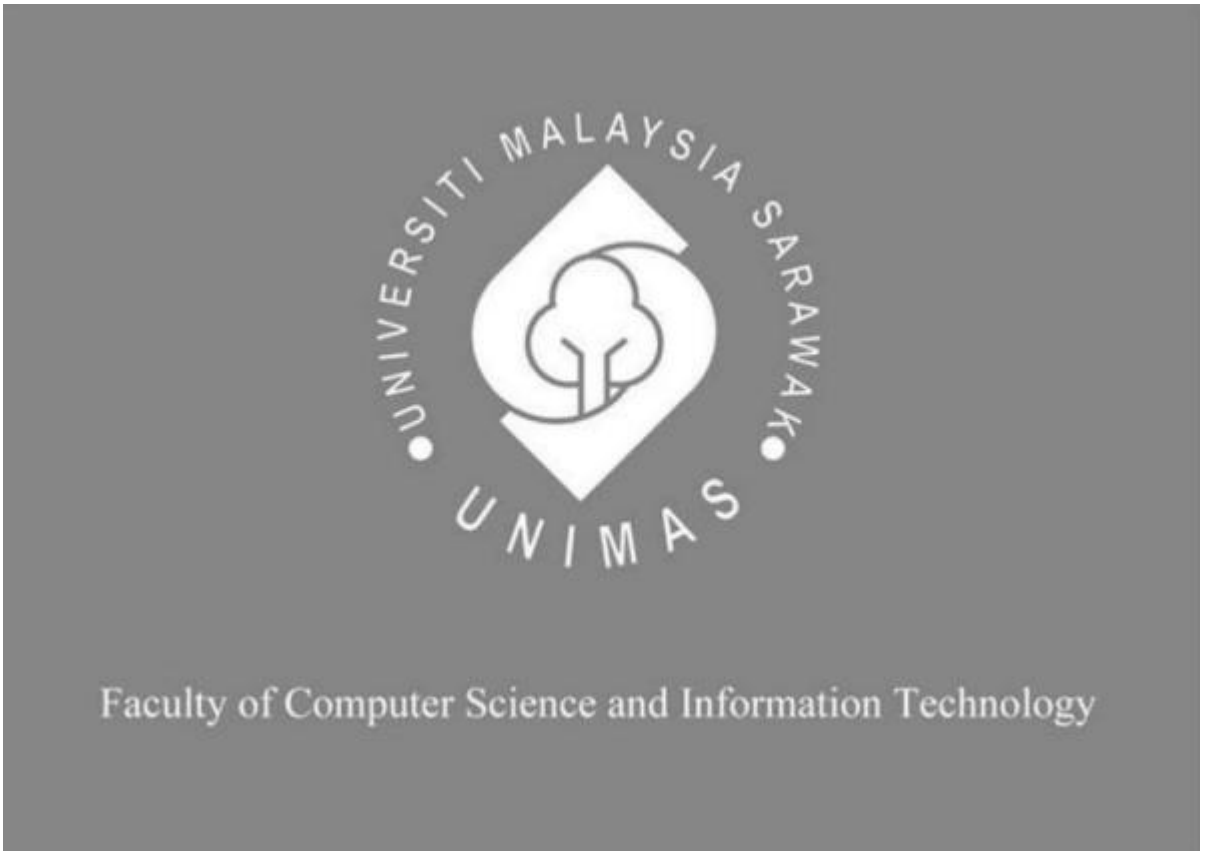

PERFORMANCE EVALUATION OF TESSERACT, EASYOCR, AND TROCR MODELS FOR OPTICAL CHARACTER RECOGNITION SYSTEMS

Lucas Chu Yen Feng

Bachelor of Computer Science with Honours

(Computational Science)

2025

Acknowledgement

I would like to express my sincere gratitude to the Faculty of Computer Science and Information Technology, University Malaysia Sarawak (UNIMAS), for providing the resources and support that made this project possible. My deepest thanks go to my supervisor, Dr. Ling Yeong Tyng, for her invaluable guidance, constructive feedback, and encouragement throughout this research. I am also grateful to my academic advisor, Dr. Sarah Samson Juan, for her advice and support throughout my academic journey. Lastly, I am deeply thankful to my family and friends for their understanding and constant encouragement throughout this endeavor.

Abstract

This research evaluates the performance of three Optical Character Recognition (OCR) methods of Tesseract, EasyOCR, and TrOCR across the Chars74k and Total Text datasets. Through K-fold cross-validation, the study analyzes character and word error rates, inference times, and generalization capabilities. Results highlight the trade-off between traditional and deep learning approaches, with TrOCR excelling in challenging scene text, EasyOCR offering a balance between accuracy and efficiency, and Tesseract excelling on cleaner text. A public survey further explores perceptions of OCR's usefulness and future relevance. Findings guide future improvements and practical deployments of OCR technologies.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Background

Today, Optical Character Recognition (OCR) is widely used across numerous sectors, including finance, healthcare, and education, to digitize printed and handwritten documents, making them easier to store, retrieve, and analyse. The COVID-19 pandemic accelerated this demand as industries quickly adapted to digital transformation. In healthcare, for example, OCR algorithms now digitize handwritten medical notes and prescriptions, facilitating quick patient information access (Alharbi & Rahman, 2021). Similarly, OCR is vital in financial services, where digital processing of paper records speeds up transactions and reduces errors. However, traditional sequential processing has shown limitations, especially with high-volume data, prompting research into faster parallel processing techniques.

OCR is a well-established technique that converts scanned images of text, such as photos and documents, into editable, searchable electronic formats (Wang, 2023). Originating in 1928, OCR initially used pattern matching to recognize characters by comparing input characters against established patterns. By the 1950s, the UK and US had developed equipment specifically for recognizing printed numbers. Research by firms like IBM in the 1960s led to improvements in recognition accuracy and initial attempts at recognizing handwritten text. The 1970s introduced third-generation OCR systems, though they were limited by low document quality. In the 1980s, as computer hardware advanced and demand grew due to widespread computer usage, commercial OCR systems proliferated, expanding to include Chinese character recognition as well (Seung, 2020).

Enhanced OCR technology, driven by deep learning, has opened a wide range of applications due to its increased accuracy and adaptability. Key applications include document digitization, text extraction from images, accessibility for visually impaired individuals, and intelligent search and retrieval. Document digitization aids sectors like legal, healthcare, and finance by simplifying data storage, searchability, and archival (Pratim, 2023). Text extraction automates data entry and multilingual processing, useful for handling data-intensive forms like invoices and receipts.

OCR faces challenges with noisy text, complex backgrounds, and degraded images. While advances have been made, issues like multi-language support and real-time application optimization are still being explored. OCR requires high-quality, high-resolution images for optimal performance. Scanned images generally yield better accuracy than camera-captured ones, which are often affected by lighting, perspective, and clarity, impacting performance (Hamad & Kaya, 2016). By addressing these technical and practical challenges, OCR technology continues to evolve, driving opportunities for automation, accessibility, and information processing across diverse industries (Pratim, 2023).

In the future, more academics are expected to invest in OCR technology for the recognition of fragmented characters, and this work demonstrates the maturity of OCR based on neural networks. There is existing research into using OCR to repair and recognise fragmented texts, and this technology may be utilised in the future to restore old, damaged books (Wang, 2023).

1.2 Problem Statement

The field of OCR is characterised by a plethora of toolsets there with different approaches including deep learning models. These tools also cater to a wide range use cases for, from recognizing text in clean, structured documents to handling complex scenarios such as multilingual texts, skewed images, and natural scene texts. (Jain, Taneja, & Kavita, 2021). Despite the widespread use of models such as LSTMs, CRNNs, and Transformers in modern OCR systems, there remains a lack of focused comparative evaluation across datasets with different levels of complexity. This project addresses that gap by assessing the performance of three representative OCR systems

1.3 Scope

This study compares the performance of OCR models which are Tesseract OCR, EasyOCR and TrOCR. Char74K, dataset of 62 classes of alphanumeric fonts, and Total Text, dataset with natural scene text. The study focusses on recognising alphanumeric characters and includes at least five typographic font changes to test model generalisation.

1.4 Objective

1. Evaluate the accuracy of LSTM, CRNN, and Transformer-based models in OCR tasks.
2. Evaluate the robustness of LSTM, CRNN, and Transformer-based models in in OCR tasks with two distinct datasets.
3. Develop a graphic user interface for the OCR system to visualise and showcase the comparative results.

1.5 Brief Methodology

This project evaluates the performance of three OCR models, Tesseract, EasyOCR, and TrOCR, using two benchmark datasets: Chars74k-Fonts and TotalText. Chars74k-Fonts consists of stylized alphanumeric characters and represents clean, structured input, while TotalText includes irregular, curved, and natural scene text, posing a greater challenge for OCR systems. These datasets were preprocessed to ensure compatibility, with word-level crops extracted from TotalText annotations.

The three OCR systems were selected for their distinct architectures. Tesseract (v4+) uses an LSTM-based recognizer with CTC decoding, representing a traditional yet hybrid approach. EasyOCR utilizes a CRNN architecture with CNNs and Bidirectional LSTMs, offering flexibility and multilingual support. TrOCR adopts a Transformer-based encoder-decoder model for end-to-end text recognition, representing state-of-the-art design in OCR.

Model performance was evaluated using three key metrics: Character Error Rate (CER), Word Error Rate (WER), and inference time per image. These metrics provide insights into both recognition accuracy and computational efficiency. Evaluations were conducted across both datasets to highlight the strengths and limitations of each model under varying input conditions.

Finally, a graphical user interface (GUI) dashboard was developed to display the comparative results interactively. The dashboard allows users to visualize performance metrics and test the models on custom inputs, enhancing accessibility and real-world applicability of the OCR systems.

1.6 Significant of the Project

This study provides valuable insights into the implementation of different OCR approaches using deep learning and traditional rule-based systems. By analysing three widely adopted OCR models which are Tesseract, EasyOCR, and TrOCR. The research highlights their strengths, limitations, and suitability across various data types and scenarios. The comparative evaluation offers practical guidance for selecting or improving OCR systems, enabling industries to enhance text recognition performance, reduce processing time, and achieve greater cost-efficiency in real-world applications.

1.7 Project Schedule

The project schedule will be projected with the task and deliverables within Final Year Project 1 and Final Year Project 2 as in the Gannt Chart depicted in Figure 1.1.

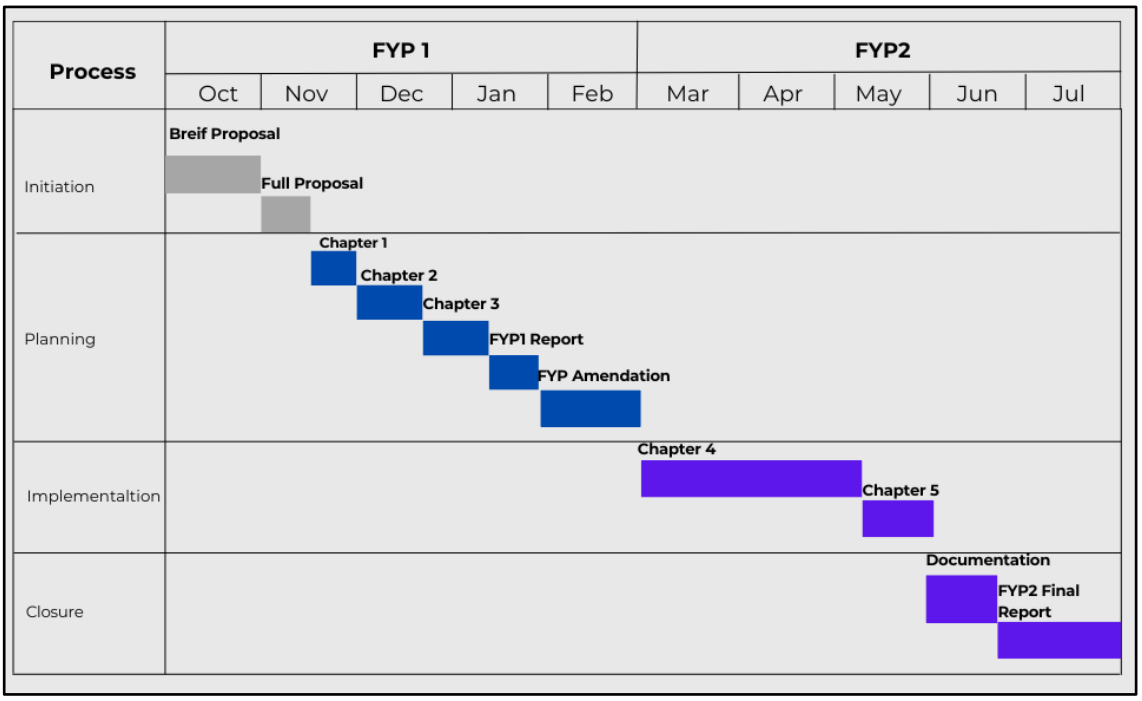

Figure 1.0.1: Gannt Chart of the Project Schedule

1.8 Project Outcome

1. Provided a comparative performance analysis of three OCR systems their respective strengths and limitations when applied to different datasets.
2. Identified how different model architectures influence OCR accuracy, robustness and inference efficiency.
3. Developed an interactive graphical user interface (GUI) dashboard to visualize evaluation metrics such as Character Error Rate (CER), Word Error Rate (WER), and inference time

1.9 Project Outline

This thesis is organized into five chapters to systematically present the research framework, implementation process, and findings. Chapter 1 introduces the project by providing a background on Optical Character Recognition (OCR) and emphasizing the importance of comparing different OCR systems powered by deep learning architectures. It outlines the problem statement, research objectives, scope, and methodology. Additionally, this chapter highlights the significance of the study, presents the project timeline, and identifies the

expected contributions.

Chapter 2 presents a comprehensive literature review, examining the historical development and recent advancements in OCR technologies. Particular attention is given to deep learning models, including Convolutional Recurrent Neural Networks (CRNNs), Long Short-Term Memory (LSTM) networks, and Transformer-based architectures. This chapter explores various approaches used in OCR tasks and discusses relevant challenges and gaps in the existing literature.

Chapter 3 outlines the methodology adopted in this study. It describes the selection and preparation of two benchmark datasets which are TextOCR and Chars74K-Fonts where both of the datasets representing different types of textual input. The chapter also details the preprocessing strategies employed to accommodate the input requirements of each OCR model, the implementation of Tesseract, EasyOCR, and TrOCR, and the evaluation metrics used for performance comparison, including accuracy, Character Error Rate (CER), Word Error Rate (WER), and inference time.

Chapter 4 focuses on the practical aspects of the project, including data engineering and the training and validation of each OCR model. It also documents the development of a web-based interface designed for both demonstration and analysis. The interface includes a dashboard for visualizing performance metrics and a demo section for testing OCR capabilities with user-supplied input.

Chapter 5 presents and discusses the results obtained from the model evaluations. It analyses the comparative performance of Tesseract, EasyOCR, and TrOCR based on the defined metrics, identifies the strengths and limitations of each system, and reflects on the scope of the study. The chapter concludes with a summary of key findings and provides recommendations for future research, including the potential use of parallel processing techniques to further optimize OCR system performance.

CHAPTER 2

LITERATURE REVIEW

Overview

This section reviews three existing OCR approaches using deep learning models. A comparison table is included in the later part of this section to summarize and provide a clear overview on the key differences between the models. The chapter sets the stage further analysis for these OCR systems with their models for their scalability and efficiency through parallel processing.

2.1 Related Works

2.1.1 LSTM Networks in Tesseract OCR

The LSTM network is a type of Recurrent Neural Network (RNN) that is intended to solve the vanishing gradient problem that normal RNNs encounter. LSTM performs well in sequence prediction problems, capturing long-term dependencies. Order dependence makes it ideal for time series, machine translation, and speech recognition. The LSTM model overcomes the issue by incorporating a memory cell, which is a container that can store information for a longer time.

The LSTM architecture has a memory cell that is controlled by three gates: input, forget, and output. These gates determine what information is added to, removed from, and output from the memory cell. This enables LSTM networks to choose store or reject information as it goes through the network, allowing them to discover long-term dependencies.

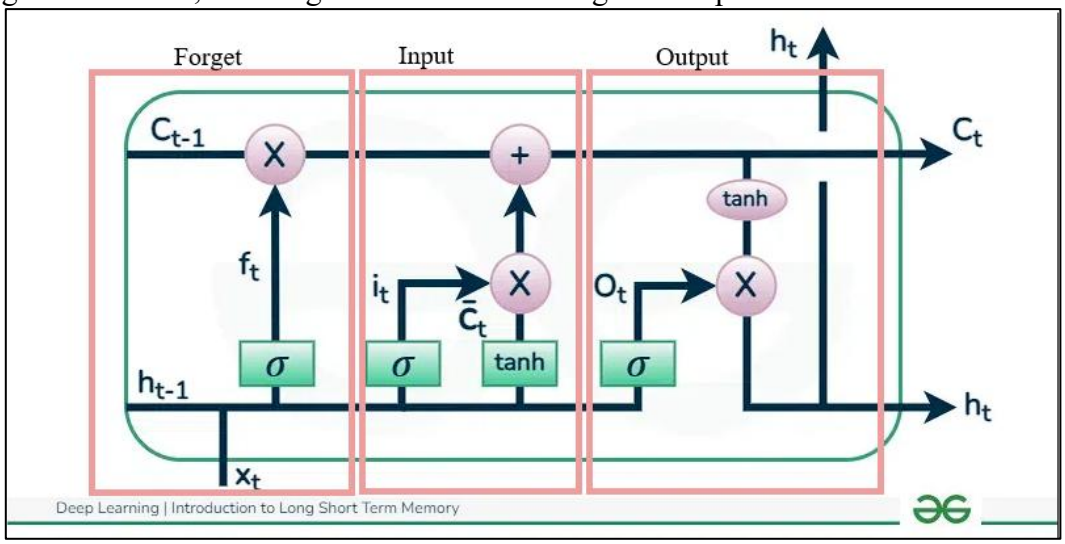

Figure 2.1: Memory Cell in a LSTM Architecture

The LSTM architecture is organized in a chain structure that includes four neural networks and various memory blocks known as cells. Cells store information, and gates manipulate memory and is shown in Figure 2.1 (GeeksforGeeks, 2024).

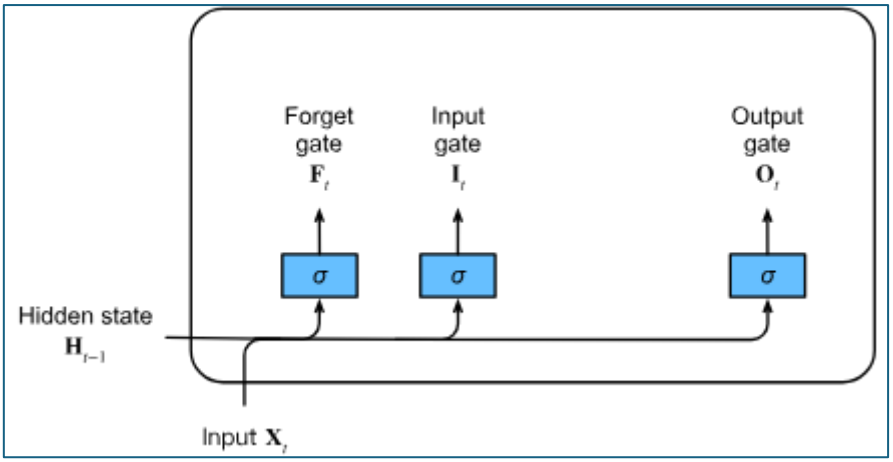

Figure 2.2: Computing the gates in a LSTM model

LSTM algorithm starts with the input handling at each step is processed by defining the weights and biases for the gates. With the initialising of the LSTM cell state and hidden state to zero or a very small value, two inputs which are the current input (x_t), and the previous hidden state (h_{t-1}) are fed into the gates as shown in Figure 2.2 (D2L.ai, 2022).

Forget Gate (f_t), Input Gate (i_t) and Output Gate (o_t):

$$f_t = \sigma\left(W_f \cdot |h_{t-1}, x_t| + b_f\right) \quad (1)$$
$$i_t = \sigma(W_i \cdot |h_{t-1}, x_t| + b_i) \quad (2)$$
$$o_t = \sigma(W_o \cdot |h_{t-1}, x_t| + b_o) \quad (3)$$

Where σ is the sigmoid activation function,W is the weight parameter and b are the bias parameter. The resulting product of the two inputs with the weight parameter and addition of the bias are then pass through a sigmoid activation function which will result binary output. Depending on the output the forget gate determine whether to keep the current value of the memory. The input gate determines how much of the input node's value should be added to the memory cell internal state (C_t). While the output gate determines whether the memory cell should influence the output at the current time step.

