A Project Report On

MEDICAL PRESCRIPTION RECOGNITION

Mini project submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
INFORMATION TECHNOLOGY
(2021-2025)
BY

T. ARCHANA	**21241A12J7**
B.SHIVANI	**21241A12D3**
E. REENA	**21241A12E4**

Under the Esteemed Guidance
of
J. ALEKHYA
Assistant Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

GOKARAJU RANGARAJU INSTITUTE OF ENGINEERING AND TECHNOLOGY

(AUTONOMOUS)

HYDERABAD

2023-24

CERTIFICATE

This is to certify that it is a bonafide record of Mini Project work entitled **"MEDICAL PRESCRIPTION RECOGNITION"** done by **T. ARCHANA (21241A12J7), B. SHIVANI (21241A12D3), E. REENA (21241A12E4)** of **B.Tech** in the Department of Information of Technology, **Gokaraju Rangaraju Institute of Engineering and Technology** during the period 2021-2025 in the partial fulfillment of the requirements for the award of degree of **BACHELOR OF TECHNOLOGY IN INFORMATION TECHNOLOGY** from GRIET, Hyderabad.

J. Alekhya	**Dr. Y J Nagendra Kumar**
Assistant Professor	Head of the Department
(Internal Guide)	

(Project External)

ACKNOWLEDGEMENT

We take the immense pleasure in expressing gratitude to our Internal guide**, J. Alekhya, Assistant Professor, Dept of IT**, GRIET. We express our sincere thanks for her encouragement, suggestions and support, which provided the impetus and paved the way for the successful completion of the project work.

We wish to express our gratitude to **Dr. Y J Nagendra Kumar,** HOD IT,our Project Coordinators **Dr. L. Sukanya** and **A. Pavithra** for their constant support during the project.

We express our sincere thanks to **Dr. Jandhyala N Murthy,** Director, GRIET**,** and **Dr. J. Praveen,** Principal, GRIET**,** for providing us the conductive environment for carrying through our academic schedules and project with ease.

We also take this opportunity to convey our sincere thanks to the teaching and non-teaching staff of GRIET College, Hyderabad.

Email: archanathota2003@gmail.com

Contact No. 6301506022

Email: bandishivaniit2021@gmail.com

Contact No. 8341495388

Email: reenaerukulla19@gmail.com

Contact No. 9014083245

DECLARATION

This is to certify that the mini-project entitled "**MEDICAL PRESCRIPTION RECOGNITION"**is a bonafide work done by us in partial fulfillment of the requirements for the award of the degree **BACHELOR OF TECHNOLOGY IN INFORMATION TECHNOLOGY** from Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad.

We also declare that this project is a result of our own effort and has not been copied or imitated from any source. Citations from any websites, books and paper publications are mentioned in the Bibliography.

This work was not submitted earlier at any other University or Institute for the award of any degree.

T. ARCHANA	**21241A12J7**
B.SHIVANI	**21241A12D3**
E. REENA	**21241A12E4**

TABLE OF CONTENTS

11 **LIST OF DIAGRAMS**

ABSTRACT

The issue of illegible handwritten prescriptions by doctors is a significant challenge in healthcare, often leading to misinterpretation of medication names and dosage instructions. To address this, our project introduces an innovative solution that leverages machine learning technology, particularly Convolutional Neural Networks (CNN) and Optical Character Recognition (OCR). The proposed system aims to seamlessly convert handwritten prescription details into easily readable digital text. Through a comprehensive dataset-driven approach, the system is trained to recognize various handwriting styles and accurately identify medication names and dosages.

The core of the system involves several key phases: pre-processing techniques such as image subtraction, noise reduction, and image resizing, followed by robust CNN classification for feature extraction. In the post-processing phase, OCR is employed to decode the recognized handwriting into digital text. This meticulous approach ensures that the system effectively handles the diverse and often distorted handwriting styles encountered in real-world prescriptions. Rigorous testing on a wide range of real-world cases demonstrates the system's reliability and effectiveness in improving prescription legibility.

