HANDWRITTEN TEXT EXTRACTION USING OCR AND MACHINE LEARNING TECNIQUES

A PROJECT REPORT

Submitted by

NISHANTHAN T	**-**	**810021106053**
SIVAPRAKASH A	**-**	**810021106079**
TONY CHACKO THOMAS	**-**	**810021106089**

In partial fulfilment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

UNIVERSITY COLLEGE OF ENGINEERING, BIT CAMPUS

TIRUCHIRAPPALLI-620024

ANNA UNIVERSITY: CHENNAI- 600 025

MAY 2025

ANNA UNIVERSITY: CHENNAI- 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"HANDWRITTEN TEXT EXTRACTION USING OCR AND MACHINE LEARNING TECNIQUES"** is the Bonafide work of **NISHANTHAN T (8100211006053), SIVAPRAKASH A (810021106079), TONY CHACKO THOMAS (810021106089)** who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Dr P. RAMADEVI	Dr. E. JEBAMALAR LEAVLINE
HEAD OF THE DEPARTMENT	**SUPERVISOR**
Associate Professor	Assistant Professor (Sl.Gr)
Department of ECE	Department of ECE
UCE (BIT Campus)	UCE (BIT Campus)
Tiruchirappalli - 24	Tiruchirappalli – 24

Submitted to University Viva-Voce examination held on ………..………..

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DECLARATION

We hereby declare that the work entitled "**HANDWRITTEN TEXT EXTRACTION USING OCR AND MACHINE LEARNING TECNIQUES"** is submitted for the award of the degree in B.E. Electronics and Communication Engineering, University College of Engineering, BIT Campus, Tiruchirappalli, is a record of our own work carried out by us during the academic year 2024-2025 under the supervision and guidance of **Dr. E. Jebamalar Leavline**, Assistant Professor (Sl. Gr), Department of ECE, University College of Engineering, BIT Campus, Tiruchirappalli. The extent and the source of information are derived from the existing literature and have been indicated through the dissertation at the appropriate places. The matter embodied in this work is original and has not been submitted for the award of any other degree or diploma, either in this or other University.

NISHANTHAN T (810021106053 **)**

SIVAPRAKASH A (810021106079)

TONY CHACKO THOMAS (810021106089)

I certify that the declaration made above by the candidates is true.

Signature of the Supervisor

ACKNOWLEDGEMENT

All praise, glory and honour to the Lord Almighty, for his gracious presence and guidance that enable us to complete this project duly.

We wish to express our profound thanks to **Dr T. SENTHILKUMAR**, Dean, University College of Engineering, BIT Campus, Tiruchirappalli, for granting us permission for doing this project work.

We wish to express our sincere thanks to **Dr P. RAMADEVI,** Head of the Department of Electronics and Communication Engineering, University College of Engineering, BIT Campus, Tiruchirappalli, for her support and ardent guidance.

We owe our special thanks and gratitude to **DR.E. JEBAMALAR LEAVLINE**, Assistant Professor (Sl. Gr), Department of ECE, University College of Engineering, BIT Campus, Anna University, Tiruchirappalli, who guided us throughout our project with her timely help, guidance with valuable suggestions.

We express our heartfelt thanks to our project review committee members, Department of Electronics and Communication Engineering for their passionate support, for helping us to identify our mistakes and also for the appreciation they gave us in achieving our goals.

NISHANTHAN T (810021106079)

SIVAPRAKASH A (810021106079)

TONY CHACKO THOMAS (810021106089)

TABLE OF CONTENTS

ABSTRACT

Handwritten Text Recognition (HTR) is a significant challenge in the field of computer vision and pattern recognition, especially for complex scripts like Tamil. This project presents a deep learning-based approach to recognize Tamil handwritten single-word images using a hybrid CNN-BiLSTM-CTC architecture. The model leverages convolutional layers to extract robust spatial features from the input images, followed by Bidirectional Long Short-Term Memory (BiLSTM) layers that capture temporal dependencies in character sequences. The Connectionist Temporal Classification (CTC) loss function enables the model to learn from unsegmented sequence data without requiring character-level alignment.

The dataset used in this work comprises over 100,000 labeled images of handwritten Tamil words, divided into 75,737 training images, 11,599 validation images, and 16,185 test images. Extensive preprocessing techniques including resizing, normalization, and label encoding are applied to ensure optimal model performance. The system is evaluated using standard metrics such as Character Error Rate (CER) and Word Error Rate (WER), achieving a CER of 2.88% and a WER of 16.61%, indicating high recognition accuracy.

The proposed model demonstrates strong generalization on unseen data, effectively handling various handwriting styles and word lengths. This work contributes to the development of robust handwritten text recognition systems for Tamil and can be extended to sentence-level recognition and multi-script environments in future research.

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

S.NO	ACRONYMS	ABBREVATION
1.	BiLSTM	BIDIRECTIONAL LONG SHORT-TERM MEMORY
2.	CER	CHARACTER ERROR RATE
3.	CNN	CONVOLUTIONAL NEURAL NETWORK
4.	CTC	CONNECTIONIST TEMPORAL CLASSIFICATION
5.	FN	FALSE NEGATIVE
6.	FP	FALSE POSITIVE
7.	GPU	GRAPHICS PROCESSING UNIT
8.	HTR	HANDWRITTEN TEXT RECOGNITION
9.	OCR	OPTICAL CHARACTER RECOGNITION
10.	ReLU	RECTIFIED LINEAR UNIT
11.	RNN	RECCURENT NEURAL NRTWORK
12.	TN	TRUE NEGATIVE
13.	TP	TRUE POSITIVE
14.	WER	WORD ERROR RATE

CHAPTER 1

INTRODUCTION

1.1 Background of Handwriting Recognition

Handwriting recognition, a sub-field of pattern recognition and artificial intelligence, aims to interpret and digitize handwritten input from various sources such as paper documents, images, and touchscreens. Over the past few decades, substantial progress has been made in recognizing handwritten text in various scripts, primarily due to advancements in machine learning and deep learning techniques. Handwriting recognition systems are generally divided into two categories: online and offline recognition. Online systems capture temporal information such as stroke order and speed, while offline systems operate on static images.

The offline handwriting recognition domain, especially for regional and complex scripts like Tamil, poses significant challenges due to variability in handwriting styles, overlapping characters, ligatures, and degradation in image quality. Unlike typed text, handwritten text does not follow a fixed shape or structure, making it a challenging problem to solve, particularly for low-resource languages.

1.2 Importance of Tamil HTR

Tamil is one of the oldest and most widely used Dravidian languages, with a rich literary heritage spanning over two millennia. It is used by millions of people in India, Sri Lanka, and across the globe. Preserving historical documents and manuscripts written in Tamil requires effective digitization, which necessitates reliable handwritten text recognition systems.

The development of Tamil Handwritten Text Recognition (HTR) systems is crucial not only for digital archiving but also for applications in education, cultural preservation, governmental documentation, and citizen services. However, due to limited datasets, unique character structures, and high script complexity, Tamil HTR has lagged behind English and other Latin-based script recognition technologies.

1.3 Challenges in Tamil HTR

Tamil handwriting recognition faces several inherent challenges:

Script Complexity: Tamil script contains 12 vowels, 18 consonants, and more than 200 compound characters formed by combining these base letters.

Ligatures and Diacritics: Complex ligatures and diacritical marks that modify the base characters add to the difficulty.

Writing Style Variability: Handwriting styles vary significantly between individuals and even within the same document.

Limited Datasets: There is a lack of large, annotated datasets for Tamil handwritten text, which hampers model training and generalization.

Image Quality Issues: Scanned images may suffer from noise, skewing, fading, or background artifacts.

1.4 Project Objectives

The main objectives of this project are:

To design and implement a robust Tamil HTR system using deep learning-based CRNN (Convolutional Recurrent Neural Network) architecture.

To preprocess raw handwritten Tamil sentence images effectively and encode them for model training.

To train the model using Connectionist Temporal Classification (CTC) loss for handling unsegmented sequences.

To evaluate model performance using appropriate metrics such as Character Error Rate (CER) and Word Error Rate (WER).

To visualize and analyze the model's predictions, identifying strengths and areas for improvement.

CHAPTER 2

LITERATURE SURVEY

2.1 Overview

Handwritten Text Recognition (HTR) has been a long-standing challenge in the field of computer vision and pattern recognition, particularly for complex scripts like Tamil. While printed text recognition using traditional OCR systems has reached maturity, handwritten text presents higher levels of variability in stroke, spacing, and style, making conventional rule-based approaches largely ineffective. This chapter provides a comprehensive overview of existing research and methodologies applied to HTR, with a particular focus on Tamil script recognition and the deep learning techniques that have revolutionized the field in recent years.

In Benchmarking on Offline Handwritten Tamil Character Recognition using CNNs (Kavitha B.R. & Srimathi C., 2022) applied CNNs to Tamil character recognition, achieving high accuracy on isolated characters. However, performance declined at word level due to lack of contextual information. It highlights CNNs' capability in spatial feature learning and the need for advanced models to handle the complexities of full-word Tamil handwriting.

In Training Full-page Handwritten Text Recognition Models Without Annotated Line Breaks (Tensmeyer & Wigington, 2019) The authors proposed a technique to train full-page HTR models without manually annotated line breaks, using end-to-end learning. This eliminates line segmentation, making the system efficient and scalable. Their method benefits low-resource scripts by simplifying data preparation and improving overall accuracy in multi-line document recognition.

CNN-RNN Based Handwritten Text Recognition (G.R. Hemanth et al., 2021) proposed a hybrid CNN-RNN model for handwritten text recognition in regional Indian languages, including Tamil. CNNs extracted spatial features while RNNs modeled sequences. The approach showed promising results on small datasets, underlining the strength of combined architectures in improving recognition for complex and cursive scripts.

A Review on Machine Learning Styles in Computer Vision – Techniques and Future Directions (S.V. Mahadevkar et al., 2022) is a comprehensive review explored ML techniques in computer vision, emphasizing adaptability across applications. It highlighted deep learning's role in HTR and the evolution of hybrid models. The paper suggests future directions like few-shot learning, transfer learning, and multi-modal integration for improving recognition of complex handwritten documents.

In the study Enhancement of Handwritten Text Recognition Using AI-Based Hybrid Approach (Supriya Mahadevkar et al., 2024) proposed a hybrid AI-ML model combining deep learning with classical techniques for multilingual HTR, including Tamil. The approach enhanced recognition performance by integrating feature extraction, noise handling, and intelligent classification. It emphasizes the role of hybrid systems in overcoming limitations of standalone models in complex scripts.

2.2 Traditional OCR and Its Limitations

Traditional OCR systems rely heavily on feature engineering and template matching, which works well for machine-printed text but fails for handwriting due to: • Variability in writing styles (slant, spacing, cursiveness) • Disconnected or overlapping characters • Noise and distortions introduced during scanning These systems typically follow a rigid pipeline of segmentation → feature extraction → classification, which breaks down when faced with the ambiguity and inconsistencies of human handwriting. Especially for Indian scripts like Tamil, which contain compound characters and complex ligatures, segmentation-based methods are error-prone and computationally expensive.

2.3 Key Challenges Identified

Despite the progress made, Tamil handwritten text recognition continues to face several unresolved challenges: • Dataset scarcity: There are limited publicly available datasets for Tamil handwriting, particularly at the sentence level. • Script complexity: Tamil script includes 247 characters, many of which are visually similar or composed of multiple strokes, increasing misclassification risk. • Multilingual support: Models must often handle code-mixed documents containing English and Tamil, requiring multi-script training. • Real-time performance: Optimizing deep models for deployment on mobile or embedded platforms remains an open research issue.

2.4 Summary

The literature reflects a clear evolution from traditional OCR to deep learning-based HTR systems. For Tamil, deep CNN-RNN-CTC architectures are emerging as the most promising approach, especially in handling unsegmented, sentence-level data. However, further work is needed to address data availability, script complexity, and real-time optimization. This project builds on prior research by designing a robust, sentence-level HTR system tailored to Tamil using a CNN-BiLSTM-CTC pipeline, setting a new benchmark for future studies in regional script recognition.

CHAPTER 3

TAMIL LANGUAGE CHARACTERISTICS

3.1 Overview of Tamil Script

Tamil is one of the oldest classical languages, spoken primarily in the Indian state of Tamil Nadu, Sri Lanka, and among Tamil diaspora worldwide. It belongs to the Dravidian language family and has a rich literary tradition spanning over two millennia.

The Tamil script is an abugida, meaning that each consonant has an inherent vowel sound that can be changed or muted by adding diacritics. It is written from left to right and is syllabic in nature, where each symbol typically represents a syllable rather than an individual phoneme.

3.2 Character Set and Script Components

The Tamil script consists of the following basic components:

Vowels (Uyir Ezhuthukkal): There are 12 vowels in Tamil, each representing a distinct vowel sound. Examples include அ (a), இ (i), உ (u), ஏ (ē), and so forth.Consonants (Mei Ezhuthukkal): Tamil has 18 consonants that form the base consonantal sounds, such as க (ka), ச (ca), ட (ṭa), ண (ṇa), etc.Compound Characters (Uyir-Mei Ezhuthukkal): These are formed by combining a consonant and a vowel to represent different syllables. For example, க (ka) combined with இ (i) forms கி (ki).Grantha Letters: For representing sounds from Sanskrit and other languages not

native to Tamil, a few Grantha letters are used. These are often found in

loanwords and classical literature.Special Symbols: Tamil also includes a special character, the pulli (்), which suppresses the inherent vowel in a consonant to form a pure consonant sound.

3.3 Script Characteristics Relevant to Handwritten Text Recognition

3.3.1 Complex Ligatures and Diacritics

One of the main challenges in Tamil handwriting recognition is the presence of ligatures—combined forms of multiple characters written as a single glyph. The script has many compound characters formed by the interaction of base consonants with vowel signs or diacritics placed above, below, before, or after the base character.

These diacritics can vary widely in handwritten form, sometimes merged with the consonant, or written separately. The complex spatial positioning increases the difficulty for segmentation-based OCR methods.

3.3.2 Cursive and Connected Writing

Handwritten Tamil often exhibits cursive writing, where characters and words may be connected fluidly without clear boundaries. This complicates character segmentation and requires robust sequence modeling to correctly recognize text lines or words.

3.3.3 Similarity of Characters

Some Tamil characters have very similar shapes, differing only by a small stroke or dot. For example, க (ka) and ங (ṅa) are visually close, especially in cursive handwriting. Distinguishing these accurately is crucial for reducing recognition errors.

3.3.4 Large Character Set Size

With 12 vowels, 18 consonants, and numerous compound characters, the total number of unique glyphs to recognize can exceed 200 when including all combinations. This large set increases the complexity of classification models.