Input Node ($\widehat{C_t}$):

$$\hat{C}_t = \tanh(W_C \cdot |h_{t-1}, x_t| + b_C) \quad (4)$$

Where,W_c is the wright parameter and b_C is the bias parameter. The computation of the input node is similar to the three gates but uses a tanh function which will result with values in range of (-1,1) as the activation function.

Memory Cell Internal State(C_t):

$$C_t = F_t \odot C_{t-1} + i_t \odot \hat{C}_t \quad (5)$$

The input gate governs how much we take new data into account and forget gate addresses how much of the cell internal state we retain. Using the Hadarmard elementwise product operator (⊙) and we will arrive with the equation above. This addresses the vanishing gradient problem, which is a common occurrence in RNNs, making LSTM models much easier to train, particularly when dealing with datasets with extended sequence lengths.

Hidden State (h_t):

$$h_t = o_t \odot tanh(C_t) \qquad (6)$$

Finally, we must define how to compute the memory cell's output, which represents the hidden state as observed by other layers. This is when the output gate comes into play. Starting with applying tanh to the memory cell internal state C_t then apply another point-wise multiplication which will results where the hidden state value is always in the interval of (-1,1). As a result, when the output gate results approach one, the memory cell's internal state allows the subsequent layer to function normally.

LSTM networks have significantly enhanced the performance of OCR systems, especially in recognizing complex fonts and text from natural scenes. A recent study by Soni et al. (2024) explores the use of LSTM networks for real-time character recognition in natural scene images. LSTM networks are used to identify the actual characters from a processed imaged. LSTMs are RNNs that excel at processing sequences, making them ideal for text data. LSTMs assess a series of features taken from a detected text section. Using this sequence, they can forecast the most likely character for each place. This method allows them to adjust for variances in character spacing and even font styles. The study highlights that LSTM-based models excel in learning contextual relationships between characters, which is crucial for handling text with distortion, noise, and complex backgrounds. By combining LSTMs with text recognition, the authors showed significant improvements in recognition accuracy, particularly in real-time applications.

Furthermore, Soni et al. (2024) also did a comparative analysis with of their proposed methods with other existing methods with different approach and their proposed method has outperform others in text extraction from natural scenes. Although effective in scene text recognition, the model is not optimized for real-time performance and requires substantial computational resources, particularly during training due to the complexity of CNN and RNN layers.

2.1.2 Hybrid CRNN in EasyOCR

CRNN term given to the proposed neural network model is a hybrid of Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). CRNN algorithm process consists of two stages the feature extraction through the convolution layers by the CNN and the sequential processing of model temporal dependencies through recurrent layers of RNN. EasyOCR utilise CNN-LSTM-Decoder model architecture where CNN models can be in a form of Visual Geometry Group (VGG) or Residual Network (ResNet).

First, the convolution layer which is used for feature extraction. It performs feature extraction by first applying filer at the convolution layer where filters are applied to a input image. A filter is very small in size usually 3x3 or 5x5 in size which convolves over the image. There can be multiple layers to detect multiple features such as specific edges, textures or patterns (Daniella, 2024).

Input Tensors(X):

$$X = H_{in} \cdot W_{in} \cdot C_{in} \qquad (7)$$

Where X is the input size, H is the height and C is the number of channels. Channels is 3 for coloured images since there is 3 layers of red, blue and green accounted for a layer. The input can also be represented as a feature map from a previous layer.

Kernels (K):

$$K = K_h \cdot K_w \cdot C_{in} \cdot C_{out} \quad (8)$$

Where $K_h \cdot K_w$ is the kernel size, C_{in} is the depth of the input and C_{out} is the number of output channels. The kernel size is usually small at least 3x3 or 5x5. This kernel will move and scan the image with a certain stride until in finish scanning the whole image. To calculate the output tensor:

$$H_{out} = \frac{H_{in} - K_h + 2P}{S} + 1 \quad (9)$$

$$W_{out} = \frac{W_{in} - K_w + 2P}{S} + 1 \quad (10)$$

$$Output\ Dimension = H_{out} \cdot W_{out} \cdot C_{out} \quad (11)$$

Where S is the strides of the kernel and P is the zero padding on the border. This will represent the size of the feature map after applying the kernels. To map out the feature map to be used for the next layer more calculation to sum up the contributions for the weight values from previous layers. This step is known as the pooling layer.

Lets say the stride, S = 1 and it is greyscales $C_{in} = 1$

$$x_{i,j} = \sum_{a=0}^{K_h - 1} \sum_{b=0}^{K_w - 1} K_{a,b} \cdot X_{(I+a,j+b)} \quad (12)$$

The feature vectors are then ready to be fed into the Recurrent Layers and LSTM is used in the study done by (Idris & Taha, 2022). They utilise a bidirectional LSTM as their recurrent layers of their OCR system.

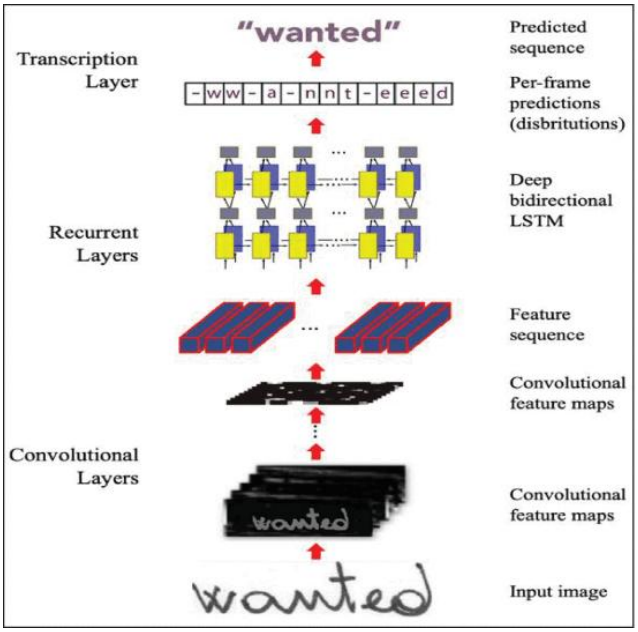

Figure 2.3: CRNN Architecture for OCR

Figure 2.3 shows the layers of the CRNN model architecture in OCR systems. According to a study by (Idris & Taha, 2022) on handwritten text detection using CRNN, the convolutional layer component of a CRNN model is generated by combining convolutional layers with max-pooling layers in the same way that CNNs are created, however fully connected layers are not required in this case. This component obtains a sequential feature representation from a pre-processed source image. The feature vectors are then extracted from the feature maps produced by the convolutional layer component, and this will be as an input for the next layers which are the recurrent layers. Each feature vector is produced by a column from left to right on the

feature maps. Following that, the feature vectors from the convolutional layers are fed into a deep bidirectional RNN which can predict each frame of the feature sequence generated by the convolutional layers as a label distribution for each one of the feature sequences. RNN may supply error differentials to its convolutional layer input, which allows us to train both the recurrent and convolutional layers in the same network. Lastly, the transcription layer aims to translate the per-frame predictions of the recurrent layers into a label sequence known as transcription. Given the predictions, the mathematical purpose of transcription is to find the label sequence with the highest probability. In this research, the study presented a model trained on the IAM dataset using CRNN, a hybrid architecture of CNN and RNN, as well as a strategy for matching pictures before the recognition process begins. They suggest that a segmentation algorithm for optimal input fitting to be developed for the system.

2.1.3 Transformer-Based Models in TrOCR

Transformer is a deep learning architecture built by Google researchers that uses the multi-head attention technique. Transformer-based architectures have gained prominence for OCR tasks due to their ability to handle long-range dependencies in textual data. Li et al. (2021) introduced TrOCR, a Transformer-based OCR framework that employs pre-trained models to improve recognition accuracy. The TrOCR framework utilizes a sequence-to-sequence learning paradigm, integrating a Vision Transformer (ViT) as the encoder and a GPT-like model as the decoder. This pre-trained approach significantly reduces the amount of labelled data required for fine-tuning, making it highly effective for both printed and handwritten text recognition. TrOCR achieves state-of-the-art performance on datasets such as IAM and PrintED53, establishing its versatility and scalability for diverse OCR tasks.

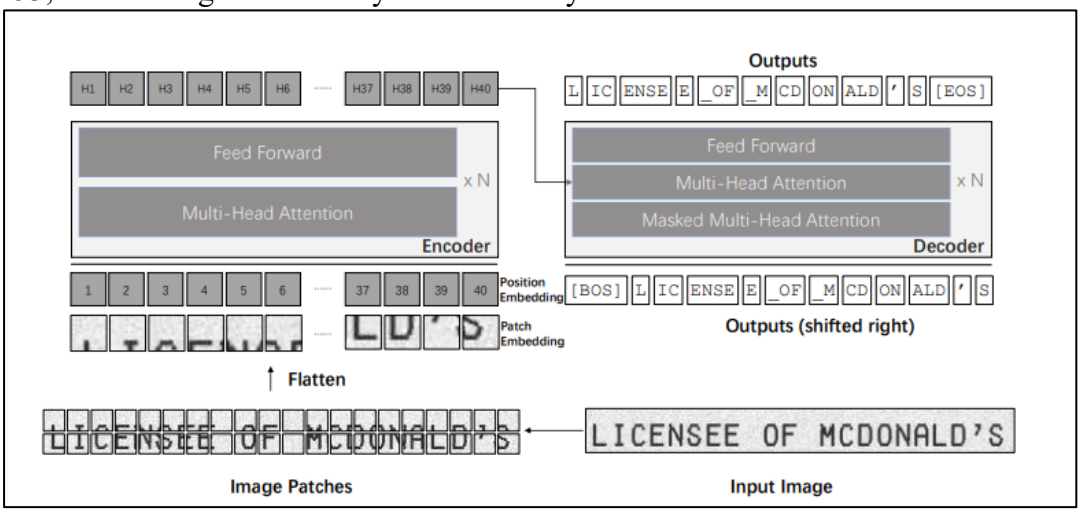

Figure 2.4: The Architecture of TrOCR

In Figure 2.4 shows the architecture of the Transformer-based OCR by Li et al. (2021), they proposed TrOCR which is a transformer-based OCR model for text recognition that includes pre-trained CV and NLP models. TrOCR uses the Transformer architecture, including an image Transformer for visual feature extraction and a text Transformer for language modelling. They use the basic form of Transformer encoder-decoder structure in TrOCR. The encoder represents image patches, while the decoder generates word piece sequences using visual cues and past predictions. First, the encoder will receive an image that will be shrunk to a fixed size and divided into precisely divisible patch sizes, after which each patch will be flattened into a vector and projected into a D-dimensional patch embedding. Similar to Vision Transformer (ViT) and Data-Efficient Image Transformer (DeiT), it will preserve a particular token that is commonly used for image classification, and it can also put together the entire image from the

patch embeddings and represents the whole image. This allows the model to focus on the full image or specific patches. After this, the decoder generates the output sequence by attending to the encoder's output via "encoder-decoder attention," in which the decoder sends queries, and the encoder sends keys/values. An attention mask assures that each output position only considers preceding positions, allowing for step-by-step production. Hidden states are projected to the vocabulary size via a linear layer, and probabilities are derived using softmax. Beam search is then used to generate the final output sequence, which ensures accuracy and consistency.

This work emphasizes the transformative potential of Transformer topologies in OCR, as well as the advantages of using large-scale pre-training for text recognition. TrOCR, unlike typical OCR, does not rely on CNNs for image interpretation and instead uses an image Transformer as an encoder. TrOCR achieves cutting-edge results for printed, handwritten, and scene text recognition workloads with no post-processing processes. While with the state of art results, the base model of their TrOCR system with a processing speed of 4.6 sentences per second which is not recommended in any real-world applications.

2.2 Comparison of existing OCR using Deep Learning models

Table 2.1: Comparison of Models between existing OCR approaches

Models	LSTM	CRNN	TrOCR
Architecture	RNN	CNN + RNN (mostly LSTM)	ViT + Decoder
Outputs	Character-level	Sequence of labels (CTC)	Word-piece predictions
Strengths	Good with noisy/distorted text	Accurate for handwriting	High accuracy, minimal training needed
Limitations	Slower, needs tuning	Needs clear input structure	Heavy, slower for real-time
Best For	Scene text, sequential data	Handwritten and structured text	Printed, handwritten, scene text
Training	Requires labeled data	End-to-end with paired data	Pre-trained; little fine-tuning needed
Scalability	Scales with cost	Scales well	Highly scalable and generalizable

Table above shows a comparative overview of three OCR models, LSTM**,** CRNN, and TrOCR, highlighting their architectural differences, output types, strengths, limitations, and training requirements. LSTM models, based on recurrent neural networks, are suitable for processing sequential or distorted text but may require extensive tuning and are generally slower. CRNN combines convolutional and recurrent layers, making it effective for handwriting and structured text using CTC-based output. TrOCR, built on a Transformer architecture with a Vision Transformer encoder and decoder, delivers high accuracy with minimal fine-tuning and strong generalization across various text types. However, it is computationally heavier and less suited for real-time applications on limited hardware. This comparison provides a foundation for selecting appropriate models based on dataset characteristics and performance needs.

2.3 Chapter Summary

This chapter reviewed three prominent deep learning-based OCR approaches: LSTM networks as used in Tesseract, CRNN architectures employed in EasyOCR, and Transformer-based models in TrOCR. Each method was examined in terms of architecture, processing flow, and suitability for different OCR tasks. LSTM models excel in handling sequential and distorted text but are limited by slower inference and higher tuning requirements. CRNNs, combining CNNs and RNNs, offer a balanced solution for structured and handwritten text recognition with end-to-end learning capability. TrOCR represents the latest advancement, utilizing pre-trained Vision Transformers and decoder modules for highly accurate text recognition with minimal fine-tuning, though at the cost of increased computational demands. The chapter concludes with a comparative table that highlights the key differences among these models, providing a foundation for understanding their strengths, limitations, and applicability in various OCR scenarios. This sets the stage for the subsequent performance evaluation in later chapters.

CHAPTER 3

METHODOLOGY

Overview

This study adopts a systematic approach to train and evaluate three OCR models which are Tesseract OCR, EasyOCR and TrOCR across two distinct datasets. The methodology is structured to ensure consistency in model evaluation and to support reproducibility through clearly defined processes. The following outlines the key methodological steps:

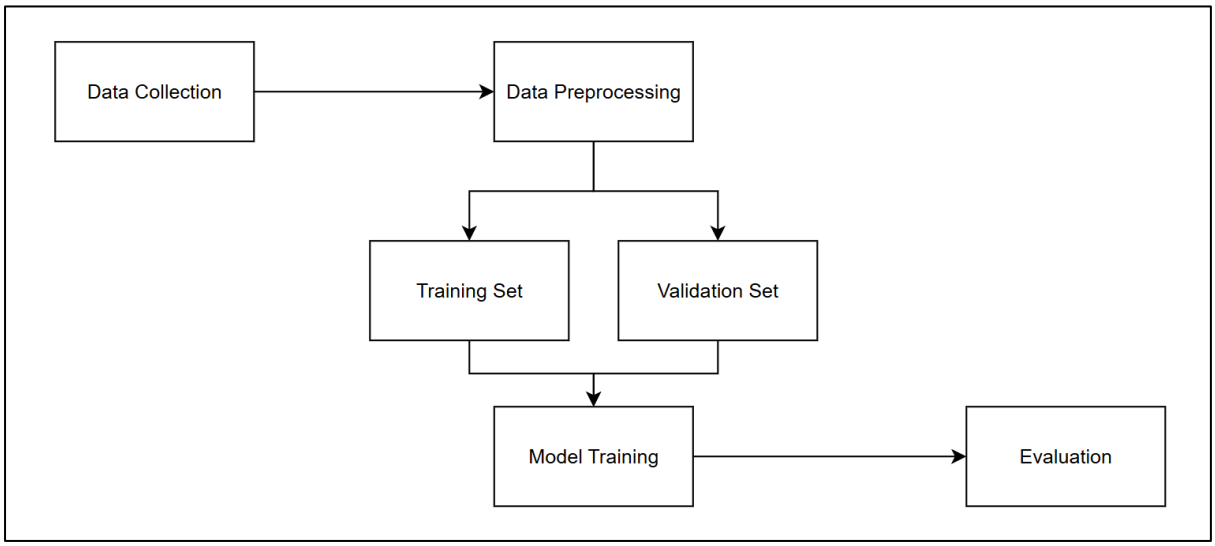

Figure 3.1: Diagram of the Training Workflow

Figure 3.1 is a diagram that depicts the process flow when conducting Training Validation and Testing of the OCR models. Data collected will be cleaned and pre-processed for training and later split into 85/15. Training and validation will be the 85 ratios of the splits then which will be later go through folding for K-Fold Cross Validation. Models outputted from training will later be refurbished for demonstration on a web application.

3.1 Hardware and Environment Setup

Table 3.1: Recommended Hardware and Environment

Hardware	Specifications
CPU	Multicore CPU
Memory	16GB of RAM
GPU	Nvidia GeForce GTX 1650 or better for CUDA optimised libraries
Storage	512GB SSD
Operating System	Windows or Linux

Table above list out the recommended hardware and environment to be established before experimenting on these OCR deep learning models. A consistent computational environment is established using a consumer-grade laptop equipped with a multi-core CPU, dedicated GPU, and ample storage.
Windows Subsystem for Linux (WSL) is required for Windows due to Tesseract Training

requires Linux-native workflows. Python virtual environments will also be utilised for dependency management to ensure compatibility for both Python-based and shell-based training pipelines.

3.2 Data Collection

Two datasets found from Kaggle[1] will be used with distinct features: digital character fonts and multi-oriented scene text.

3.2.1 Chars74k

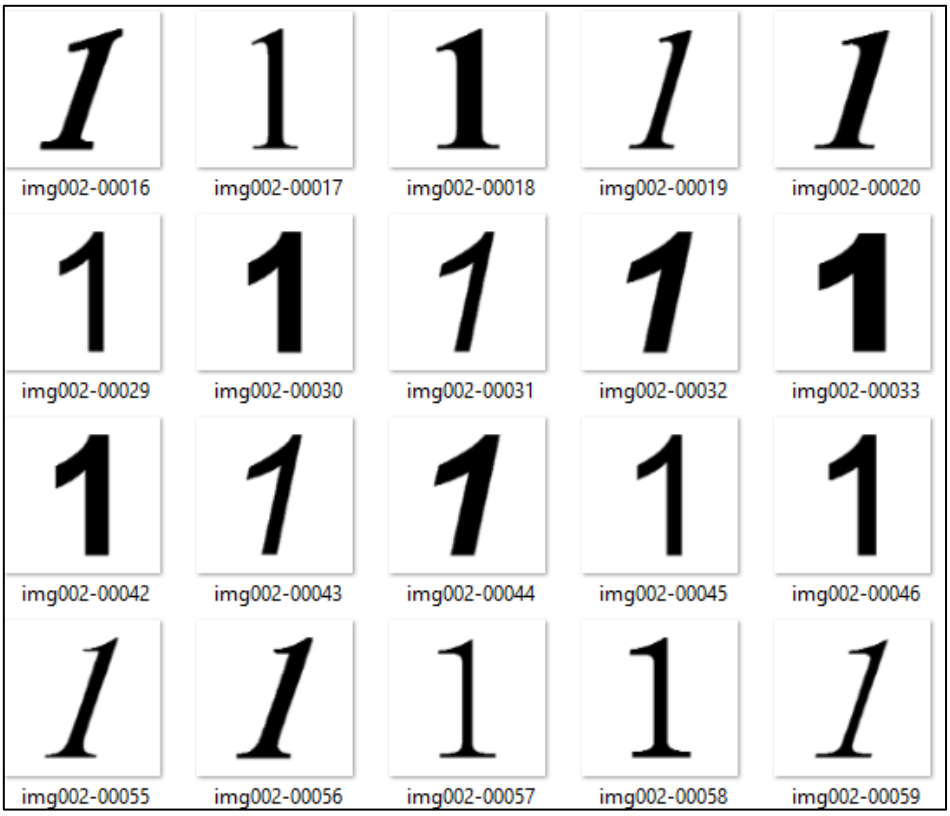

Figure 3.2: Sample data of number '1' from Chars74k

First will be a character level dataset, Chars74k, which is a structured dataset of 62 alphanumeric classes where each of them comes with at least 1,000 data per class. This comes out to more 60,000 of data to be utilised for training and evaluation. Figure 3.2 shows that the images provided are all in 128x128 pixels with a white background which provide a very clean input without any preprocessing.