Our research not only aims to enhance the clarity and accuracy of handwritten prescriptions but also seeks to instill confidence in pharmacists and patients by reducing uncertainties associated with misinterpretation. The integration of advanced machine learning techniques underscores the potential for a transformative impact on healthcare by minimizing the risks associated with unreadable prescriptions. This project promises to contribute significantly to improving the overall healthcare experience by ensuring precise communication of medication information.

Keywords: Medical Prescription Recognition, CNN, LSTM, RNN

Domain: Machine learning, Image Processing

1. INTRODUCTION

1.1 Introduction to Project

In today's rapidly advancing healthcare landscape, technological innovations are essential for improving patient care, increasing efficiency, and reducing human error. A significant area where technology can make a profound impact is medical prescription handling. The proposed Medical Prescription Optical Character Recognition (OCR) system represents a groundbreaking advancement in this field, addressing critical issues and transforming the management of medical prescriptions.

Medical prescriptions are vital to healthcare, providing instructions for administering medications and outlining treatment plans. However, the current manual process of handling and interpreting prescriptions is prone to errors and inefficiencies. Misreading a prescription due to illegible handwriting or other factors can have severe consequences for patients. These challenges highlight the urgent need for a robust and reliable solution to enhance the accuracy and efficiency of prescription management.

Leveraging state-of-the-art technologies such as YOLOv5 for object detection and deep learning models for text recognition, the Medical Prescription OCR system offers a comprehensive approach to managing prescriptions. By accurately identifying and locating medicine names on prescriptions, the system minimizes the risk of misinterpretation, ensuring patients receive the correct medications and dosages. Additionally, the system enhances accessibility by making prescription information easily retrievable and interpretable, benefiting both healthcare professionals and patients.

This technology empowers patients by providing them with valuable information about their treatment regimens. Access to detailed data about medications, including their chemical composition, dosage, precautions, and potential side effects, enables patients to make informed decisions about their health. Furthermore, by automating prescription interpretation, the system significantly reduces the administrative burden on healthcare providers, allowing them to focus on more critical aspects of patient care.

In an increasingly data-driven healthcare environment, the Medical Prescription OCR system holds the potential to improve patient safety, enhance the quality of healthcare delivery, and streamline administrative processes. This project not only showcases technical expertise but also represents a meaningful contribution to the broader field of healthcare technology, with far-reaching implications for patient well-being and the efficiency of healthcare services.

CNN:

Convolutional neural networks, or CNNs, are a type of deep learning approach designed primarily for processing and analyzing visual input, including images and videos. It is widely used in several fields, including computer vision, photo identification, and natural language processing. CNNs are inspired by the architecture and functions of the human visual cortex.

CNNs are different from traditional neural networks in that they are able to automatically learn hierarchical representations of visual data. They employ a variety of interconnected layers to achieve this, including convolutional, pooling, and completely linked layers. In order to localize important features like as edges, corners, and textures in small portions of the input data, the convolutional layers apply filters.

CNNs are highly effective at tasks like object recognition, picture classification, and image segmentation because they can capture translation invariance and spatial interdependence. They have revolutionized fields like as autonomous driving, medical imaging, and facial recognition. CNN designs such as AlexNet, VGGNet, and ResNet have demonstrated remarkable performance in a number of computer vision challenges, and their techniques have influenced the development of additional deep learning models.
In general, CNNs are an essential component as, on the whole, they have shown to be a successful technique for extracting meaningful information from visual input.

