PROJECT REPORT

on

Ink to AI: Advanced Handwritten Recognition using DL

Submitted in partial fulfillment of the requirements for the award of

BACHELOR OF ENGINEERING

IN

INFORMATION TECHNOLOGY

Submitted by

JAYAVARAPU SRI GOWRI SANKAR—23BQ5A1208

KOTA SAI KRISHNA—23BQ5A1209

MANEPALLI JASWANTH SAI—23BQ5A1210

Under the esteemed guidance of

Dr. N. ASHOK M.Tech, Ph.D;

Associate Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

NAMBUR(V),PEDAKAKANI(M),GUNTUR-522508,TELno:08732118036,

www.vvitguntur.com, approved by AICTE,permanently affiliated to JNTUK Accredited by NAAC with "A" grade, Accredited by NBA for 3 years

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

NAMBUR

BONAFIDE CERTIFICATE

Certified that this project report **"Ink to AI: Advanced Handwritten Recognition using DL" is** the bonafide work of by "**JAYAVARAPU SRI GOWRI SANKAR (23BQ5A1208), KOTA SAI KRISHNA(23BQ5A1209), MANEPALLI JASWANTH SAI(23BQ5A1210)**" who carried out the project under my guidance in the year 2025 towards partial fulfillment of the requirements of the Degree of Bachelor of Technology in Information Technology from Jawaharlal Nehru Technological University, Kakinada. The results embodied in this report have not been submitted to any other University for the award of any degree.

Signature of the Supervisor
Dr. N.ASHOK
Associate Professor , IT.

Signature of the Head of the Department
Dr. A. KALAVATHI
Professor ,IT.

Submitted for Viva voce Examination held on

EXTERNAL EXAMINER

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

NAMBUR

CERTIFICATE OF AUTHENTICATION

We solemnly declare that this project report **"Ink to AI: Advanced Handwritten Recognition using DL"** is the Bonafide work done purely by us ,carried out under the supervision of Dr. N.ASHOK, towards partial fulfillment of the requirements of the Degree of Bachelor of Technology in Information Technology from Jawaharlal Nehru Technological University, Kakinada during the year 2025.

It is further certified that this work has not been submitted, either in part of in full, to any other department of the Jawaharlal Nehru Technological University,or any other University, institution or else where, or for publication in any form.

Signature of the Student

JAYAVARAPU SRI GOWRI SANKAR– 23BQ5A1208 _____

KOTA SAI KRISHNA– 23BQ5A1209 _____

MANEPALLI JASWANTH SAI – 23BQ1A1210 _____

ACKNOWLEDGEMENT

We take this opportunity to express our deepest gratitude and appreciation to all those people who made this project work easier with words of encouragement, motivation, discipline, and faith by offering different places to look to expand my ideas and help me towards the successful completion of this project work.

First and foremost, we express our deep gratitude to **Mr. Vasireddy Vidyasagar**, Chairman,Vasireddy Venkatadri Institute of Technology for providing necessary facilities throughout the Information Technology program.

We express our sincere thanks to **Dr.Y.Mallikarjuna Reddy**, Principal, Vasireddy Venkatadri Institute of Technology for his constant support and cooperation throughout the Information Technology program.

We express our sincere gratitude to **Dr.A. Kalavathi**, Professor & HOD, Information Technology, Vasireddy Venkatadri Institute of Technology for her constant encouragement, motivation and faith by offering different places to look to expand my ideas. We would like to express our sincere gratitude to our guide **Dr.N.Ashok** for his insightful advice, motivating suggestions, invaluable guidance, help and support in successful completion of this project.

We would like to take this opportunity to express our thanks to the **teaching and non-teaching** staff in the Department of InformationTechnology,VVIT for their invaluable help and support.

J.SRI GOWRI SANKAR
K.SAI KRISHNA
M.JASWANTH SAI

ABSTRACT

Handwritten Digit and Character recognition is a key problem in **artificial intelligence** with many applications in different areas, such as **postal automation, banking, data entry**, and analysis of historical documents. The problem, though much eased by progress in algorithms, presents ongoing challenges due to **varied handwriting styles, different character sets, and inherent noise**, which continue to challenge reliable recognition systems.

This abstract introduces a **deep learning-based solution** for solving these issues, taking advantage of the **feature extraction strength of Convolutional Neural Networks (CNNs)**. Our designed system will be able to obtain high **accuracy and efficiency** for handwritten digit and character classification using optimized network structures, state-of-the-art training methods, and **robust data augmentation strategies**. The emphasis will be placed on designing a **robust model that can generalize** to a broad range of handwritten data so as to improve the **automation and digitization** of handwritten data.

This research specifically leverages a **multi-layer CNN architecture**, including optimized kernel sizes and **dropout layers**, to mitigate overfitting and capture intricate variations in handwriting. Furthermore, the model's performance will be **rigorously evaluated against benchmark datasets**, such as EMNIST and extensions including more complex character sets, to validate its generalization capability. Our proposed system aims to **significantly outperform** traditional machine learning techniques and existing deep learning models in terms of both **recognition accuracy and inference speed**.

The successful deployment of this solution promises to **streamline data processing workflows** across various industries, dramatically **reducing manual effort** and potential human error in data transcription. Ultimately, this work contributes a **highly efficient and accurate method** toward the ultimate goal of fully automating the recognition of diverse handwritten scripts.

Keywords- Deep Learning, Convolutional Neural Networks (CNNs), feature extraction, Handwritten Recognition, Generalization, Accuracy.

TABLE OF CONTENTS

CHAPTER -1

1.1 INTRODUCTION

Handwritten character and digit recognition is a classic and fundamental problem in the field of computer vision and pattern recognition. The task involves teaching a machine to accurately identify and interpret handwritten text or numerals, a challenge due to the vast diversity in human handwriting styles, sizes, and orientations. Historically, this problem was tackled using traditional machine learning techniques, which required extensive feature engineering and often struggled to achieve high accuracy.

The emergence of **deep learning** has revolutionized the approach to this problem. Specifically, **Convolutional Neural Networks (CNNs)** have proven to be exceptionally effective for image-based tasks like handwriting recognition. CNNs are a class of neural networks designed to automatically and adaptively learn spatial hierarchies of features from input data, such as images.

This ability to learn features directly from raw pixels, without the need for manual feature extraction, has led to significant breakthroughs. By leveraging multiple layers of convolutions, pooling, and fully connected layers, CNNs can capture intricate patterns and subtle variations in handwritten characters, leading to state-of-the-art performance and enabling the development of robust, real-world recognition systems.

1.2 PROBLEM STATEMENT

The problem statement for handwritten character and digit recognition is as follows:

Before the rise of deep learning, systems for handwriting recognition relied on **hand-crafted features**. This meant that human experts had to manually identify and program the specific characteristics of each character (e.g., lines, curves, loops). This approach faced several key challenges:

Variability in Handwriting: Human handwriting is highly inconsistent, varying in style, size, slant, and stroke thickness. Traditional methods struggled to generalize across these vast differences, leading to low accuracy.

Manual Feature Engineering: The process of defining features was labor-intensive, time-consuming, and required deep domain expertise. Any change or addition to the problem (e.g., a new language or font) meant the entire feature extraction process had to be re-engineered.

Segmentation Issues: Cursive handwriting, where characters are connected, presented a major problem. Traditional methods often required a separate, complex step to "segment" or separate individual characters before they could be classified, a process prone to errors.

CHAPTER-2

SYSTEM ANALYSIS

2.1-REQUIREMENT ANALYSIS

2.2.1 Non-Functional Requirements

Performance Requirements: The system is designed to use an optimized CNN network structure and efficient training methods to achieve high efficiency in handwritten digit and character classification.

Software Quality Attributes:

Accuracy :A primary goal is to achieve high accuracy for handwritten digit and character classification, with one related literature survey noting a hybrid CNN-SVM model achieved an accuracy of 99.28%.

Scalability: The architecture involves a distributed Data/Train Cluster with Training Worker(s) and a Dataset Store, suggesting a design intended to handle large datasets and scale training operations.

Reliability: The system emphasizes the design of a robust model that can generalize to a broad range of handwritten data to improve the reliability of the recognition system.

2.2 Hardware Requirements

The **Ink to AI** project uses a **dual-architecture setup** to manage its intensive Deep Learning operations. Training requires a **GPU Node** within the Data/Train Cluster, leveraging **CUDA/cuDNN** for efficient, large-scale CNN matrix calculations. For user-facing **real-time recognition**, an **Edge/Inference Server** with dedicated server GPUs ensures high-performance, low-latency classification. This approach optimizes both model development efficiency and responsive service delivery for the Web/Mobile UI, enhancing the automation and digitization of handwritten data.