[1] www.kaggle.com/datasets

3.2.2 Total Text

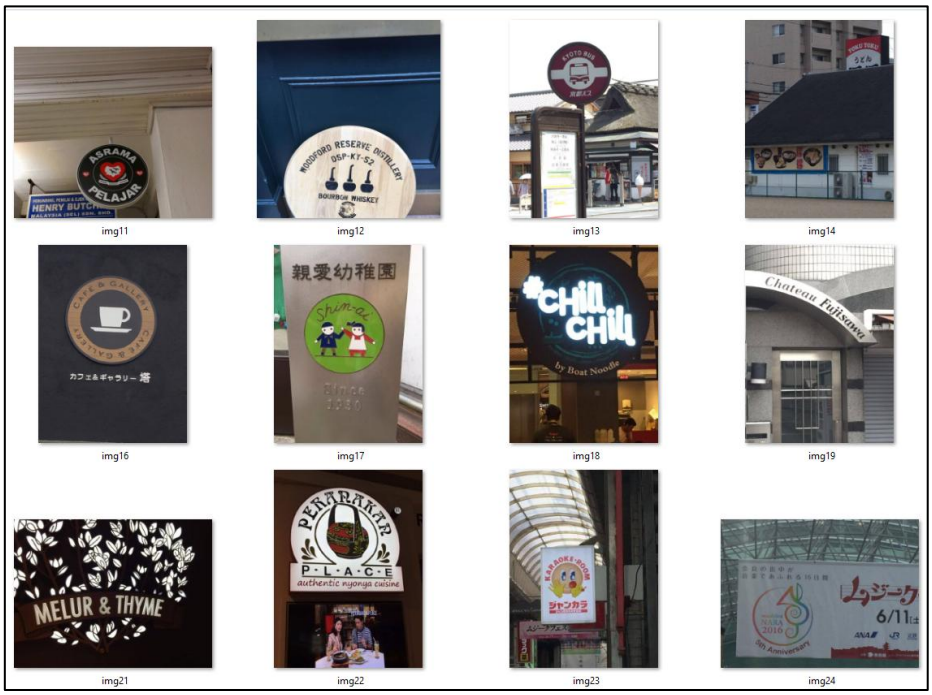

Figure 3.3: Sample of Total Texts dataset

Figure 3.3 shows some sample from Total Text which is a word-level, scene-text dataset containing curved and multi-oriented text. The ground truth is provided for the pair of region coordinates and labels for each word in the image.

3.3 Data Preprocessing

A unified preprocessing strategy is applied to both datasets to ensure fair evaluation. This includes:

- Standardizing image formats and sizes.
- Aligning image data with corresponding ground truth text.
- Structuring datasets to support K-Fold Cross validation strategies.

Each dataset undergoes cleaning, formatting, and pairing with its ground truth labels. For Tesseract OCR, images are needed to be converted to .tif or .png format and followed with a ground truth text file and separated into different folders for training and validation.

3.3.1 Chars74k Preprocessing

	image_path	label
2	datasets\Char64k\Sample001\img001-00001.png	0
3	datasets\Char64k\Sample001\img001-00002.png	0
4	datasets\Char64k\Sample001\img001-00003.png	0
5	datasets\Char64k\Sample001\img001-00004.png	0
6	datasets\Char64k\Sample001\img001-00005.png	0
7	datasets\Char64k\Sample001\img001-00006.png	0
8	datasets\Char64k\Sample001\img001-00007.png	0
9	datasets\Char64k\Sample001\img001-00008.png	0
10	datasets\Char64k\Sample001\img001-00009.png	0
11	datasets\Char64k\Sample001\img001-00010.png	0
12	datasets\Char64k\Sample001\img001-00011.png	0
13	datasets\Char64k\Sample001\img001-00012.png	0
14	datasets\Char64k\Sample001\img001-00013.png	0
15	datasets\Char64k\Sample001\img001-00014.png	0
16	datasets\Char64k\Sample001\img001-00015.png	0
17	datasets\Char64k\Sample001\img001-00016.png	0

Figure 3.4: Annotations CSV file for Chars74k

For Chars74k since all the classes of characters are in different is hard to manage the number of directories. Therefore, the data will be all read and stored in a CSV file for ease of uses for customization later needed for each of the models. The data are now labelled and stored in a CSV file for ease of access.

3.3.2 Total Text Preprocessing

Figure 3.5: “img13.jpg” from Total Text

```
x: [[332 382 459 466 403 355]], y: [[246 207 193 232 244 274]], ornt: [u'c'], transcriptions: [u'KYOTO']
x: [[483 530 558 540 508 473]], y: [[199 215 241 272 247 236]], ornt: [u'c'], transcriptions: [u'BUS']
x: [[389 536 536 393]], y: [[462 448 490 506]], ornt: [u'#'], transcriptions: [u'#']
x: [[268 379 380 273]], y: [[739 746 771 766]], ornt: [u'#'], transcriptions: [u'#']
x: [[268 380 380 276]], y: [[775 778 805 802]], ornt: [u'#'], transcriptions: [u'#']
x: [[262 393 388 263]], y: [[830 830 856 851]], ornt: [u'#'], transcriptions: [u'#']
x: [[274 379 374 282]], y: [[878 881 908 904]], ornt: [u'#'], transcriptions: [u'#']
x: [[270 385 388 276]], y: [[928 925 950 950]], ornt: [u'#'], transcriptions: [u'#']
x: [[861 892 899 875]], y: [[ 925  925 1041 1040]], ornt: [u'#'], transcriptions: [u'#']
x: [[838 866 869 852]], y: [[ 940  940 1024 1029]], ornt: [u'#'], transcriptions: [u'#']
x: [[805 836 838 810]], y: [[ 976  981 1003 1007]], ornt: [u'#'], transcriptions: [u'#']
```

Figure 3.6: Polygonal Ground Truth of "img13.jpg" from Total Text

Total Text comes with provided text files of polygonal data of the text region and the transcription of the region shown in Figure 3.6. Most special characters for example Japanese characters in Figure 3.5 are shown as '#' as transcription listed in Figure 3.6 therefore it needs to be removed for consistency. The ground truth provided is difficult to normalise for three of the OCR models training workflow. Therefore, each valid words will be cropped based on the polygonal coordinates into rectangular images with it labels and store into a CSV.

3.4 Model Training Strategy

Each of the model is trained using a tailored strategy.

- Tesseract OCR will be trained using the Tesstrain pipeline, which is managed through a Makefile that orchestrates the underlying Python training scripts. Since Make is a Unix-based tool, a Linux environment is required to execute the training commands from PowerShell.
- EasyOCR[2] will be trained with the training pipeline provide in the GitHub repository. A configuration file can be altered to customise the training pipeline which can later be ran with scripts on a Jupyter Notebook file that calls all the scripts.
- TrOCR will be trained using PyTorch with standard training loops and hyperparameter controls.

3.5 K-Fold Cross Validation

K-Fold Cross-Validation is a reliable method for assessing the performance of machine learning models. It improves generalization by dividing the dataset into multiple subsets, using different combinations for training and testing across several iterations (Chugani, 2024).

[2] https://github.com/JaidedAI/EasyOCR

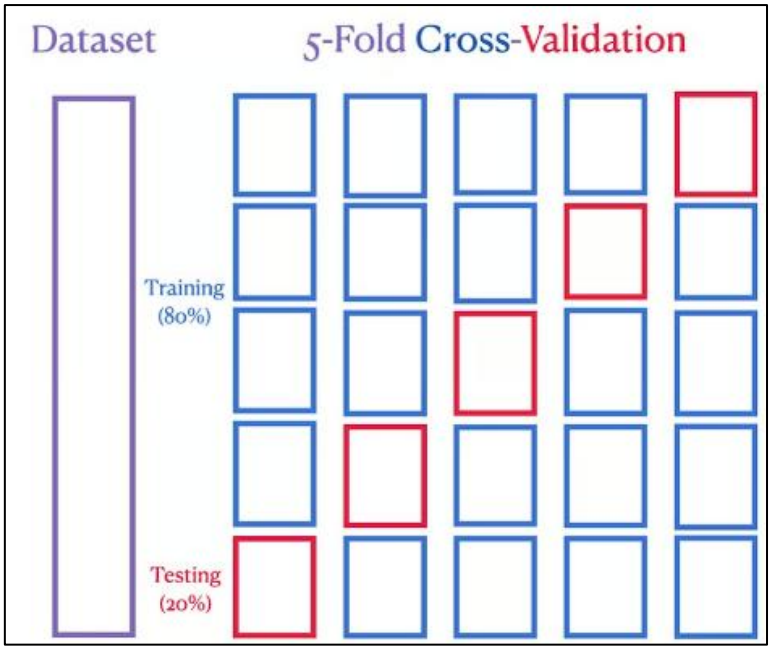

Figure 3.7: 5-Folds Cross Validation (Chugani, 2024)

The Figure 3.7 shows five unique non-overlapping datasets, and each part will serve as the evaluation set in one of the five iterations. Ensuring all segments is used for both training and evaluating.

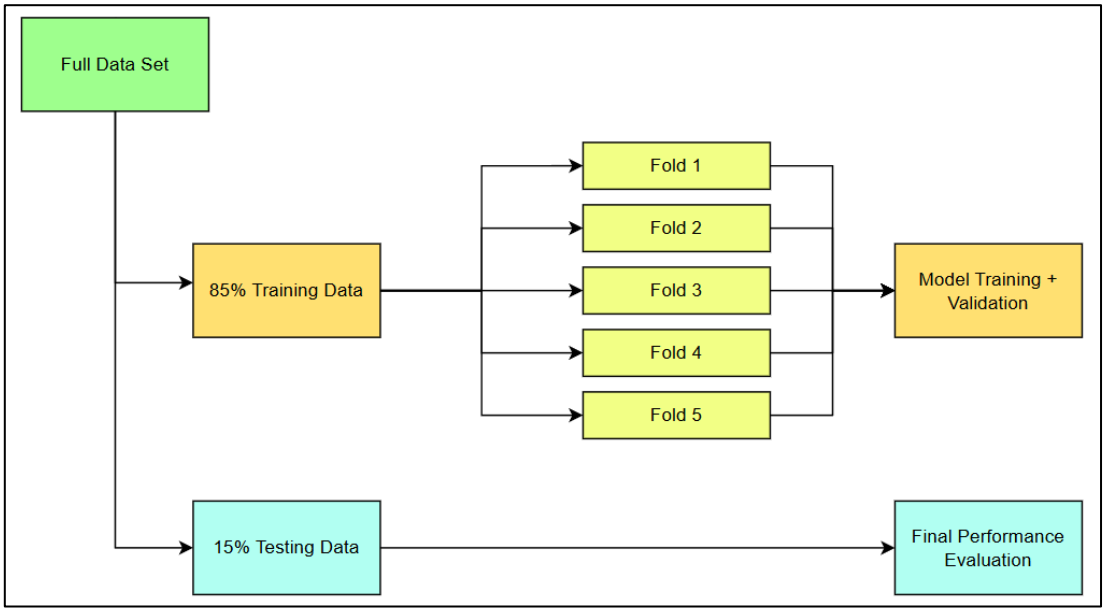

Figure 3.8: Data Training and Evaluation Splits

Figure 3.8 shows the flow of the dataset splitting from the main dataset into training and testing subsets with a 85/15 ratio. Then within the training set, K-Fold validation is used to iteratively train and validate models across multiple folds. The training data is then split into for each fold 80/20 for training and evaluation which will be practiced for the three OCR models.

3.6 Key Evaluation Metrics

To evaluate the performance of the deep learning models in OCR tasks, several key metrics are utilized. Character Error Rate (CER) provides a fine-grained measure of accuracy by calculating the edit distance between predicted and ground truth text at the character level, making it suitable for assessing low-level recognition precision. Word Error Rate (WER) offers a broader perspective by measuring errors at the word level, which is especially relevant for datasets like Total Text that contain natural language text. In addition to accuracy-based metrics, Inference Time is also recorded to assess the efficiency and responsiveness of each model during prediction. Together, these metrics offer a comprehensive view of both the effectiveness and computational performance of the tested OCR systems.

3.7 Web Interface

To provide an interactive and user-friendly platform for showcasing OCR model performance, a web-based graphical user interface (GUI) was developed using the Streamlit [3] library. Streamlit is an open-source Python framework that allows for rapid development of data-driven web applications with minimal effort and code complexity. Its simplicity and seamless integration with Python make it ideal for prototyping and deploying machine learning interfaces.

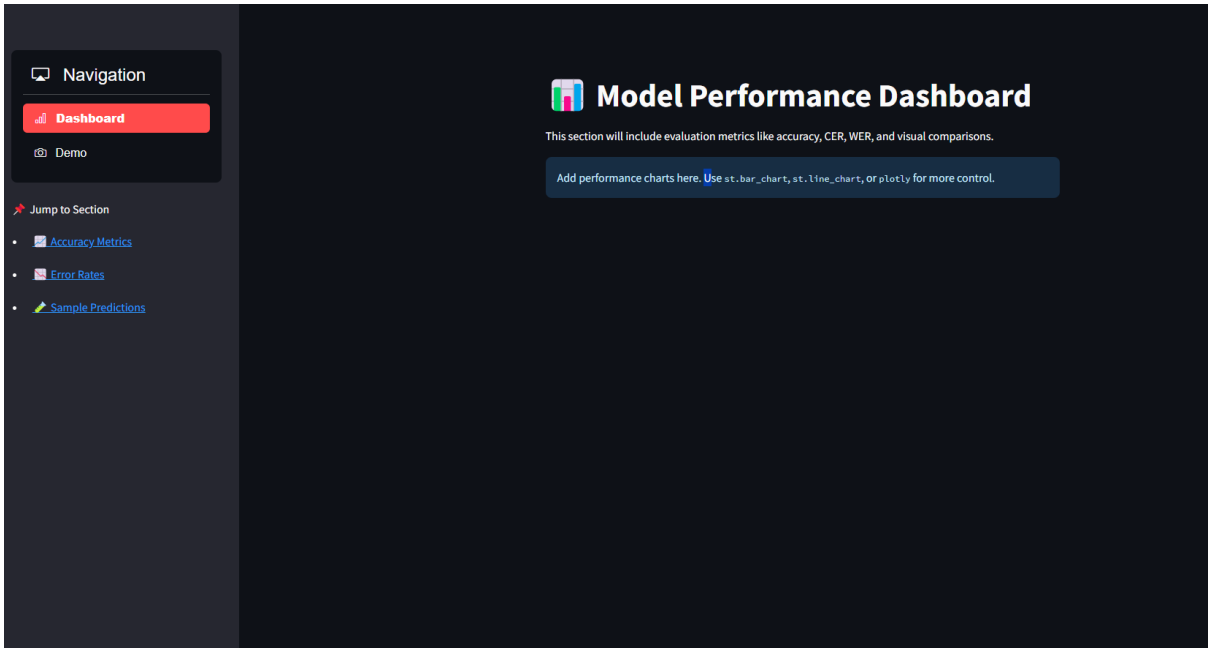

Figure 3.9: Sample of a Simple Web Interface with Sidebar

Figure 3.9 shows a sample of a Dashboard to display all the essential results from the experiments to showcase and compare the three models. The Streamlit application enables users to upload images and view OCR predictions generated by different trained models (e.g., TrOCR, EasyOCR, and Tesseract). Upon image upload, the backend processes the input through the selected model and displays the recognized text, alongside performance metrics such as Character Error Rate (CER), Word Error Rate (WER), and inference time. Additionally, the interface provides visual comparisons between ground truth labels and predicted outputs, facilitating qualitative evaluation.

[3] https://streamlit.io

Key interactive components include file upload widgets, dropdown menus for model selection, and dynamically rendered plots and tables for performance metrics. The modular design of the app allows for easy updates and integration of new features or models. Overall, Streamlit played a critical role in making the evaluation results accessible and interpretable for both technical and non-technical users.

3.8 Public Sentiment Survey as Supplementary Method

A survey was conducted to assess public familiarity with Optical Character Recognition (OCR) technology and perceptions of its reliability in practical applications. The survey was administered online using Google Forms and comprised a total of twelve questions: two demographic questions (age range and occupational background) and ten dichotomous (Yes/No) questions focusing on participants' experience with OCR technologies, perceptions of its accuracy, and its role in daily or professional contexts.

The survey was distributed to a diverse group of respondents, and the questions were intentionally designed to be straightforward and accessible, ensuring ease of participation across varying levels of technical expertise. The complete survey instrument is included in Appendix A.

CHAPTER 4

IMPLEMENTATION

Overview

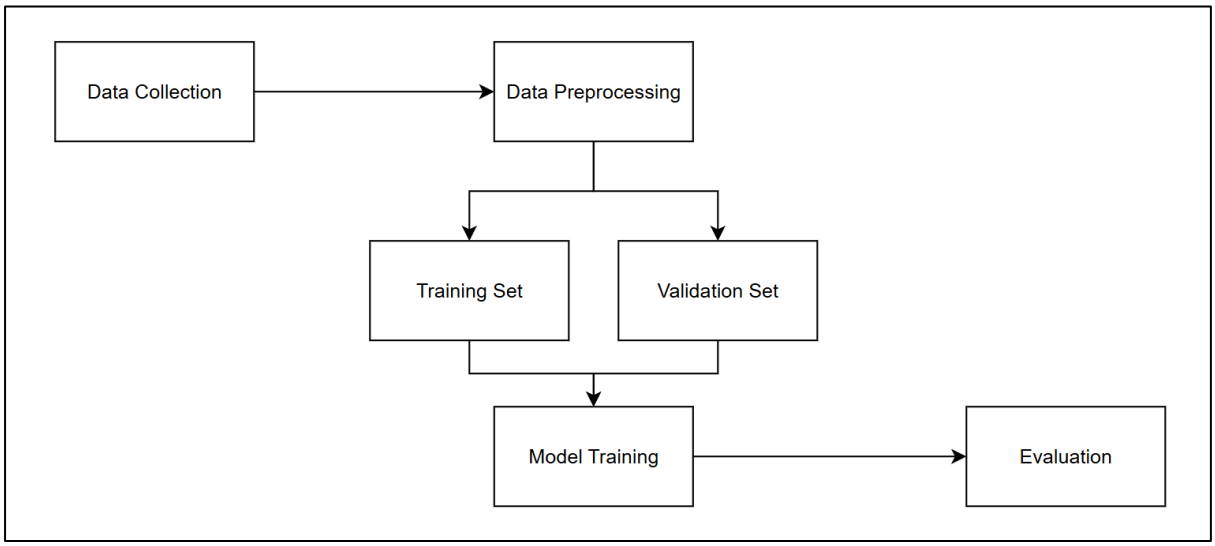

Figure 4.1: Training, Validation and Testing Process for the OCR models

This chapter describes the implementation, training, and evaluation of three OCR models conducted as part of this study. The training process, illustrated in Figure 4.1, follows a structured approach that incorporates both training and validation within a five-fold cross-validation scheme. By applying this method, the reliability and generalizability of the results are strengthened, allowing for a more robust assessment of the performance across different data splits.

The chapter is organized into three main sections. The first section details the training and validation results for the three OCR models, presenting the outcomes across all five folds and highlighting key performance metrics. The second section describes the development of a web-based interface built using Streamlit. This interface serves as both a dashboard for visualizing performance metrics and a demonstration platform that allows end-users to interact with the trained OCR models and evaluate their capabilities. The final section presents the results of a public sentiment survey conducted to assess perceptions of OCR technology. This survey explores its usage across various fields, user trust and satisfaction, as well as anticipated future significance. Together, these sections provide a comprehensive examination of the experimental outcomes, practical implementation, and user perceptions related to OCR technology.

4.1 Hardware Specification

The laptop used for the experiments and deployment of the OCR models. Its readily accessible and can be easily setup during the process to run memory intensive task required by the nature of these deep learning models.

Table 4.1: Hardware During Deployment and Training of the OCR Models

Hardware	Description
Processor: AMD Ryzen 5 5600H	AMD Ryzen 5 in a Lenovo Legion 5 is used to conduct this comparative study. 6 Cores and 12 Threads
Memory: 16GB RAM (single channel)	16GB RAM is enough for smooth execution, especially when dealing with large datasets
Graphic Processing Unit (GPU): NVIDIA GTX 1650 with 4GB VRAM	4GB VRAM is the minimum requirement when dealing with deep learning models, which involve millions of parameters that need to be stored and processed. Limited VRAM can determine your input parameters of your models.
Storage: 500GB SSD + 1TB SSD	The storage is large enough to store datasets and model weights. SSD also provides faster read and write operations, especially during training phases where data loading is crucial.

Table 4.1 has listed the hardware specification of the laptop device during the implementation of the OCR models training regiments. It is sufficient for the training of deep learning models.

4.2 Dataset Collection

The datasets shown later are being used for experimentation for training and testing of the OCR models for performance evaluation.