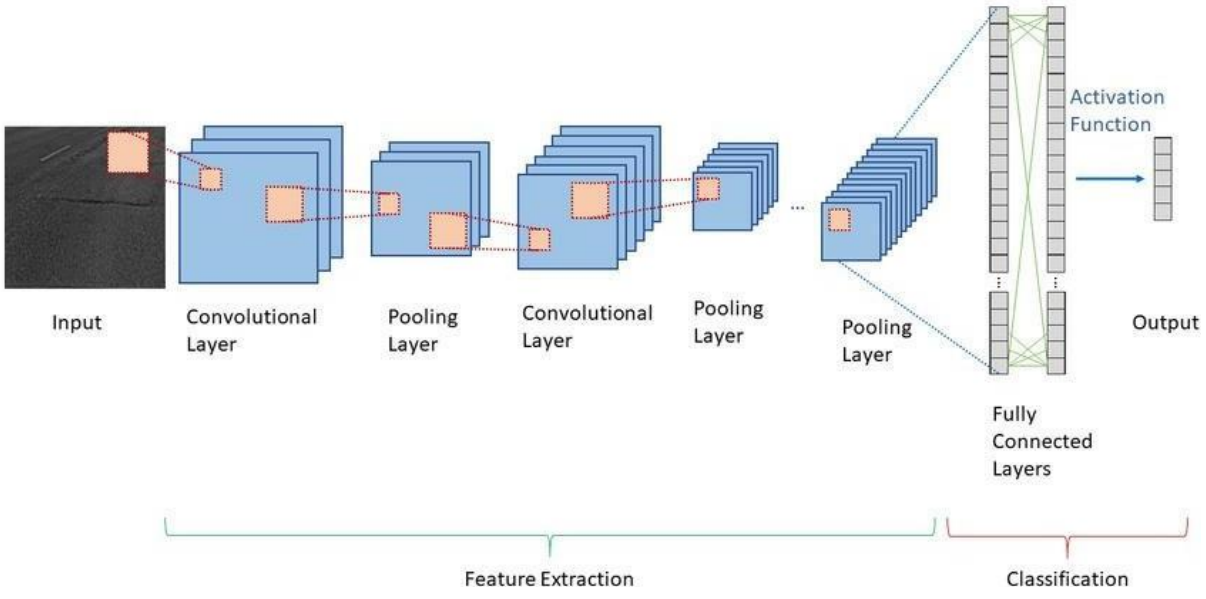


Fig no. 1 CNN

The process of pooling entails summing the features that fall inside the filter's coverage area after swiping a two-dimensional filter over each feature map channel.

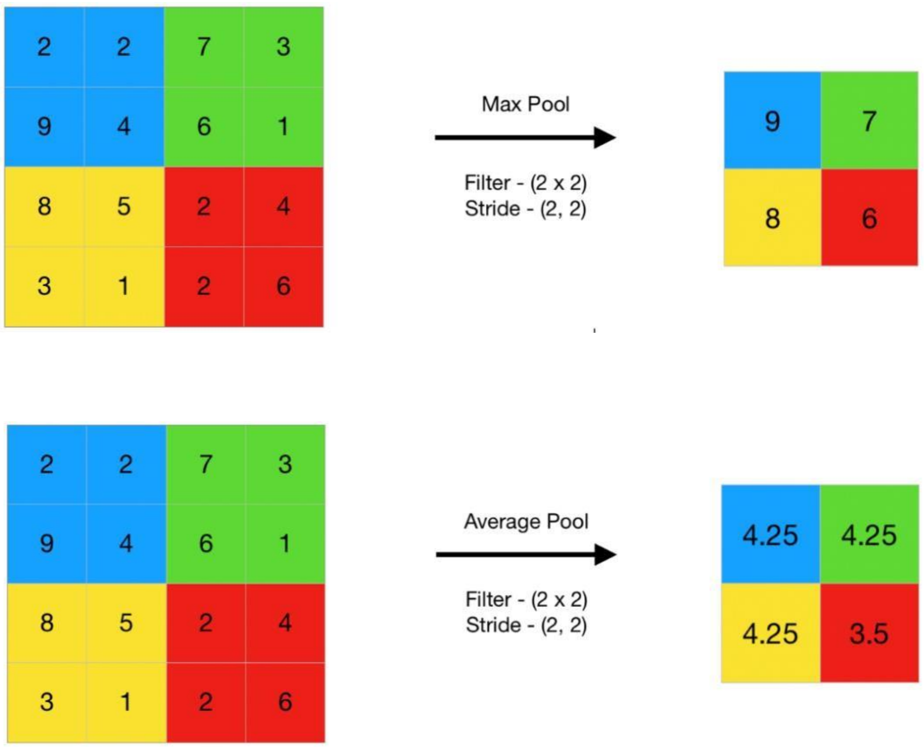
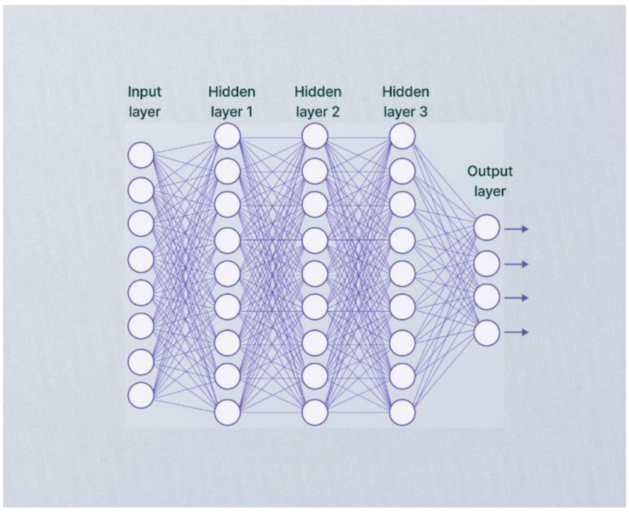