NETWORK : 10 Mbps

STORAGE : 100 GB

RAM : 8 GB

PROCESSOR : Intel core i3 or above

GPU:For Better Performance

2.3 Software Requirements

I. Frontend Requirements:

The frontend is responsible for capturing the user's handwriting and displaying the results.

Component	Technology	Purpose in Project
User Interface Framework	**React.js**	Provides a responsive, fast-loading interface for users to interact with the system.
Input Method	**HTML Input Type="File"**	Enables the user to **Upload Image** of the handwritten character or digit.
Image Preprocessing (Client-Side)	**JavaScript Libraries**	Performs immediate client-side steps like converting the canvas drawing to a **grayscale** image and **resizing it to 28x28 pixels** before sending it to the backend for recognition.
Communication	**JavaScript Fetch API / Axios**	Sends the preprocessed image data to the Backend's REST API endpoint as a POST request and handles the real-time prediction response.

II. Backend Requirements

The backend hosts the Deep Learning model, manages the REST API, and handles core prediction logic. The project's architecture suggests a REST Controller running on a Python server.

Component	Technology	Purpose in Project
Core Programming Language	**Python**	Used for data manipulation, running the machine learning model, and implementing the API logic.
Web/API Framework	**Flask, Django, or FastAPI**	**Flask** (Micro-framework, simple for serving the inference API) or **Django** (Full-featured, good for modular scaling). FastAPI is an alternative known for high speed.
Deep Learning Framework	**TensorFlow/Keras**	Loads the trained model (.h5 or .pkl files) and executes the predict() function for real-time inference.
Model Serving Libraries	**NumPy**	**NumPy** for manipulating tensors (or Keras utility functions) for any final server-side preprocessing (e.g., thinning, slant correction).
GPU Acceleration	**NVIDIA CUDA Toolkit** and **cuDNN**	Essential to enable the deep learning frameworks (TensorFlow/PyTorch) to utilize the GPU Node for fast training and high-performance inference.

III. Data Base Requirements

The project requires storage for three distinct purposes: training data, model artifacts, and operational logs.

Component	Technology	Purpose in Project
Training Dataset	**EMNIST**	The foundational dataset used to train and validate the CNN model.
Results/Logs Database	**PostgreSQL, MySQL** (Relational)	Used to store operational data, such as system performance logs (latency, throughput), training results (accuracy, confusion matrices), and potentially a log of user predictions/errors for future model improvement (Admin function).

IV.Development Environment:

Integrated Development Environment (IDE): PyCharm or Visual Studio Code (VS Code)or Google CoLab are powerful IDEs suitable for managing the Python and JavaScript components of the full-stack application.

Version Control: Git/GitHub is essential for tracking changes, managing the code repository, and facilitating collaboration.

Containerization: Docker is the primary tool for containerizing the application components (Frontend, Backend REST Controller, Inference Service) to ensure consistency between development and deployment environments.

2.4 Existing System

Classical machine learning (non-deep learning) systems for handwritten recognition follow a two-stage process: first, feature extraction converts raw image data into descriptive numerical features. These features are then fed into a separate classification algorithm. Popular choices for these classifiers include Support Vector Machines (SVMs), which use an optimal hyperplane to classify data in a high-dimensional space. Other options are the simple, non-parametric K-Nearest Neighbors (KNN) model , or traditional Multilayer Perceptrons (MLPs). Notably, MLPs lack the specialized convolutional layers of deep learning, requiring features to be manually engineered and fed to the network.

2.5 Proposed Solution

The proposed CNN methodology for handwritten recognition is executed in two main phases: data preparation and model architecture/training. The data preparation phase involves collecting datasets like MNIST/EMNIST, preprocessing images by resizing them to 28×28 pixels, converting to grayscale, and normalizing pixel values. The data is then divided into training, validation, and test sets for unbiased evaluation. The second phase focuses on the CNN model, utilizing convolutional layers for feature extraction and the ReLU function for non-linearity. This is followed by pooling and flatten layers to reduce dimensions and prepare the data for the final fully connected layers. The output layer uses a Softmax function to produce classification probabilities. The system is trained using the Adam optimizer and a categorical cross-entropy loss function over multiple epochs, adjusting weights via backpropagation. Finally, model evaluation uses unseen data and metrics like accuracy and a confusion matrix to measure generalization and ensure system reliability.

2.6 Modules

1. User Interface / Client (User Focus)

The **User Interface (UI) / Client** is the system's entry point, designed to facilitate interaction with the end-user. Its primary function is to allow the **User** to either capture or upload an image containing the handwritten character or digit. After processing, this module receives the classification result and confidence score from the backend and presents them to the user. Furthermore, it provides the functionality for the **User** to export or save the recognized output.

2. Controller (Workflow Orchestration)

The **Controller** serves as the central orchestration logic of the application, managing the communication between the front-end and the deep learning services. It receives the initial image classification request from the **User** and directs the image through the necessary preprocessing steps before triggering the prediction in the CNN Model. The Controller also manages administrative tasks, such as handling the **Admin's** command to initiate the model training process and delivering the final evaluation report.

3. Preprocessor (Image Standardization)

The **Preprocessor** module is essential for data standardization, ensuring that any input image—whether from a user or the training dataset—is in the correct format for the model. When a new image is uploaded, it handles critical preliminary steps like **resizing** the image (e.g., to 28×28 pixels), converting it to **grayscale**, and **normalizing** the pixel values. For more robust handwriting, it includes complex steps like image thresholding, thinning, slant correction, and segmentation.

4. Dataset Manager (Admin Data Prep)

The **Dataset Manager** is responsible for all operations related to the training and testing data required for the deep learning model. Its main function, utilized by the **Admin**, is to load the necessary datasets (such as MNIST or EMNIST) and logically split them into distinct training, validation, and test sets to ensure model generalization is measured fairly. It ensures the data is ready to be consumed by the training process.

5. CNN Model (Core Intelligence)

The **CNN Model** is the core intelligence of the project, embodying both the feature extraction and classification logic. It is the component that the **Admin** trains over multiple epochs to minimize prediction error. When a user requests a prediction, the CNN Model executes a forward pass on the preprocessed image to produce the raw probabilities of the recognized character.

6. Evaluator (Admin Performance Audit)

The **Evaluator** module functions exclusively to support the **Admin's** oversight of the model's quality. After the training loop is complete, it assesses the generalization ability of the trained model by testing it on unseen data. Its primary functions include calculating key performance metrics, such as **accuracy**, and generating a **confusion matrix**.

7. Decision Algorithm (Final Prediction)

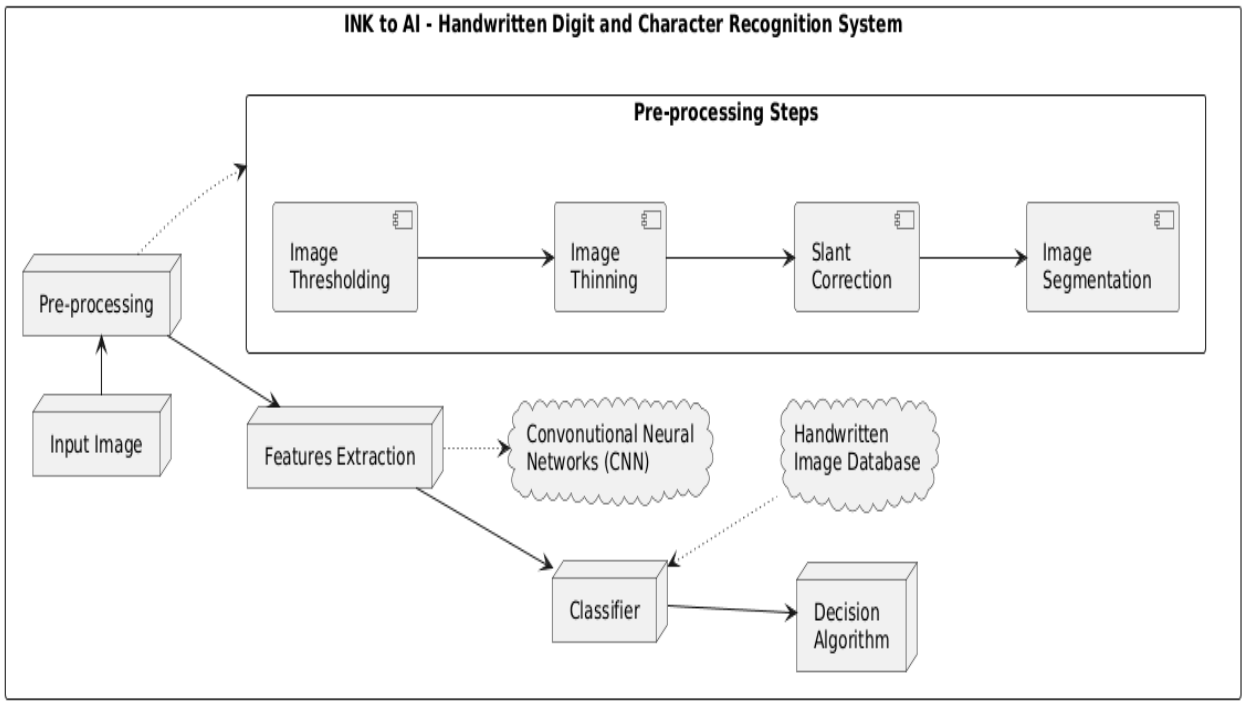
The **Decision Algorithm** is the final step in the inference chain, responsible for translating the model's output into a definitive answer. It processes the probability vector generated by the CNN's Softmax output layer. The algorithm then selects the character or digit with the highest probability score as the final, recognized result to be returned to the user.

