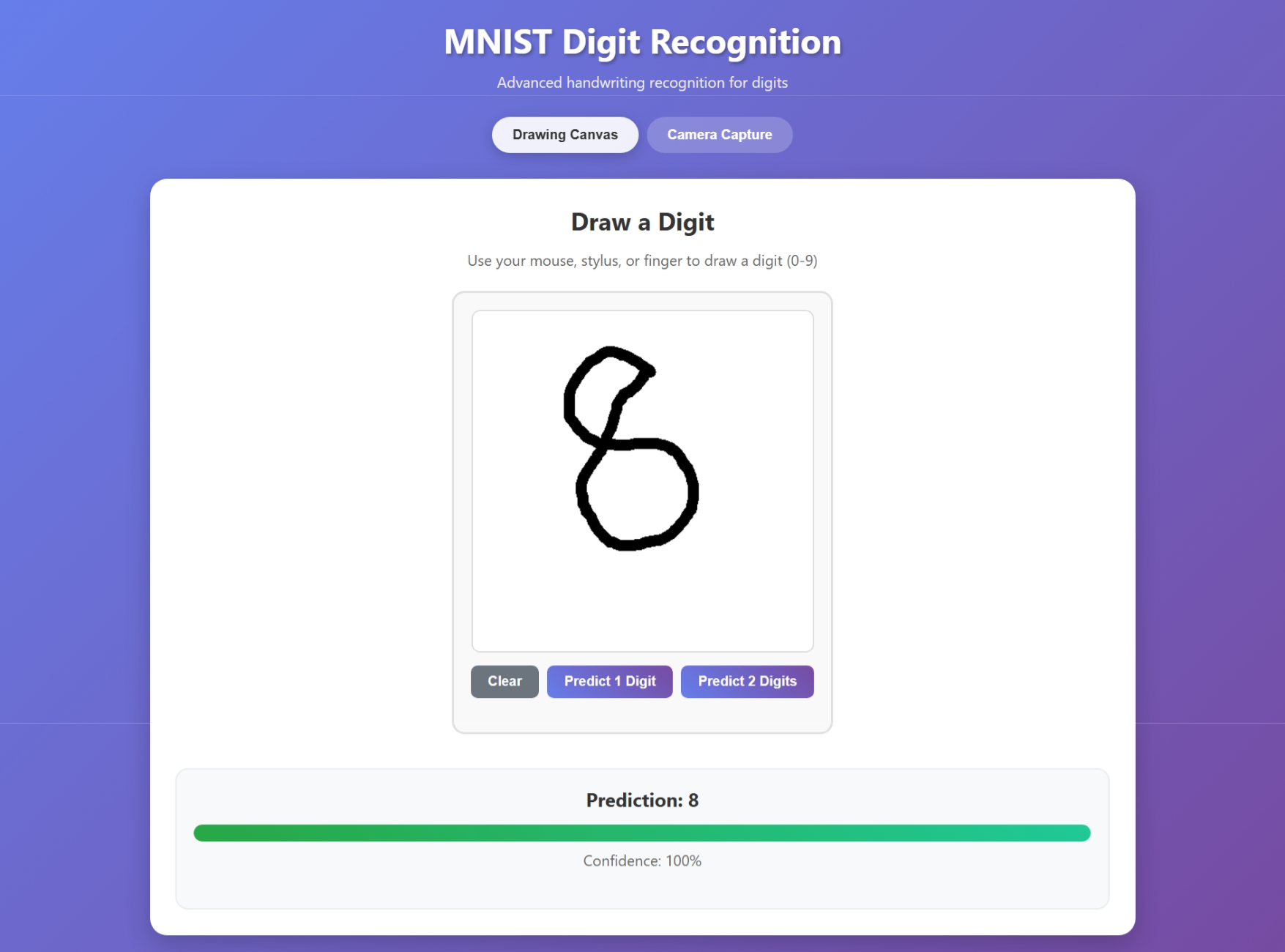
| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **MDL\_ID** | **Number** | **Unique identifier for each model, serves as the primary key** |
| **MDL\_Name** | **Text** | **Descriptive name of the model, e.g., ImprovedCNN** |
| **MDL\_Version** | **Text** | **Version identifier to track model updates, e.g., "1.0"** |
| **MDL\_Weights** | **Text** | **Filename or storage path for the model weights file, e.g., "mnist\_model\_best.pth"** |
| **MDL\_TrainedOn** | **Date** | **The date on which the model was trained** |
| **MDL\_Accuracy** | **Number** | **The test accuracy percentage achieved by the model during training** |