Fig no. 1.1 Types of Pooling

Convolutional Neural Networks (CNNs) consist of an input layer that receives the raw image data, hidden layers which include convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected layers for combining features, and an output layer that produces the final classification or prediction.



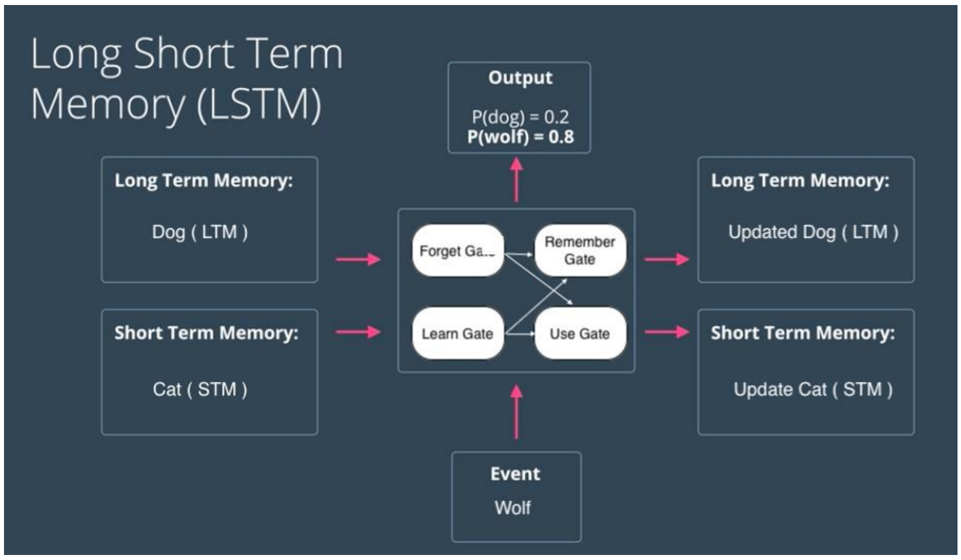
<u>Fig no. 1.2 Types of Layers</u>

LSTM:

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem and effectively capture long-term dependencies in sequential data. Unlike traditional RNNs, LSTM networks include specialized memory cells and gating mechanisms (such as input, output, and forget gates) that regulate the flow of information through the network.

LSTM networks are widely used in various tasks involving sequential data, such as natural language processing (NLP), speech recognition, time series forecasting, and handwriting recognition. Their ability to retain and selectively update information over long sequences makes them particularly effective for modeling complex temporal relationships and handling inputs of varying lengths.

In summary, LSTM networks enhance the capabilities of traditional RNNs by allowing for better long-term memory retention and management of sequential data, making them a crucial tool in modern deep learning applications.



<u>Fig no. 2 LSTM Architeture</u>

<u>1.2 Existing System</u>

Existing systems in medical prescription recognition using machine learning leverage OCR technology and deep learning models like CNNs to digitize and interpret handwritten prescriptions accurately. These systems enhance efficiency in healthcare by speeding up prescription processing, and integrating seamlessly with electronic health records (EHR) systems.

<u>1.3 Proposed System:</u>

The proposed system for medical prescription recognition employs LSTM and RNN architectures to better handle sequential dependencies and contextual information in handwritten text. This approach improves the accuracy of interpreting complex and varied handwriting styles, ensuring more reliable digitization and processing of medical prescription.