4.2.1 Char74k

Figure 4.2: Sample of the Chars74k dataset

Chars74k [4]is use for the training and testing of the OCR models. The dataset with the version of around 63,000 data is used. The dataset comprises of 128x128 images and 62 classes of computer font from [0-9][A-Z][a-z]. Each class have 1,016 variation of computer fonts with

[4] https://www.kaggle.com/datasets/supreethrao/chars74kdigitalenglishfont

the combinations of any variations of italic, bold and normal. Originally the dataset comes separated with their respective class of characters. Figure 4.2 shows the dataset used in one of the models training where the dataset is shuffled and split for training, validation and testing. It is labelled before and then split therefore the dataset will have ground truths as validation.

4.3.2 TotalText

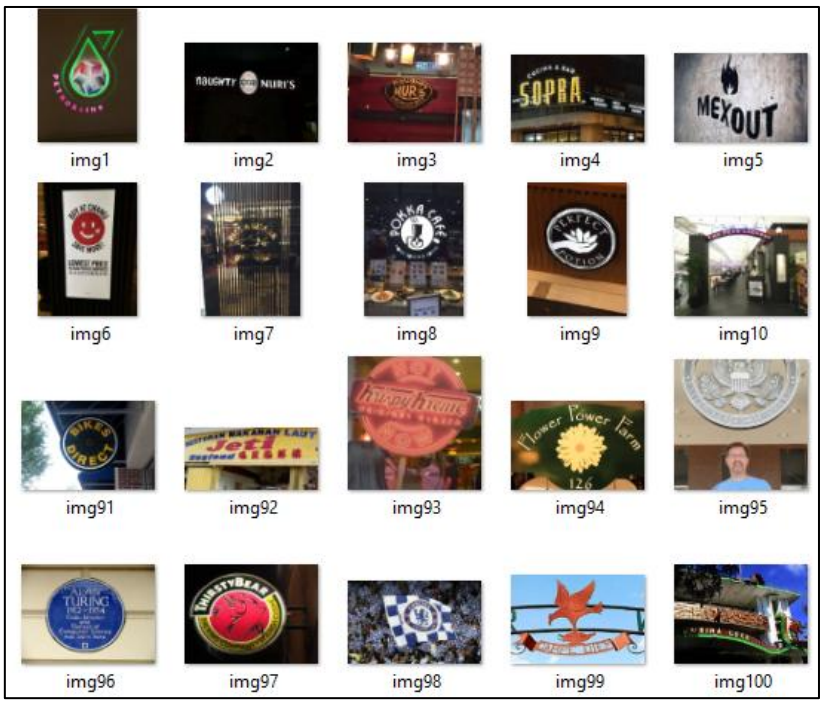

Figure 4.3: Image Data from the Total Text Dataset

Total Text [5]is a medium sized scene text dataset with 1555. Figure 4.3 shows a sample of Total Text which is a word-level English curve text dataset with orientations of horizontal, multi-oriented, and curved. There are many texts in a single image with multiple regions to consider therefore it is very complex. Since the text region are provided from the dataset's author, each word is cropped rectangularly based on the provided polygonal text regions to simplify the complexity of the data.

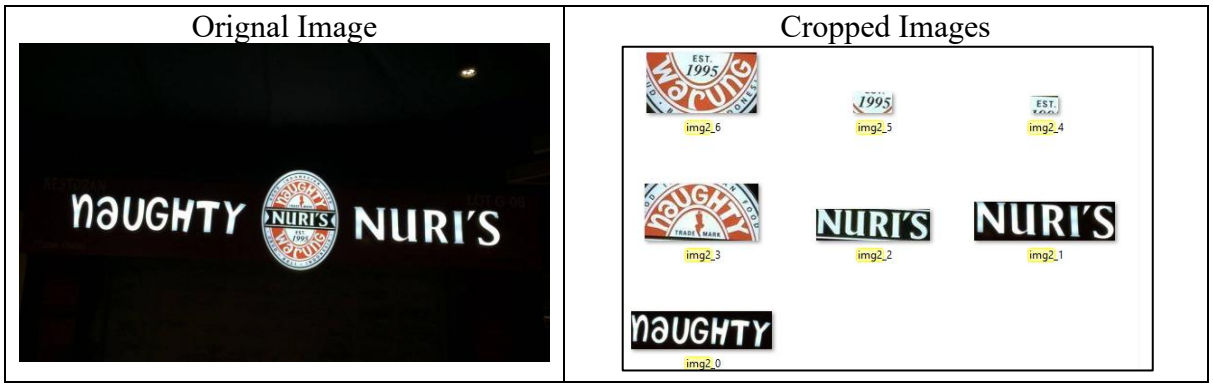

Figure 4.4: Comparison of the cropping of Total Text Images

[5] https://github.com/cs-chan/Total-Text-Dataset

This means that the data size also has been increase to around 10,000. For example, in Figure 4.4, a single images data is separated into 7 separate images. There are all cropped based on the given text region ground truth coordinates and word labels provided for the uncropped data is also based on the text regions. This will provide a good representation of the accuracy for the OCR models in terms of word error rates due to it comparing with a single word instead of comparing to a sentence of words if the original single image was used.

4.3 Training and Validation for Tesseract OCR

Tesseract is a widely adopted open-source OCR engine with strong support for customization and training. Using the official Tesstrain[6] repository, custom models can be trained for specific datasets and text recognition tasks.

4.3.1 Systems Prerequisites and Python Environment Setup

To ensure a reproducible and Linux-friendly development environment on a Windows system, this project utilizes:

- **Windows Subsystem for Linux (WSL)** for running Linux binaries.
- **Visual Studio Code** with the Remote – WSL extension for seamless integration.
- **Python virtual environment** within WSL for dependency isolation.

a) Windows Subsystem for Linux (WSL)

Enable WSL through PowerShell with "wsl –install" then for integration with Visual Studio Code. WSL allows Linux binaries to run natively on Windows.

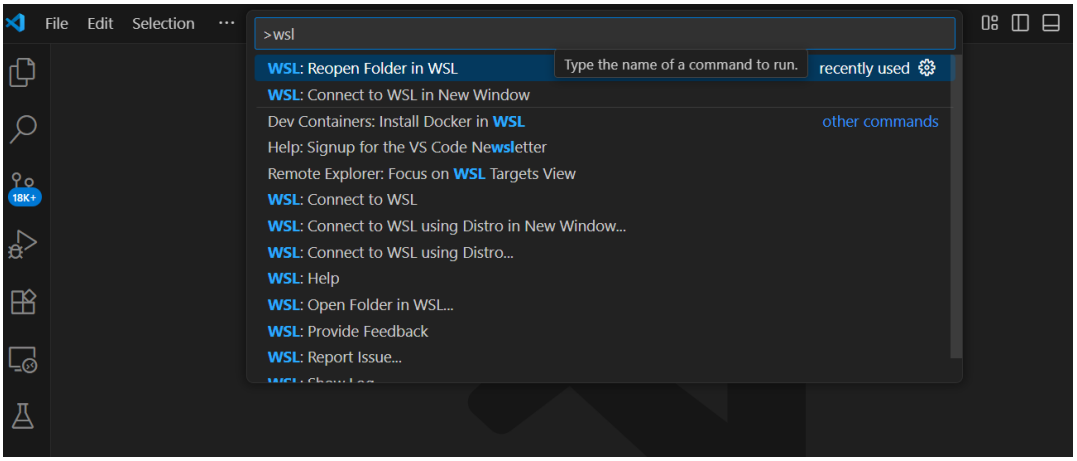

Figure 4.5: Connect to the WSL environment with Visual Studio Code

Then install the Remote – WSL extension in VS Code and connect using the command palette. Pressing (F1 > Remote WSL) and search for WSL shown in Figure 4.5, just download reopen the folder or connect to WSL in a new window.

[6] https://github.com/tesseract-ocr/tesstrain

b) Python 3 (inside WSL)

Installing Python3 for will be essential even if the device already have Python 3 install. Tesseract training will need to run python files and libraries such Pillow which is important for image processing.

```
bash
sudo apt install python3
```

c) Tesstrain Repository

Clone or download the Tesstrain Github Repository to a desired working directory. This provide all the scripts needed for a end-to-end training process.

4.3.2 Installing Required Packages and Tools

1. Tesseract OCR (5.3.X)

The project uses the recent version with training support enabled. This can be install from Ubuntu repositories.

```
bash

sudo apt install tesseract-ocr libtesseract-dev libleptonica-dev
```

2. GNU Utilities

Several tools are required by the training scripts, such as make, wget and unzip are standard on most Linux distributions. This is required for basic file,shell and text manipulation of many operations performed on Linux.

```
bash

sudo apt install make wget unzip
```

3. Python Libraries

tesstrain > requirements.txt

```
1  Pillow>=6.2.1
2  python-bidi>=0.4
3  matplotlib
4  pandas
5
6
7  #Extra:
8  sklearn
```

Figure 4.6: Python libraries used for Tesseract Training

Figure 4.6 shows all the python libraries installed for the training. Python libraries used are Pillow, python-bidi, matplotlib, pandas which can be found in the Tesstrain repository's

requirement text file. Other than that, sklearn will be included to modify the training for k-fold cross-validation to perfectly split data for each fold.

4.3.3 Data Preparation

There will be many differences between the models in terms the set up of the data before inputted for training. The following will show a sample of the preprocessing required for the Tesseract OCR training.

4.3.3.1 Image Format

Only the following formats are supported:

- .tif
- .png, .bin.png, .nrm.png

Images must be in the format of TIFF with the extension of .tif or PNG and have the extension of .png , .bin.png, or .nrm.png. The trainer scripts will not detect images with other formats other than this.

4.3.3.2 Provide Ground Truth Data

Image will be paired with the ground truth or label text file all into desire folder or the default “data/[model name]-ground-truth” path in main tesstrain folder

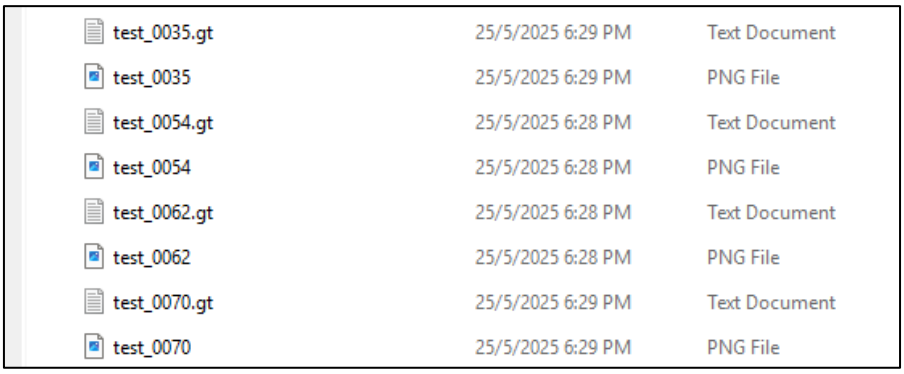

Figure 4.7: Sample of Prepared Data (Tesseract)

Each training image must have a matching ground truth .gt.txt file with the same base filename as shown in Figure 4.7. As for the Total Text dataset, images are cropped with the word or line level with its label.

4.3.4 Training Workflow

4.3.4.1 Default Training

To begin training:

```
bash
make training
```

This will use the default values from the Makefile. Even the directory to your training datasets will be in the default path.

4.3.4.2 Custom Training

```
Variables

  MODEL_NAME         Name of the model to be built. Default: foo
  START_MODEL        Name of the model to continue from (i.e. fine-tune). Default: ''
  PROTO_MODEL        Name of the prototype model. Default: OUTPUT_DIR/MODEL_NAME.traineddata
  WORDLIST_FILE      Optional file for dictionary DAWG. Default: OUTPUT_DIR/MODEL_NAME.wordlist
  NUMBERS_FILE       Optional file for number patterns DAWG. Default: OUTPUT_DIR/MODEL_NAME.numbers
  PUNC_FILE          Optional file for punctuation DAWG. Default: OUTPUT_DIR/MODEL_NAME.punc
  DATA_DIR           Data directory for output files, proto model, start model, etc. Default: data
  OUTPUT_DIR         Output directory for generated files. Default: DATA_DIR/MODEL_NAME
  GROUND_TRUTH_DIR   Ground truth directory. Default: OUTPUT_DIR-ground-truth
  TESSDATA_REPO      Tesseract model repo to use (_fast or _best). Default: _best
  TESSDATA           Path to the directory containing START_MODEL.traineddata
                     (for example tesseract-ocr/tessdata_best). Default: ./usr/share/tessdata
  MAX_ITERATIONS     Max iterations. Default: 10000
  EPOCHS             Set max iterations based on the number of lines for training. Default: none
  DEBUG_INTERVAL     Debug Interval. Default:  0
  LEARNING_RATE      Learning rate. Default: 0.0001 with START_MODEL, otherwise 0.002
  NET_SPEC           Network specification (in VGSL) for new model from scratch. Default: [1,36,0,1 C
  FINETUNE_TYPE      Fine-tune Training Type - Impact, Plus, Layer or blank. Default: ''
  LANG_TYPE          Language Type - Indic, RTL or blank. Default: ''
  PSM                Page segmentation mode. Default: 13
  RANDOM_SEED        Random seed for shuffling of the training data. Default: 0
  RATIO_TRAIN        Ratio of train / eval training data. Default: 0.90
  TARGET_ERROR_RATE  Stop training if the character error rate (CER in percent) gets below this value
  LOG_FILE           File to copy training output to and read plot figures from. Default: OUTPUT_DIR/
```

Figure 4.8: List of Variables from the Make file in Tesstrain

Customise the training by changing the variables listed in Figure 4.8 is used to change the training configurations used for training.

Key variables:

- MODEL_NAME: Output model name.
- START_MODEL: Base pretrained language (e.g., eng).
- EPOCHS or MAX_ITERATIONS: Training duration control
- PSM: Page Segmentation Modes

```
make training \
  MODEL_NAME=char74k1 \
  START_MODEL=eng \
  GROUND_TRUTH_DIR=../char74k_test/char74k-ground-truth \
  MAX_ITERATIONS=42834 \
  LEARNING_RATE=0.002 \
  PSM=10 \
  DEBUG_INTERVAL=1000 \
  TESSDATA=../tessdata_best \
  TESSDATA_REPO=_best \
```

Figure 4.8: Sample of Make training command

Figure 4.8 shows a sample of the training command used during experiment of this research. The training automatically generates .lstmf files and these files store bounding boxes, text, and image data. Generating these files for large datasets (e.g., Char74k) may take 8–12 hours.

4.3.4.3 Page Segmentation Mode (PSM)

PSM determines the layout assumption for input images.

```
Page segmentation modes:
  0    Orientation and script detection (OSD) only.
  1    Automatic page segmentation with OSD.
  2    Automatic page segmentation, but no OSD, or OCR. (not implemented)
  3    Fully automatic page segmentation, but no OSD. (Default)
  4    Assume a single column of text of variable sizes.
  5    Assume a single uniform block of vertically aligned text.
  6    Assume a single uniform block of text.
  7    Treat the image as a single text line.
  8    Treat the image as a single word.
  9    Treat the image as a single word in a circle.
 10    Treat the image as a single character.
 11    Sparse text. Find as much text as possible in no particular order.
 12    Sparse text with OSD.
 13    Raw line. Treat the image as a single text line,
       bypassing hacks that are Tesseract-specific.
```

Figure 4.9: Tesseract Page Segmentation Modes Descriptions

Figure 4.9 shows the list for page segmentation modes (PSM), which determine how the behaviour of the OCR. Therefore, choosing the correct PSM will affect the accuracy of the OCR based on the pattern of data that it is dealing with. For Chars74k PSM 10 is used since it only contains single character images while PSM 8 is used for Total Text since it is cropped to contain a single word with various orientation.

4.3.4.4 Implementing K-Fold Cross Validation

```python
def split_file(input_file, ratio):
    """
    Splits a text file into list.train and  list.eval with lines ratio.
    """
    if not isinstance(input_file, pathlib.Path):
        input_file = pathlib.Path(input_file)
    if not input_file.exists():
        print(f"'{input_file}' not exists!")
        return False
    lines = input_file.read_text().splitlines()

    split_point = int(ratio * len(lines))
    output_dir = input_file.resolve().parent
    train_list = pathlib.Path(output_dir, 'list.train')
    eval_list = pathlib.Path(output_dir, 'list.eval')

    with open(train_list, 'w', newline='\n') as f1, open(
        eval_list, 'w', newline='\n'
    ) as f2:
        f1.write('\n'.join(lines[:split_point]))
        f2.write('\n'.join(lines[split_point:]))
    return True
```

Figure 4.10: Code for generate_eval_train.py

There are limitations to the default script since the sequence of the data is predetermined during the training initialisation it be difficult to implement K-fold Cross Validation. The data splits for training are handled when the original python file called “generate_eval_train.py” shown in Figure 4.10: It does not support K-Fold as it only splits once based on a fixed ratio.

```python
def kfold_split(input_file, num_folds, eval_fold):
    """
    Splits a text file into list.train and list.eval using K-Fold cross-validation.
    Only the selected eval_fold will be used as eval, others as train.
    """
    input_file = pathlib.Path(input_file)
    if not input_file.exists():
        print(f"'{input_file}' does not exist!")
        return False

    lines = input_file.read_text().splitlines()
    kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)
    fold_indices = list(kf.split(lines))

    if eval_fold >= num_folds:
        print(f"Invalid eval_fold {eval_fold}, must be between 0 and {num_folds - 1}")
        return False

    train_idx, eval_idx = fold_indices[eval_fold]
    train_lines = [lines[i] for i in train_idx]
    eval_lines = [lines[i] for i in eval_idx]

    output_dir = input_file.resolve().parent
    train_list = pathlib.Path(output_dir, 'list.train')
    eval_list = pathlib.Path(output_dir, 'list.eval')

    with open(train_list, 'w', newline='\n') as f1, open(eval_list, 'w', newline='\n') as f2:
        f1.write('\n'.join(train_lines))
        f2.write('\n'.join(eval_lines))

    print(f"[OK] Fold {eval_fold}: {len(train_lines)} train lines, {len(eval_lines)} eval lines.")
    return True
```

Figure 4.11: Code for kfoldsplit.py

A new script “kfoldsplit.py” will replace the previous “generate_eval_train.py” and the line it is called in the Make file in line 202 to 204.

Arguments include:

- Number of folds
- Target fold index (0-based)

The sequence and folds of the data can now be determined through the inputted arguments with the defined functions shown in Figure 4.11. If there are 5 folds, there will be 5 different distinct evaluation folds of index 0 to 4 can be manually inputted to train or evaluate a certain fold of data. The script will be integrated into the Makefile.

```
202    $(OUTPUT_DIR)/list.eval \
203    $(OUTPUT_DIR)/list.train: $(ALL_LSTMF) | $(OUTPUT_DIR)
204        $(PY_CMD) generate_eval_train.py $(ALL_LSTMF) $(RATIO_TRAIN)
```

Figure 4.12: Previous Snippet of the Makefile in tesstrain

```
202    $(OUTPUT_DIR)/list.eval \
203    $(OUTPUT_DIR)/list.train: $(ALL_LSTMF) | $(OUTPUT_DIR)
204        $(PY_CMD) kfoldsplit.py $(ALL_LSTMF) 5 2
```

Figure 4.13: New Snippet of the Makefile in tesstrain

Figure 4.12 and Figure 4.13 shows the changes made in the original Makefile from the tesstrain folder. The 2nd and 3rd argument of 5 and 2 can be seen in Figure 4.11, where 5 will be the number of folds while 2 will be the sequence of the fold from 0 to 4. These value will be manually configured for the folds dataset to train and validate with.

4.3.3.5 Validation

After Training is completed for all folds, evaluation of each fold is done manually with tesseract commands. It will evaluate through the list of data that is split for a fold.

```
lstmeval \
  --model data/char74k5/checkpoints/char74k5_checkpoint \
  --traineddata data/char74k5/char74k5.traineddata \
  --eval_listfile data/char74k5/list.eval \
  --verbosity 2
```

Figure 4.14: Tesseract commands for Evaluation

The "lstmeval" command is a tesseract function to evaluate a model which runs a thorough inference with the evaluation data listed during the training of each model for each fold. Figure 4.14 shows a example of the command used to evaluate the 5th fold of the model trained with Chars74k dataset.

Variables to configure:

- model : the directory of the checkpoints folder for the model where the best checkpoint will be used for evaluation
- traineddata : the directory to the modelname.traineddata file
- eval_listfile : the directory to the list.eval file generated during training
- verbosity : details shown during evaluation.

After evaluating, the only output will be CER and WER of the evaluation which will be shown in the shell. The metrics will then be recorded as the validation accuracy score for each fold.