Chapter 3

SYSTEM DESIGN

3.1 System Architecture

A System Architecture is the conceptual model that defines the structure, behavior, and more views of a system. An architecture description is a formal description and representation of a system, organized in a way that supports reasoning about the structures and behaviors of the system. A System Architecture can consist of system components and the sub-systems developed, that will work together to implement the overall system.



3.2 UML Diagrams

UML is an acronym that stands for Unified Modeling Language. Simply put, UML is a modern approach to modeling and documenting software. In fact, it's one of the most popular business process modeling techniques. It is based on diagrammatic representations of software components.

As the old proverb says: "a picture is worth a thousand words". By using visual representations, we are able to better understand possible flaws or errors in software or business processes. Mainly, UML has been used as a general-purpose modeling language in the field of software engineering. However, it has now found its way into the documentation of several business processes or workflows.

For example, activity diagrams, a type of UML diagram, can be used as a replacement for flowcharts. They provide both a more standardized way of modeling workflows as well as a wider range of features to improve readability and efficacy. Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding.

There are two broad categories of diagrams and they are again divided into subcategories –

- Structural Diagrams
- Behavioral Diagrams

Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable. These static parts are represented by classes, interfaces, objects, components, and nodes.

The four structural diagrams are –

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram

Behavioral Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered. Behavioral diagrams basically capture the dynamic aspect of a system. UML has the following five types of behavioral diagrams

- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Use Case Diagram:

A **use case diagram** is a foundational tool in system analysis and design. At its core, it's a simplified, graphical representation of a user's interaction with a system, illustrating the **relationship between users (or actors)** and the **different use cases** they are involved with. Use cases themselves are typically represented by **circles or ellipses**.

Purpose and Value

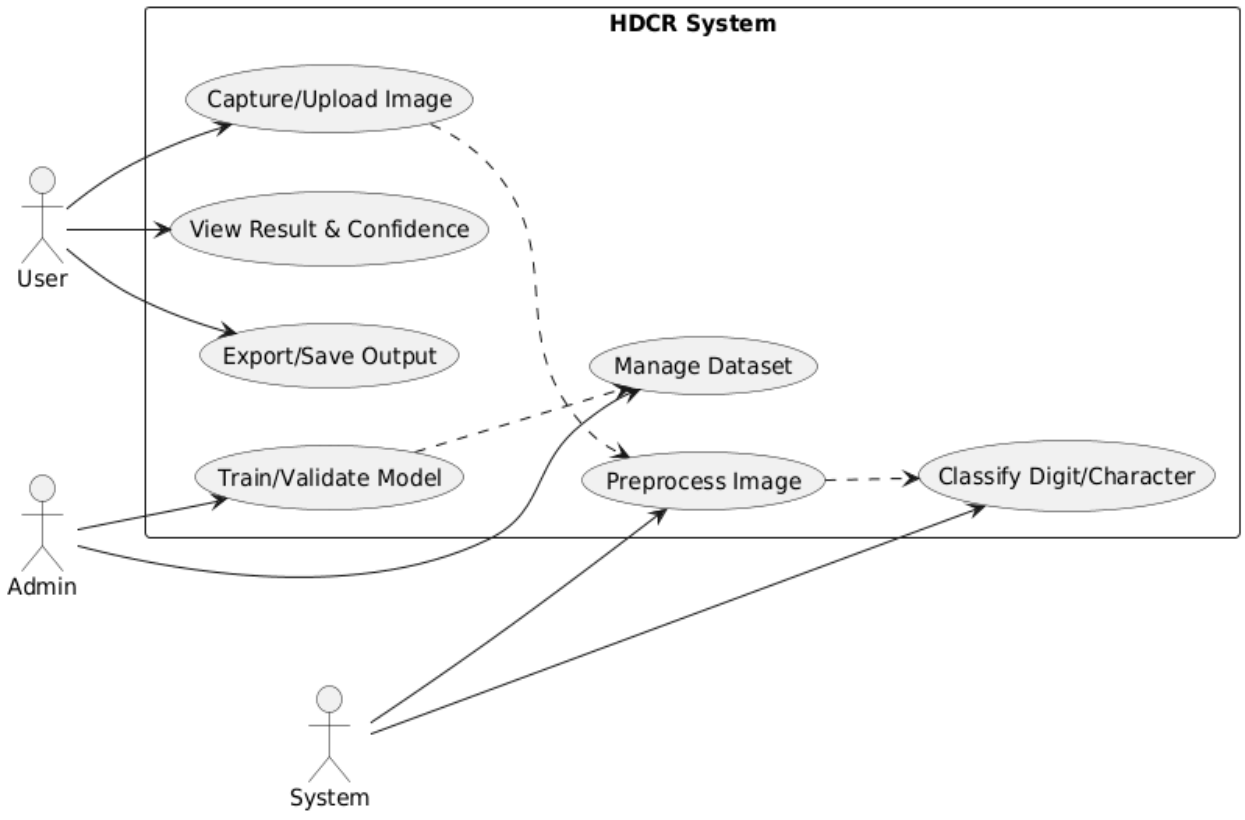
While a single use case may delve into extensive detail about every possibility, the diagram provides a crucial **high-level view** of the system's intended functionality. It's often referred to as the "**blueprint for your system**" because it offers a clear, simplified map of what the system must actually accomplish.

The primary purposes of a use case diagram are:

- **Gathering System Requirements:** They help to capture the functionalities and **design requirements** of a system, including both **internal and external influences**. By analyzing the system's intended functions, use cases are prepared and the necessary **actors** are identified.
- **Providing an Outside View:** Once initial analysis is complete, these diagrams are modeled to present the external, or **outside view**, of the system.
- **Stakeholder Communication:** Due to their simple, real-world mimicking nature, they serve as an excellent communication tool. They help **stakeholders** understand how the system is going to be designed by conveying the **requirements in laypeople's terms**.
- **Identifying Influences:** They explicitly help identify the **external and internal factors** influencing the system and show the **interactions among the requirements and actors**.

In essence, the use case diagram presents a clear, initial overview. Additional documentation and diagrams are then used to build upon this foundation and provide a complete functional and technical view of the system.

USE CASE DIAGRAM:



Sequence Diagram:

A **sequence diagram** is a type of **UML (Unified Modeling Language)** interaction diagram that illustrates how **objects interact with each other** and in what **order** those interactions occur. It visually represents the **time sequence** of events in a specific scenario, often associated with the realization of a use case. They are sometimes also referred to as **event diagrams** or **event scenarios**.

Core Components

A sequence diagram is built from the following core elements:

1. Lifelines (Vertical Lines)

- These **parallel vertical lines** represent the different **processes, objects, or classes** that are involved in the scenario and exist simultaneously.
- If a lifeline represents an **object**, it demonstrates a role. Leaving the instance name blank (e.g., just the Class Name) indicates an **anonymous or unnamed instance**.
- **Destruction:** If an object is removed from memory, its lifeline ends with an **'X'** drawn at the bottom, and the dashed line ceases below that point.