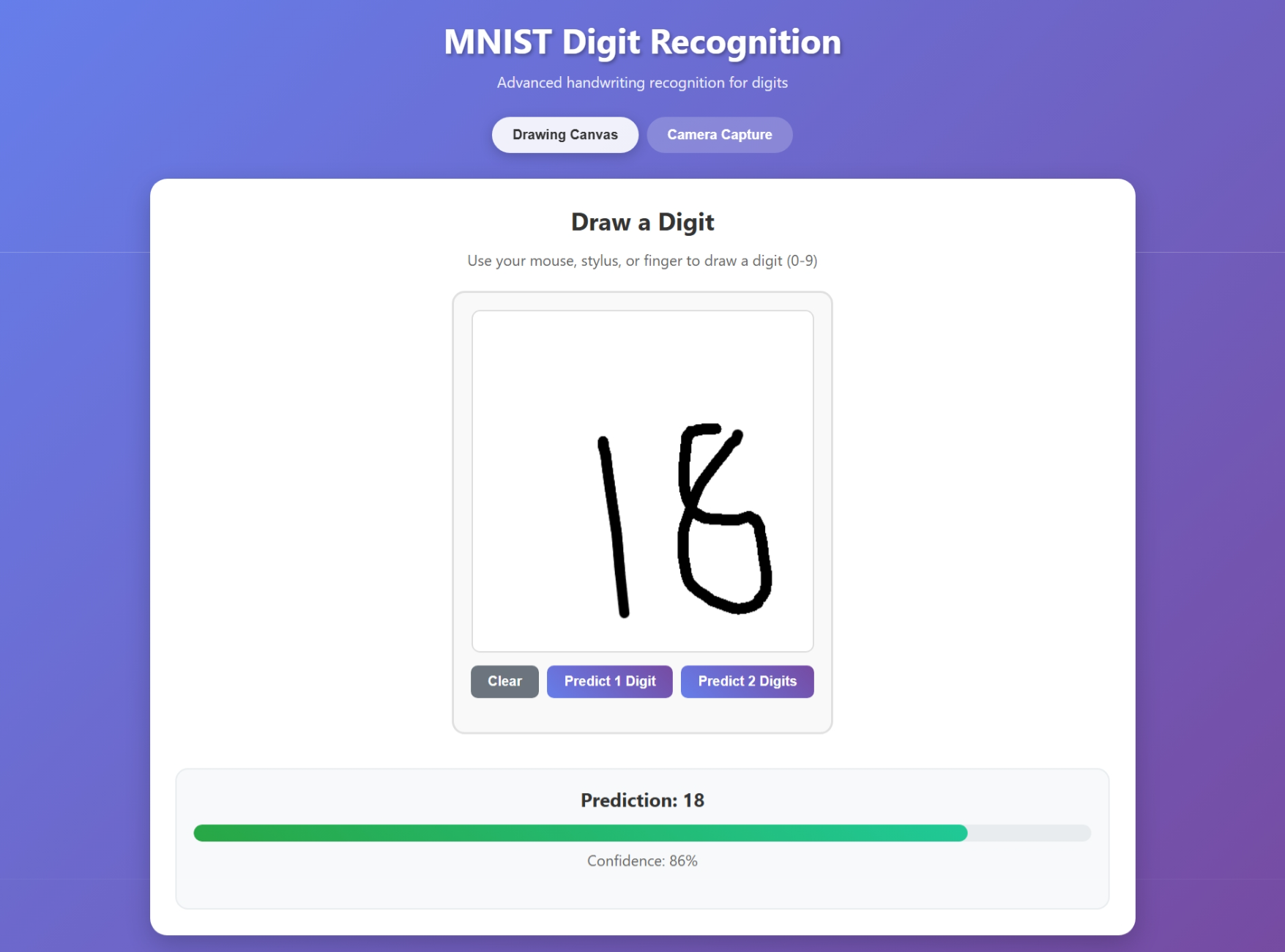
**6. HEALTH\_STATUS (HST) – Logical / Runtime**