RNN

Recurrent Neural Networks (RNNs) are specialized neural networks designed to process sequential data by maintaining memory through loops. They excel in tasks where understanding the sequence of inputs is crucial, such as predicting text or analyzing time series data. RNNs are known for their ability to handle inputs of varying lengths and capture dependencies over time, making them valuable in fields like natural language processing and time series forecasting.

How RNN Works:

1. **Input & Hidden State:** The network receives an input from the sequence and combines it with the current hidden state.
2. **Activation Function:** The combined information is passed through an activation function that determines the output of the current step.
3. **Updated Hidden State:** The hidden state is updated based on the current input, previous hidden state, and the activation function's output.
4. **Sequence Processing:** These steps (1-3) are repeated for each element in the sequence, allowing the network to learn long-term dependencies.

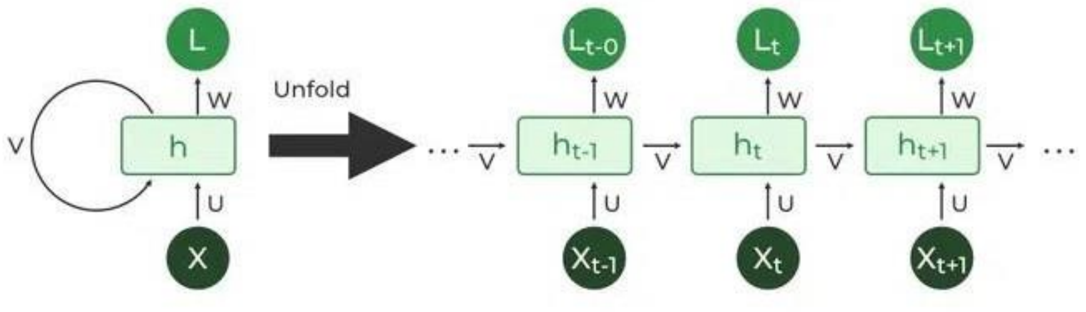


Fig no. 3 RNN Architecture

2. REQUIREMENT ENGINEERING

2.1 Hardware Requirements

- Processor –Intel core i5 and above
- Memory – 8GB RAM (16GB recommended)
- Input devices – Keyboard Mouse
- Internet

2.2 Software Requirements

- Operating System - Any OS capable of running Python and web browsers (Mac, Windows, Linux).
- Front-end - HTML, CSS.
- Libraries - OpenCV, Pillow, Flask.
- Languages - Python 3.x.
- Tools Required - Anaconda, Jupyter Notebook.

3. LITERATURE SURVEY

The proposed model for medical prescription recognition begins by scanning the prescription with a mobile camera and proceeds through three main phases. In the pre-processing phase, the images are normalized, converted to black and white, and divided into three parts: the doctor's name, prescribed medicines and instructions, and the doctor's contact information. In the processing phase, the part containing prescribed medicines is extracted and analyzed using a Convolutional Neural Network (CNN). The CNN employs convolutional, ReLU, and max-pooling layers for feature extraction, followed by fully connected layers for classification. The convolutional layers detect features, ReLU layers introduce nonlinearity, and max-pooling reduces the input size to prevent overfitting. The pooled feature map is flattened for further processing in an Artificial Neural Network (ANN). In the post-processing phase, additional handwritten prescriptions are collected to improve training, and Optical Character Recognition (OCR) is applied if accuracy is below 50%. The OCR results are compared with a dataset of medicine names to identify the nearest match. This comprehensive approach aims to enhance the accuracy and reliability of identifying prescribed medicines and their dosages from medical prescriptions.[1]