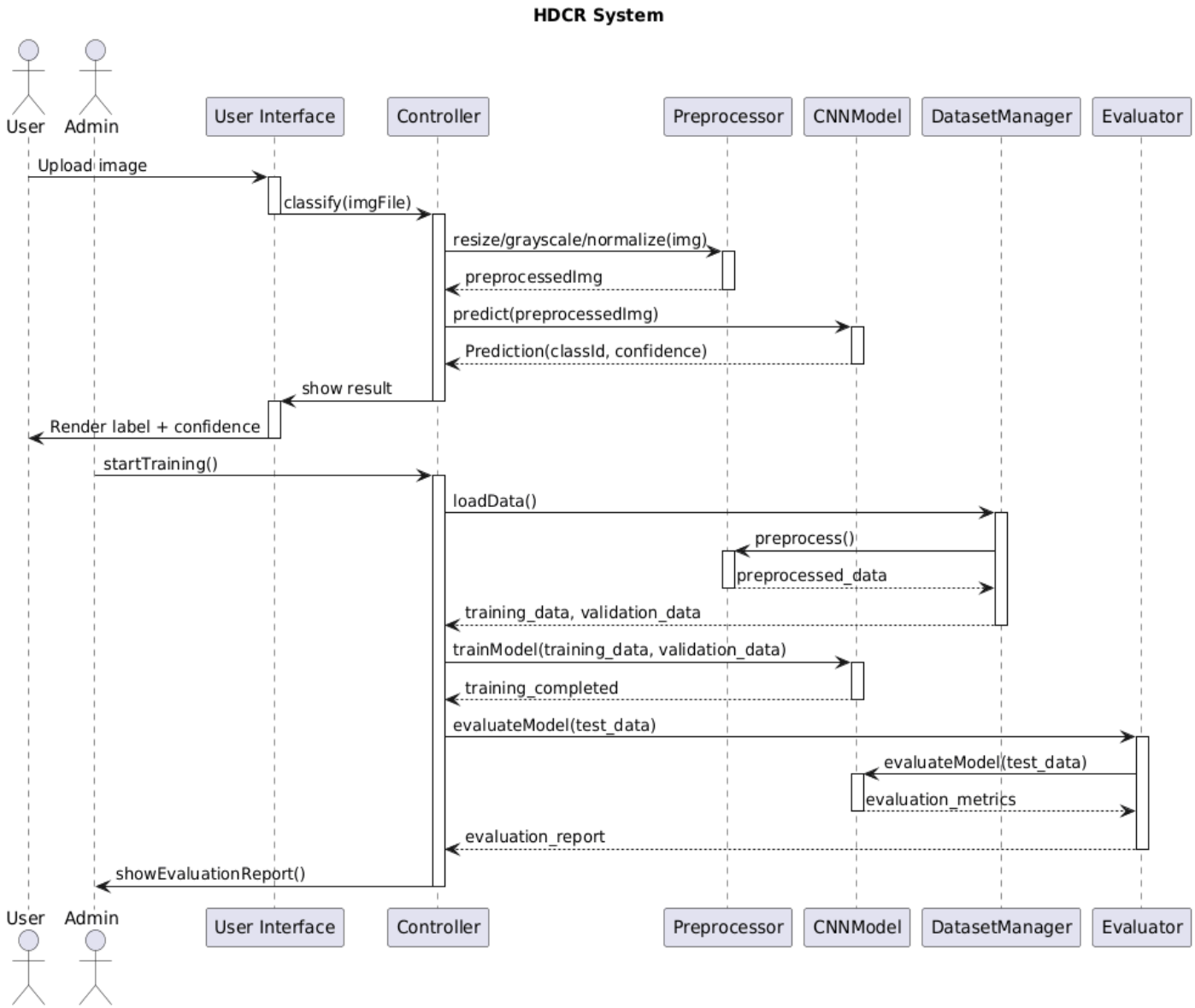
2. Messages (Horizontal Arrows)

- **Horizontal arrows** show the **messages** (or method calls) exchanged between the objects, arranged in the sequential order in which they occur. The message name is written above the arrow.
- **Synchronous Calls:** Represented by a **solid arrowhead** ($\blacktriangleright$). When a caller sends a synchronous message, it **must wait** for the called method to complete and return a result before it can continue processing (like invoking a subroutine).
- **Asynchronous Messages:** Represented by an **open arrowhead** ($\rightarrow$). When a caller sends an asynchronous message, it **does not wait** for a response and can continue processing immediately. These are common in **multithreaded, event-driven applications**, and **message-oriented middleware**.
- **Reply Messages (Returns):** Represented by a **dashed line** with an open arrowhead ($\dashrightarrow$). This indicates the return of control or a value back to the caller object.

3. Activation Boxes (Execution Specifications)

- These are **opaque rectangles** drawn on top of a lifeline.
- They represent the **period during which an object is performing an operation** or is "active" in response to a message. In UML, this is formally called an **Execution Specification**.
- When an object calls a method on **itself (self-delegation)**, a new, smaller activation box is added on top of the existing one to indicate a further level of processing.

SEQUENCE DIAGRAM:



Class Diagrams:

A **Class Diagram** is a fundamental type of **static structure diagram** in the **Unified Modeling Language (UML)**. It serves as the **main building block of object-oriented modeling** by describing the **structure of a system**.

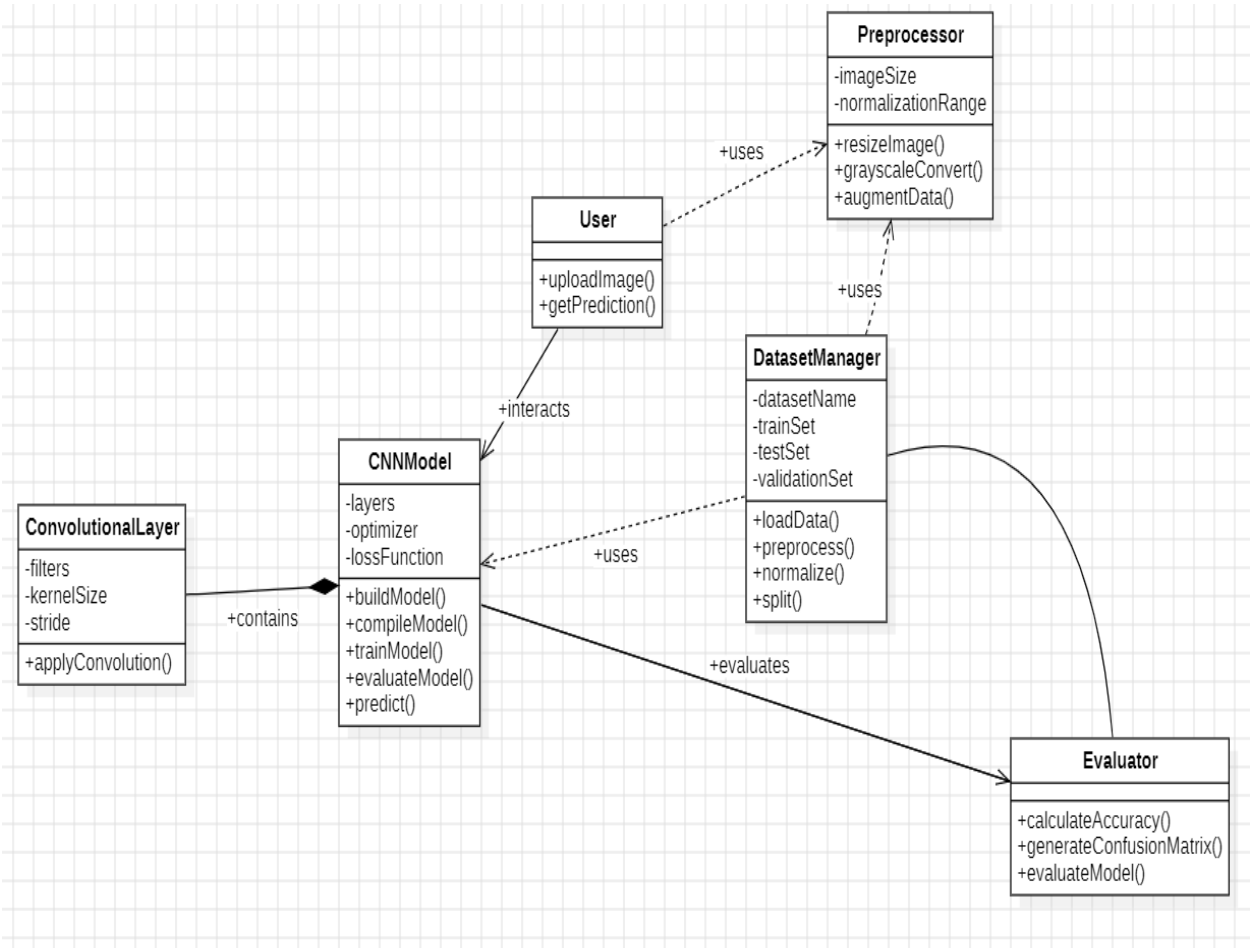
It achieves this by showing:

1. The system's **classes**.
2. The **attributes** (data/properties) each class holds.
3. The **operations** (or methods/behavior) each class can perform.
4. The **relationships** (associations, inheritance, dependencies) among these classes.

Purpose and Use

Class diagrams are highly versatile and are used for various purposes across the software development lifecycle:

- **Conceptual Modeling:** They are used for general conceptual modeling of the structure of an application, providing a blueprint of the system's architecture.
- **Detailed Design:** They are essential for detailed modeling, directly translating the design into programming code. The classes in the diagram represent the main elements, interactions, and the actual classes to be programmed.
- **Data Modeling:** They can also be effectively used for data modeling, representing the structure of a database or information system.