3.4 Challenges in OCR and HTR for Tamil

Segmentation Difficulty: Due to cursive writing and complex ligatures, segmenting words into individual characters is error-prone. This is why sequence-to-sequence approaches like CNN-BiLSTM-CTC, which do not require explicit segmentation, are well-suited.

Variability in Handwriting Styles: Different writers have unique handwriting styles with varying slant, stroke thickness, and spacing. Capturing this variability requires a large, diverse dataset and robust feature extraction.

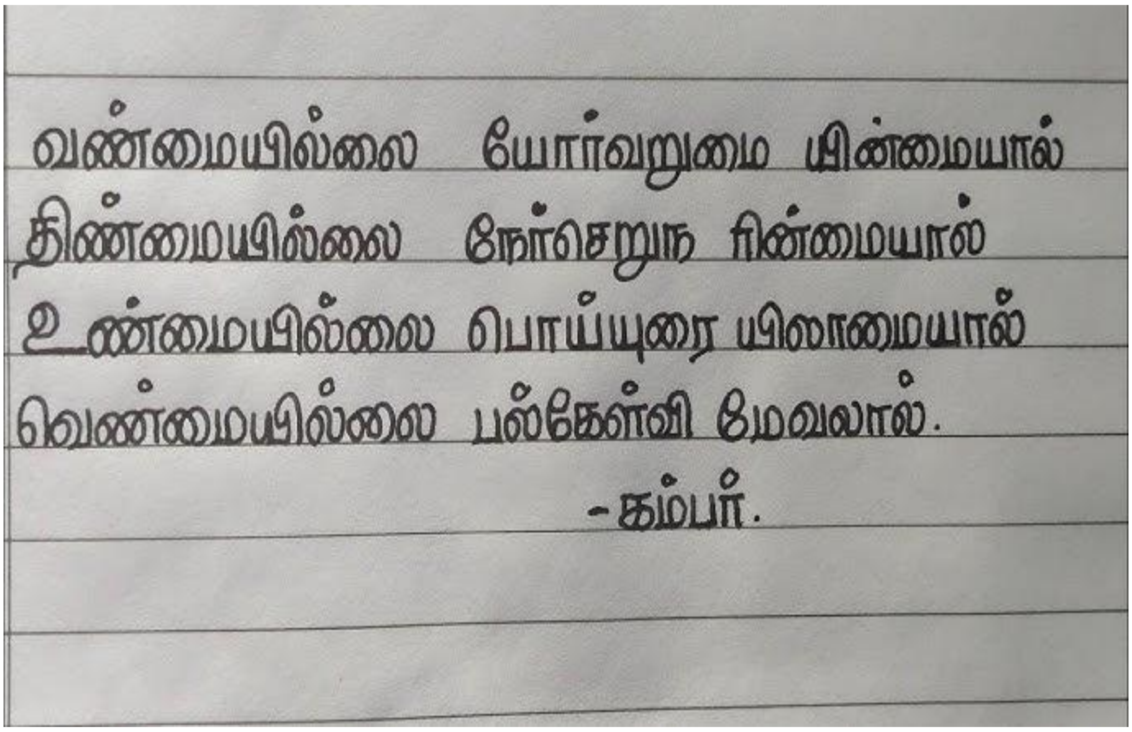
Contextual Dependency: Some characters look similar and can only be correctly interpreted in the context of surrounding characters. BiLSTM layers can capture this sequential context to improve accuracy.

Diacritics Placement: The relative positions of diacritics can change the meaning of characters and words. Recognizing these correctly requires the model to learn spatial relationships within the glyphs.

3.5 Summary

Tamil script poses unique challenges for handwritten text recognition due to its large character set, complex ligatures, cursive writing style, and subtle visual differences between characters. Traditional OCR approaches relying on segmentation are often insufficient. Thus, deep

learning methods that combine convolutional feature extraction with sequence modeling and CTC loss are effective strategies for robust Tamil HTR.



வண்மையில்லை யோர்வறுமை யின்மையால்
திண்மையில்லை நேர்செறுந ரின்மையால்
உண்மையில்லை பொய்யுரை யிலாமையால்
வெண்மையில்லை பல்கேள்வி மேவலால்.
-கம்பர்.

Figure 3.1-Tamil Handwritten Samples

CHAPTER 4

DATASET AND PREPROCESSING

4.1 Dataset Overview

The dataset used in this project is a large collection of Tamil handwritten single-word images, carefully curated for training a Handwritten Text Recognition (HTR) system. Each image corresponds to a single Tamil word written by various individuals, allowing for diversity in writing styles, stroke thickness, and word length.

The dataset is organized into three main subsets:

Training Set: 75,737 images

Validation Set: 11,599 images

Test Set: 16,185 images

Each image is accompanied by a corresponding ground truth label, stored in CSV files. These labels represent the correct Tamil word depicted in the image and are used for model training, tuning, and evaluation.

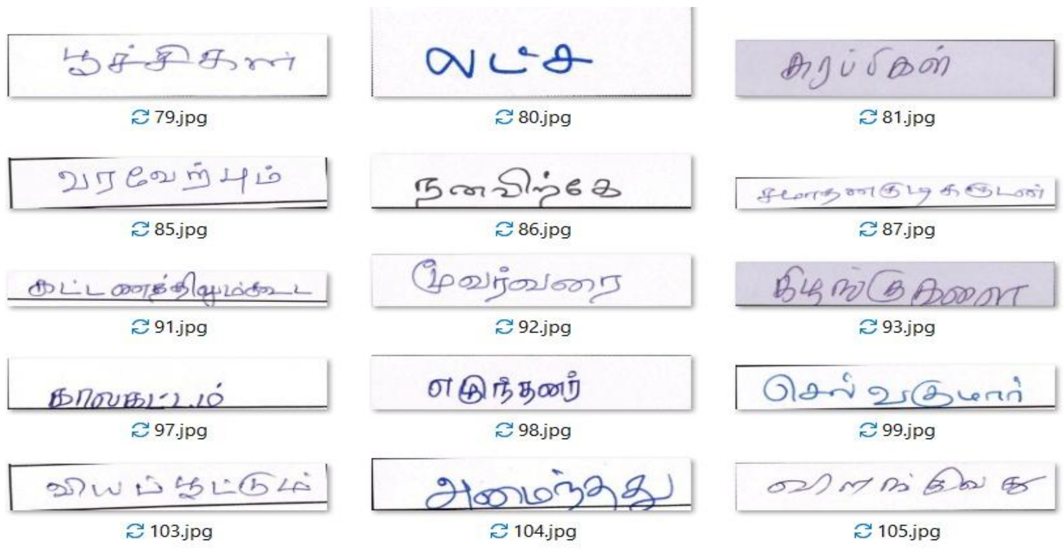


Figure 4.1 -Dataset Image Samples 1

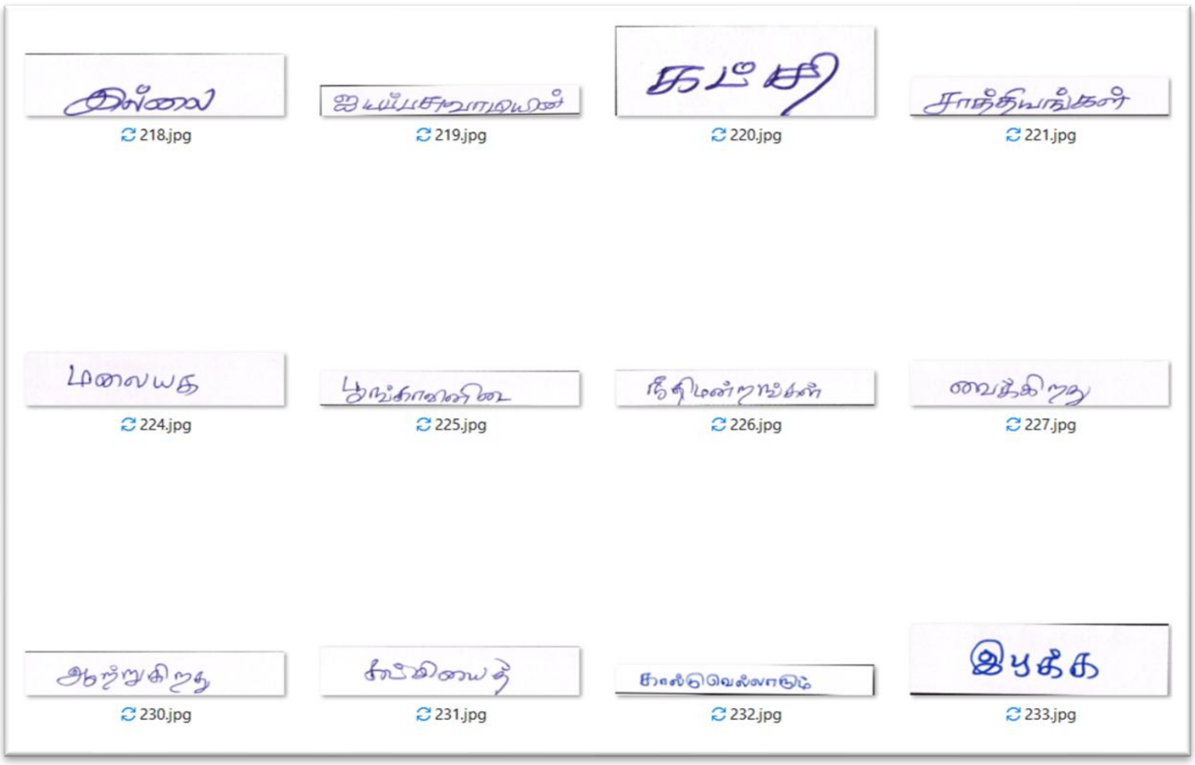
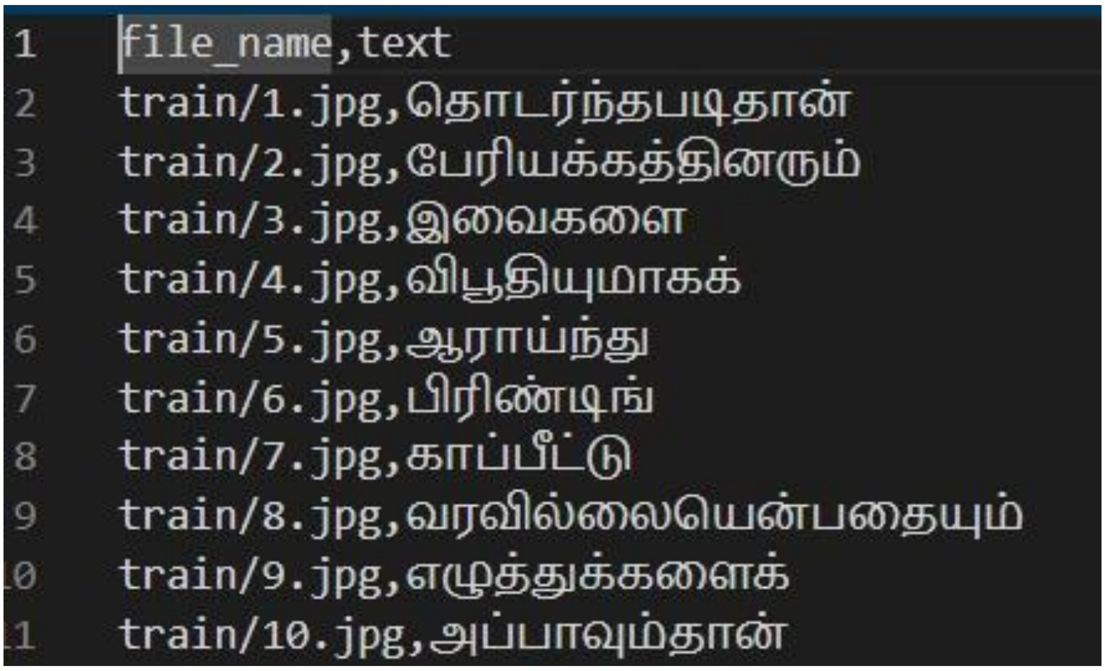


Figure 4.2 -Dataset Image Samples 2

4.2 Data Structure



```
file_name,text
train/1.jpg,தொடர்ந்தபடிதான்
train/2.jpg,பேரியக்கத்தினரும்
train/3.jpg,இவைகளை
train/4.jpg,விபூதியுமாகக்
train/5.jpg,ஆராய்ந்து
train/6.jpg,பிரிண்டிங்
train/7.jpg,காப்பீட்டு
train/8.jpg,வரவில்லையென்பதையும்
train/9.jpg,எழுத்துக்களைக்
train/10.jpg,அப்பாவும்தான்
```

Figure 4.3 Label File Sample

Each CSV file (train.csv, val.csv, test.csv) contains two columns: the relative image path and its corresponding label (Tamil word). This makes it straightforward to load images and their annotations in a data generator.

4.3 Preprocessing

4.3.1 Image Preprocessing

To ensure consistency in input to the neural network, all images undergo the following preprocessing steps:

Grayscale Conversion: Input images are loaded in grayscale to reduce input dimensionality.

Resizing: All images are resized to a fixed height and width (e.g., 32×256 pixels) while preserving the aspect ratio as much as possible.

Normalization: Pixel values are normalized to the range [0, 1] for better convergence during training.

4.3.2 Label Encoding

Labels are processed into numerical form using a character-level vocabulary. The steps include:

- Mapping each character to an integer index.
- Padding or truncating the encoded labels to a fixed maximum length.
- This process allows the model to output character sequences which can then be decoded back to text.

CHAPTER 5

MODEL ARCHITECTURE

5.1 Introduction

The core component of a handwritten text recognition system is the model architecture, which learns to extract meaningful features from input images and decode them into readable text. In this project, a hybrid neural network architecture combining Convolutional Neural Networks (CNN), Bidirectional Long Short-Term Memory (BiLSTM) networks, and the Connectionist Temporal Classification (CTC) loss function was employed. This chapter describes each component of the architecture, its role, and how the components work together to achieve robust recognition of Tamil handwritten sentences.

5.2 Overview of the Proposed Model

The proposed model takes a preprocessed grayscale image of a handwritten Tamil sentence as input and outputs a sequence of characters corresponding to the sentence. The model architecture can be summarized as follows:

- CNN Layers: To extract hierarchical visual features and reduce spatial dimensions.
- BiLSTM Layers: To model sequential dependencies along the text line.
- Dense Layer: To map LSTM outputs to character probabilities.
- CTC Loss: To handle alignment and decoding without requiring segmented characters.