The proposed model for medical prescription recognition involves scanning the prescription with a mobile camera and processing it through three main phases. In the pre-processing phase, images are normalized, converted to black and white, and divided into three sections: doctor's name, prescribed medicines and instructions, and doctor's contact information. The processing phase extracts the prescribed medicines and analyzes them using a Convolutional Neural Network (CNN) with convolutional, ReLU, and max-pooling layers for feature extraction, followed by fully connected layers for classification. Convolutional layers detect features, ReLU layers add nonlinearity, and max-pooling reduces input size to prevent overfitting. The pooled feature map is flattened and processed in an Artificial Neural Network (ANN). In the post-processing phase, additional handwritten prescriptions are collected to improve training, and Optical Character Recognition (OCR) is used if accuracy falls below 50%, with results compared to a medicine name dataset to find the closest match. This approach aims to improve the accuracy and reliability of identifying prescribed medicines and dosages from medical prescriptions.[2]

This research aims to develop an automatic verification system using deep learning to ensure prescription dispensing accuracy and reduce medication errors in pharmacies. The system includes two models: an image classification model using raw blister pack images processed with Histograms of Oriented Gradients (HOG) and a Convolutional Neural Network (CNN), and a text classification model that extracts and matches imprints on blister packs using CRAFT, Keras-OCR, and text correction. The dataset comprises 200 types of blister packs, each with 300 high-quality images taken in controlled conditions. The system's accuracy is determined by a majority vote between the models, achieving 95.83% accuracy for image classification and 92% for text classification, with an overall accuracy of 94.23%. [3]

Deciphering a doctor's handwritten prescription is a common challenge for patients and even some pharmacists, often leading to negative consequences due to misinterpretation. This difficulty arises partly because doctors use Latin abbreviations and medical terminology unfamiliar to most people. This paper demonstrates the development of a system that uses Artificial Neural Networks (ANN) to recognize handwritten English medical prescriptions. By employing a Deep Convolution Recurrent Neural Network for training, the supervised system segments input images and processes them to detect and classify characters into 64 predefined categories. The results indicate that the proposed system achieves high recognition rates with an accuracy of 98%.[4]

Doctors often write prescriptions hastily and in illegible handwriting, posing challenges for patients and pharmacists who must interpret them accurately. These prescriptions frequently include abbreviations, cursive writing, and even regional languages, complicating understanding further. This project aims to develop a recognition system that translates physicians' handwritten prescriptions across various languages. The system will function autonomously as an application, processing uploaded prescription images through initial image pre-processing and word segmentation. Deep learning techniques such as CNN, RNN, and LSTM will be employed to train the model for each language, utilizing Unicode for character encoding. The system will integrate fuzzy search and market basket analysis to optimize results from a pharmaceutical database, presenting structured outputs to users for clarity and accuracy in medication dispensation. [5]

4. TECHNOLOGY

4.1 ABOUT PYTHON

Python's environment has evolved significantly, enhancing its capabilities for statistical analysis. It strikes a fine balance between scalability and elegance, placing a premium on efficiency and code readability. Python is renowned for its emphasis on program readability, featuring a straightforward syntax that is beginner-friendly and encourages concise code expression through indentation. Noteworthy aspects of this high-level language include dynamic system functions and automatic memory management.