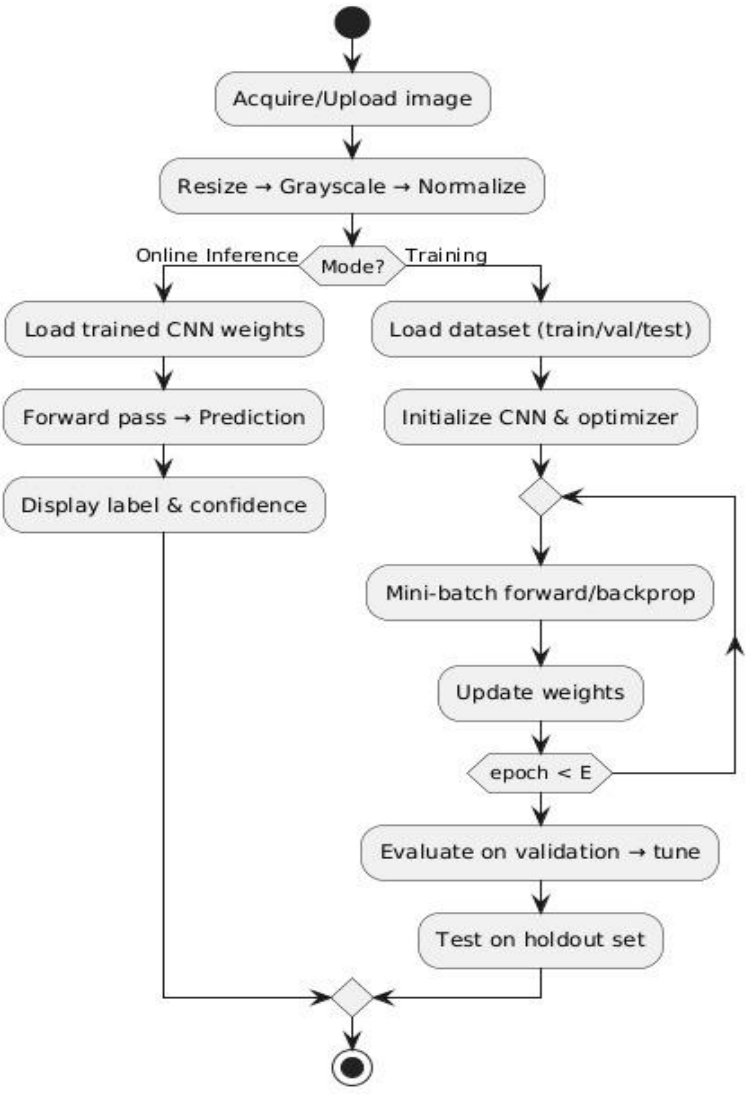


Activity Diagrams:

Activity Diagrams are a type of **behavioral diagram** in the Unified Modeling Language (**UML**) used to visually illustrate the **flow of control** in a system. They essentially depict **workflows** and the steps involved in the execution of a **use case**.

Key Aspects of Activity Diagrams

- **Focus on Flow:** An activity diagram focuses on the **conditions of flow** and the **sequence** in which activities happen. They describe or depict what causes a particular event.
- **Modeling Activities:** They model the steps involved in an operation, showing the transition from one activity to the next.
- **Start to Finish:** They portray the control flow from a **start point** to a **finish point**, showing the various **decision paths** that exist while the activity is being executed.
- **Sequential and Concurrent Processing:** Activity diagrams can depict both **sequential processing** (activities happening one after another) and **concurrent processing** (activities happening at the same time) of activities.
- **Application:** They are widely used in **business and process modeling** where their primary use is to depict the dynamic aspects of a system.
- **Similarity to Flowcharts:** An activity diagram is very similar to a **flowchart** in its visual representation of a process.



State Diagrams:

A **State Machine Diagram**, also known as a **State Chart** or **State Transition Diagram**, is a UML behavioral diagram that shows the **order of states** that an **object** or component undergoes within a system.

It is a powerful way to capture and model the behavior of **event-based systems**, focusing on how the state of an object changes in response to specific events.

Types of State Machine Diagrams

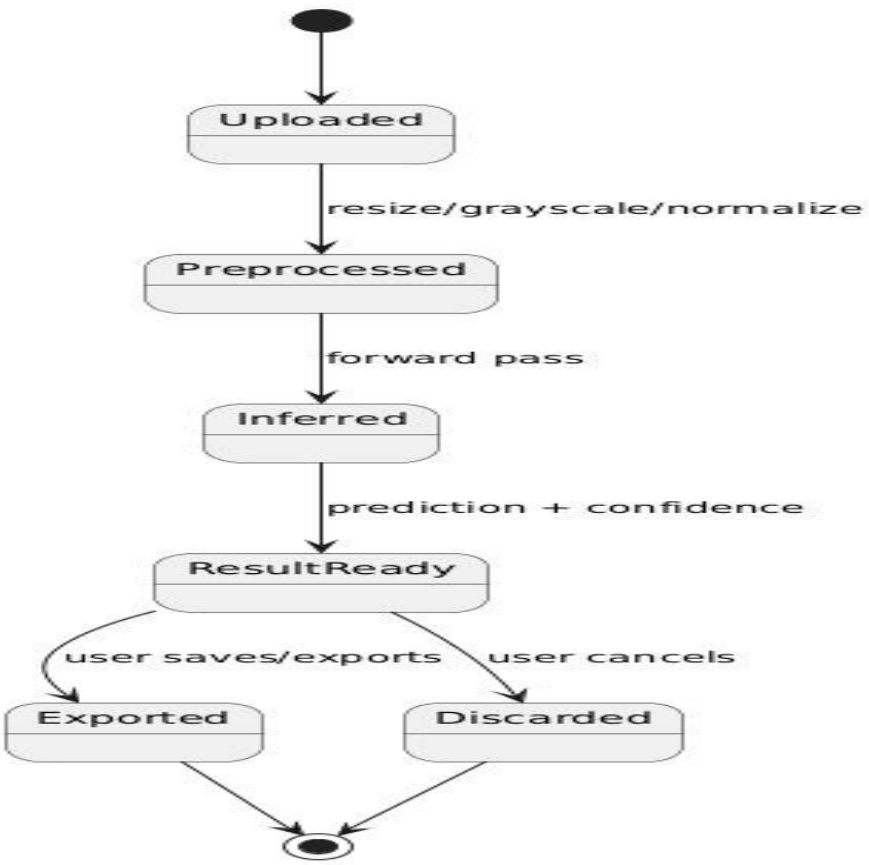
UML defines two main types of state machine diagrams:

1. Behavioural State Machine

- **Purpose:** This type records the **behavior of an object** within the system.
- **Focus:** It depicts the **implementation** of a particular entity or component.
- **Goal:** It fundamentally models the **behavior of the system** as a whole through the lens of its constituent objects' states.

2. Protocol State Machine

- **Purpose:** This type captures the **behavior of a protocol** (e.g., communication protocol, usage protocol).
- **Focus:** It depicts the change in the state of the protocol and **parallel changes** within the system.
- **Limitation:** It focuses on the allowable sequence of operations and **does not portray the internal implementation** details of a particular component, unlike the Behavioral State Machine.



CHAPTER 4

SYSTEM IMPLEMENTATION

4.1 Technologies Used

1.Programming Language

Python

Python is the primary programming language used for this project due to its simplicity, extensive machine learning ecosystem, and strong community support. Python enables seamless integration of deep learning frameworks, visualization libraries, and dataset utilities.

2. Deep Learning Frameworks

TensorFlow & Keras

TensorFlow and its high-level API Keras are used for designing, training, and evaluating the Convolutional Neural Network (CNN).

Key functionalities:

- Implementation of **LeNet-5 architecture**
- Use of layers such as **Conv2D**, **Pooling**, **Flatten**, and **Dense**
- Model training using **Adam optimizer**
- Loss computation using **Categorical Crossentropy**
- GPU-accelerated training support (via Colab)

These frameworks simplify model creation while providing flexibility for experimentation.

3. Dataset Technology

EMNIST (Extended MNIST) Dataset

The project uses the EMNIST dataset, a large-scale extension of MNIST containing:

- Handwritten digits (0–9)
- Uppercase letters (A–Z)
- Lowercase letters (a–z)

Its 28×28 grayscale images make it ideal for training character recognition systems and evaluating model generalization across diverse handwriting styles.

4. Scientific Computing & Data Processing Libraries

NumPy

NumPy is used for:

- Numerical computations
- Reshaping and normalizing image arrays
- Label encoding and preprocessing

Its ability to efficiently handle large matrices accelerates model training workflows.

TensorFlow Datasets (TFDS)

Used for:

- Loading EMNIST
- Splitting data into train/test sets
- Automatic preprocessing utilities

TFDS ensures seamless dataset handling with minimal code.