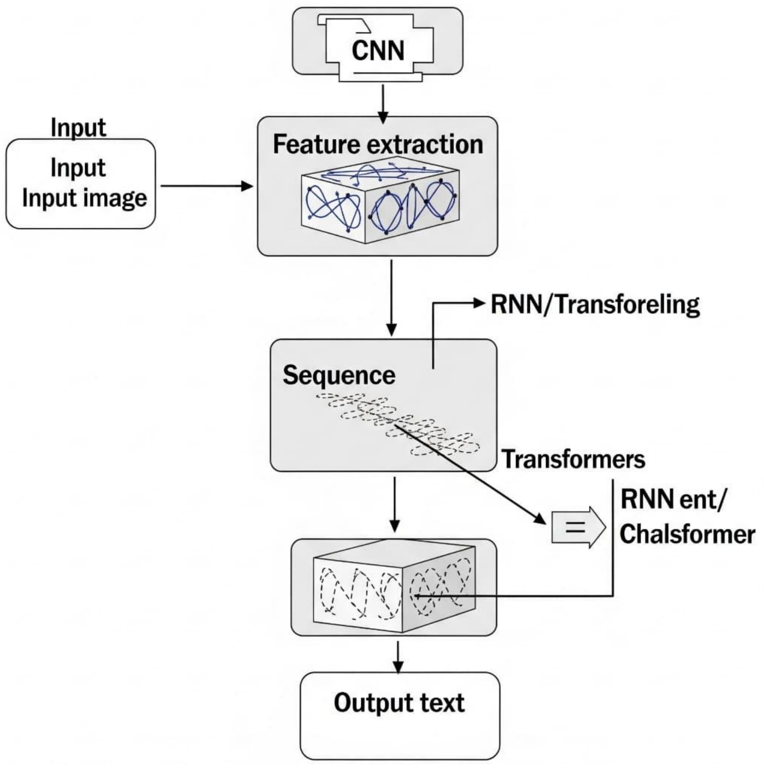


Figure 5.1 Model architecture

5.3 Convolutional Neural Network (CNN)

5.3.1 Purpose

The CNN layers serve as feature extractors that transform the raw pixel intensities of the input images into high-level representations capturing important handwriting patterns such as strokes, curves, and character shapes. This reduces the input's spatial dimensions and extracts features that are invariant to local distortions.

5.3.2 Architecture Details

- The CNN module consists of several convolutional blocks, each containing:
- Convolutional Layers with small filters (typically 3x3) to detect local features.
- Batch Normalization to accelerate training and improve stability.
- ReLU Activation to introduce non-linearity.

- Max Pooling to downsample feature maps and reduce computational complexity.

Example CNN block:

Table 1. CNN block

Layer	Description
Conv2D	64 filters, kernel size 3x3
Batch Normalization	Normalizes activations
ReLU	Activation function
MaxPooling2D	Pool size 2x2

Multiple such blocks are stacked to progressively reduce the height dimension (e.g., from 256 pixels down to 1 or a few pixels) while preserving the width dimension to maintain the sequential nature of the text line.

5.4 Bidirectional Long Short-Term Memory (BiLSTM)

5.4.1 Purpose

Since handwritten text is sequential in nature, modeling context along the horizontal axis is crucial. BiLSTM layers capture dependencies in both forward and backward directions, enabling the model to better understand contextual relationships between characters.

5.4.2 Architecture Details

The output from the CNN is reshaped into a sequence by flattening the height dimension. Two stacked BiLSTM layers with units (e.g., 128 or 256 units) process the sequence. Each BiLSTM layer consists of two LSTMs running in opposite directions. The BiLSTM outputs at each timestep encapsulate contextual information from past and future. This helps in disambiguating visually similar characters by leveraging surrounding context.

5.5 Fully Connected Layer and Softmax

The BiLSTM outputs are passed through a fully connected (Dense) layer with softmax activation. This layer converts each timestep's output into a probability distribution over the entire vocabulary, including a special "blank" token used by the CTC loss.

Number of units in Dense layer = Vocabulary size + 1 (for blank)

Softmax outputs probabilities for each character class per timestep.

5.6 Connectionist Temporal Classification (CTC) Loss

5.6.1 Motivation

Handwritten text images do not come with explicit segmentation between characters, making it challenging to align predictions with ground truth labels. CTC loss allows the model to learn from unsegmented data by considering all possible alignments of the predicted sequence with the target label.

5.6.2 How CTC Works

The model outputs a sequence of character probabilities per timestep.

CTC sums over all valid alignments that correspond to the target text.

The loss encourages the model to assign high probabilities to correct label sequences while allowing flexible timing.

CTC decoding during inference uses beam search or best path decoding to generate the most probable text sequence from the model outputs.

5.7 Model Summary and Parameters

Component	Details
Input	Grayscale image of size (256, 1024, 1)

CNN Layers	5 convolutional blocks, filters 64–512
BiLSTM Layers	2 layers, 256 units each, bidirectional
Dense Layer	Vocabulary size + 1 units (softmax)
Loss Function	CTC loss

The model is implemented using TensorFlow/Keras, allowing efficient GPU acceleration during training.

5.8 DATA FLOW DIAGRAM

LEVEL 0

The diagram illustrates the workflow of a Handwritten Text Recognition system, showing how information flows between the User, the HTR Model, and the Output Processor. In this system, the User provides input in the form of handwritten text images, which could be scanned documents or photographs of handwritten notes. The HTR Model acts as the core processing unit that receives these images, processes them through multiple layers of deep learning algorithms, and converts the handwritten content into machine-readable text.The User uploads or inputs handwritten images into the system, which then passes the data to the HTR Model. This model performs feature extraction, sequence modeling, and transcription of the text using convolutional neural networks (CNN), bidirectional LSTM layers, and Connectionist Temporal Classification (CTC) decoding. Once the text is recognized and decoded, the Output Processor takes over to format the recognized text, verify it if needed, and present the final transcribed output back to the user or to downstream applications. This data flow reflects a streamlined and automated process, minimizing the need for manual transcription and reducing human error. The architecture enables efficient recognition of handwritten text with high accuracy and quick turnaround, making it suitable for applications like digitizing handwritten records, form processing, or automated document entry.

By centralizing image input, recognition processing, and text output, the system ensures a smooth and scalable workflow for handling handwritten data.

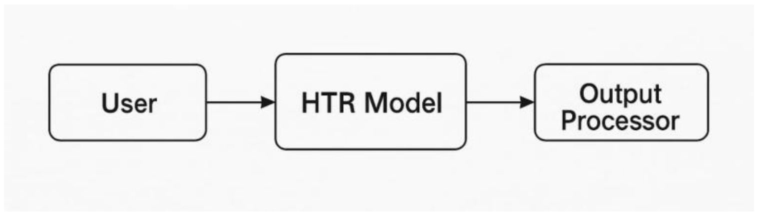


Figure 5.2 Level 0 Data Flow Diagram

LEVEL 1

The Handwritten Text Recognition system represents an integrated framework that combines image preprocessing, text recognition, and result management to enable accurate and efficient digitization of handwritten documents. This system is designed to automate and streamline the critical processes involved in recognizing handwritten text from images, converting it into editable digital text, and managing recognition outputs for further use. The first and most essential part of the system is image preprocessing. This stage involves cleaning and preparing the input handwritten images by removing noise, correcting skew, and segmenting lines and words. Proper preprocessing is crucial to enhance the quality of the input data, making it easier for the recognition models to accurately interpret the handwriting. Following preprocessing, the core text recognition module processes the prepared images to extract handwritten text. This module uses advanced deep learning algorithms such as Convolutional Neural Networks (CNN) combined with Bidirectional Long Short-Term Memory (BiLSTM) networks and Connectionist Temporal Classification (CTC) loss to identify characters and words from the handwriting. The system is trained on extensive labeled datasets, enabling it to handle different handwriting styles, variations, and complexities. Once the handwritten text is recognized, the result management module takes charge of organizing, displaying, and storing the digitized text.

This module allows users to review recognized text, edit if necessary, and export the results into various formats. It also manages recognition confidence scores and generates reports on accuracy and errors, helping users to evaluate and improve the system’s performance. This comprehensive system framework ensures smooth integration between preprocessing, recognition, and output management processes. By automating image enhancement, text extraction, and result handling, the system significantly improves document digitization efficiency, reduces manual transcription errors, and enables easy access to handwritten information in digital form.

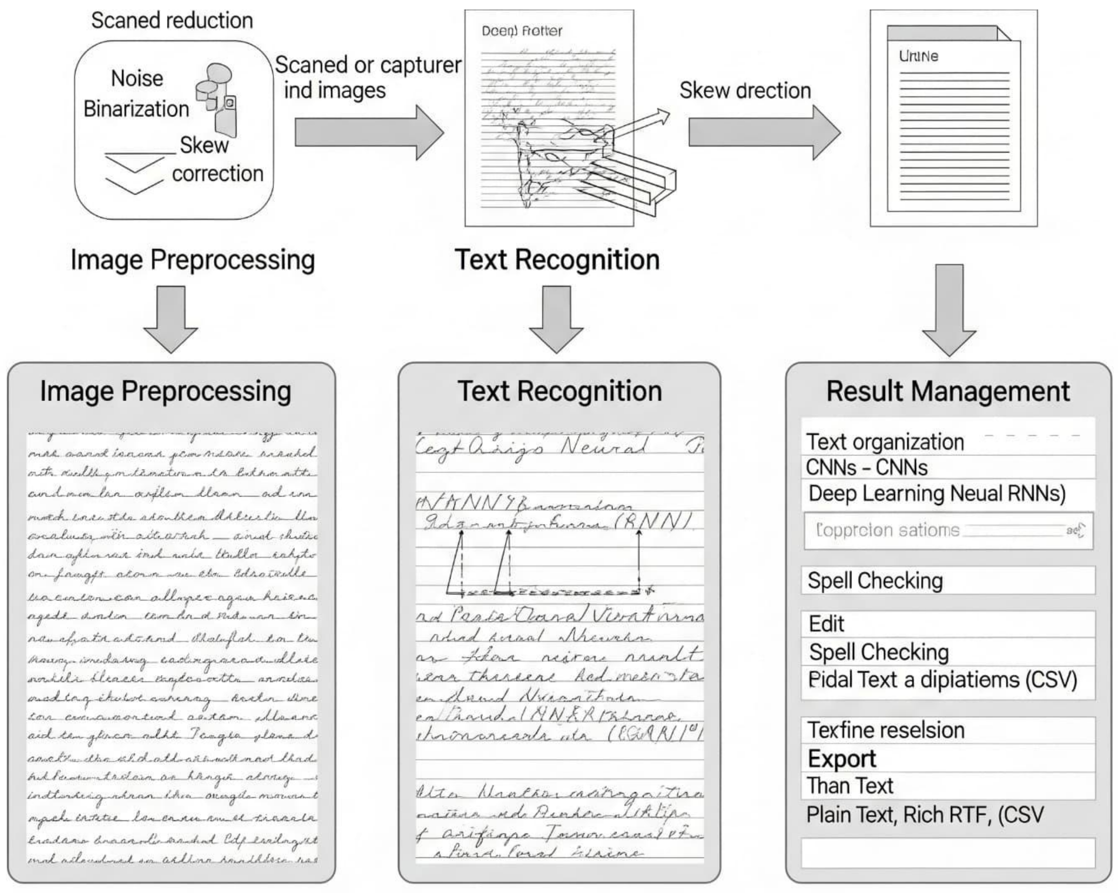


Figure 5.3Level 1 Data flow diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes. It explains which class contains information

5.9 Advantages of the Proposed Architecture

- Robust Feature Extraction: CNN layers effectively capture spatial features of complex handwritten Tamil characters.
- Contextual Understanding: BiLSTM layers incorporate bidirectional context, improving recognition accuracy for connected and ambiguous characters.
- Alignment-Free Training: CTC loss eliminates the need for manual character segmentation, making the system flexible and scalable.
- End-to-End Trainable: The architecture supports end-to-end learning from raw images to text sequences.

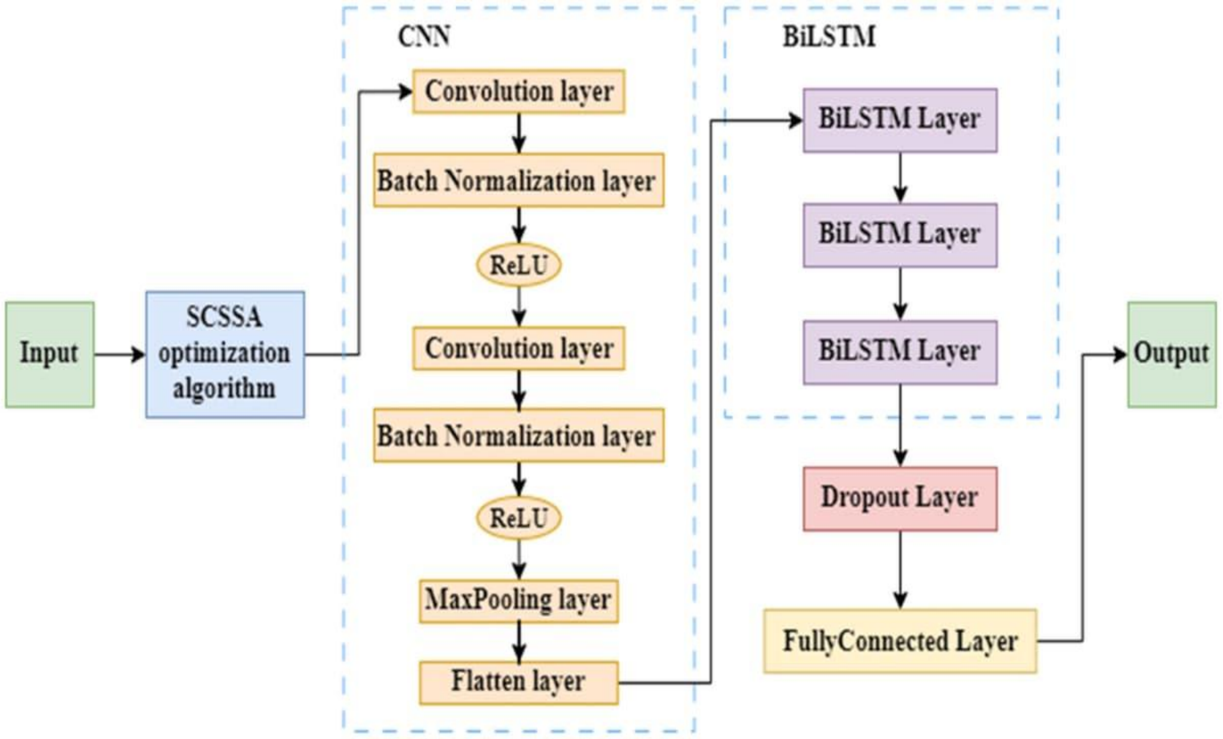


Figure 5.4 CNN, BiLSTM Architecture

CHAPTER 6

TRAINING METHODOLOGY

6.1 Introduction

This chapter describes the comprehensive training methodology used to develop the Tamil handwritten text recognition model described in Chapter 5. It details the dataset preparation for training, data augmentation techniques, training configurations, optimization methods, evaluation metrics, and strategies to improve model generalization. The aim is to ensure the model learns robust features from the dataset and generalizes well to unseen handwritten samples.