4.4 Training and Validation for EasyOCR

EasyOCR framework is an open-source OCR model. EasyOCR provides a modular training pipeline built with Pytorch, supporting configuration and dataset formats. The training was conducted using the official trainer module available in EasyOCR's GitHub repository, customised to suit the requirements of the selected datasets and experimental design.

4.3.1 Required Packages, Repositories

Table 4.2: Required Packages and Repositories for EasyOCR

Python :	Version 3.6 or above
Pytorch :	Version 1.6.0 or above (CUDA support for GPU acceleration)
EasyOCR Repository[7]:	Clone the official source code to your project folder
Python packages:	Listed from "/EasyOCR/requirement.txt": • torch • torchvision>=0.5 • opencv-python-headless • scipy • numpy • Pillow • scikit-image • python-bidi • PyYAML • Shapely • pyclipper • ninja

Table 4.2 shows the requirement setup needed for EasyOCR. The trainer scripts are all include in the EasyOCR repository clone it to a desired folder. For torch and torch vision is better to install it through the official website[8] due to the default installation from pip install might not be the version with CUDA support for GPU acceleration.

[7] https://github.com/JaidedAI/EasyOCR
[8] https://pytorch.org

4.4.2 Dataset Preparation

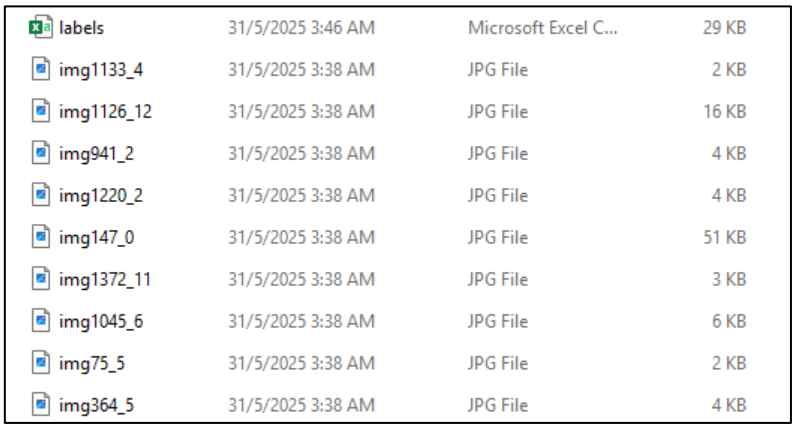

Figure 4.15: Folder of data for Training with EasyOCR

The data needs to be prepared before input for training as shown in Figure 4.15. All the image data information for filename and the label must be store in a CSV file name "labels.csv" with column headers of "filename" and "words". This must be done for both training and validation data files which are separated.

```
filename,words
img1001_4.jpg,Alpaca
img1001_7.jpg,hats
img1002_5.jpg,BUILDERS
img1003_2.jpg,SPECIALS
img1004_3.jpg,own
img1004_6.jpg,SON
img1007_0.jpg,OSCARS
img1007_2.jpg,OF
img1007_4.jpg,HANDBAGS
img1008_0.jpg,CONNOISSENR
img1009_12.jpg,DINNER
```

Figure 4.16: Sample of labels.csv of validation data for Total Text

Sample of the CSV file with the images is shown above in Figure 4.16. This is the format of for the training script to reference the ground truth of the images.

4.4.3 Model and Training Configuration

Training pipeline supports various model components. Therefore, before training the configuration files where a sample can be found in "trainer/config_files". The following configuration listed below in Table 4.3

Table 4.3: Training Configuration for EasyOCR

Training Parameters:	
Workers	6
Batch Size	32
Number of Iterations	8000
Validation interval	8000
Batch max length	34

Input Processing:	
Image height	64
Image width	64 (Chars74k) / 400 (Total Text)
Image colour	False (grayscale)
Contrast adjustment	0
Padding	True
Model Architecture:	
Transformation	None
Feature Extraction	VGG
Sequence Modelling	Bi-LSTM
Prediction	CTC
Prediction decoding	Greedy
Input channel	1
Output channel	256
Hidden layer	256

The total iteration was set to 8000 with 32 batch size will definitely go through the full dataset especially Chars74k's 60,000 data. Evaluation then was performed at the end of training Validation Interval setting in the configuration file. The default optimizer used was Adadelta in the code of the training script.

4.4.4 Training Workflow

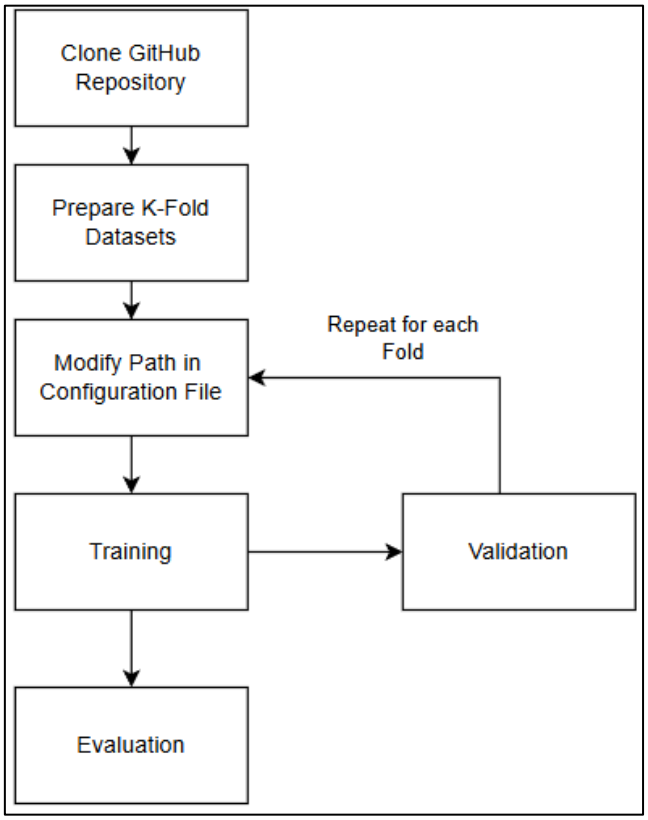

Figure 4.17: Training Workflow of EasyOCR

Figure 4.17 shows the process flow of training using EasyOCR's trainer. Each model was trained per fold using the K-Fold Cross validation method using the train.py script which was controlled in a "trainer.ipynb" Notebook file. The configuration file which was defined in .yaml format which included training paths, character sets, and also the model hyperparameters. The training will be initiated from the notebook file and then logging of training and validation loss will be saved to "log_train.txt" in the experiment name's directory. The model checkpoints of at minimum per 10,000 iterations or the best accuracy will be saved.

For K-fold cross-validation, a manual approach was used by splitting each training fold into separate directories, rather than integrating the splitting logic directly into the training loop. A 5-fold configuration was applied to the training data. This method was chosen to avoid modifying the original training code, which is complex and not easily adaptable for dynamic data partitioning within the loop.

4.5 Training and Validation of TrOCR

The implementation of the TrOCR will be entirely on the Python's library of PyTorch which provides the training loop for deep learning models. The model used will be the pretrained models from the HugginFace models.

4.5.1 Required Packages

The training workflow of TrOCR relies on a set of fundamental Python packages and libraries that enable data processing, model construction, and training. The table below show the list of python packages required.

Table 4.4: List of Python packages and its Description

Python packages	Description
Py Torch	Provides GPU-accelerated deep learning capabilities
Transformers	Enables access to Hugging Face's pretrained models and their associated processing pipelines.
Pandas	Facilitates structure data handling and preprocessing of the dataset annotation files
Pillow	Enables image loading and conversion to the required RGB format
Scikit-learn	Provides K-fold cross-validation for robust model evaluation across the dataset.
Tqdm	Enables progress tracking and visual feedback for the steps during training or validation.
Torchvision	Supports basic data augmentation and image resizing
jiwer	Provides contents for calculating word error rate (WER) and character error rate (CER), the primary metrics used for evaluation.

4.5.2 Data Preparation

All the training data and associated labels will be stored in a CSV annotation file. The training loop will then load the data from the CSV annotation file containing both the image paths and labels using Pandas. Paths are normalized to match the project folder's path convention.

```
image_path,label
datasets\TotalText\Test_Cropped\img1000_0.jpg,drunk
datasets\TotalText\Test_Cropped\img1000_1.jpg,CRAFT
datasets\TotalText\Test_Cropped\img1000_2.jpg,BEER
datasets\TotalText\Test_Cropped\img1000_3.jpg,COFFEE
datasets\TotalText\Test_Cropped\img1000_4.jpg,RESTAURAN
datasets\TotalText\Test_Cropped\img100_0.jpg,GEMBIRA
datasets\TotalText\Test_Cropped\img100_1.jpg,LOKA
datasets\TotalText\Test_Cropped\img1055_0.jpg,TAJ
datasets\TotalText\Test_Cropped\img1055_1.jpg,MAHAL
datasets\TotalText\Test_Cropped\img1055_2.jpg,INDIA
```

Figure 4.18: Sample annotations CSV file for the data

Figure 4.18 shows the dataset that was processed to generate a CSV file containing paths to the image data. During training, the CSV file will be read and loaded later on processed for training. K-fold Cross-Validation will be implemented into the training loop itself where the dataset is partitioned into training and validation splits using scikit-learn's KFold.

4.5.2.1 Data Loading and Preprocessing

```python
class OCRDataset(Dataset):
    def __init__(self, image_paths, texts, processor):
        self.image_paths = image_paths
        self.texts = texts
        self.processor = processor

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        try:
            image = Image.open(image_path).convert("RGB")
        except Exception as e:
            print(f"Error loading image: {image_path}. Skipping. Error: {e}")
            return self.__getitem__((idx + 1) % len(self.image_paths))

        text = str(self.texts[idx])
        encoding = self.processor(images=image, text=text,
                                  return_tensors="pt",
                                  padding="max_length",
                                  truncation=True,
                                  max_length=128)
        encoding = {k: v.squeeze() for k, v in encoding.items()}
        return encoding
```

Figure 4.0.19: Class Function for Data Loading and Preprocessing

The class function code shown in Figure 4.19 encapsulates the essential data loading and preprocessing steps. By combining Pillow (PIL) for image handling and Huggin Face's processor for feature and label generation, this class ensures every training data is correctly formatted for direct use in the training loop.

4.5.3 Training Flow

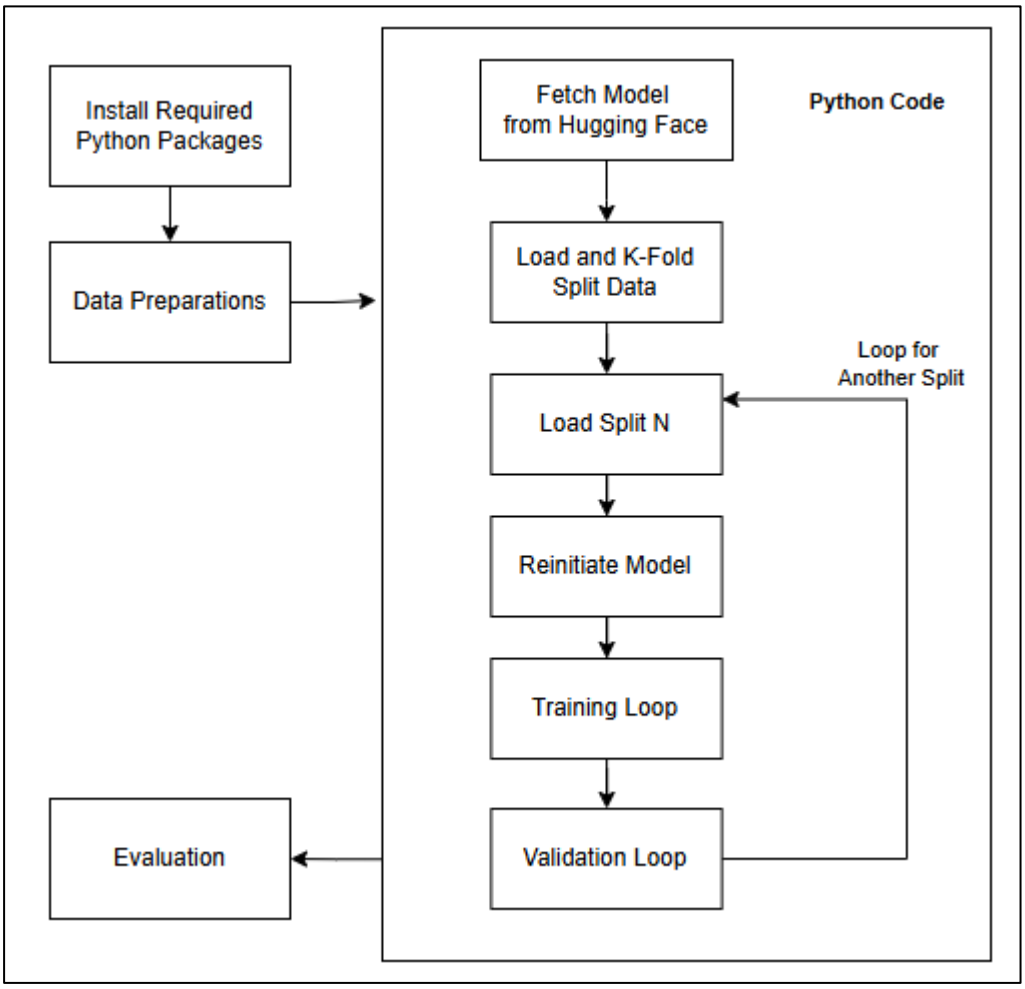

Figure 4.20: End-to-end Training Flow of TrOCR

The training routine operates with the K-fold framework in mind, looping over the defined number of folds and executing the training loop. The following Python code is outlined in Figure 4.20 and the full code can be seen at Appendix B.

1. **Data Loading:** Training and validation data are loader through the class defined data loader, and configured with batch sizes and pin memory settings for optimized GPU access.
2. **Model and Processor Intilisation**: The beginning of each fold, the pretrained TrOCR model (VisionEncoderDecoderModel) for "microsoft/trocr-base-printed" and its processor (TrOCRProcessor) are reloaded, ensuring isolation between folds.
3. **Optimization and Mixed Precision Training**:

Table 4.5 Hyperparameters configuration for the training of TrOCR

Hyperparameter	**Value**
Optimizer	AdamW
Learning rate	5e-5
Precision	Mixed (FP16)

Loss function	CrossEntropy (masked for padding tokens)

Table 4.5 shows that the model was trained using the AdamW optimizer with a learning rate of 5e-5. To reduce the GPU memory usage and accelerate training, mixed precision training was employed using Pytorch's "torch.cuda.amp" module.The AdamW optimizer is utilised for parameter updates and Mixed precision training is implemented via PyTorch's GradScaler and autocast to balance precision and performance.

4. **Training Loop**: At each iteration, a batch of data is fed through the model, yielding a loss. The loss is then scaled, backpropagated, and the optimiser is stepped. Finally the model outputs and labels are decoded to strings, allowing the calculation of interim training metrics (WER, CER).
5. **Validation**: At defined intervals, the model is switched to evaluation mode. The validation lss, WER and CER are computed across the validation dataset. Results are then logged to a CSV files, and the best performing model across folds is saved based on the least validation loss.

It is worth noting that the TrOCR model was trained with a relatively limited setup so only 100 iterations per fold and a batch size of 8. This training regime was intentionally kept short due to computational constraints, providing an approximate indication of the model's performance. Despite this, TrOCR achieved competitive results across both the Chars74k and Total Text datasets, highlighting its strong potential when trained more extensively.

4.6 Web Prototype

To effectively present an interactive visualization and practical demonstration of the OCR model's performance, a web-based prototype was developed using the Streamlit framework. Streamlit is a lightweight Python-based tool that allows for the rapid development of web apps tailored for machine learning applications.

The prototype was implemented using Python logics for the backend, with the up forth mentioned Streamlit for the frontend and user interface. With that the interface will consists of two primary tabs with the "Dashboard" and the "Demo" tab, designed to showcase the evaluation and demonstration of the research.

4.6.1 Dashboard Page

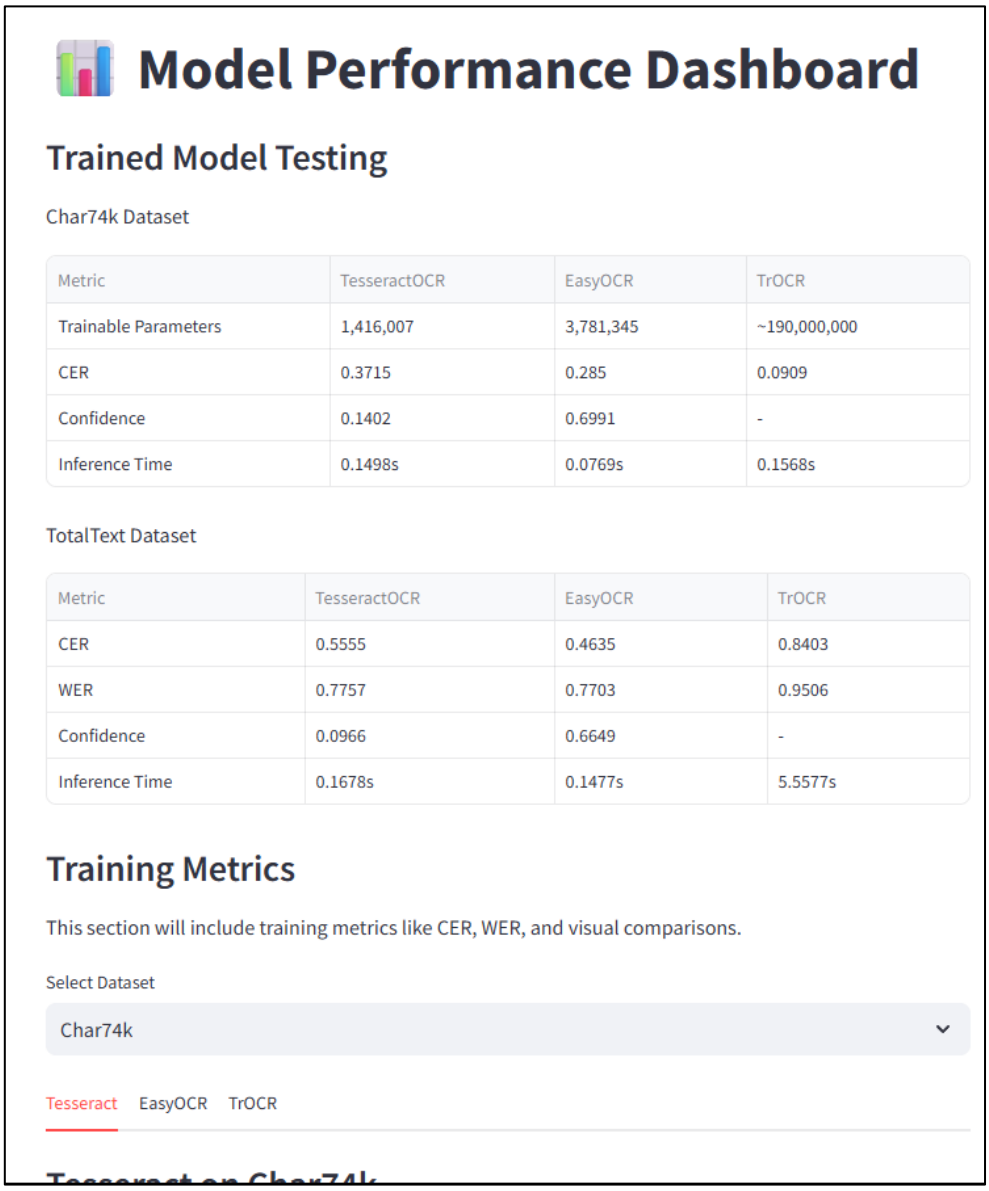

Figure 4.21: Dashboard Page of the Web Prototype

The dashboard tab shown in Figure 4.21 provides a comparative view of the evaluation metrics across the three OCR models used in the experiment. This page displays metrics such as accuracy and inference time in the form of interactive line charts, allowing users to quickly assess performance differences across models.