5. Visualization Tools

Matplotlib

Matplotlib is used for:

- Plotting training accuracy and loss
- Visualizing sample images
- Displaying predicted vs. true labels

These visual tools help in analyzing model performance and debugging.

6. Environment & Execution Platform

Google Colab

Google Colab provides:

- Free GPU/TPU acceleration
- Notebook-based execution
- Cloud storage integration
- Easy dependency management

4.2 Approach and Implementation

1. Problem Understanding

Handwritten character recognition is a complex task due to:

- Variations in handwriting styles
- Differences in stroke width, slant, and spacing
- Noise and distortions in images

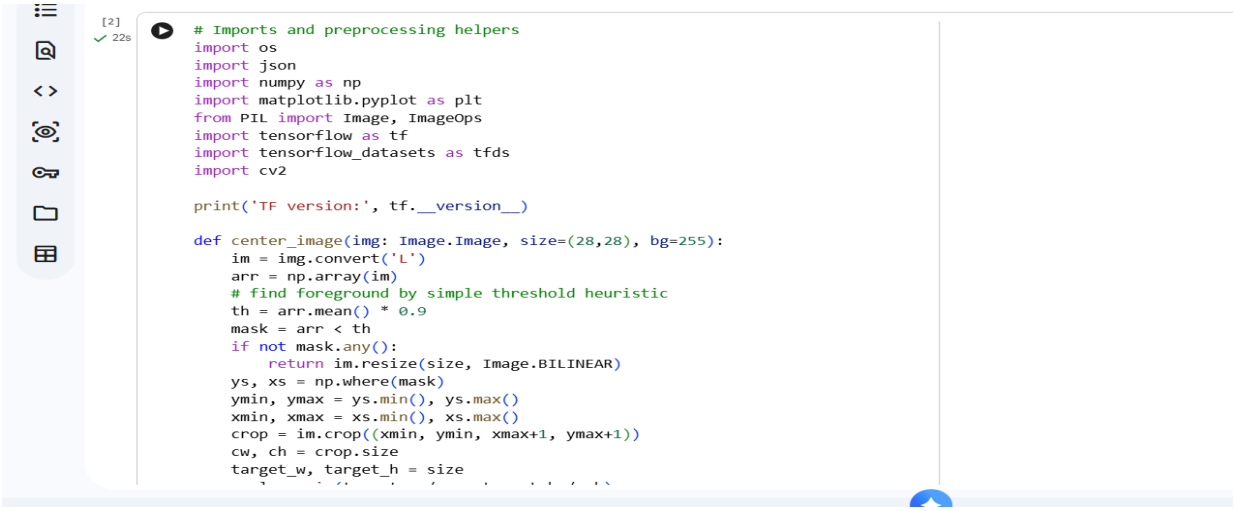
To address these challenges, the project adopts a **Convolutional Neural Network (CNN)**–based approach, particularly inspired by the classical **LeNet-5 architecture**, which is well suited for digit and character recognition tasks.

2. Proposed Approach

The proposed system uses the following deep-learning pipeline:

Step 1: Dataset Acquisition

- EMNIST dataset is used, which includes digits (0–9), uppercase (A–Z), and lowercase (a–z) characters.
- Loaded using **TensorFlow Datasets (TFDS)**.
- Dataset is split into **training** and **testing** sets.



Step 2: Data Preprocessing

To ensure consistency and improve model accuracy, several preprocessing steps are performed:

- **Reshaping**
  Convert images to size **28×28×1** to match CNN input.
- **Normalization**
  Pixel values scaled from **0–255** to **0–1** to enhance training stability.

- **One-Hot Encoding**
  Convert labels into categorical form for multi-class classification.
- **Optional Rotation Correction**
  EMNIST images may require transposing and flipping depending on the split.

These steps help reduce model overfitting and improve feature learning.

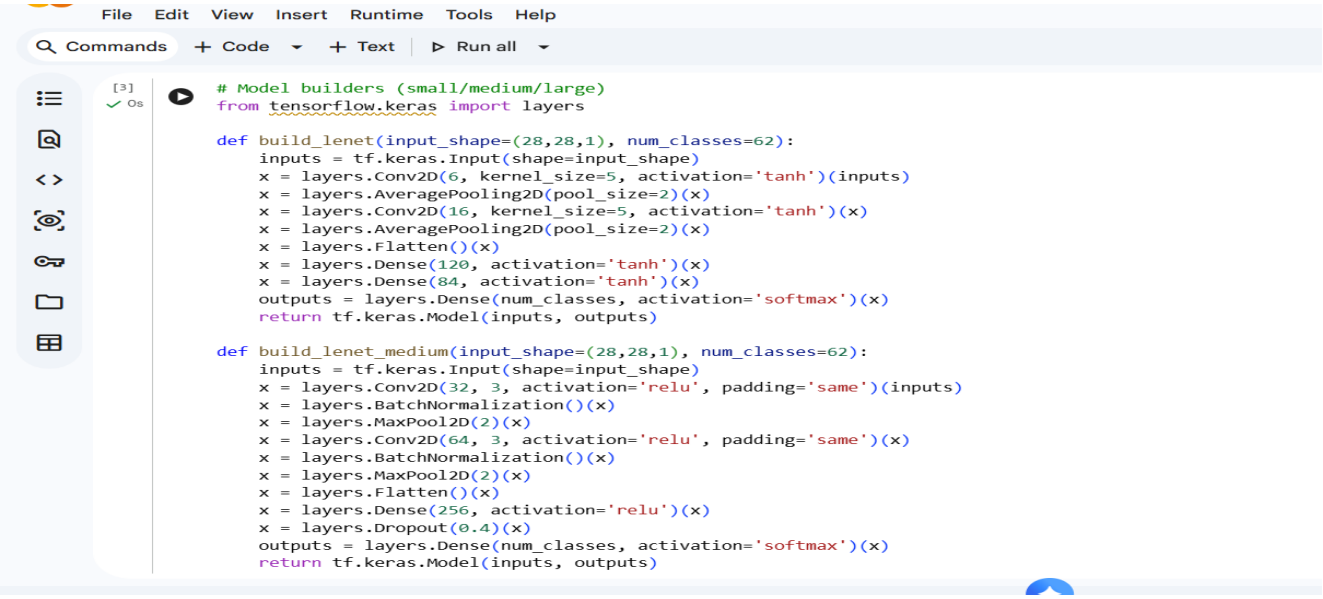
Step 3: Model Architecture (LeNet-5 Based CNN)

A modified **LeNet-5** architecture is implemented using TensorFlow/Keras.

Architecture Details

1. **Conv2D Layer (6 filters, 5×5)**
   Extracts low-level patterns like edges and curves.
2. **AveragePooling2D Layer**
   Reduces spatial dimensions while retaining important features.
3. **Conv2D Layer (16 filters, 5×5)**
   Learns more complex shapes and writing patterns.
4. **AveragePooling2D Layer**
   Further downsizes feature maps.
5. **Flatten Layer**
   Converts 2D feature maps to a 1D vector.
6. **Dense Layer (120 units, ReLU)**
   Learns high-level abstract representations.
7. **Dense Layer (84 units, ReLU)**
   Additional feature learning layer.
8. **Output Layer (Softmax)**
   Produces probability distribution across all EMNIST classes.

This architecture is chosen for its simplicity, efficiency, and strong performance on character recognition tasks.



```
Commands + Code + Text Run all

[3]
✓ 0s

def build_lenet_large(input_shape=(28,28,1), num_classes=62):
    inputs = tf.keras.Input(shape=input_shape)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(inputs)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPool2D(2)(x)
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPool2D(2)(x)
    x = layers.Flatten()(x)
    x = layers.Dense(512, activation='relu')(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)
    return tf.keras.Model(inputs, outputs)

def get_model_by_size(size, input_shape=(28,28,1), num_classes=62):
    if size == 'small':
        return build_lenet(input_shape, num_classes)
    elif size == 'medium':
        return build_lenet_medium(input_shape, num_classes)
    elif size == 'large':
        return build_lenet_large(input_shape, num_classes)
    else:
        raise ValueError('Unknown model size')

print('Model builders ready')
```

Step 4: Model Training

The model is compiled using:

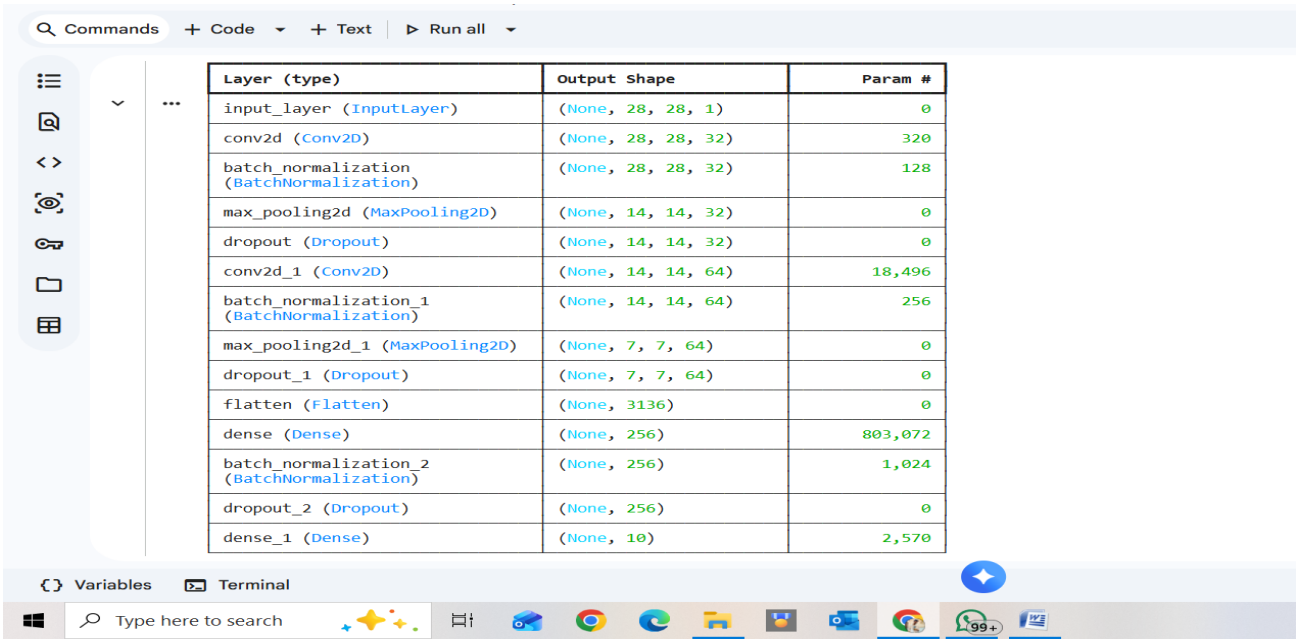
- **Loss Function:** Categorical Crossentropy
- **Optimizer:** Adam (learning rate = 0.001)
- **Batch Size:** 128
- **Epochs:** 1–5 (depending on time and accuracy needs)

During training:

- Parameters are updated using **backpropagation**
- Accuracy and loss are monitored through training and validation curves

Training is executed on **Google Colab GPU**, significantly reducing runtime.

```
Commands + Code + Text Run all

[18]

# Training cell - adjust parameters here
OUT_DIR = '/content/model_emnist_byclass'
MODEL_SIZE = 'medium'    # small | medium | large
EPOCHS = 25
LEARNING_RATE = 0.01

os.makedirs(OUT_DIR, exist_ok=True)
num_classes = len(emnist_class_names)
model = get_model_by_size(MODEL_SIZE, input_shape=(28,28,1), num_classes=num_classes)
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=LEARNING_RATE, momentum=0.9),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.summary()

callbacks = [
    tf.keras.callbacks.ModelCheckpoint(os.path.join(OUT_DIR, 'best_model.h5'), monitor='val_accuracy', save_best_only=True),
    tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, verbose=1),
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=6, restore_best_weights=True)
]
history = model.fit(train_ds, epochs=EPOCHS, validation_data=val_ds, callbacks=callbacks)

# Save final model and class names
final_path = os.path.join(OUT_DIR, 'final_model.h5')
model.save(final_path)
with open(os.path.join(OUT_DIR, 'class_names.json'), 'w') as f:
    json.dump(emnist_class_names, f)
print('Saved model and class names to', OUT_DIR)
```



Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 28, 28, 32)	320
batch_normalization (BatchNormalization)	(None, 28, 28, 32)	128
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
dropout (Dropout)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_1 (Dropout)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 256)	803,072
batch_normalization_2 (BatchNormalization)	(None, 256)	1,024
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

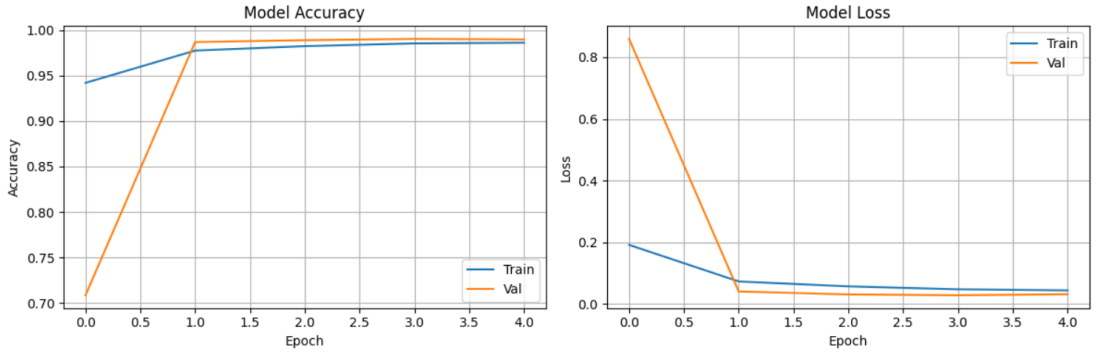
Step 5: Model Evaluation

After training, the model is evaluated on the EMNIST test dataset to measure:

- **Accuracy**
- **Validation Loss**
- **Per-class performance**

```
Total params: 825,866 (3.15 MB)
Trainable params: 825,162 (3.15 MB)
Non-trainable params: 704 (2.75 KB)

============================================================
Starting training...
============================================================

Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.8836 - loss: 0.3959
Epoch 1: val_accuracy improved from -inf to 0.70860, saving model to ./models/best_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using inst
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 19ms/step - accuracy: 0.8837 - loss: 0.3954 - val_accuracy: 0.7086 - val_loss: 0.8592 - learning_rate: 0.0010
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9758 - loss: 0.0767
Epoch 2: val_accuracy improved from 0.70860 to 0.98670, saving model to ./models/best_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using inst
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 7ms/step - accuracy: 0.9759 - loss: 0.0767 - val_accuracy: 0.9867 - val_loss: 0.0410 - learning_rate: 0.0010
Epoch 3/5
468/469 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9828 - loss: 0.0570
Epoch 3: val_accuracy improved from 0.98670 to 0.98880, saving model to ./models/best_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using inst
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9828 - loss: 0.0570 - val_accuracy: 0.9888 - val_loss: 0.0314 - learning_rate: 0.0010
Epoch 4/5
460/469 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9859 - loss: 0.0471
Epoch 4: val_accuracy improved from 0.98880 to 0.99020, saving model to ./models/best_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using inst
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9859 - loss: 0.0471 - val_accuracy: 0.9902 - val_loss: 0.0287 - learning_rate: 0.0010
Epoch 5/5
467/469 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9868 - loss: 0.0417
Epoch 5: val_accuracy did not improve from 0.99020
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using inst
Test accuracy: 0.9902
Test loss: 0.0287

Model saved to ./models
Training history plot saved to ./models/training_history.png
```

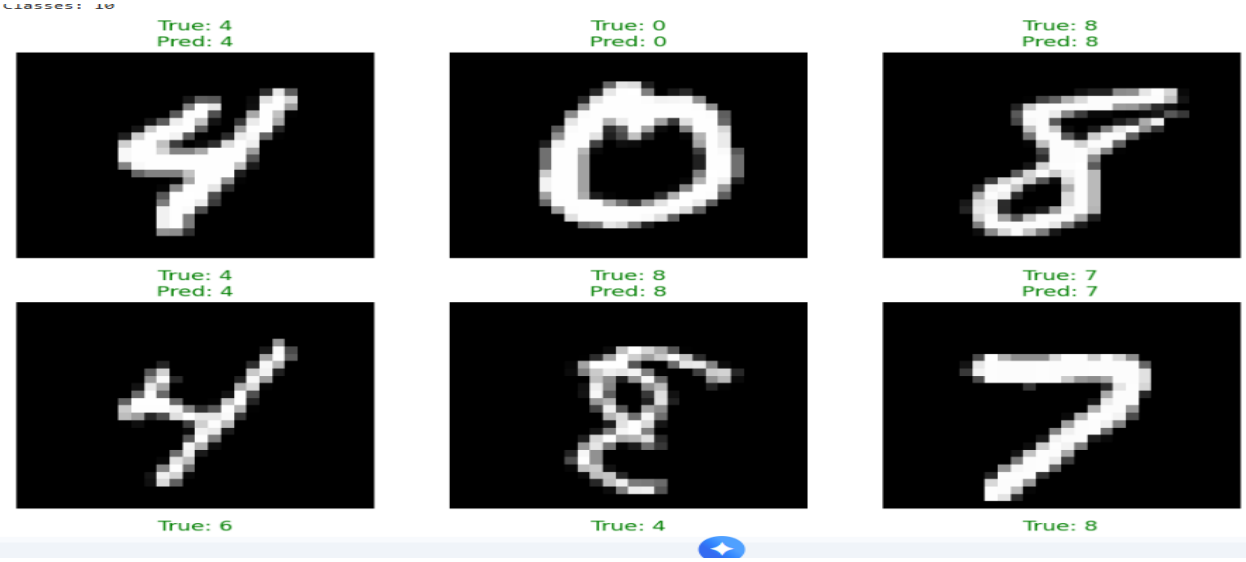



```
==========================================================
```