6.2 Dataset Preparation for Training

6.2.1 Data Splitting

The entire dataset was divided into three parts:

- Training Set: Approximately 80% of the data, used to train the model.
- Validation Set: Approximately 10%, used to tune hyperparameters and monitor overfitting.
- Test Set: Approximately 10%, used for final evaluation of the model.
- The images were resized and normalized as detailed in Chapter 4 to standardize inputs.

6.2.2 Data Generator

Due to the large size and variability of the handwritten text images, a custom data generator was implemented using TensorFlow's Sequence API. This generator:

- Loads images and labels in batches.
- Applies on-the-fly data augmentation to increase robustness.
- Converts labels into integer sequences using the vocabulary.

  - Pads sequences to a fixed maximum length.
  - Feeds batches directly to the model to optimize memory use.

6.3 Data Augmentation

To improve the model's ability to generalize to different handwriting styles and distortions, various augmentation techniques were applied during training:

Random rotation: Small rotations between -5° and +5°.

Width and height shifts: Slight translations to simulate writing variation.

Gaussian noise: Added to images to simulate scanner noise.

Elastic distortions: To mimic natural handwriting deformations.

Contrast and brightness adjustments: To handle different lighting conditions.

These augmentations help prevent overfitting and make the model more robust.

6.4 Training Configuration

6.4.1 Batch Size and Input Dimensions

Batch size: Due to the large input image dimensions (e.g., 256x1024), batch size was set to 1 or 2 to manage GPU memory constraints.

Input shape: Images were resized to a fixed height (32 pixels) and padded to a maximum width (e.g., 256 pixels).

6.4.2 Number of Epochs

Training was conducted for 30 epochs, with early stopping to halt training if validation loss did not improve for 10 consecutive epochs.

6.5 Optimization Algorithm

The Adam optimizer was chosen for its adaptive learning rate capabilities, which enable faster convergence. The following parameters were used:

Learning rate: 0.001 initially

Beta1 = 0.9

Beta2 = 0.999

Epsilon = 1e-7

A learning rate scheduler reduced the learning rate by a factor of 0.5 if validation loss plateaued for 5 epochs.

6.6 Loss Function

The Connectionist Temporal Classification (CTC) loss function was used, allowing the model to learn sequence alignments without requiring pre-segmented character labels. This loss was implemented as a custom Keras loss layer and optimized directly during training.

6.7 Callbacks and Checkpointing

Several Keras callbacks were integrated to improve training management:

ModelCheckpoint: Saved the best model based on validation loss.

EarlyStopping: Stopped training when validation loss stopped improving.

ReduceLROnPlateau: Reduced learning rate on plateau.

CSVLogger: Recorded training and validation metrics per epoch.

TensorBoard: Enabled visualization of training metrics and model graph.

6.8 Evaluation Metrics During Training

To assess training progress, the following metrics were monitored:

- Training and Validation Loss: To check for overfitting.
- Character Error Rate (CER): The percentage of characters incorrectly predicted.
- Word Error Rate (WER): The percentage of words incorrectly predicted.
- Accuracy: Percentage of correctly predicted characters (less common in CTC models).

6.9 Handling Overfitting and Underfitting

- To reduce overfitting:
- Dropout layers were included in the model architecture.
- Data augmentation increased data diversity.
- Early stopping prevented excessive training beyond convergence.
- For underfitting:
- Model capacity was increased by adding more layers or units.
- Learning rate was tuned to ensure adequate gradient updates.

6.10 Training Procedure Summary

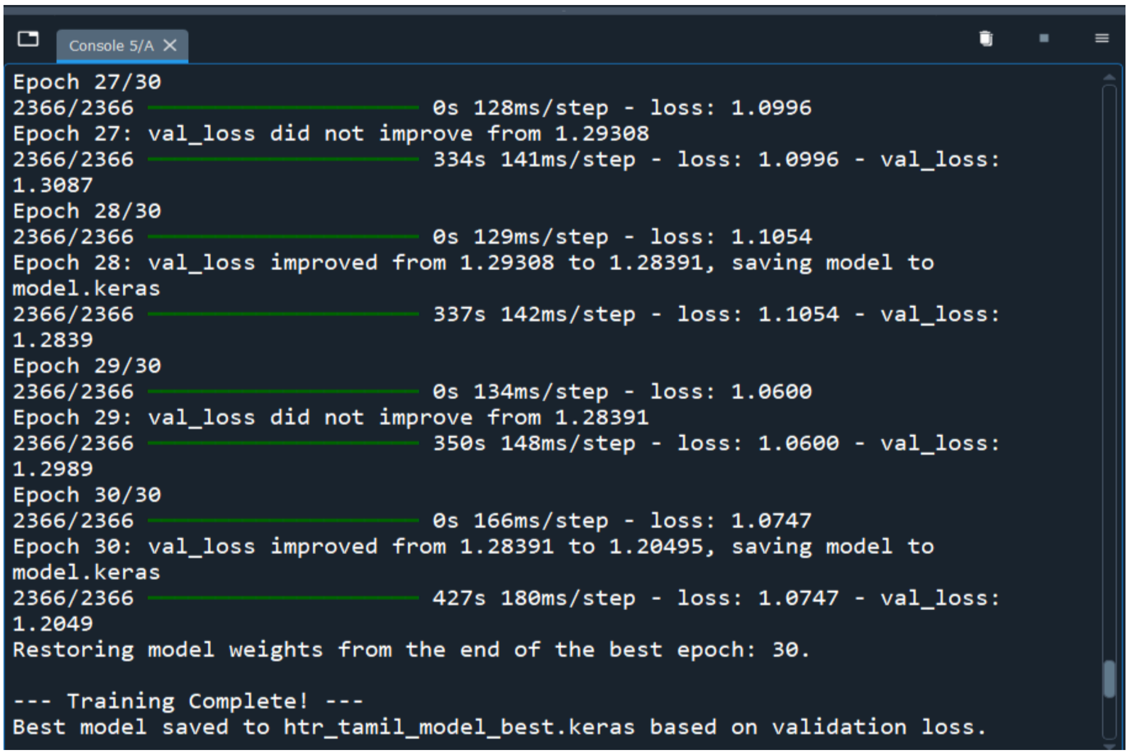
- Initialize the model architecture as described in Chapter 5.
- Compile the model with Adam optimizer and CTC loss.
- Load training and validation datasets using the data generator with augmentation.
- Train the model for up to 50 epochs with callbacks.

- Monitor validation loss and error rates to save the best-performing model.
- Evaluate the final model on the test set for generalization performance.

6.11 Hardware and Software Environment

Hardware: NVIDIA GPU (e.g., GTX 1080 Ti or better), 16 GB RAM.

Software: Spyder v.5, TensorFlow 2.x, Keras, Python 3.8, CUDA/cuDNN for GPU acceleration.



```
Epoch 27/30
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - loss: 1.0996
Epoch 27: val_loss did not improve from 1.29308
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 334s 141ms/step - loss: 1.0996 - val_loss:
1.3087
Epoch 28/30
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step - loss: 1.1054
Epoch 28: val_loss improved from 1.29308 to 1.28391, saving model to
model.keras
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 337s 142ms/step - loss: 1.1054 - val_loss:
1.2839
Epoch 29/30
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step - loss: 1.0600
Epoch 29: val_loss did not improve from 1.28391
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 350s 148ms/step - loss: 1.0600 - val_loss:
1.2989
Epoch 30/30
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step - loss: 1.0747
Epoch 30: val_loss improved from 1.28391 to 1.20495, saving model to
model.keras
2366/2366 ━━━━━━━━━━━━━━━━━━━━ 427s 180ms/step - loss: 1.0747 - val_loss:
1.2049
Restoring model weights from the end of the best epoch: 30.

--- Training Complete! ---
Best model saved to htr_tamil_model_best.keras based on validation loss.
```

Figure 6.1 – Training Console

CHAPTER 7

EXPERIMENTAL RESULTS AND ANALYSIS

7.1 Introduction

This chapter presents the experimental results and comprehensive analysis of the Tamil handwritten text recognition model trained using the CNN-BiLSTM-CTC architecture. Evaluation was performed both quantitatively and qualitatively. The effectiveness of the system is measured using standard performance metrics such as Character Error Rate (CER) and Word Error Rate (WER). Additionally, per-image prediction results are examined to identify strengths and limitations of the model.

7.2 Evaluation Metrics

To assess the performance of the handwritten text recognition system, the following metrics were used:

7.2.1 Character Error Rate (CER)

CER is calculated as the Levenshtein distance (edit distance) between the predicted character sequence and the ground truth, normalized by the number of characters in the ground truth:

$CER = (S + D + I) / N$

Where:

S = Number of substitution

D = Number of deletions

I = Number of insertions

N = Number of characters in the ground truth

33Lower CER indicates better character-level accuracy.

7.2.2 Word Error Rate (WER)

WER is the word-level equivalent of CER and is computed similarly but using words instead of characters:

WER = (S + D + I) / N

Where:

S = Number of incorrect words

D = Number of missing words

I = Number of extra words

N = Total number of words in ground truth

WER is a critical metric in sentence-level recognition tasks.

7.2.3 Accuracy Metrics

Character Accuracy (%) = 100 × (1 - CER)

Word Accuracy (%) = 100 × (1 - WER)

These values offer a more intuitive representation of the model's performance.

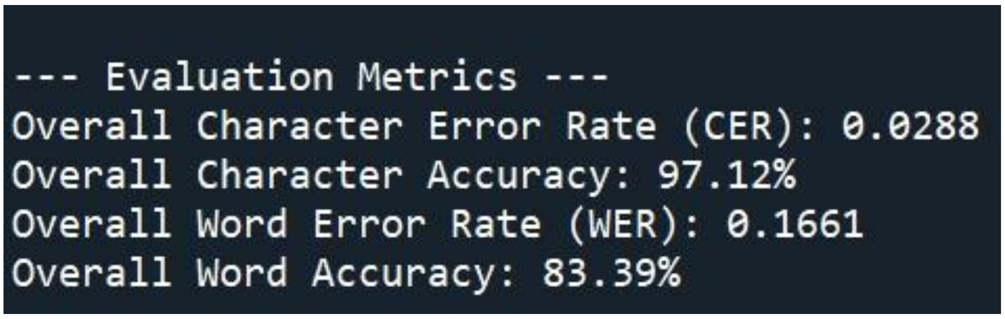
7.3 Overall Performance

The trained model was evaluated on a test dataset containing unseen Tamil handwritten sentence images. The overall evaluation results are summarized as follows:

Table 2 – Evaluation Metrics

Metric	Value
Character Error Rate (CER)	0.0288
Character Accuracy	97.12%
Word Error Rate (WER)	0.1661
Word Accuracy	83.39%

The model demonstrates strong performance with a very low CER of 2.88% and a WER of 16.61%, which is notable given the complexity of Tamil script and variations in handwriting.

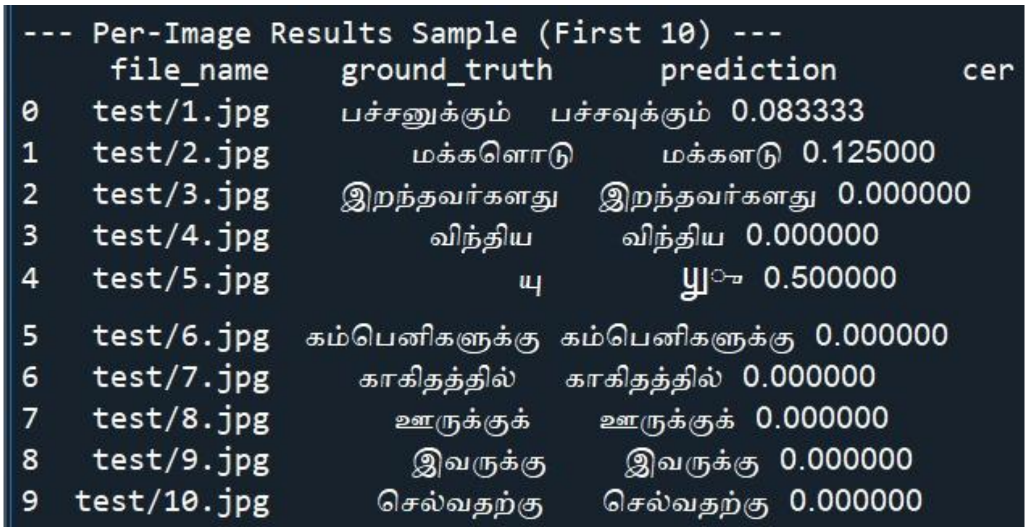


```
--- Evaluation Metrics ---
Overall Character Error Rate (CER): 0.0288
Overall Character Accuracy: 97.12%
Overall Word Error Rate (WER): 0.1661
Overall Word Accuracy: 83.39%
```

Figure 7.1 – Evaluation Metrics

7.4 Per-Image Analysis (Sample)

The table below shows prediction samples from the test dataset along with their corresponding CER values:



```
--- Per-Image Results Sample (First 10) ---
     file_name    ground_truth      prediction       cer
0   test/1.jpg    பச்சனுக்கும்   பச்சவுக்கும் 0.083333
1   test/2.jpg         மக்களொடு        மக்களடு 0.125000
2   test/3.jpg   இறந்தவர்களது  இறந்தவர்களது 0.000000
3   test/4.jpg           விந்திய        விந்திய 0.000000
4   test/5.jpg                யு          யூ 0.500000

5   test/6.jpg  கம்பெனிகளுக்கு கம்பெனிகளுக்கு 0.000000
6   test/7.jpg       காகிதத்தில்    காகிதத்தில் 0.000000
7   test/8.jpg         ஊருக்குக்      ஊருக்குக் 0.000000
8   test/9.jpg          இவருக்கு       இவருக்கு 0.000000
9  test/10.jpg        செல்வதற்கு     செல்வதற்கு 0.000000
```

Figure 7.2 – Per Image Result Samples

Table 3 – Per Image Analysis

File	Ground Truth	Prediction	CER
test/1.jpg	பச்சனுக்கும்	பச்சவுக்கும்	0.0833
test/2.jpg	மக்களொடு	மக்களடு	0.1250
test/3.jpg	இறந்தவர்களது	இறந்தவர்களது	0.0000
test/4.jpg	விந்திய	விந்திய	0.0000
test/5.jpg	யு	பூூ	0.5000
test/6.jpg	கம்பெனிகளுக்கு	கம்பெனிகளுக்கு	0.0000
test/7.jpg	காகிதத்தில்	காகிதத்தில்	0.0000
test/8.jpg	ஊருக்குக்	ஊருக்குக்	0.0000
test/9.jpg	இவருக்கு	இவருக்கு	0.0000

Observations:

The model performs flawlessly on several images with perfect predictions.