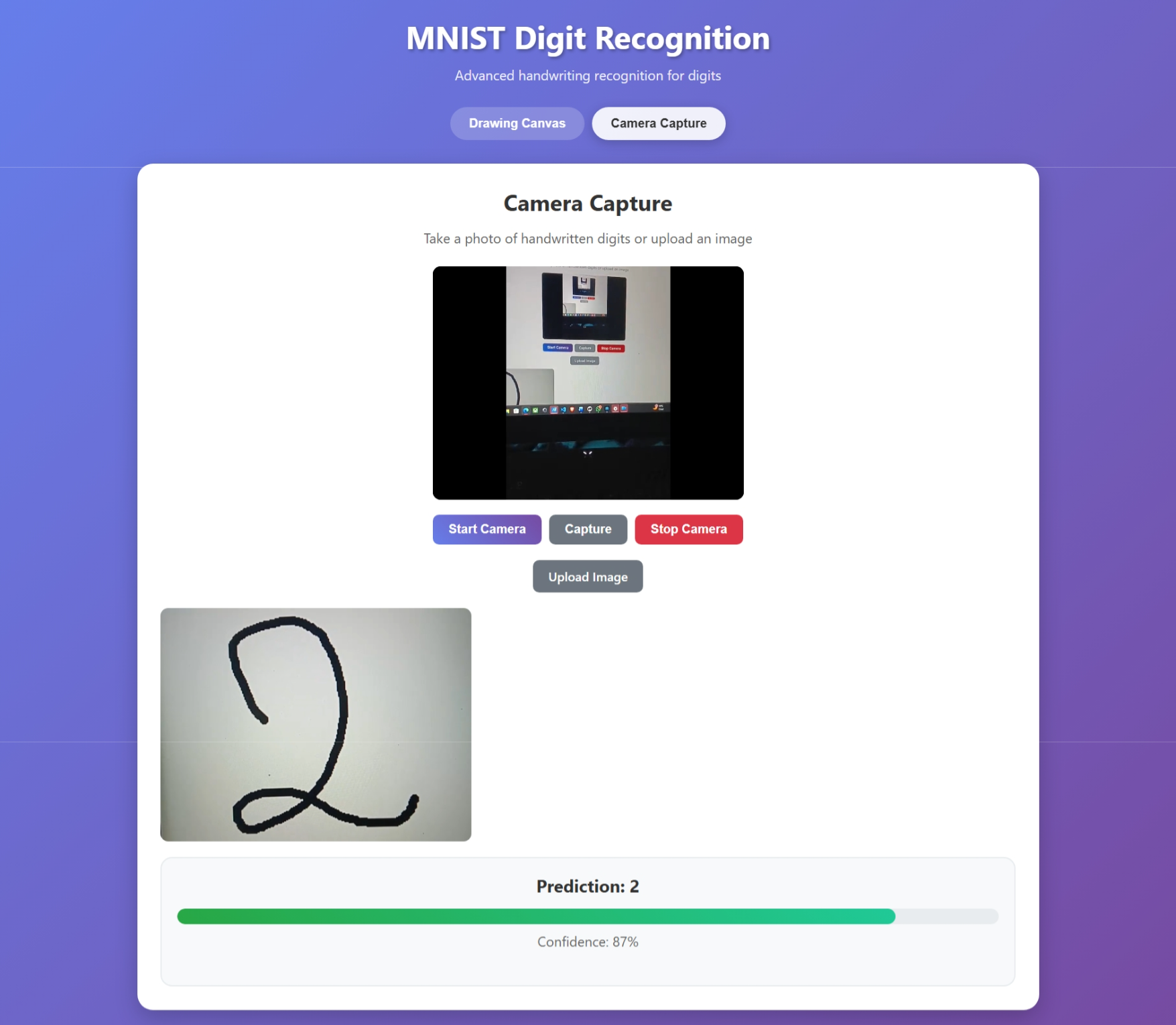
| **Field Name** | **Data Type** | **Description** |
| --- | --- | --- |
| **HST\_Status** | **Text** | **Overall health status of the system: "healthy" or "error".** |
| **HST\_ModelLoaded** | **Boolean** | **Indicates if the CNN model is currently loaded (true/false).** |
| **HST\_Device** | **Text** | **Specifies the device used for inference (e.g., "cpu" or "cuda").** |
| **HST\_Message** | **Text** | **Additional status messages or error details for diagnostics.** |

**Annexure E**

**Screen Shots**

****

****

****