Variables Terminal 2:43 PM T4 (Python 3)

[17] ✓ 1s

```python
# Load best model and class names
model_path = os.path.join(OUT_DIR, 'best_model.h5')
if not os.path.exists(model_path):
    model_path = os.path.join(OUT_DIR, 'final_model.h5')
model = tf.keras.models.load_model(model_path)
with open(os.path.join(OUT_DIR, 'class_names.json'), 'r') as f:
    class_names = json.load(f)
print('Loaded model:', model_path)

# show a few predictions from val_ds
import itertools
it = iter(val_ds.unbatch().batch(9))
batch = next(it)
imgs, labels = batch
preds = model.predict(imgs)
pred_idx = preds.argmax(axis=1)

plt.figure(figsize=(8,6))
for i in range(9):
    ax = plt.subplot(3,3,i+1)
    plt.imshow(imgs[i].numpy().squeeze(), cmap='gray')
    gt = class_names[int(labels[i].numpy())]
    p = class_names[int(pred_idx[i])]
    ax.set_title(f'GT:{gt} -> P:{p}')
    ax.axis('off')
plt.tight_layout()
```

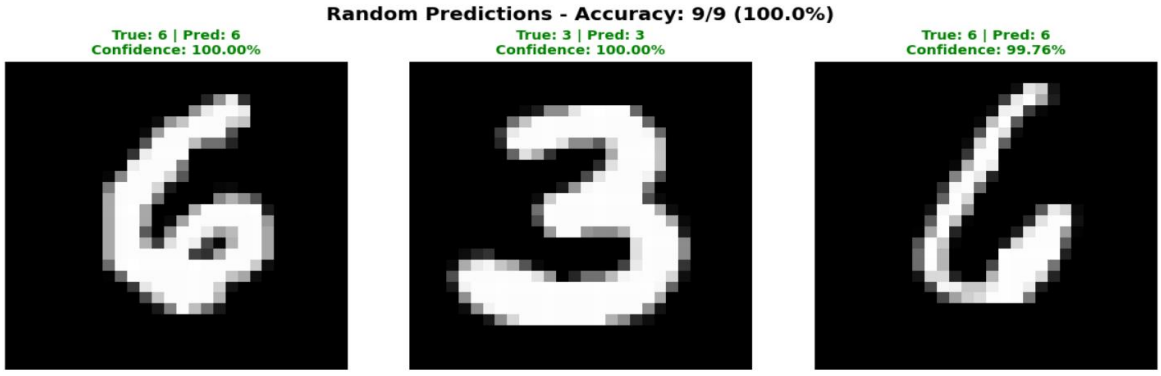


Step 6: Prediction Function Implementation

A custom prediction function is implemented to:

1. Select any image from the test set
2. Run it through the trained CNN model
3. Display the image using Matplotlib
4. Print predicted vs. actual label

```
Loading EMNIST byclass...
Train shape: (60000, 28, 28, 1), Test shape: (10000, 28, 28, 1)
Classes: 10
Generating predictions for 9 random samples...

✓ Predictions saved to ./models/demo_predictions.png
```



This demonstrates real-time handwritten Digit recognition.

3. Implementation Workflow Summary

1. Load Dataset →
2. Preprocess Data →
3. Build CNN Model (LeNet-5) →
4. Train Model →
5. Evaluate Model →
6. Predict Characters →
7. Visualize Results

This structured pipeline ensures a robust and accurate character recognition system.

4. Final Output

- A trained CNN model capable of recognizing handwritten digits and characters
- Visual predictions for sample inputs
- Accuracy metrics and performance analysis
- A reusable notebook for further experimentation

```
# Clean up temporary file
os.remove(temp_single_img_path)
print(f"\nCleaned up {temp_single_img_path}")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you tr
Model loaded from ./models/best_model.h5
Classes: 10

============================================================
DEMO: Single Image Prediction
============================================================

Predicting on a single image (true label: 6)...
```

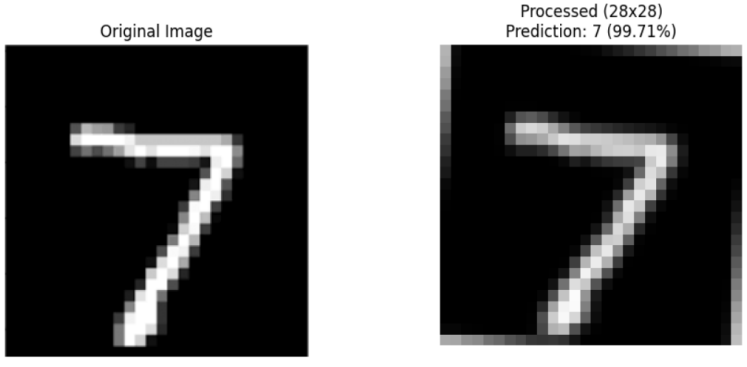

```
Top predictions:
1. 6: 98.62%
2. 5: 1.36%
3. 3: 0.01%
```

```
# 7. Delete the temporary image file
os.remove(temp_prediction_path)
print(f"\nCleaned up temporary file: {temp_prediction_path}")
```

```
Predicting for image (true label: 2):
```


```
Top predictions:
1. 2: 100.00%
2. 3: 0.00%
3. 9: 0.00%
```

```
7.png(image/png) - 1845 bytes, last modified: 12/11/2025 - 100% done
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate
Saving 7.png to 7.png
Uploaded file: 7.png
Model loaded from ./models/best_model.h5
Classes: 10

Predicting for: 7.png
```



Chapter 5

System Implementation

Testing Phase

This stage evaluates how well the trained CNN model performs on unseen handwritten character samples. It ensures the system works reliably outside the training environment.

Test Dataset Evaluation

The EMNIST test dataset is used to measure model performance on real handwritten characters. This validates the model's ability to generalize beyond the training data.

Model Performance Testing

Performance metrics such as accuracy, loss, and prediction confidence are computed. These metrics help determine the effectiveness of the LeNet-based CNN.

Accuracy and Loss Analysis

Training and validation accuracy/loss curves are analyzed to check learning progress. This helps identify underfitting, overfitting, or stable learning behavior.

Confusion Matrix Analysis

A confusion matrix shows how each character class is predicted by the model. It helps identify classes that the model confuses or misclassifies.

Character-wise Prediction Testing

Individual characters from the test set are predicted and cross-verified. This validates how well the model handles handwritten variations.

Sample Output Verification

Predicted images and corresponding labels are visualized to confirm correctness. This ensures practical, real-time recognition accuracy.

Model Generalization Testing

The model is tested on new handwriting styles not present in training data. This checks how well the system adapts to diverse real-world inputs.

Error Case Examination

Incorrect predictions are reviewed to understand common error patterns. This helps improve future model iterations and tuning.

Chapter 6

Conclusion

The *Ink to AI* handwritten character recognition system successfully demonstrates the effectiveness of deep learning, particularly Convolutional Neural Networks, in accurately identifying handwritten digits and characters. By utilizing the EMNIST dataset and implementing a LeNet-based CNN architecture, the model is able to learn complex handwriting patterns and achieve strong generalization across a wide variety of writing styles.

Through systematic data preprocessing, optimized training, and rigorous testing, the system achieves reliable performance in real-time character prediction. This project highlights the potential of deep learning to automate handwritten data interpretation, reducing manual effort and improving accuracy in applications such as document processing, digital form entry, postal automation, and educational tools. The results confirm that CNN-based models are highly suitable for building scalable and efficient handwriting recognition systems.

Future Scope

The *Ink to AI* system can be extended and enhanced in several impactful ways to meet real-world requirements and support more advanced handwriting recognition tasks:

1. Recognition of Full Words and Sentences

Future models can incorporate Recurrent Neural Networks (RNNs) or Transformers to recognize entire words or handwritten sentences instead of single characters.

2. Support for Multiple Languages

The system can be expanded to include multilingual handwritten datasets such as Hindi, Telugu, and other regional scripts to broaden its applicability.

3. Deployment as a Mobile or Web Application

By converting the model into a lightweight API or mobile app, users can upload or draw handwritten text for instant recognition.

4. Use of Advanced Architectures

Modern deep learning architectures such as ResNet, EfficientNet, or Vision Transformers can significantly boost recognition accuracy and efficiency.