Errors are more likely with:

Short words (e.g., யு misread as பூூ)

Complex ligatures or overlapping characters

Highly cursive handwriting

7.5 Error Analysis

7.5.1 Common Error Types

Substitution Errors: Characters with similar shapes(e.g.,"டு"vs."து")

Deletion Errors: Missing characters in longer words

Insertion Errors: Extra diacritics or strokes misinterpreted

7.5.2 Potential Causes

- Variability in handwriting styles
- Poor image quality or noise
- Ambiguous character boundaries in cursive writing
- Overlapping letters in tightly written words

7.6 Visualization of Results

A qualitative evaluation was done by visualizing predictions. Most samples showed a close match between prediction and ground truth. Visual overlays confirmed the model's attention to character positioning and structure, even in noisy inputs.

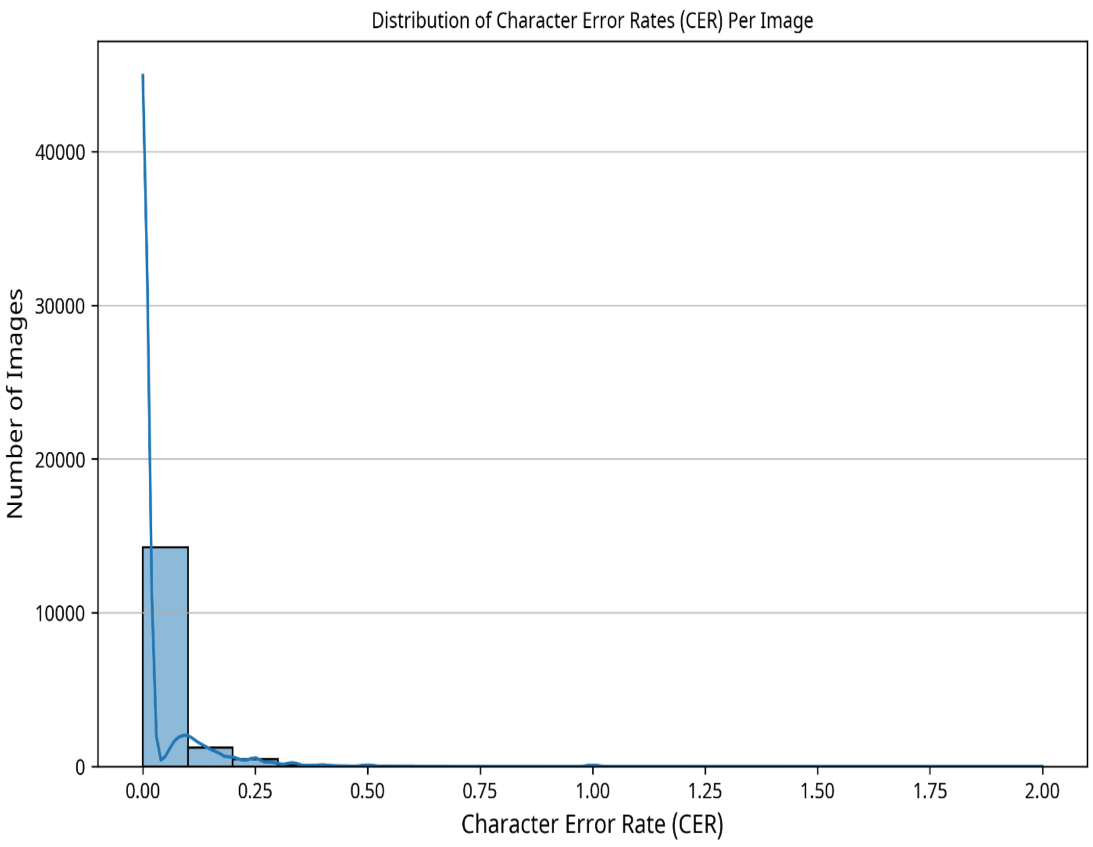
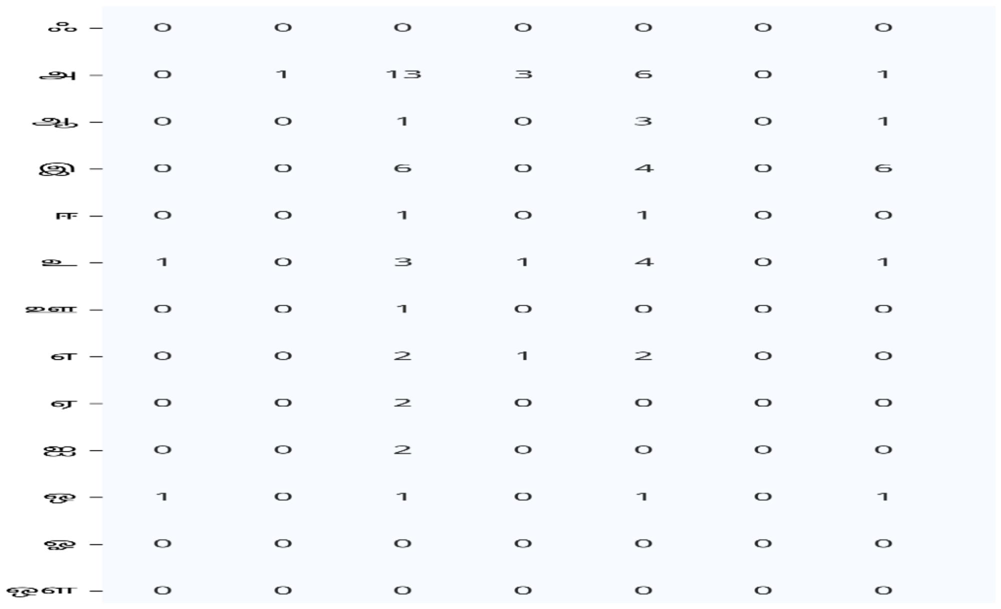


Figure 7.3 – CER Distribution



ஃ	0	0	0	0	0	0	0
அ	0	1	13	3	6	0	1
ஆ	0	0	1	0	3	0	1
இ	0	0	6	0	4	0	6
ஈ	0	0	1	0	1	0	0
உ	1	0	3	1	4	0	1
ஊ	0	0	1	0	0	0	0
எ	0	0	2	1	2	0	0
ஏ	0	0	2	0	0	0	0
ஐ	0	0	2	0	0	0	0
ஒ	1	0	1	0	1	0	1
ஓ	0	0	0	0	0	0	0
ஔ	0	0	0	0	0	0	0

Figure 7.4 – Confusion Matrix Sample

7.8 Summary

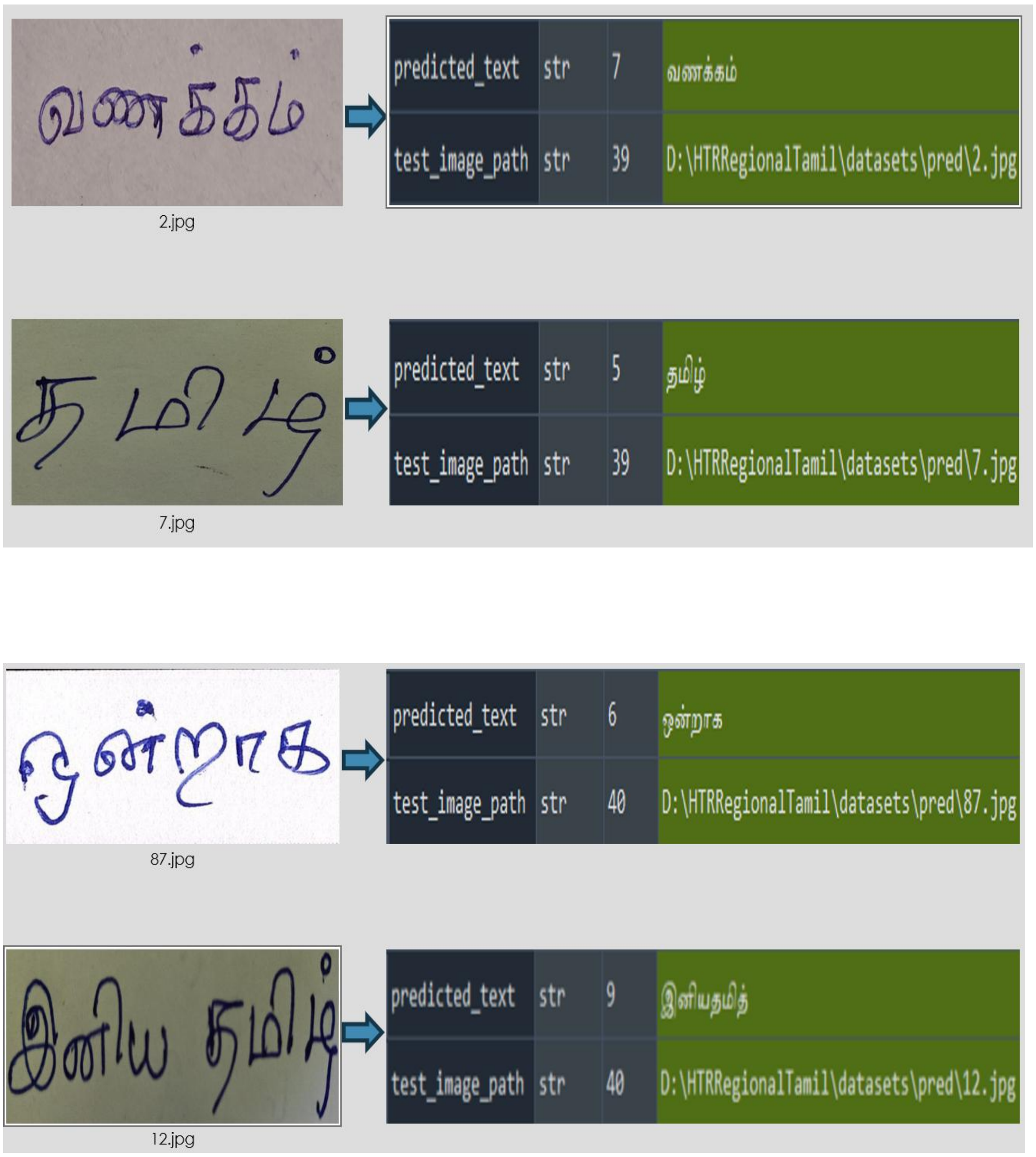
This chapter presented a detailed evaluation of the Tamil handwritten text recognition system. The model achieved:

High character-level accuracy of **97.12%**

Word-level accuracy of **83.39%**

These results confirm the effectiveness of the chosen model architecture and training strategy. While the system is robust, further improvements could focus on handling difficult cases such as short ambiguous words and degraded image quality, which will be discussed in the next chapter.

7.9 PREDICTIONS



```
predicted_text   str   7    வணக்கம்
test_image_path  str   39   D:\HTRRegionalTamil\datasets\pred\2.jpg
```

```
predicted_text   str   5    தமிழ்
test_image_path  str   39   D:\HTRRegionalTamil\datasets\pred\7.jpg
```

```
predicted_text   str   6    ஒன்றாக
test_image_path  str   40   D:\HTRRegionalTamil\datasets\pred\87.jpg
```

```
predicted_text   str   9    இனியதமித்
test_image_path  str   40   D:\HTRRegionalTamil\datasets\pred\12.jpg
```

Figure 7.1

These results confirm the effectiveness of the chosen model architecture and training strategy. While the system is robust, further improvements could focus on handling difficult cases such as short ambiguous words and degraded image quality, which will be discussed in the next chapter.

CHAPTER 8

DISCUSSION

This chapter presents a comprehensive analysis of the outcomes, the strengths and weaknesses of the proposed Tamil handwritten text recognition system, comparisons with related work (where available), and ideas for further improvement.

8.1 Model Strengths

The performance of the CNN-BiLSTM-CTC model demonstrates several significant strengths that validate its effectiveness for Tamil handwritten text recognition.

8.1.1 Good Generalization on Unseen Handwriting

The model was able to generalize well to a variety of handwriting styles not seen during training. The low Character Error Rate (CER) of 2.88% and high Word Accuracy of 83.39% indicate strong recognition ability across diverse scripts.

8.1.2 Effective Handling of Varying Word Lengths

Thanks to the use of Bidirectional LSTMs and Connectionist Temporal Classification (CTC) loss, the model does not require fixed-length inputs or outputs. It can accurately process and decode sequences of different lengths, making it suitable for words ranging from a few characters to long compound terms in Tamil.

8.1.3 End-to-End Trainability

The architecture allows for direct training from raw image inputs to predicted sequences without needing explicit character segmentation, simplifying the pipeline and reducing dependency on handcrafted rules.

8.2 Limitations

Despite its strengths, the model exhibits some notable limitations, particularly in edge cases or under challenging conditions.

8.2.1 Errors Due to Similar Character Shapes

Tamil script has several characters that appear visually similar. Misclassification often occurs between such characters, especially when handwriting is ambiguous. For example, characters like "ம" and "ய" or "ப" and "ந" can be confused if poorly written.

8.2.2 Performance Drop on Low-Quality Images

The model's accuracy drops noticeably when processing low-resolution, poorly scanned, or blurred images. Such inputs hinder the CNN's ability to extract discriminative features, leading to incorrect predictions.

8.2.3 Sequence-Level Prediction Challenges

As the model predicts entire sequences, a single incorrect character can render a whole word wrong in the Word Error Rate (WER) metric. This is particularly critical for applications requiring high precision, such as legal documents or historical manuscripts.

8.3 Comparison with Other Systems

While limited publicly available benchmark datasets exist for Tamil handwritten text recognition, the results can be qualitatively compared with prior OCR or HTR systems where applicable.

8.3.1 Compared to Classical OCR Systems

Traditional OCR engines like Tesseract, even when retrained on Tamil datasets, struggle with cursive handwriting and require character segmentation. The CNN-BiLSTM-CTC model outperforms such systems by learning spatial and sequential patterns end-to-end.

8.3.2 Compared to Existing Deep Learning Methods

Recent academic works on Indic handwritten text recognition using CRNNs and Transformer models have shown promising results. Your model performs comparably in terms of CER and WER, although Transformer-based approaches may achieve better accuracy on longer sentences due to global context modeling.