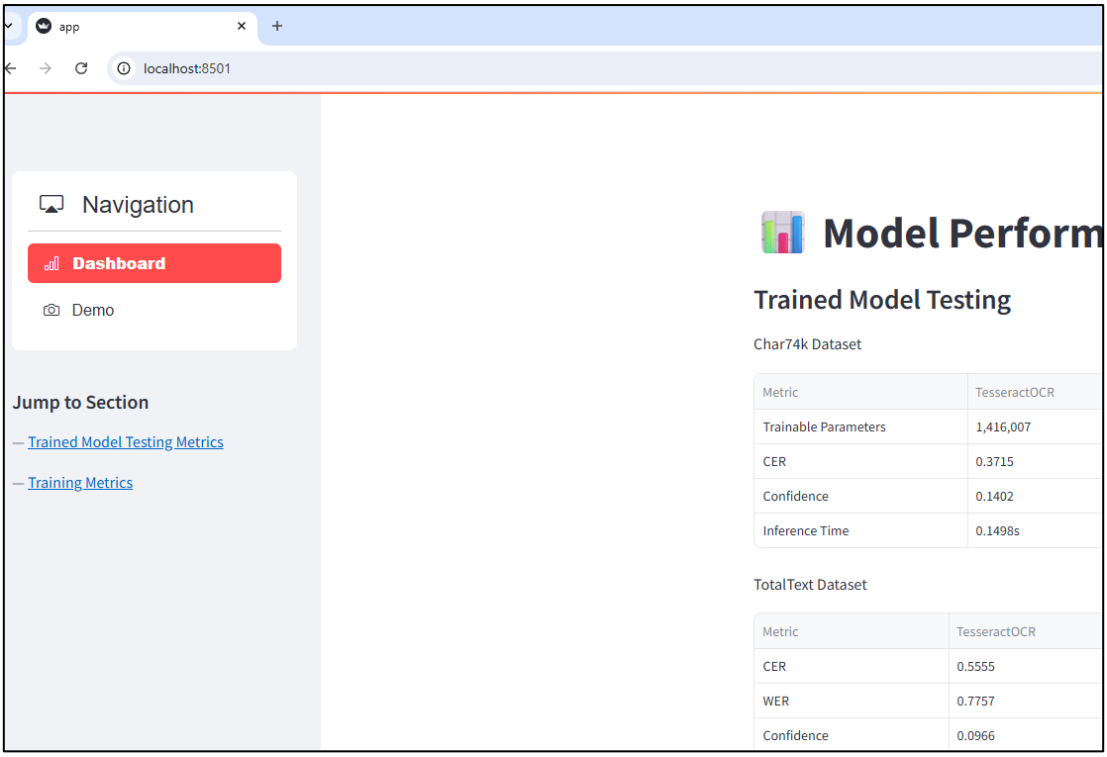

Figure 4.22: Side Navigation bar of the Web Prototype

A side navigation bar in Figure 4.22, allows for seamless navigation between sections of the Dashboard, making it convenient for the user to quickly access specific charts or performance metrics. The sidebar can be configured to be always visible or collapsible, depending on user preference.

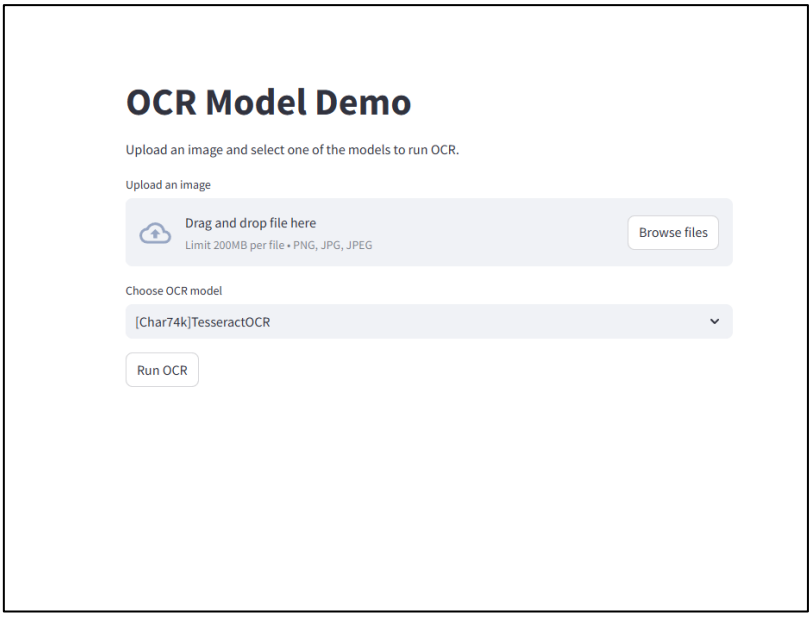

Figure 4.23: Demo page for the Web Prototype

The **Demo** page in Figure 4.23 provides an interactive environment for end-users to test the trained OCR models. Users can upload an image and select one of the trained OCR models (such as LSTM, CRNN, or TrOCR) to perform text recognition.

4.6.2 Demo Page

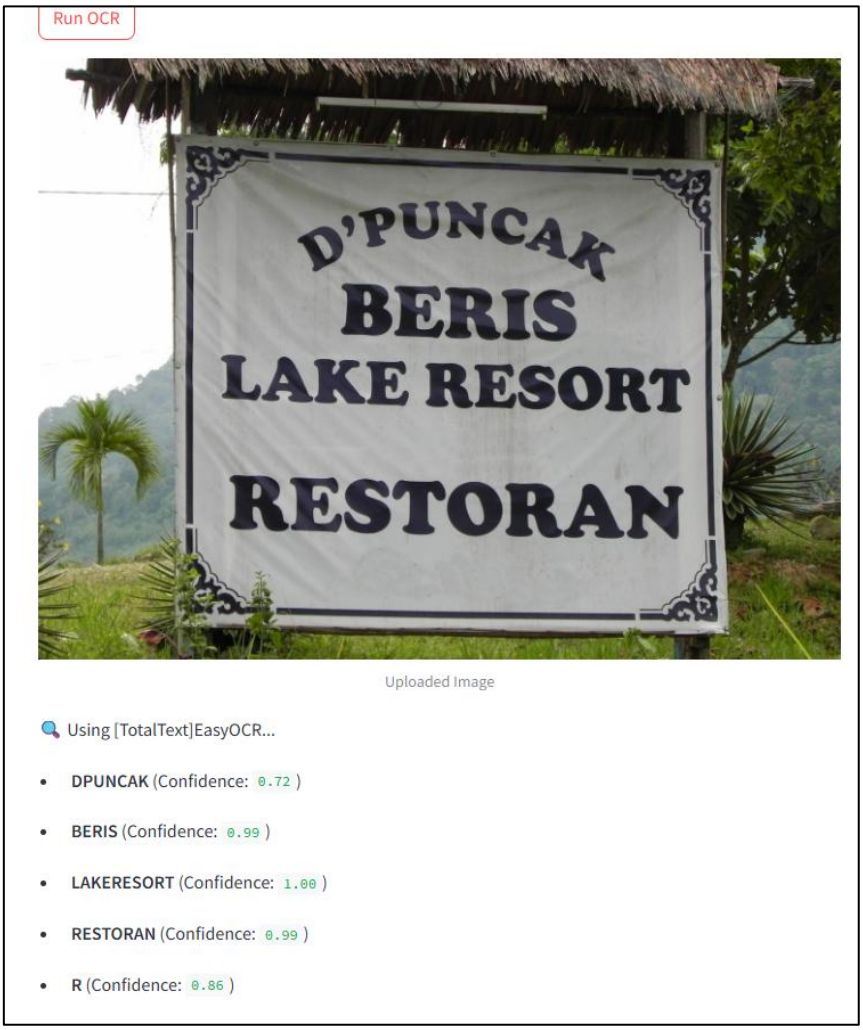

Figure 4.24: Sample of an OCR Demonstration

Once an image is uploaded, it is rendered on the page, and the extracted text appears immediately below the image shown in Figure 4.24. This allows for a side-by-side review of the results, making it easy to compare the performance of the different models and assess their effectiveness across varied input images.

4.7 Survey Results

The survey received a total of 25 responses from participants across a range of disciplines, including Information Technology (IT)/Software Development, Computer Science/Data Science, Engineering, Finance/Accounting, Legal/Law, Education, and the Arts. There are 60% of them from IT backgrounds. The age of respondents ranged from 20 to 25 years. Both of these charts are shown in Figure 4.25 and Figure 4.26.

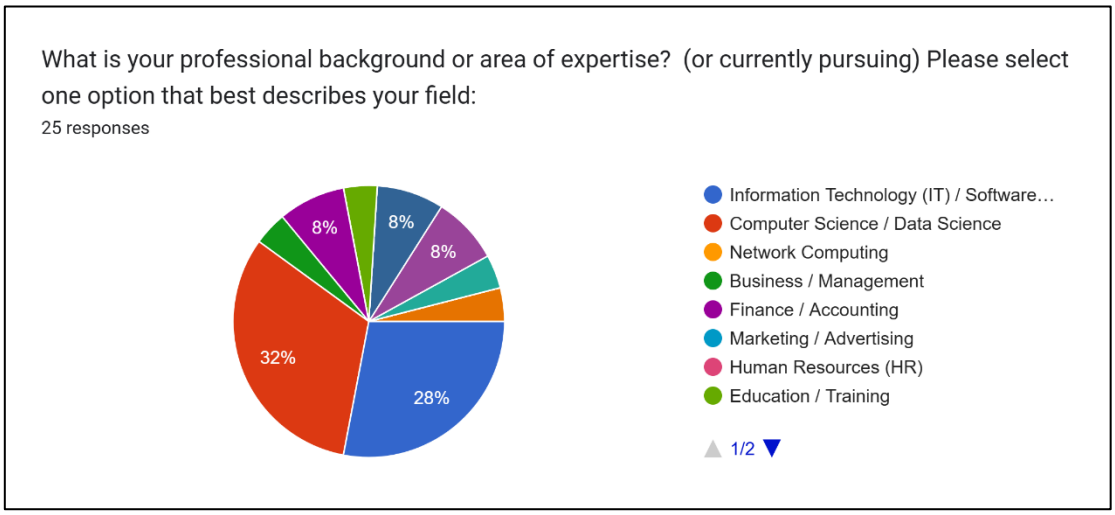

Figure 4.25: Background of the Respondent

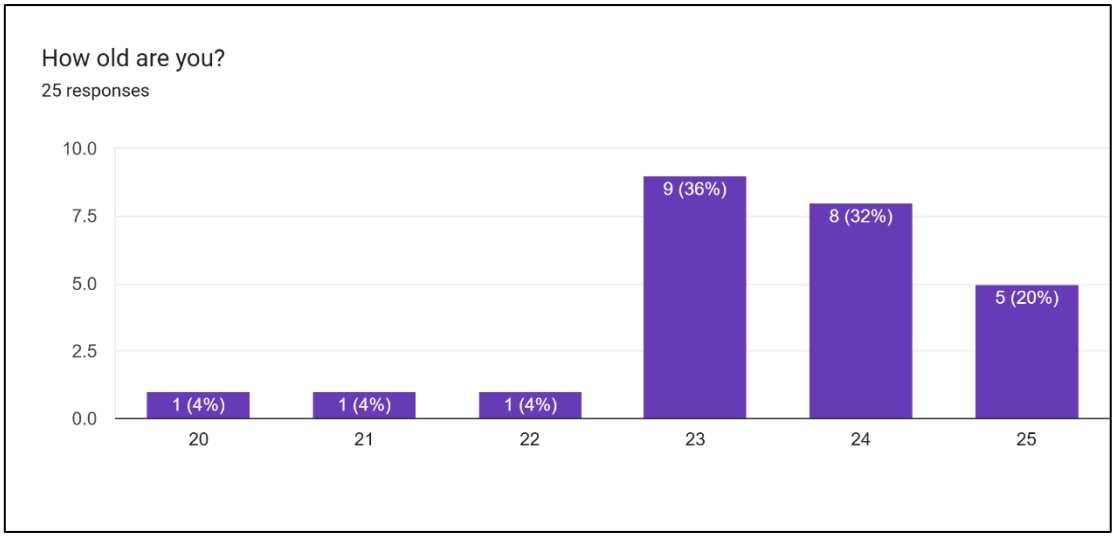

Figure 4.26: Age of the Respondent

4.7.1 General Awareness and Usage of OCR

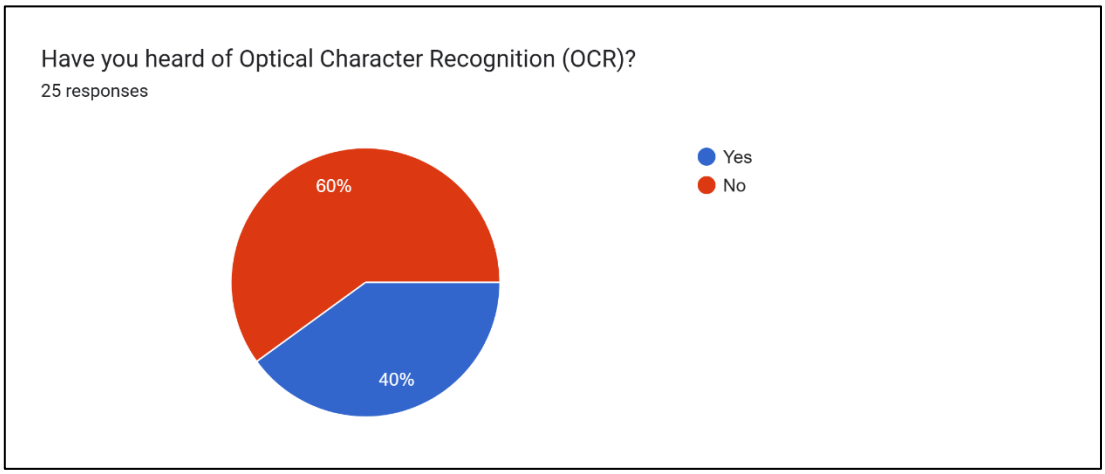

Figure 4.27: Respondent's Awareness of OCR

The chart in Figure 4.27 shows the awareness of OCR technology. Only 40% of the participants indicated that they had heard of OCR or were familiar with the term, suggesting that the technology is still relatively niche across many industries. This result highlights that, despite its increasing relevance in fields such as data entry, digitization, and document processing, OCR has not yet achieved widespread recognition.

Use Cases

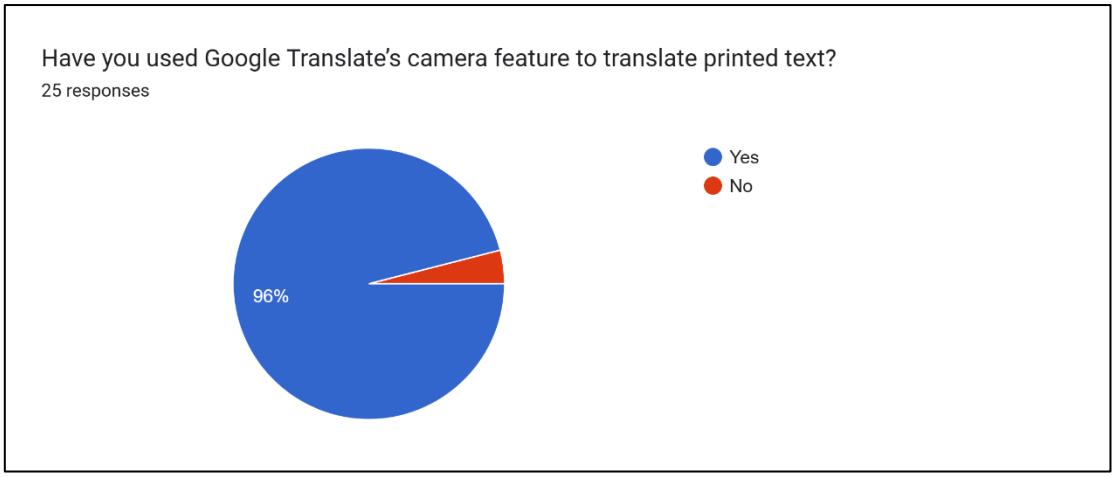

Figure 4.28: Respondent's Usage of Google Translates' Camera Function

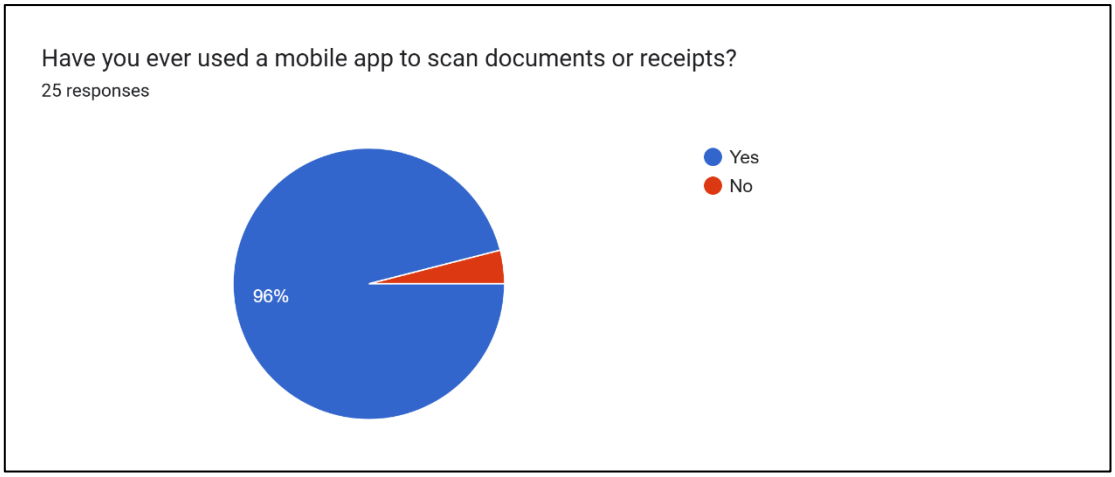

Figure 4.29: Respondent's Usage of Document Scanners Application

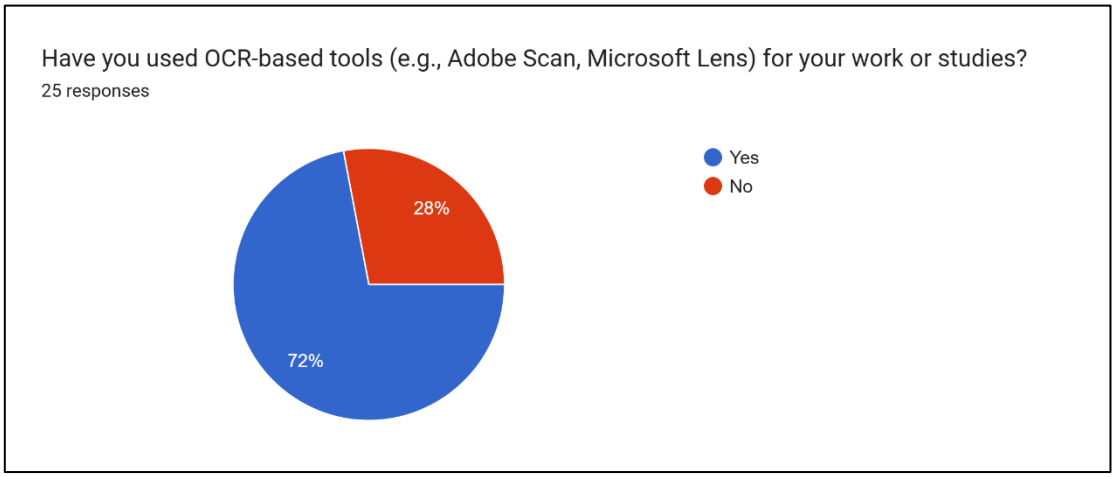

Figure 4.30: Respondent's Usage of OCR-based tools for Work

Even though only 40% of the respondents indicated that they were familiar with the term "OCR," a significantly higher proportion of them reported using OCR-related tools in practice. As illustrated in Figure 4.28 and Figure 4.29, 96% of the participants responded "Yes" when asked about their usage of the Google Translate camera feature and mobile document scanner applications, respectively. Meanwhile, Figure 4.30 shows that 72% of respondents have used OCR-based tools in their work or studies.

These results suggest that, despite relatively low formal awareness of the term, OCR technology is already embedded in common digital practices and is being utilized extensively across every day and in professional contexts.