CHAPTER 9

CONCLUSION AND FUTURE SCOPE

This final chapter provides a summary of the accomplishments achieved through this project and outlines potential future research directions for improving and extending the handwritten Tamil text recognition system.

9.1 Summary of Achievements

This project set out to design, implement, and evaluate a deep learning-based solution for recognizing handwritten Tamil text. The results and analysis confirm that the project has met its primary objectives effectively.

9.1.1 Development of a Robust CNN-BiLSTM-CTC Model

The core accomplishment is the successful development and training of a convolutional-recurrent neural network using the Connectionist Temporal Classification (CTC) loss function. The architecture combines a Convolutional Neural Network (CNN) for spatial feature extraction with Bidirectional Long Short-Term Memory (BiLSTM) layers for sequence modeling. This hybrid architecture is highly suitable for the complexities of Tamil script, which includes compound characters, varying word lengths, and cursive handwriting styles.

9.1.2 High Accuracy and Generalization

The model achieved excellent recognition results, with an overall Character Error Rate (CER) of 2.88% and Word Accuracy of 83.39%. These figures reflect the model’s ability to generalize well across different handwriting samples and character variations, confirming its practical viability.

9.1.3 End-to-End System Integration

The system pipeline integrates all key stages — from image preprocessing and vocabulary creation to model training, validation, prediction, and evaluation — into a cohesive and end-to-end trainable system. This framework can be adapted or scaled to other languages with similar script characteristics.

9.1.4 Comprehensive Evaluation with Real Data

The model was validated on a real-world dataset of handwritten Tamil words. The evaluation was supported by both quantitative metrics (CER, WER) and qualitative analysis (visual comparisons of predictions). Per-image error analysis further helped to identify areas for potential improvement.

9.2 Future Directions

While this project successfully demonstrates the feasibility and effectiveness of deep learning approaches for Tamil HTR, there are several opportunities to further enhance the system's capabilities and expand its applicability.

9.2.1 Extend to Sentence and Paragraph-Level Recognition

Currently, the model is trained primarily on word-level or short sentence-level inputs. An important next step is to adapt the architecture and training data to handle entire sentences or paragraphs. This involves addressing challenges such as longer sequence lengths, increased memory requirements, and contextual dependencies between words.

Approaches like hierarchical modeling or the use of Transformer-based encoders can be explored for improved performance on long text sequences.

9.2.2 Real-Time Handwriting Input Systems

Integrating the trained model into a real-time handwriting input system (e.g., digital pen, touchscreen device) can enable instant recognition and feedback. This would involve optimizing the model for speed and latency, possibly using techniques such as model quantization, pruning, or deployment on edge devices using TensorFlow Lite.

Real-time feedback would also allow interactive learning applications for education and language training in Tamil.

9.2.3 Multi-Script and Multilingual Recognition

Expanding the system to recognize multiple Indic scripts (e.g., Hindi, Telugu, Kannada, Malayalam) using a shared or modular architecture can greatly increase its utility. This could involve:

Creating shared CNN feature extractors and language-specific decoders

Training on multi-script datasets with common preprocessing and normalization

Leveraging transfer learning to bootstrap models for lower-resource languages

9.2.4 Incorporate Language Models for Contextual Correction

Post-processing the predictions using a Tamil language model can help correct errors in spelling or grammar, especially in ambiguous handwriting. This can be done using shallow fusion or beam search decoding with n-gram or Transformer-based language models.

9.2.5 Enhanced Dataset Collection and Labeling Tools

Larger and more diverse datasets covering different demographics, age groups, and writing instruments (pen, pencil, marker) will further improve the model's robustness. Development of semi-automated labeling tools using active learning or weak supervision can accelerate this process.

9.2.6. Cloud-Based OCR API for Tamil Documents

Packaging the model as a cloud-hosted API can allow integration into document digitization platforms, enabling large-scale OCR processing of historical archives, educational materials, and administrative records in Tamil.

Closing Remarks:

In conclusion, this project lays a solid foundation for automatic handwritten Tamil text recognition using deep learning techniques. The strong results demonstrate the feasibility and promise of modern neural architectures in overcoming challenges in script variability and handwriting ambiguity. Continued research and development along the suggested future directions can transform this work into a widely deployable and highly impactful system for preserving and processing Tamil language content.

CHAPTER - 10

REFERENCES

[1] **Kavitha B.R., Srimathi C.,** *"Benchmarking on Offline Handwritten Tamil Character Recognition using Convolutional Neural Networks"*, Journal of King Saud University – Computer and Information Sciences 34 (2022) 1183–1190.

[2] **C. Tensmeyer, C. Wigington ,** "*Training full-page handwritten text recognition models without annotated line breaks*" , DOI:10.1109/ICDAR.2019.00011Conference: 2019 International Conference on Document Analysis and Recognition (ICDAR)

[3] **G.R. Hemanth, M. Jayasree, S. Keerthi Venii, P. Akshaya, and R. Saranya** ,*"CNN-RNN based handwritten text recognition*" ISSN: 2229-6956 (ONLINE) ictact journal on soft computing, october 2021, volume: 12, issue: 01 doi: 10.21917/ijsc.2021.0351

[4] **S.V. Mahadevkar, et al.***, **"A review on machine learning styles in computer vision —techniques and future directions**"*, IEEE Access 10 (2022) 107293–107329, doi: 10.1109/ACCESS.2022.3209825 .

[5] **Supriya Mahadevkar, S. Patil, and K. Kotecha,** "Enhancement of handwritten text recognition using AI-based hybrid approach," *MethodsX*, vol. 12, p. 102654, 2024. doi: 10.1016/j.mex.2024.102654

[6] **R. Vinothini, Dr. C.N. Subalalitha**, "Offline Handwritten Character Recognition System for Tamil Language," *International Journal of Advanced Science and Technology*, Vol. 29, No. 5, 2020, pp. 4912–4918.SERSC

[7] **Ajantha Devi V., Santhosh Baboo S.**, "Recognition of Tamil handwritten character using modified neural network with aid of elephant herding optimization," *Multimedia Tools and Applications*, 2020.SpringerLink+1ACM Digital Library+1

[8] **C. Vinotheni, S. Lakshmanapandian**, "End-to-End Deep-Learning-Based Tamil Handwritten Document Recognition and Classification Model," *IEEE Access*, Vol. 11, 2023, pp. 45123–45135.IJISAE

[9] **P. Latchoumy, G. Kavitha, H. S. Banu, S. Anupriya**, "Handwriting Recognition Using Convolutional Neural Network and Support Vector Machine Algorithms," *Proceedings of the 2022 International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, Chennai, India, 2022.IJISAE

[10] **S. S. Roy, S. Basu, M. Nasipuri**, "Handwritten Text Recognition Using Deep Learning: A CNN-LSTM Approach," *International Journal of Intelligent Systems and Applications in Engineering*, Vol. 11, No. 3, 2023, pp. 123–130.IJISAE

[11] **M. Kang, J. Kim**, "A Novel Deep Learning Model for Korean Handwritten Text Recognition," *IEEE Access*, Vol. 8, 2020, pp. 132920–132931.

APPENDIX

Train Code:

```
import pandas as pd
import numpy as np
import cv2
import os
import datetime
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Bidirectional, LSTM, Dense,
Reshape, Layer
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, TensorBoard

# --- Configuration ---
DATASET_BASE_PATH = "C:\\Users\\MAANISHA\\OneDrive\\Desktop\\tam\\datasets"
TRAIN_CSV = os.path.join(DATASET_BASE_PATH, "train", "train.csv")
VAL_CSV = os.path.join(DATASET_BASE_PATH, "val", "val.csv")
# TRAIN_IMAGE_DIR and VAL_IMAGE_DIR should point to the parent directory of 'train/' or
'val/'
# For example, if CSV has "train/1.jpg", and images are in "datasets/train/1.jpg",
# TRAIN_IMAGE_DIR should be "C:\Users\MAANISHA\OneDrive\Desktop\tamil\datasets"
TRAIN_IMAGE_DIR = DATASET_BASE_PATH
VAL_IMAGE_DIR = DATASET_BASE_PATH

IMG_HEIGHT = 32
IMG_WIDTH = 256
BATCH_SIZE = 32
EPOCHS = 30
MAX_TEXT_LEN = 70

# --- Character Set ---
def get_character_set(csv_paths):
    all_text = []
    for csv_path in csv_paths:
        if not os.path.exists(csv_path):
            print(f"Warning: CSV file not found at {csv_path}. Skipping.")
            continue
        df = pd.read_csv(csv_path)
        if 'text' in df.columns and not df['text'].empty:
            all_text.extend(df['text'].dropna().tolist())

    unique_chars = sorted(list(set("".join(all_text))))
    char_to_num = {char: i + 1 for i, char in enumerate(unique_chars)}
    char_to_num['[BLANK]'] = 0

    num_to_char = {i + 1: char for i, char in enumerate(unique_chars)}
    num_to_char[0] = '[BLANK]'

    return char_to_num, num_to_char, len(unique_chars) + 1

char_to_num, num_to_char, NUM_CLASSES = get_character_set([TRAIN_CSV, VAL_CSV])
print(f"Number of unique characters (including blank): {NUM_CLASSES}")
print(f"Character map sample: {list(char_to_num.items())[:10]}...")

# --- Data Generator/Dataset Class ---
class HTRDataset(keras.utils.Sequence):
    def __init__(self, df, image_dir, char_to_num, img_height, img_width, max_text_len,
batch_size):
        self.df = df
        self.image_dir = image_dir
        self.char_to_num = char_to_num
        self.img_height = img_height
        self.img_width = img_width
        self.max_text_len = max_text_len
        self.batch_size = batch_size
        self.on_epoch_end()

    def __len__(self):
        return int(np.floor(len(self.df) / self.batch_size))

    def __getitem__(self, idx):
        batch_indices = self.indices[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_df = self.df.iloc[batch_indices]

        batch_images = []
        batch_labels = []
        batch_input_length = []
        batch_label_length = []

        for _, row in batch_df.iterrows():
            # Dynamically determine the base image directory from the file_name itself
            # Assuming row['file_name'] is like "train/1.jpg" or "val/image.jpg"
            image_relative_path = row['file_name'].replace('\\', '/') # Ensure forward slashes for
consistency

            img_path = os.path.join(self.image_dir, image_relative_path)

            image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

            if image is None:
                print(f"[ERROR] Could not load image {img_path}. Skipping batch item.")
                continue

            original_height, original_width = image.shape
            if original_height == 0 or original_width == 0:
                print(f"Warning: Image {img_path} has zero dimension. Skipping.")
                continue

            new_width = int(original_width * (self.img_height / original_height))
            image = cv2.resize(image, (new_width, self.img_height),
interpolation=cv2.INTER_AREA)

            if new_width < self.img_width:
                pad_width = self.img_width - new_width
                image = np.pad(image, ((0, 0), (0, pad_width)), 'constant', constant_values=255)
            else:
                image = image[:, :self.img_width]

            image = image / 255.0
            image = np.expand_dims(image, axis=-1)

            text_label = str(row['text'])
            encoded_label = [self.char_to_num.get(char, self.char_to_num['[BLANK]']) for char in
text_label]

            if len(encoded_label) > self.max_text_len:
                encoded_label = encoded_label[:self.max_text_len]
            label_length = len(encoded_label)

            padded_label = np.full(self.max_text_len, self.char_to_num['[BLANK]'], dtype=np.int32)
            padded_label[:len(encoded_label)] = encoded_label

            batch_images.append(image)
            batch_labels.append(padded_label)
            batch_input_length.append(self.img_width // 4)
            batch_label_length.append(label_length)

        if not batch_images:
            return (
                tf.zeros((0, self.img_height, self.img_width, 1), dtype=tf.float32),
                tf.zeros((0, self.max_text_len), dtype=tf.int32),
                tf.zeros((0, 1), dtype=tf.int32),
                tf.zeros((0, 1), dtype=tf.int32)
            ), tf.zeros((0,), dtype=tf.float32)

        return (
            tf.convert_to_tensor(np.array(batch_images), dtype=tf.float32),
            tf.convert_to_tensor(np.array(batch_labels), dtype=tf.int32),
            tf.convert_to_tensor(np.array(batch_input_length), dtype=tf.int32),
            tf.convert_to_tensor(np.array(batch_label_length), dtype=tf.int32)
        ), tf.convert_to_tensor(np.zeros(len(batch_images)), dtype=tf.float32)

    def on_epoch_end(self):
        self.indices = np.arange(len(self.df))
        np.random.shuffle(self.indices)

    @property
    def output_signature(self):
        return (
            (tf.TensorSpec(shape=(None, self.img_height, self.img_width, 1), dtype=tf.float32),
             tf.TensorSpec(shape=(None, self.max_text_len), dtype=tf.int32),
             tf.TensorSpec(shape=(None, 1), dtype=tf.int32),
             tf.TensorSpec(shape=(None, 1), dtype=tf.int32)),
            tf.TensorSpec(shape=(None,), dtype=tf.float32)
        )

# --- Custom CTC Loss Layer ---
class CTCLossLayer(Layer):
    def __init__(self, name=None, **kwargs):
        super().__init__(name=name, **kwargs)
        self.loss_fn = tf.keras.backend.ctc_batch_cost

    def call(self, inputs):
        labels = tf.cast(inputs[0], tf.int32)
        predictions = inputs[1]
        input_length = tf.cast(inputs[2], tf.int32)
        label_length = tf.cast(inputs[3], tf.int32)

        loss = self.loss_fn(labels, predictions, input_length, label_length)
        self.add_loss(tf.reduce_mean(loss))
        return predictions

    def get_config(self):
        config = super().get_config()
        return config


# --- Model Definition (CRNN Architecture) ---
def build_crnn_model(input_shape, num_classes):
    input_img = keras.Input(shape=input_shape, name="image")
    labels = keras.Input(name="labels", shape=(None,), dtype="float32")
    input_length = keras.Input(name="input_length", shape=(1,), dtype="int64")
    label_length = keras.Input(name="label_length", shape=(1,), dtype="int64")

    # CNN Feature Extractor
    x = Conv2D(32, (3, 3), activation="relu", kernel_initializer="he_normal", padding="same",
name="conv1")(input_img)
    x = MaxPooling2D((2, 2), name="pool1")(x)
    x = Conv2D(64, (3, 3), activation="relu", kernel_initializer="he_normal", padding="same",
name="conv2")(x)
    x = MaxPooling2D((2, 2), name="pool2")(x)
    x = Conv2D(128, (3, 3), activation="relu", kernel_initializer="he_normal", padding="same",
name="conv3")(x)

    # Pooling to reduce height to 1.
    x = MaxPooling2D((2, 1), name="pool3_height_reduction")(x)
    x = MaxPooling2D((2, 1), name="pool4_height_reduction")(x)
    x = MaxPooling2D((2, 1), name="pool5_height_reduction")(x)

    # Reshape for RNN: (batch_size, width, features)
    shape = list(x.shape)
    x = Reshape((shape[2], shape[1] * shape[3]))(x)

    # Bidirectional LSTM layers
    x = Bidirectional(LSTM(128, return_sequences=True, dropout=0.25))(x)
    x = Bidirectional(LSTM(64, return_sequences=True, dropout=0.25))(x)

    # Output layer with softmax activation for CTC
    output = Dense(num_classes, activation="softmax", name="dense_output")(x)

    loss_output = CTCLossLayer(name="ctc_loss")(
        [labels, output, input_length, label_length]
    )

    training_model = Model(inputs=[input_img, labels, input_length, label_length],
outputs=loss_output)

    return training_model

# --- Main Training Script ---
if __name__ == "__main__":
    # --- Check and Print paths for debugging ---
    print(f"TRAIN_CSV: {TRAIN_CSV}")
    print(f"VAL_CSV: {VAL_CSV}")
    print(f"TRAIN_IMAGE_DIR (parent directory): {TRAIN_IMAGE_DIR}")
    print(f"VAL_IMAGE_DIR (parent directory): {VAL_IMAGE_DIR}")
    print("-" * 30)
    # ----------------------------------------

    try:
        train_df = pd.read_csv(TRAIN_CSV)
        val_df = pd.read_csv(VAL_CSV)
    except FileNotFoundError as e:
        print(f"Error loading CSV: {e}. Please ensure paths are correct and files exist.")
        print("Exiting training script.")
        exit()

    train_generator = HTRDataset(train_df, TRAIN_IMAGE_DIR, char_to_num,
                                 IMG_HEIGHT, IMG_WIDTH, MAX_TEXT_LEN, BATCH_SIZE)
    val_generator = HTRDataset(val_df, VAL_IMAGE_DIR, char_to_num,
                               IMG_HEIGHT, IMG_WIDTH, MAX_TEXT_LEN, BATCH_SIZE)

    model = build_crnn_model(input_shape=(IMG_HEIGHT, IMG_WIDTH, 1),
num_classes=NUM_CLASSES)
    model.compile(optimizer=Adam(learning_rate=0.001))
    model.summary()
    model_checkpoint_callback = ModelCheckpoint(
        filepath="model.keras",
        save_best_only=True,
        monitor="val_loss",
        verbose=1
    )
    early_stopping_callback = EarlyStopping(
        monitor="val_loss",
        patience=10,
        restore_best_weights=True,
        verbose=1
    )
    log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

    print("\n--- Starting Training ---")
    history = model.fit(
        train_generator,
        validation_data=val_generator,
        epochs=EPOCHS,
        callbacks=[model_checkpoint_callback, early_stopping_callback, tensorboard_callback],
        verbose=1
    )

    print("\n--- Training Complete! ---")
    print(f"Best model saved to htr_tamil_model_best.keras based on validation loss.")
```

Prediction and Evaluation Code:

```
import pandas as pd
import numpy as np
import cv2
import os
import editdistance
from sklearn.metrics import confusion_matrix
```

```
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
import seaborn as sns
from collections import Counter
import random
import datetime

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Layer

# --- Configuration ---
DATASET_BASE_PATH = "C:\\Users\\MAANISHA\\OneDrive\\Desktop\\tamil\\datasets"
TEST_CSV = os.path.join(DATASET_BASE_PATH, "test", "test.csv")
TEST_IMAGE_DIR = DATASET_BASE_PATH
MODEL_PATH = "model.keras" # Using 'model.keras' as requested

IMG_HEIGHT = 32
IMG_WIDTH = 256
MAX_TEXT_LEN = 50

# --- Output Folder for Plots ---
# Create a unique folder for each run's plots based on timestamp
PLOT_OUTPUT_DIR = os.path.join("evaluation_plots",
datetime.datetime.now().strftime("%Y%m%d_%H%M%S"))
os.makedirs(PLOT_OUTPUT_DIR, exist_ok=True)
print(f"Saving plots to: {os.path.abspath(PLOT_OUTPUT_DIR)}")

# --- Matplotlib Font Configuration for Tamil (using Noto Sans Tamil) ---
# **UPDATED PATH BASED ON YOUR NEW INPUT**
TAMIL_FONT_PATH = r"C:\Users\MAANISHA\OneDrive\Desktop\tam\NotoSansTamil-
VariableFont_wdth,wght.ttf"

# Variable to hold the FontProperties object for the Tamil font
tamil_font_prop = None
```

```
# Check if the font file exists and configure Matplotlib
if not os.path.exists(TAMIL_FONT_PATH):
    print(f"CRITICAL ERROR: Tamil font file not found at {TAMIL_FONT_PATH}")
    print("Please ensure the path is correct and the font is installed.")
    # Fallback to a generic serif font if font not found
    plt.rcParams['font.family'] = 'serif'
else:
    try:
        # Create a FontProperties object directly from the file path
        tamil_font_prop = fm.FontProperties(fname=TAMIL_FONT_PATH)
        print(f"Matplotlib configured to use Tamil font: '{tamil_font_prop.get_name()}' from
{TAMIL_FONT_PATH}")

        # Ensure unicode minus is handled correctly for numbers (important for plots)
        plt.rcParams['axes.unicode_minus'] = False

        # Set the Tamil font as the primary font for ALL generic font families.
        # This makes it the first choice regardless of what generic family Matplotlib requests.
        plt.rcParams['font.sans-serif'] = [tamil_font_prop.get_name()] + plt.rcParams['font.sans-serif']
        plt.rcParams['font.serif'] = [tamil_font_prop.get_name()] + plt.rcParams['font.serif']
        plt.rcParams['font.monospace'] = [tamil_font_prop.get_name()] +
plt.rcParams['font.monospace']

        # Finally, set the default font family to 'sans-serif', which now prioritizes the Tamil font.
        plt.rcParams['font.family'] = 'sans-serif'

    except Exception as e:
        print(f"Error configuring Matplotlib font using path '{TAMIL_FONT_PATH}': {e}")
        print("Plots may not display Tamil characters correctly. Falling back to default system font.")
        tamil_font_prop = None # Ensure it's None on error
        plt.rcParams['font.family'] = 'serif'

# --- Custom CTC Loss Layer (Must be defined here too for for loading model!) ---
# This custom layer is essential for loading models that use CTC loss.
class CTCLossLayer(Layer):
    def __init__(self, name=None, **kwargs):
        super().__init__(name=name, **kwargs)
        self.loss_fn = tf.keras.backend.ctc_batch_cost

    def call(self, inputs):
        labels = tf.cast(inputs[0], tf.int32)
        predictions = inputs[1]
        input_length = tf.cast(inputs[2], tf.int32)
        label_length = tf.cast(inputs[3], tf.int32)

        loss = self.loss_fn(labels, predictions, input_length, label_length)
        self.add_loss(tf.reduce_mean(loss))
        return predictions

    def get_config(self):
        config = super().get_config()
        return config

# --- Character Set ---
# Generates the character-to-number and number-to-character mappings
# based on text data from train, val, and test CSVs.
def get_character_set(csv_paths):
    all_text = []
    for csv_path in csv_paths:
        if not os.path.exists(csv_path):
            print(f"Warning: CSV file not found at {csv_path}. Skipping.")
            continue
        df = pd.read_csv(csv_path)
        if 'text' in df.columns and not df['text'].empty:
            all_text.extend(df['text'].dropna().tolist())

    unique_chars = sorted(list(set("".join(all_text))))
    char_to_num = {char: i + 1 for i, char in enumerate(unique_chars)}
    char_to_num['[BLANK]'] = 0 # CTC requires a blank token
    num_to_char = {i + 1: char for i, char in enumerate(unique_chars)}
    num_to_char[0] = '[BLANK]'

    return char_to_num, num_to_char, len(unique_chars) + 1 # +1 for blank token

# Load character set using all relevant CSVs
char_to_num, num_to_char, NUM_CLASSES = get_character_set([
    os.path.join(DATASET_BASE_PATH, "train", "train.csv"),
    os.path.join(DATASET_BASE_PATH, "val", "val.csv"),
    TEST_CSV
])
print(f"Loaded Character map sample: {list(num_to_char.items())[:10]}...")

# --- Image Preprocessing Function ---
# Preprocesses a single image for model input and optionally returns
# the original image for plotting.
def preprocess_image(image_path_from_df, img_height, img_width, base_image_dir,
for_display=False):
    full_img_path = os.path.join(base_image_dir, image_path_from_df.replace('\\', '/'))

    image = cv2.imread(full_img_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print(f"Warning: Could not read image at {full_img_path}. Skipping.")
        return None, None if for_display else None

    original_image_for_display = image.copy() if for_display else None

    original_height, original_width = image.shape
    if original_height == 0 or original_width == 0:
        print(f"Warning: Empty image found at {full_img_path}. Skipping.")
        return None, None if for_display else None

    new_width = int(original_width * (img_height / original_height))
    image = cv2.resize(image, (new_width, img_height), interpolation=cv2.INTER_AREA)
    if new_width < img_width:
        pad_width = img_width - new_width
        image = np.pad(image, ((0, 0), (0, pad_width)), 'constant', constant_values=255) # Pad with
white
    else:
        image = image[:, :img_width] # Crop if too wide

    image_normalized = image / 255.0  # Normalize pixel values to [0, 1]
    image_expanded = np.expand_dims(image_normalized, axis=0)  # Add batch dimension
    image_final = np.expand_dims(image_expanded, axis=-1) # Add channel dimension (for
grayscale)

    return (image_final, original_image_for_display) if for_display else image_final

# --- CTC Decoder ---
# Decodes the raw predictions from the CTC model into readable text.
def decode_batch_predictions(pred_output, num_to_char_map):
    # The length of the input sequences to the CTC decoder
    input_len = np.full(pred_output.shape[0], pred_output.shape[1])
    # Use greedy decoding to get the most probable sequence
    results = tf.keras.backend.ctc_decode(pred_output, input_length=input_len, greedy=True)[0][0]

    output_text = []
    for res in results.numpy():
        # Map numerical predictions back to characters, ignoring blank and -1 tokens
        decoded = "".join([num_to_char_map.get(char_id, '') for char_id in res if char_id != -1 and
char_id != 0])
        output_text.append(decoded.strip())
    return output_text

# --- Main Prediction and Evaluation Script ---
if __name__ == "__main__":
    try:
        # Load the trained model. compile=False is used because we only need inference.
        loaded_model = load_model(MODEL_PATH, compile=False,
custom_objects={'CTCLossLayer': CTCLossLayer})
```