4.7.2 Trust and Perceived Efficiency of OCR

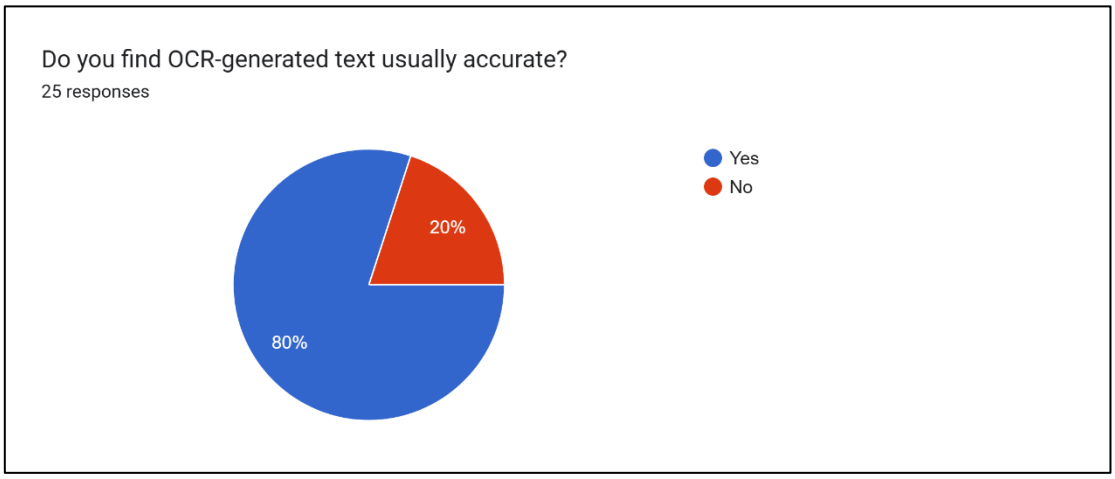

Figure 4.31: Respondent's opinion on OCR's Accuracy

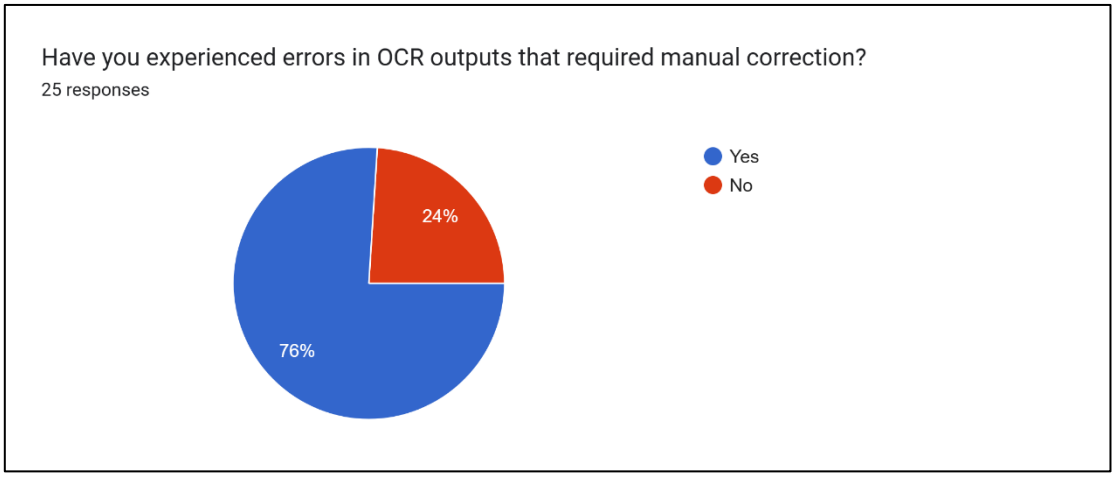

Figure 4.32: Needs to Correct OCR Outputs

Figure 4.31 illustrates the respondents' perceptions of OCR accuracy. A significant **80%** of participants stated that OCR-generated text is generally accurate. However, as shown in Figure 4.32, **76%** of respondents indicated that they have experienced a need to manually correct OCR outputs.

These results reveal an important nuance: although the majority of participants recognize OCR as an accurate technology, a substantial proportion still encounter errors that require manual intervention. This highlights the current limitations of OCR tools and underscores the need for ongoing improvements in reliability and precision.

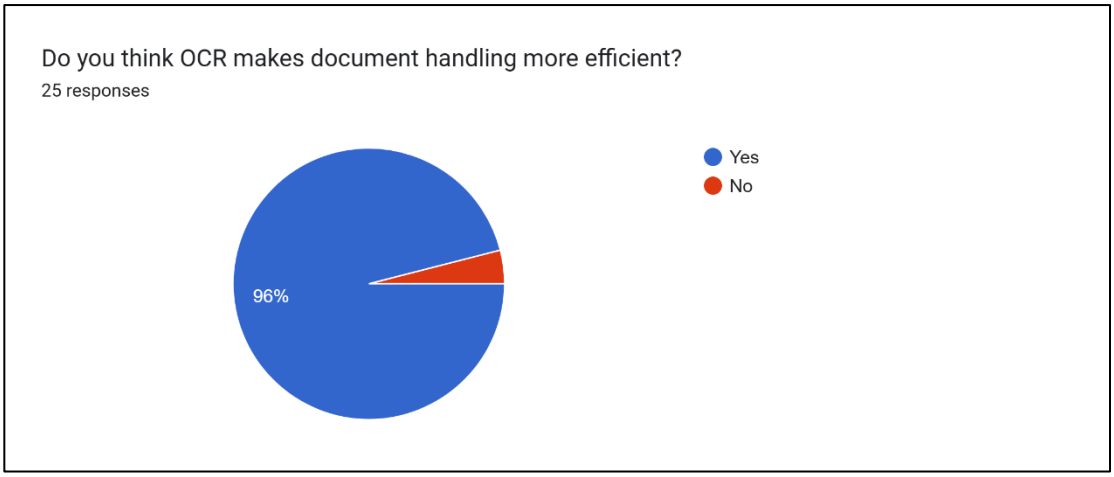

Figure 4.33: Respondent's Opinion on OCR Document Handling Efficiency

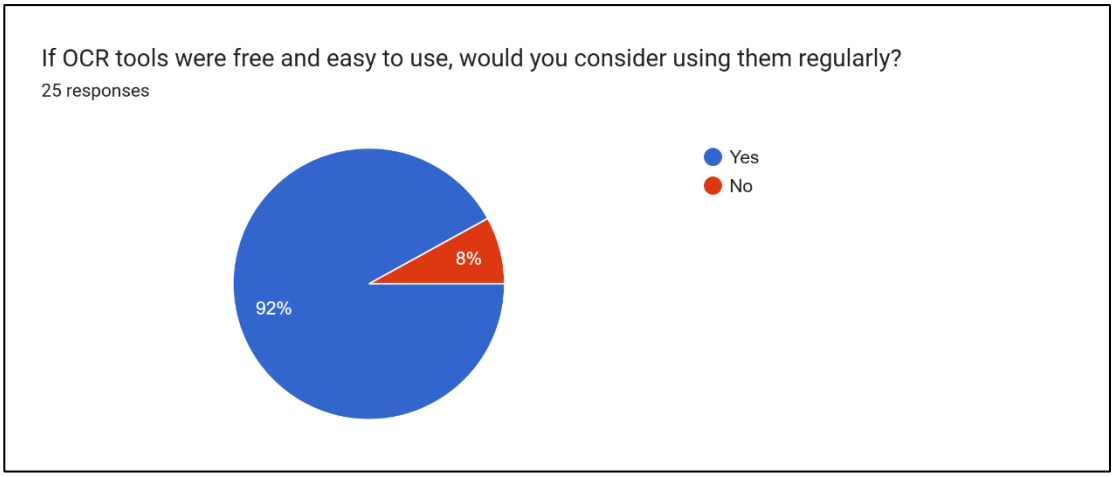

Figure 4 0.24: Respondent's Eagerness to use OCR

The survey results indicate strong perceptions of OCR's efficiency and accessibility. As illustrated in Figure 4.33, **96%** of respondents agreed that OCR technology helps make document handling more efficient, highlighting its perceived value in streamlining routine tasks. Additionally, Figure 4.34 shows that **92%** stated they would consider using OCR tools more regularly if these were free and easy to use.

These findings underscore the high potential for broader OCR adoption, suggesting that improvements in accessibility, ease of use, and cost can further catalyze its widespread integration across both professional and personal contexts.

4.7.3 Outlook and Importance of OCR

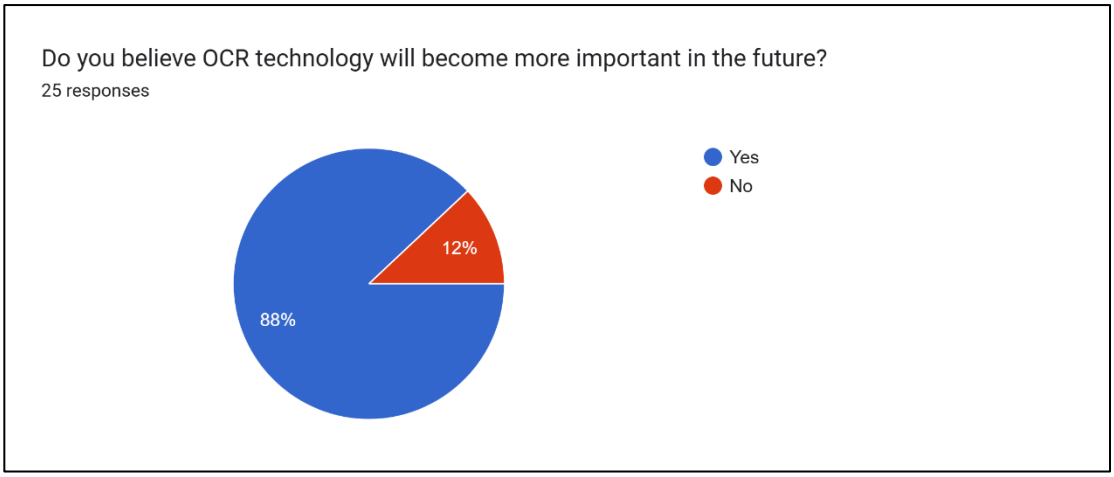

Figure 4.35: Respondent on Future Importance of OCR

As presented in Figure 4.35, a significant 88% of respondents expressed the belief that OCR technology will become increasingly important in the future. This sentiment emphasizes the growing role OCR is expected to play in digitalization, suggesting a strong trend towards its wider adoption across industries and disciplines. The results point to a favourable outlook for OCR and its continuing evolution as a vital component of modern information processing.

4.8 Chapter Summary

This chapter details the implementation and experimental procedures used to evaluate three OCR models: Tesseract, EasyOCR, and TrOCR. It outlines the preprocessing steps, dataset handling, and model-specific configurations used to ensure a fair comparison across both the Chars74k-Fonts and TotalText datasets. A manual 5-fold cross-validation strategy was employed to assess the models' robustness without modifying the original training pipelines. Each model was evaluated using Character Error Rate (CER), Word Error Rate (WER), and inference time as key performance metrics. The implementation results highlight the strengths and limitations of each approach across structured and unstructured text scenarios. In addition, the chapter introduces a graphical user interface (GUI) developed to visualize model outputs and enable user interaction through real-time OCR testing. This comprehensive evaluation framework forms the basis for the discussion and conclusion in the following chapter.

In addition to the quantitative evaluation, this chapter includes findings from a user-based survey designed to assess the usability and effectiveness of the developed graphical user interface (GUI). Survey responses indicated that the interface was user-friendly and effectively supported the comparison of model outputs. This qualitative feedback complements the experimental results and validates the system's practical applicability. Overall, this chapter lays the groundwork for interpreting the comparative outcomes and their implications in the following discussion chapter.

CHAPTER 5

EVALUATION AND ANALYSIS

Overview

This chapter presents the results and analysis of the three OCR models, Tesseract, EasyOCR, and TrOCR, which are evaluated on the Chars74k and Total Text datasets. The performance of each model is examined in terms of character error rate (CER), word error rate (WER), inference time, and stability across K-fold cross-validation. The results highlight the trade-offs between speed, accuracy, and robustness across clean and challenging datasets. The chapter concludes with a summary of key findings and recommendations for future work.

5.1 Tesseract OCR Training Results

The performance of the Tesseract OCR model was evaluated using K-fold cross-validation across two distinct datasets, Chars74k and Total Text. The results, including Character Error Rate (CER), Word Error Rate (WER), Root-Mean-Square loss (RMS), and Delta, are presented to assess both the accuracy and reliability of the model.

In the context of OCR training, the RMS metric refers to the root-mean-square loss, which measures the average magnitude of error between the model's predicted outputs and the ground truth labels. Lower RMS values indicate closer alignment between the model's output embeddings and the actual labels, suggesting better model performance. Meanwhile, the Delta metric captures the relative difference in loss across training and evaluation, serving as an indicator of the model's stability and consistency. Small Delta values imply low overfitting or underfitting, reflecting robust learning behaviour across the cross-validation folds.

The results of this evaluation are presented and discussed in the following sections for both Chars74k and Total Text datasets.

5.1.1 Chars74k (Tesseract)

Table 5.1: Chars74k K-Fold Cross-Validation Results

Chars74k	Fold1	Fold2	Fold3	Fold4	Fold5
Rms	0.98%	0.96%	0.95%	0.97%	1.00%
Delta	4.63%	4.46%	4.20%	4.21%	4.72%
Train CER	0.15	0.146	0.136	0.142	0.15
Val CER	0.148	0.1522	0.13904	0.15176	0.14354

Table 5.2: Mean and Standard Deviation for Table 5.1

Metrics	Mean	Std Deviation
Train CER	0.1448	0.00593
Val CER	0.1469	0.00561

Table 5.1 presents the K-fold cross-validation results for the Chars74k dataset, including RMS,

Delta, and character error rate (CER) across five folds. The results in Table 5.2 summarize the average and standard deviation for both training and validation CER, highlighting the model's performance on a relatively clean and well-structured dataset.

The model achieved a low average training CER of 0.1448 ± 0.00593 and a similarly low average validation CER of 0.1469 ± 0.00561. The minimal gap between training and validation error rates confirms that the model generalizes effectively, with very low risk of overfitting. The RMS remained stable across folds, averaging approximately 0.969%, and the Delta metric varied between 4.20% and 4.72%, indicating consistent optimization behaviour throughout training.

Overall, these results highlight the model's strong ability to recognize characters in the Chars74k dataset, achieving highly accurate and stable performance across all splits.

5.1.2 Total Text (Tesseract)

Table 5.3: Total Text K-Fold Cross-Validation Results

Total Text	Fold1	Fold2	Fold3	Fold4	Fold5
Rms	2.96%	3.02%	3.02%	2.96%	3.04%
Delta	14.95%	15.96%	16.06%	15.37%	16.30%
Train CER	0.557	0.564	0.558	0.552	0.572
Train WER	0.698	0.719	0.701	0.696	0.709
Val CER	0.584	0.554	0.579	0.574	0.565
Val WER	0.731	0.700	0.727	0.719	0.710

Table 5.4: Mean and Standard Deviation for Table 5.3

Metrics	Mean	Std Deviation
Train CER	0.5608	0.00779
Train WER	0.7046	0.00945
Val CER	0.5712	0.01177
Val WER	0.7174	0.01278

Table 5.3 presents the K-fold cross-validation results for the Total Text dataset, including RMS, Delta, and error rates across five folds. Table 5.4 summarizes the mean and standard deviation for the training and validation character error rate (CER) and word error rate (WER).

The results demonstrate the increased difficulty posed by the Total Text dataset, as evidenced by higher error rates compared to cleaner, character-level datasets such as Chars74k. Despite this complexity, the low gap between training and validation metrics confirms that the model captures the data characteristics effectively and generalizes well across folds.

Specifically, the average training and validation CER were 0.5608 ± 0.00779 and 0.5712 ± 0.01177, respectively, while the average training and validation WER were 0.7046 ± 0.00945 and 0.7174 ± 0.01278. The relatively low standard deviations across all metrics further indicate the model's stability and reliability across different data splits, suggesting robust performance despite the challenging nature of the Total Text dataset.

5.2 EasyOCR Training Results

The performance of the EasyOCR model was evaluated using K-fold cross-validation on the Chars74k dataset. The results, including training and validation losses as well as character error rate (CER) across five folds, are presented to assess both the efficacy and the consistency of the training process.

5.2.1 Chars74k (EasyOCR)

Table 5.5: Chars74k K-Fold Cross-Validation Results

Chars74k	Fold1	Fold2	Fold3	Fold4	Fold5
train loss	0.127	0.128	0.133	0.125	0.127
valid loss	0.625	0.663	0.688	0.688	0.615
Train CER	0.152	0.151	0.160	0.159	0.151
Val CER	0.063	0.125	0.094	0.188	0.156

As shown in Table 5.5, the training loss remained relatively low across all folds, averaging roughly 0.128, indicating a successful optimization process. The validation loss varied more substantially, ranging from 0.615 to 0.688, which reflects the increased challenge posed by certain data splits within the dataset.

Table 5.6: Mean and Standard Deviation for Table 5.5

Metrics	Mean	Std Deviation
Train CER	0.154	0.00456
Val CER	0.125	0.04941

Table 5.6 shows that the average training CER was 0.1448 ± 0.00456, while the average validation CER was 0.1469 ± 0.04941. The low and consistent training CER suggests that the model effectively learned character-level features from the training data, while the slightly higher and more variable validation CER highlights variations across data splits, suggesting areas for further optimization or data augmentation.

Overall, these results confirm that the EasyOCR model is capable of achieving strong character-level performance on the Chars74k dataset, with low error rates and acceptable variability across folds.

5.2.2 Total Text (EasyOCR)

Table 5.7: Chars74k K-Fold Cross-Validation Results

Total Text	Fold1	Fold2	Fold3	Fold4	Fold5
train loss	1.493	1.510	1.654	1.571	1.404
valid loss	2.313	2.326	2.318	2.400	2.439
Train CER	0.322	0.318	0.324	0.336	0.335
Train WER	0.583	0.575	0.581	0.602	0.613
Val CER	0.547	0.534	0.200	0.408	0.517
Val WER	0.643	0.533	0.400	0.571	0.800

Table 5.8: Mean and Standard Deviation for Table 5.7

Metrics	Mean	Std Deviation
Train CER	0.3271	0.00807
Train WER	0.5906	0.01595
Val CER	0.4410	0.14553
Val WER	0.5895	0.14710

The EasyOCR model was further evaluated on the Total Text dataset using K-fold cross-validation. The results, including training and validation loss, character error rate (CER), and word error rate (WER), are presented in Table 5.7, while the statistical summary across five folds is shown in Table 5.8.

The results indicate that the model achieved an average training CER of 0.3271 ± 0.0081 and an average training WER of 0.5906 ± 0.0159, suggesting that the model can reliably learn character-level and word-level patterns despite the higher complexity and variability of the dataset. The average validation CER was 0.4410 ± 0.1455, and the average validation WER was 0.5895 ± 0.1471, indicating that the model maintains a reasonable level of performance across different data splits, although it is impacted by certain challenging instances present in the Total Text dataset.

Notably, the higher standard deviations for the validation CER and WER metrics imply that the model's performance varied more significantly between folds, highlighting areas where targeted data augmentation or fine-tuning could further stabilize and improve its generalization capabilities.

5.3 TrOCR Training Results

The results for TrOCR training results, including training and validation loss, character error rate (CER), and their associated statistics across five folds, are presented in the following sections.

5.3.1 Chars74k (TrOCR)

Table 5.9: TrOCR K-Fold Cross-Validation Results

Chars74k	Fold1	Fold2	Fold3	Fold4	Fold5
train loss	0.922	0.971	1.059	0.825	1.045
valid loss	0.269	0.251	0.348	0.335	0.294
Train CER	0.269	0.231	0.275	0.238	0.256
Val CER	0.300	0.255	0.245	0.285	0.245

Table 5.10: Mean and Standard Deviation of Table 5.9

Metrics	Mean	Std Deviation
Train CER	0.2537	0.01903
Val CER	0.2660	0.02510

The performance of TrOCR on the Chars74k dataset is summarized in Table 5.9, with statistical measures across five folds presented in Table 5.10. The model achieved an average training CER of 0.2537 ± 0.0190 and an average validation CER of 0.2660 ± 0.0251.

These results highlight the model's strong ability to learn character-level features from the Chars74k dataset, yielding low error rates with relatively tight standard deviations across all folds. The small gap between training and validation CER suggests that the TrOCR model generalizes well to unseen data and maintains stable performance across different splits. This underscores its suitability for character recognition tasks in relatively clean and structured datasets.

5.3.2 Total Text (TrOCR)

The TrOCR model was further evaluated on the Total Text dataset, which presents a higher degree of complexity due to its diverse text instances and challenging scene text conditions. The results across five folds are presented in Table 5.11, with statistical summaries in Table 5.12.

Table 5.11: Total Text K-Fold Cross-Validation Results

Total Text	Fold1	Fold2	Fold3	Fold4	Fold5
train loss	2.509	2.110	2.155	2.311	2.353
valid loss	1.625	1.584	1.957	1.203	1.843
Train CER	0.530	0.399	0.352	0.415	0.371
Train WER	0.663	0.525	0.600	0.625	0.606
Val CER	0.576	0.409	0.670	0.405	0.694
Val WER	0.675	0.655	0.760	0.610	0.715

Table 5.12: Mean and Standard Deviation for Table 5.11

Metrics	Mean	Std Deviation
Train CER	0.4134	0.06952
Train WER	0.6037	0.05031
Val CER	0.5508	0.13858
Val WER	0.6830	0.05729

On average, the model achieved a training CER of 0.4134 ± 0.0695 and a training WER of 0.6037 ± 0.0503, indicating that the TrOCR model was able to effectively learn character- and word-level features despite the dataset's difficulty. The average validation CER was 0.5508 ± 0.1386, and the average validation WER was 0.6830 ± 0.0573, suggesting a moderate gap between training and validation performance. This gap can be attributed to the varied and challenging nature of the Total Text dataset.