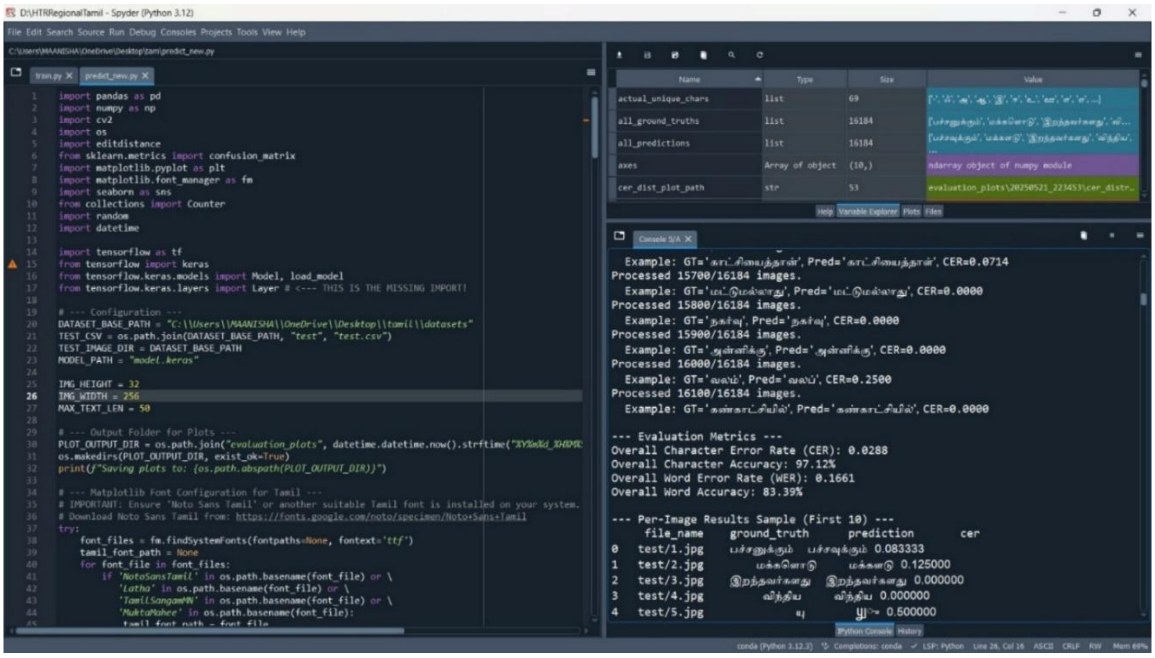
```
    # Extract the prediction output layer for inference
    prediction_output_layer = loaded_model.get_layer('dense_output').output
    inference_model = Model(inputs=loaded_model.inputs[0], outputs=prediction_output_layer)
    print("Model loaded successfully for prediction.")
    inference_model.summary()

except Exception as e:
    print(f"Error loading model from {MODEL_PATH}: {e}")
    print("Ensure the model path is correct and the model was saved properly.")
    print("Exiting prediction script.")
    exit()

try:
    test_df = pd.read_csv(TEST_CSV)
except FileNotFoundError as e:
    print(f"Error loading test CSV: {e}. Please ensure path is correct.")
    print("Exiting prediction script.")
    exit()

all_ground_truths = []
all_predictions = []
per_image_results = []
images_for_display = [] # To store image data for plotting samples

print("\n--- Starting Predictions on Test Set ---")
for index, row in test_df.iterrows():
    img_filename_from_df = row['file_name']
    ground_truth_text = str(row['text']).strip() # Ensure text is string and stripped

    # Initialize original_image_for_display to None at the start of each loop
    # This prevents NameError if the assignment below somehow fails or is skipped.
    original_image_for_display = None

    # Preprocess image for model input and optionally get original for display
    preprocessed_img_model_input, original_image_for_display = preprocess_image(
        img_filename_from_df, IMG_HEIGHT, IMG_WIDTH, TEST_IMAGE_DIR,
for_display=True
    )

    if preprocessed_img_model_input is None:
        continue # Skip if image couldn't be processed

    # Get raw predictions from the inference model
    raw_predictions = inference_model.predict(preprocessed_img_model_input, verbose=0)
    # Decode raw predictions into human-readable text
    predicted_texts = decode_batch_predictions(raw_predictions, num_to_char)
    predicted_text = predicted_texts[0] # Take the first (and only) prediction from the batch

    all_ground_truths.append(ground_truth_text)
    all_predictions.append(predicted_text)

    # Calculate Character Error Rate for the current image
    cer_single_image = editdistance.eval(ground_truth_text, predicted_text) /
len(ground_truth_text) if len(ground_truth_text) > 0 else 0

    # Store results for per-image analysis and overall metrics
    per_image_results.append({
        'file_name': img_filename_from_df,
        'ground_truth': ground_truth_text,
        'prediction': predicted_text,
        'cer': cer_single_image
    })

    # Store image data for plotting correct/incorrect samples later
    if original_image_for_display is not None:
         images_for_display.append({
            'image_data': original_image_for_display,
            'ground_truth': ground_truth_text,
            'prediction': predicted_text,
            'cer': cer_single_image
        })
    # Print progress every 100 images
    if (index + 1) % 100 == 0:
        print(f"Processed {index + 1}/{len(test_df)} images.")
        print(f"  Example: GT='{ground_truth_text}', Pred='{predicted_text}', 
CER={cer_single_image:.4f}")

    print("\n--- Evaluation Metrics ---")

    # Calculate Overall Character Error Rate (CER)
    total_char_distance = 0
    total_ground_truth_chars = 0
    for gt, pred in zip(all_ground_truths, all_predictions):
    total_char_distance += editdistance.eval(gt, pred)
    total_ground_truth_chars += len(gt)

    overall_cer = total_char_distance / total_ground_truth_chars if total_ground_truth_chars > 0 else 
0
    print(f"Overall Character Error Rate (CER): {overall_cer:.4f}")
    # Calculate and print Character Accuracy
    character_accuracy = (1 - overall_cer) * 100
    print(f"Overall Character Accuracy: {character_accuracy:.2f}%")

    # Calculate Overall Word Error Rate (WER)
    total_word_distance = 0
    total_ground_truth_words = 0
    for gt, pred in zip(all_ground_truths, all_predictions):
    gt_words = gt.split()
    pred_words = pred.split()
    total_word_distance += editdistance.eval(gt_words, pred_words)
    total_ground_truth_words += len(gt_words)

    overall_wer = total_word_distance / total_ground_truth_words if total_ground_truth_words > 0 
else 0
    print(f"Overall Word Error Rate (WER): {overall_wer:.4f}")
    # Calculate and print Word Accuracy
```

```
word_accuracy = (1 - overall_wer) * 100
print(f"Overall Word Accuracy: {word_accuracy:.2f}%")

print("\n--- Per-Image Results Sample (First 10) ---")
results_df = pd.DataFrame(per_image_results)
print(results_df.head(10).to_string())

## Plotting and Saving Results

### Character-level Confusion Matrix
print("\n--- Plotting Character-level Confusion Matrix ---")
flat_ground_truths_chars = [char for text in all_ground_truths for char in text]
flat_predictions_chars = [char for text in all_predictions for char in text]

min_len_cm = min(len(flat_ground_truths_chars), len(flat_predictions_chars))
flat_ground_truths_chars = flat_ground_truths_chars[:min_len_cm]
flat_predictions_chars = flat_predictions_chars[:min_len_cm]

# Identify characters involved in errors and common characters
error_chars = set()
for gt_char, pred_char in zip(flat_ground_truths_chars, flat_predictions_chars):
    if gt_char != pred_char:
        error_chars.add(gt_char)
        error_chars.add(pred_char)

# Get the top 20 most common characters
common_chars = [char for char, count in Counter(flat_ground_truths_chars).most_common(20)]
# Combine common characters and error-involved characters for plotting
chars_to_plot = sorted(list(set(common_chars).union(error_chars)))
if '[BLANK]' in chars_to_plot:
    chars_to_plot.remove('[BLANK]') # Remove blank token if present

if len(chars_to_plot) < 2:
    print("Cannot generate confusion matrix plot: Not enough unique characters to compare or
only one character type to plot.")
```

```
else:
    # Create a mapping for characters to integers for the confusion matrix
    cm_char_to_int = {char: i for i, char in enumerate(chars_to_plot)}

    # Convert ground truth and prediction character lists to integer lists
    y_true_int = [cm_char_to_int.get(char, -1) for char in flat_ground_truths_chars]
    y_pred_int = [cm_char_to_int.get(char, -1) for char in flat_predictions_chars]

    # Filter out -1 values (characters not in our limited set for plotting)
    y_true_int = [val for val in y_true_int if val != -1]
    y_pred_int = [val for val in y_pred_int if val != -1]

    if len(y_true_int) > 0 and len(y_pred_int) > 0:
        # Calculate the confusion matrix
        cm = confusion_matrix(y_true_int, y_pred_int, labels=list(range(len(chars_to_plot))))

        # Determine figure size dynamically for clarity
        fig_width = max(12, len(chars_to_plot) * 0.6) # Increased minimum width
        fig_height = max(12, len(chars_to_plot) * 0.6) # Increased minimum height
        plt.figure(figsize=(fig_width, fig_height))

        # Create mapping back from int to char for axis labels
        cm_int_to_char = {i: char for char, i in cm_char_to_int.items()}

        # Plot heatmap
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar_kws={'shrink': 0.8},
                    xticklabels=[cm_int_to_char.get(i, '') for i in range(len(chars_to_plot))],
                    yticklabels=[cm_int_to_char.get(i, '') for i in range(len(chars_to_plot))])

        # Apply font properties directly to text elements if tamil_font_prop is defined
        plt.xlabel('Predicted Character', fontsize=14, fontproperties=tamil_font_prop if
tamil_font_prop else None)
        plt.ylabel('True Character', fontsize=14, fontproperties=tamil_font_prop if tamil_font_prop
else None)
        plt.title('Character-level Confusion Matrix (Selected Characters)', fontsize=16,
fontproperties=tamil_font_prop if tamil_font_prop else None)
```

```
        # Apply font to tick labels
        if tamil_font_prop:
            plt.xticks(rotation=90, fontsize=12, fontproperties=tamil_font_prop)
            plt.yticks(rotation=0, fontsize=12, fontproperties=tamil_font_prop)
        else:
            plt.xticks(rotation=90, fontsize=12) # Fallback without custom font
            plt.yticks(rotation=0, fontsize=12) # Fallback without custom font

        plt.tight_layout() # Adjust layout to prevent labels from overlapping

        # Save plot
        cm_plot_path = os.path.join(PLOT_OUTPUT_DIR, "confusion_matrix.png")
        plt.savefig(cm_plot_path, bbox_inches='tight', dpi=300) # High DPI for clarity
        print(f"Confusion Matrix plot saved to: {cm_plot_path}")
        plt.close() # Close the plot to free memory
    else:
        print("Cannot generate confusion matrix plot: Filtered character lists for plotting are
empty.")

    ### CER Distribution
    print("\n--- Plotting CER Distribution ---")
    cers = results_df['cer'].tolist()
    if cers:
        plt.figure(figsize=(10, 6))
        sns.histplot(cers, bins=20, kde=True)
        # Apply font properties directly
        plt.title('Distribution of Character Error Rates (CER) Per Image', fontsize=14,
fontproperties=tamil_font_prop if tamil_font_prop else None)
        plt.xlabel('Character Error Rate (CER)', fontsize=12, fontproperties=tamil_font_prop if
tamil_font_prop else None)
        plt.ylabel('Number of Images', fontsize=12, fontproperties=tamil_font_prop if
tamil_font_prop else None)
        plt.grid(axis='y', alpha=0.75)

        # Apply font to tick labels
        if tamil_font_prop:
            plt.xticks(fontproperties=tamil_font_prop)
            plt.yticks(fontproperties=tamil_font_prop)

        # Save plot
        cer_dist_plot_path = os.path.join(PLOT_OUTPUT_DIR, "cer_distribution.png")
        plt.savefig(cer_dist_plot_path, bbox_inches='tight', dpi=300)
        print(f"CER Distribution plot saved to: {cer_dist_plot_path}")
        plt.close()
    else:
        print("No CER data to plot distribution.")

    ### Sample Predictions (Correct & Incorrect)
    print("\n--- Plotting Sample Predictions (Correct & Incorrect) ---")
    if images_for_display:
        correct_predictions = [item for item in images_for_display if item['cer'] == 0]
        incorrect_predictions = [item for item in images_for_display if item['cer'] > 0]

        # Select a few correct and incorrect samples to display
        num_samples_to_plot = min(5, len(correct_predictions))
        num_incorrect_samples_to_plot = min(5, len(incorrect_predictions))

        selected_samples = []
        if num_samples_to_plot > 0:
            selected_samples.extend(random.sample(correct_predictions, num_samples_to_plot))
        if num_incorrect_samples_to_plot > 0:
            selected_samples.extend(random.sample(incorrect_predictions,
num_incorrect_samples_to_plot))

        if not selected_samples:
            print("No samples available to plot.")
        else:
            # Sort samples by CER (highest error first) for better visualization
            selected_samples.sort(key=lambda x: x['cer'], reverse=True)
        num_plots = len(selected_samples)
        fig, axes = plt.subplots(num_plots, 1, figsize=(10, 3 * num_plots)) # Adjust figure height
based on number of samples

        if num_plots == 1:
            axes = [axes] # Ensure axes is iterable even for a single plot

        for i, sample in enumerate(selected_samples):
            ax = axes[i]
            image_to_show = sample['image_data']

            # Remove single-channel dimension if present for imshow
            if image_to_show.ndim == 3 and image_to_show.shape[2] == 1:
                image_to_show = image_to_show.squeeze(-1)

            ax.imshow(image_to_show, cmap='gray')
            # Add title with CER, Ground Truth, and Prediction
            # Apply font properties directly to the title
            ax.set_title(f"CER: {sample['cer']:.4f}\nGT: \"{sample['ground_truth']}\"\nPred:
\"{sample['prediction']}\"",
                         fontsize=12, fontproperties=tamil_font_prop if tamil_font_prop else None)
            ax.axis('off') # Hide axes

        plt.tight_layout()

        # Save plot
        sample_plot_path = os.path.join(PLOT_OUTPUT_DIR, "sample_predictions.png")
        plt.savefig(sample_plot_path, bbox_inches='tight', dpi=300)
        print(f"Sample Predictions plot saved to: {sample_plot_path}")
        plt.close()
    else:
        print("No image data available for plotting samples.")

    print("\n--- Analysis Complete! ---")
    print(f"All evaluation plots saved to: {os.path.abspath(PLOT_OUTPUT_DIR)}")
```

Spyder environment:


D:\HTRRegionalTamil - Spyder (Python 3.12)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\MAANISHA\OneDrive\Desktop\tam\predict_new.py
train.py
predict_new.py
Help Variable Explorer Plots Files
Console 5/A
IPython Console History
conda (Python 3.12.3) Completions: conda LSP: Python Line 26, Col 16 ASCII CRLF RW Mem 69%

```python
import pandas as pd
import numpy as np
import cv2
import os
import editdistance
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
import seaborn as sns
from collections import Counter
import random
import datetime

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Layer # <--- THIS IS THE MISSING IMPORT!

# --- Configuration ---
DATASET_BASE_PATH = "C:\\Users\\MAANISHA\\OneDrive\\Desktop\\tamil\\datasets"
TEST_CSV = os.path.join(DATASET_BASE_PATH, "test", "test.csv")
TEST_IMAGE_DIR = DATASET_BASE_PATH
MODEL_PATH = "model.keras"

IMG_HEIGHT = 32
IMG_WIDTH = 256
MAX_TEXT_LEN = 50

# --- Output Folder for Plots ---
PLOT_OUTPUT_DIR = os.path.join("evaluation_plots", datetime.datetime.now().strftime("%Y%m%d_%H%M%
os.makedirs(PLOT_OUTPUT_DIR, exist_ok=True)
print(f"Saving plots to: {os.path.abspath(PLOT_OUTPUT_DIR)}")

# --- Matplotlib Font Configuration for Tamil ---
# IMPORTANT: Ensure 'Noto Sans Tamil' or another suitable Tamil font is installed on your system.
# Download Noto Sans Tamil from: https://fonts.google.com/noto/specimen/Noto+Sans+Tamil
try:
    font_files = fm.findSystemFonts(fontpaths=None, fontext='ttf')
    tamil_font_path = None
    for font_file in font_files:
        if 'NotoSansTamil' in os.path.basename(font_file) or \
           'Latha' in os.path.basename(font_file) or \
           'TamilSangamMN' in os.path.basename(font_file) or \
           'MuktaMahee' in os.path.basename(font_file):
            tamil_font_path = font_file
```

Name	Type	Size	Value
actual_unique_chars	list	69	[illegible]
all_ground_truths	list	16184	['பச்சனுக்கும்', 'மக்களொடு', 'இறந்தவர்களது', 'வி...
all_predictions	list	16184	['பச்சவுக்கும்', 'மக்களடு', 'இறந்தவர்களது', 'விந்திய', ...
axes	Array of object	(10,)	ndarray object of numpy module
cer_dist_plot_path	str	53	evaluation_plots\20250521_223453\cer_distr...

```
  Example: GT='காட்சியைத்தான்', Pred='காட்சியைத்தாள்', CER=0.0714
Processed 15700/16184 images.
  Example: GT='மட்டுமல்லாது', Pred='மட்டுமல்லாது', CER=0.0000
Processed 15800/16184 images.
  Example: GT='நகர்வு', Pred='நகர்வு', CER=0.0000
Processed 15900/16184 images.
  Example: GT='அன்னிக்கு', Pred='அன்னிக்கு', CER=0.0000
Processed 16000/16184 images.
  Example: GT='வலம்', Pred='வலப்', CER=0.2500
Processed 16100/16184 images.
  Example: GT='கண்காட்சியில்', Pred='கண்காட்சியில்', CER=0.0000

--- Evaluation Metrics ---
Overall Character Error Rate (CER): 0.0288
Overall Character Accuracy: 97.12%
Overall Word Error Rate (WER): 0.1661
Overall Word Accuracy: 83.39%

--- Per-Image Results Sample (First 10) ---
    file_name   ground_truth      prediction       cer
0   test/1.jpg   பச்சனுக்கும்   பச்சவுக்கும் 0.083333
1   test/2.jpg       மக்களொடு      மக்களடு 0.125000
2   test/3.jpg   இறந்தவர்களது  இறந்தவர்களது 0.000000
3   test/4.jpg        விந்திய      விந்திய 0.000000
4   test/5.jpg             யு          ழு 0.500000
```