Although the higher standard deviation in validation CER highlights increased variability across folds, the results confirm that the TrOCR model maintains relatively robust generalization, achieving competitive character and word error rates in complex scene text scenarios.

5.4 Concluding Summary

Table 5.13: Final Comparison with the OCR Models

Model	Dataset	Train CER	Val CER	Train WER	Val WER
Tesseract	Chars74k	0.1448	0.1469	N/A	N/A
	Total Text	0.5608	0.5712	0.7046	0.7174
EasyOCR	Chars74k	0.1540	0.1250	N/A	N/A
	Total Text	0.3271	0.4410	0.5906	0.5895
TrOCR	Chars74k	0.2537	0.2660	N/A	N/A
	Total Text	0.4134	0.5508	0.6037	0.6830

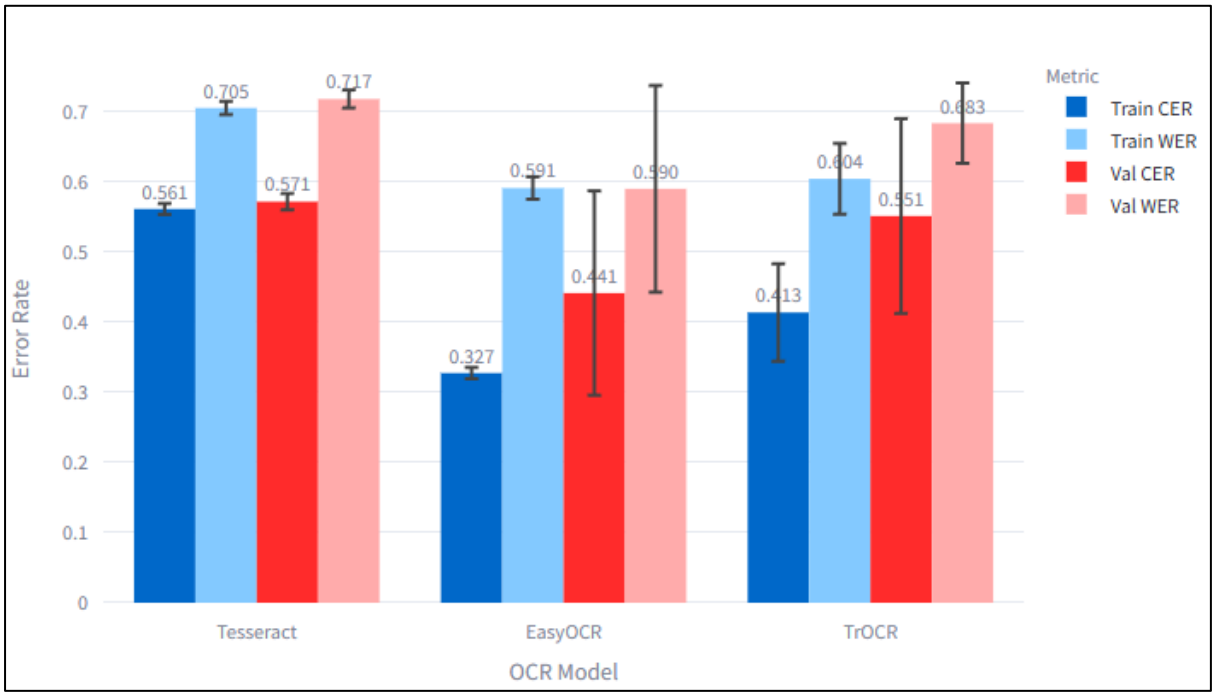

Figure 5.1: Graph Comparison of the OCR Models

Table 5.14: Inferences Time of the OCR Models

Model	Inference Time (Char74k)	Inference Time (Total Text)
Tesseract	0.1498s	0.1678s
EasyOCR	0.0769	0.1477s
TrOCR	0.15679s	5.5577s

As shown in Table 5.13 and Figure 5.1, The results across the three models, Tesseract, EasyOCR, and TrOCR, reveal distinct performance profiles when applied to the Chars74k and Total Text datasets. Table 5.14 shows the inference time of each model on each dataset. The results will highlight distinct trade-offs between accuracy and speed when applied to the Chars74k and Total Text datasets.

On the relatively clean and structured Chars74k dataset, both Tesseract and EasyOCR achieved remarkably low character error rates (CER), averaging 0.14 and 0.13, respectively, with

inference times of approximately 0.15 seconds (Tesseract) and 0.08 seconds (EasyOCR). TrOCR, despite a slightly higher CER of 0.25, still demonstrated competitive character recognition and a comparable inference time (~0.16 seconds).

For the more challenging Total Text dataset, deep learning–based methods outperformed the traditional Tesseract engine in terms of accuracy. TrOCR achieved the best character error rate (0.41) and word error rate (0.68), followed closely by EasyOCR (CER 0.44, WER 0.59). In contrast, Tesseract registered higher error rates (CER 0.57, WER 0.72). In terms of speed, EasyOCR was the most efficient deep learning model, achieving an inference time of roughly 0.15 seconds, while TrOCR required approximately 5.56 seconds per image due to its transformer-based architecture.

Overall, the results demonstrate that deep learning approaches (TrOCR and EasyOCR) provide stronger and more robust recognition capabilities for challenging scene text, outperforming traditional OCR methods. TrOCR is best suited for precision-oriented applications where robust text understanding is required despite longer processing times, whereas EasyOCR offers an attractive balance between accuracy and efficiency, making it ideal for deployments where both quality and speed matter.

5.5 Discussion

The results clearly highlight the trade-offs between traditional and deep learning–based OCR approaches across both clean and challenging datasets. The traditional Tesseract engine performed admirably on the clean Chars74k dataset, achieving a character error rate (CER) of approximately 0.14 and a word error rate (WER) that was competitive for its category. Its inference time of roughly 0.15 seconds per character image confirms its efficiency for character-level OCR tasks. However, its limitations became apparent when tested on the more complex Total Text dataset, yielding higher error rates (CER ≈ 0.57, WER ≈ 0.72) despite maintaining a relatively low inference time (~0.168 seconds per image).

In contrast, deep learning–based methods such as EasyOCR and TrOCR demonstrated stronger performance across both datasets. EasyOCR achieved low error rates (CER ≈ 0.14–0.44, WER ≈ 0.59–0.72) while maintaining highly efficient inference times (~0.08 seconds for Chars74k and ~0.15 seconds for Total Text), making it well-suited for applications where both quality and speed are priorities. TrOCR, a Transformer-based approach, delivered the best accuracy overall, achieving the lowest error rates across the challenging Total Text dataset (CER ≈ 0.41, WER ≈ 0.68). However, its inference time (~5.56 seconds per image) was significantly higher due to its complex architecture. Notably, TrOCR achieved these results despite a highly constrained training regime with only 100 iterations per fold and batch size of 8 which is suggesting significant potential for further improvement if trained for longer or with larger batch sizes.

Overall, the findings underscore the strength of deep learning–based methods for challenging scene text recognition tasks. TrOCR leads in precision, making it ideal for accuracy-critical applications, while EasyOCR provides an effective balance between precision and efficiency. Meanwhile, Tesseract remains a strong candidate for clean, character-level text due to its competitive speed and lower computational demands.

5.6 Future Work and Recommendation

While this study has demonstrated the effectiveness of deep learning–based OCR methods, including TrOCR and EasyOCR, there remain several avenues for further enhancement. First, extending the training regime for TrOCR beyond the initial 100 iterations per fold and increasing the batch size could enable the model to learn richer, more robust feature representations. This may result in lower error rates and improved performance, especially on challenging scene text such as the Total Text dataset.

In addition, applying advanced data augmentation techniques such as distortion, noise injection, and lighting variations can help with improvement of the model's ability to generalize across diverse text instances and environments. This would be especially valuable when dealing with complex scene text where variations in font, orientation, and background are common.

Optimizing inference speed also presents an important area for future work. Although TrOCR achieved the best character and word error rates, its inference times were considerably longer than those of Tesseract and EasyOCR. Techniques such as model pruning, quantization, and knowledge distillation can be explored to reduce the computational load, making TrOCR a more viable option for real-time or low-resource deployment.

Future studies can also investigate the performance of these OCR models across a wider range of datasets, including multi-lingual and multi-script text instances, to better understand their versatility and robustness. Lastly, exploring ensemble approaches such as combining the precision of TrOCR with the efficiency of EasyOCR and the reliability of Tesseract which could yield highly accurate, adaptable, and efficient OCR pipelines suitable for a variety of real-world applications.

Appendices

Appendix A:

Google Form for Survey

1. What is your **professional background** or **area of expertise**? (or currently pursuing) * Dropdown
   Please **select one** option that best describes your field:

   Mark only one oval.

   - ( ) Information Technology (IT) / Software Development
   - ( ) Computer Science / Data Science
   - ( ) Network Computing
   - ( ) Business / Management
   - ( ) Finance / Accounting
   - ( ) Marketing / Advertising
   - ( ) Human Resources (HR)
   - ( ) Education / Training
   - ( ) Healthcare / Medical
   - ( ) Engineering (e.g., Electrical, Mechanical, Civil)
   - ( ) Legal / Law
   - ( ) Arts / Design / Creative Media
   - ( ) Logistics / Supply Chain
   - ( ) Customer Service / Support
   - ( ) Sales / Retail
   - ( ) Research / Academia

2. How old are you? *

   _____

3. Have you heard of Optical Character Recognition (OCR)? *

   Mark only one oval.

   - ( ) Yes
   - ( ) No

4. Have you ever used a mobile app to scan documents or receipts? *

Mark only one oval.

- Yes
- No

5. Have you used Google Translate's camera feature to translate printed text? *

Mark only one oval.

- Yes
- No

6. Have you used OCR-based tools (e.g., Adobe Scan, Microsoft Lens) for your work or studies? *

Mark only one oval.

- Yes
- No

7. Has a bank or company ever asked you to scan your ID for verification? *

Mark only one oval.

- Yes
- No

8. Do you think OCR makes document handling more efficient? *

Mark only one oval.

- ◯ Yes
- ◯ No

9. Do you find OCR-generated text usually accurate? *

Mark only one oval.

- ◯ Yes
- ◯ No

10. Have you experienced errors in OCR outputs that required manual correction? *

Mark only one oval.

- ◯ Yes
- ◯ No

11. If OCR tools were free and easy to use, would you consider using them regularly? *

Mark only one oval.

- ◯ Yes
- ◯ No

12. Do you believe OCR technology will become more important in the future? *

Mark only one oval.

- ◯ Yes
- ◯ No

Appendix B:

```
import os
import pandas as pd
from PIL import Image
from sklearn.model_selection import KFold
from torch.utils.data import Dataset, DataLoader
import torch
from torch.optim import AdamW
from transformers import TrOCRProcessor, VisionEncoderDecoderModel
from tqdm import tqdm, trange
from torch.amp import autocast, GradScaler
from jiwer import wer, cer
from datetime import datetime
from torchvision import transforms

# Load processor + model
model_name = "microsoft/trocr-base-printed"
model = VisionEncoderDecoderModel.from_pretrained(model_name)
processor = TrOCRProcessor.from_pretrained(model_name)

model.config.decoder_start_token_id = processor.tokenizer.cls_token_id
model.config.pad_token_id = processor.tokenizer.pad_token_id

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

os.makedirs("models", exist_ok=True)

# Dataset
class OCRDataset(Dataset):
    def __init__(self, image_paths, texts, processor):
        self.image_paths = image_paths
        self.texts = texts
        self.processor = processor

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        try:
            image = Image.open(image_path).convert("RGB")
        except Exception as e:
            print(f'Error loading image: {image_path}. Skipping. Error: {e}')
            return self.__getitem__((idx + 1) % len(self.image_paths))

        text = str(self.texts[idx])
        encoding = self.processor(images=image, text=text,
                                  return_tensors="pt",
```

```
                        padding="max_length",
                        truncation=True,
                        max_length=128)
        encoding = {k: v.squeeze() for k, v in encoding.items()}
        return encoding

# Data
csv_path = r"datasets\TotalText\train.csv"
df = pd.read_csv(csv_path)

print(f"Loaded {len(df)} rows")

# Optional: sample smaller subset (for testing)
n_samples = 1000
if len(df) >= n_samples:
    df = df.sample(n=n_samples, random_state=42).reset_index(drop=True)
    print(f"Sampled {n_samples} rows")
else:
    print(f"Using all {len(df)} rows")

# Update paths
df["image_path"] = df["image_path"].apply(lambda p: p.replace("\\", os.sep))


image_paths = df["image_path"].tolist()
labels = df["label"].astype(str).tolist()
assert len(image_paths) == len(labels)

# KFold setup
num_folds = 5
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

best_val_loss = float("inf")
best_model_path = ""

# KFold training
for fold, (train_idx, val_idx) in enumerate(kf.split(df)):
    print(f"\n====== Fold {fold + 1} / {num_folds} ======")
    print(f'Fold:{fold+1}, Train set: {len(train_idx)}, Val set:{len(val_idx)}')

    train_df = df.iloc[train_idx]
    val_df = df.iloc[val_idx]
    train_dataset = OCRDataset(train_df["image_path"].tolist(),
train_df["label"].astype(str).tolist(), processor)
    val_dataset = OCRDataset(val_df["image_path"].tolist(),
val_df["label"].astype(str).tolist(), processor)

    train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True, pin_memory=True,
drop_last=False)
    val_loader = DataLoader(val_dataset, batch_size=8, pin_memory=True, drop_last=False)
```

```
    model = VisionEncoderDecoderModel.from_pretrained(model_name).to(device)
    model.config.decoder_start_token_id = processor.tokenizer.cls_token_id
    model.config.pad_token_id = processor.tokenizer.pad_token_id

    optimizer = AdamW(model.parameters(), lr=5e-5)
    scaler = GradScaler()

    train_iterator = iter(train_loader)
    max_iterations = 100
    train_loss_total = 0

    progress_bar = trange(max_iterations, desc=f"Fold {fold+1} Training", ncols=100)
    train_preds = []
    train_refs = []

    for iteration in progress_bar:
        model.train()
        try:
            batch = next(train_iterator)
        except StopIteration:
            train_iterator = iter(train_loader)
            batch = next(train_iterator)

        pixel_values = batch["pixel_values"].to(device)
        labels = batch["labels"].to(device)
        labels[labels == processor.tokenizer.pad_token_id] = -100

        with autocast(device_type='cuda'):
            outputs = model(pixel_values=pixel_values, labels=labels)
            loss = outputs.loss

        train_loss_total += loss.item()

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        optimizer.zero_grad()

        # Accumulate train preds for metrics (optional: sample every 5 iters for speed)
        if (iteration + 1) % 5 == 0:
            with torch.no_grad():
                generated_ids_train = model.generate(pixel_values)
                pred_str_train = processor.batch_decode(generated_ids_train,
skip_special_tokens=True)

                decoded_labels_train = labels.clone()
                decoded_labels_train[decoded_labels_train == -100] =
processor.tokenizer.pad_token_id
```

```
            label_str_train = processor.batch_decode(decoded_labels_train,
skip_special_tokens=True)

            train_preds.extend(pred_str_train)
            train_refs.extend(label_str_train)

        # Validation + metrics
        if (iteration + 1) % 100 == 0 or (iteration + 1) == max_iterations:
            model.eval()
            val_loss = 0
            val_preds, val_refs = [], []

            with torch.no_grad():
                for val_batch in tqdm(val_loader, desc=f"Validation @ Fold {fold+1} Iter
{iteration+1}"):
                    pixel_values = val_batch["pixel_values"].to(device)
                    labels = val_batch["labels"].to(device)
                    labels[labels == processor.tokenizer.pad_token_id] = -100

                    outputs = model(pixel_values=pixel_values, labels=labels)
                    val_loss += outputs.loss.item()

                    generated_ids = model.generate(pixel_values)
                    pred_str = processor.batch_decode(generated_ids, skip_special_tokens=True)

                    decoded_labels = labels.clone()
                    decoded_labels[decoded_labels == -100] = processor.tokenizer.pad_token_id
                    label_str = processor.batch_decode(decoded_labels, skip_special_tokens=True)

                    val_preds.extend(pred_str)
                    val_refs.extend(label_str)

            avg_train_loss = train_loss_total / (iteration + 1)
            avg_val_loss = val_loss / len(val_loader)

            train_wer_score = wer(train_refs, train_preds) if train_preds else None
            train_cer_score = cer(train_refs, train_preds) if train_preds else None
            val_wer_score = wer(val_refs, val_preds)
            val_cer_score = cer(val_refs, val_preds)

            print(f"☑ Fold {fold+1} | Iter {iteration+1} | "
                f"Train Loss: {avg_train_loss:.4f} | "
                f"Val Loss: {avg_val_loss:.4f} | "
                f"Train WER: {train_wer_score:.4f} | Train CER: {train_cer_score:.4f} | "
                f"Val WER: {val_wer_score:.4f} | Val CER: {val_cer_score:.4f}")

            # Save best model
            if avg_val_loss < best_val_loss:
                best_val_loss = avg_val_loss
                best_model_path = f"models/best_model_fold{fold+1}.pt"
```

```
            torch.save(model.state_dict(), best_model_path)
            print(f"🌸 Saved new best model at {best_model_path}")

        # Log
        log_file_path = f"totaltext_fold{fold+1}.csv"
        if not os.path.exists(log_file_path):
            with open(log_file_path, "w") as f:
                f.write("iteration,train_loss,val_loss,train_wer,train_cer,val_wer,val_cer,timesta
mp\n")
        with open(log_file_path, "a") as log_file:
            log_file.write(f"{iteration+1},{avg_train_loss:.4f},{avg_val_loss:.4f},"
                           f"{train_wer_score:.4f},{train_cer_score:.4f},"
                           f"{val_wer_score:.4f},{val_cer_score:.4f},{datetime.now()}\n")

        # Clear train preds for next val period
        train_preds, train_refs = [], []


# Final save
final_model = VisionEncoderDecoderModel.from_pretrained(model_name)
final_model.load_state_dict(torch.load(best_model_path))
print(best_model_path)
final_model.save_pretrained("models/best_trained_trocr")
processor.save_pretrained("models/best_trained_trocr")
print("☑ Best model and processor saved to 'models/best_trained_trocr_totaltext'")
```

References

Alharbi, A., & Rahman, M. (16 9, 2021). Review of Recent Technologies for Tackling COVID-19. *SN Computer Science, 2*(6). doi:10.1007/s42979-021-00841-z

Chugani, V. (21 6, 2024). *A Comprehensive Guide to K-Fold Cross Validation*. Retrieved from Datacamp: https://www.datacamp.com/tutorial/k-fold-cross-validation

D2L.ai. (2022). *9.2. Long Short Term Memory (LSTM) — Dive into Deep Learning 0.14.4 documentation*. Retrieved from d2l.ai: https://d2l.ai/chapter_recurrent-modern/lstm.html

Daniella. (2024). *Understanding convolutional neural networks (CNN)*. Retrieved from Innovatiana.com: https://en.innovatiana.com/post/convolutional-neural-network

GeeksforGeeks. (2024). *Deep Learning | Introduction to Long Short Term Memory*. Retrieved from GeeksfoGeeks.

Idris, A., & Taha, D. (2022). Handwritten Text Recognition Using CRNN. doi:10.1109/iccitm56309.2022.10032003

Jain, P., Taneja, K., & Kavita, H. (2021). Which OCR toolset is good and why? A comparative study. *Kuwait Journal of Science, 48*(2). doi:10.48129/kjs.v48i2.9589

Pratim, S. P. (30 8, 2023). Advancements in OCR: A Deep Learning Algorithm for Enhanced Text Recognition. *International journal of inventive engineering and sciences, 10*(8), 1-7.

Seung, J. (2020). OCR RELATED TECHNOLOGY TRENDS. *European Journal of Engineering and Technology, 8*(1). Retrieved from https://www.idpublications.org/wp-content/uploads/2019/12/Full-Paper-OCR-RELATED-TECHNOLOGY-TRENDS.pdf

Wang, J. (15 12, 2023). A Study of The OCR Development History and Directions of Development. *Highlights in Science Engineering and Technology, 72*, 409-415. doi:10.54097/bm665j77