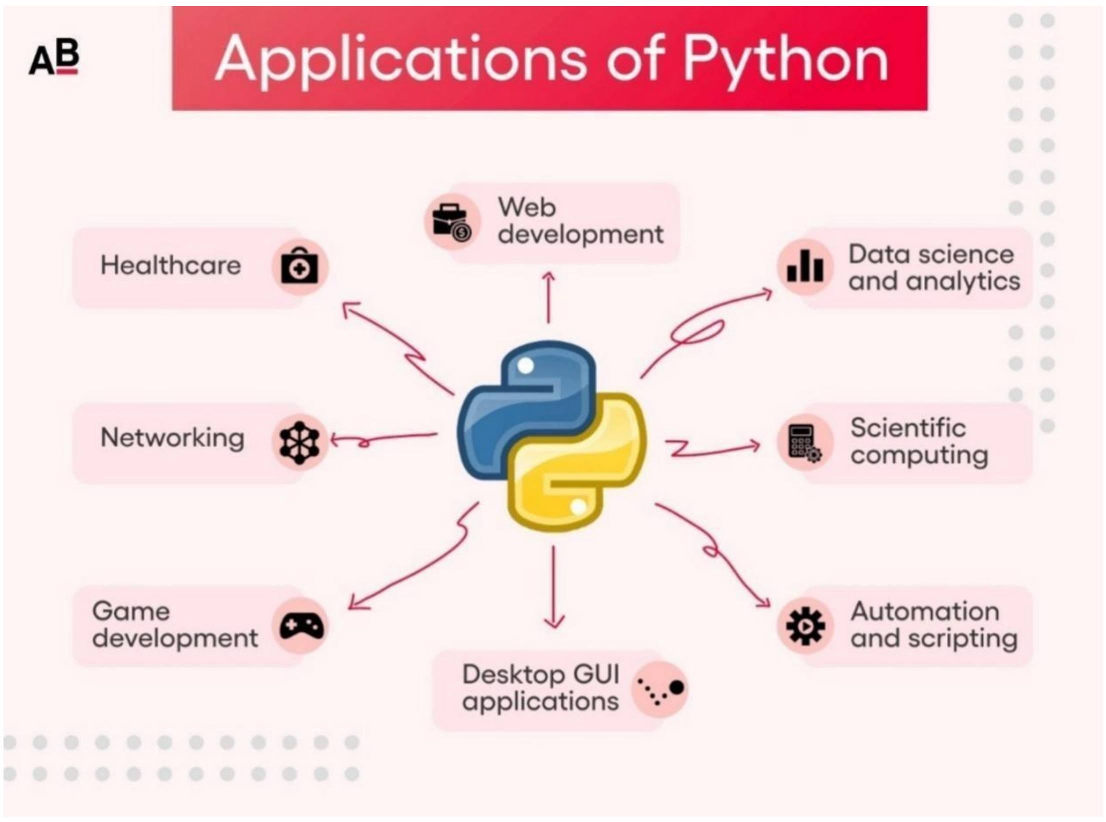


Fig no. 4 Applications of Python

Python is used for:

- Web development
- Data science and machine learning
- Artificial intelligence
- Scientific computing
- Desktop GUI applications
- Automation and scripting
- Game development
- Networking
- Healthcare

4.2 PYTHON IS WIDELY USED IN MACHINE LEARNING

Python is widely favored in machine learning for its flexibility and open-source nature. It provides extensive functionality for mathematical computations and scientific operations, making it indispensable in developing and deploying machine learning models. Python's simple syntax and vast libraries accelerate the development process, reducing coding time significantly. This makes it a preferred choice for machine learning practitioners seeking efficiency and robustness in their projects.

The major Python libraries used in machine learning are as follows:

4.2.1 PANDAS

Pandas is a Python library used for statistical analysis, data cleaning, exploration, and manipulation. Typically, datasets contain both useful and extraneous information. Pandas helps to make this data more readable and relevant.

4.2.2 NUMPY

NumPy is a Python library utilized for numerical data reading, cleaning, exploration, and manipulation. It provides powerful data structures for efficient computation with large arrays and matrices, making the data more accessible and manageable.

4.2.3 MATPLOTLIB

Matplotlib is a Python library for plotting graphs. Built on NumPy arrays, it allows for the creation of a wide range of graph types, from basic plots to bar graphs, histograms, scatter plots, and more.

4.2.4 SCIKIT-LEARN

Scikit-learn is a Python library for machine learning. It provides tools for machine learning and statistical modeling, including classification, regression, and clustering.

4.2.5 TENSORFLOW & PYTORCH

Essential libraries for deep learning, used to create and deploy neural networks. They provide robust tools for developing complex models and facilitating machine learning workflows.

4.2.6 INTERPRETED LANGUAGE

Python executes code line by line, without the need for prior compilation. This approach facilitates quicker development cycles and simplifies the debugging process. As a result, developers can iterate and test their code more efficiently.

4.2.7 CROSS-PLATFORM COMPATIBILITY

Python code runs seamlessly on multiple operating systems, including Windows, macOS, Linux, and Unix-based systems, without requiring modifications. This versatility ensures that Python applications can be deployed across diverse environments with ease.

4.2.8 YOLO v5 MODEL

YOLO v5 is an upgraded version of the YOLO real-time object detection system, known for its speed and accuracy in identifying objects within images and videos.

It is widely used in applications requiring real-time object detection such as autonomous vehicles, surveillance systems, image captioning, and augmented reality.

How YOLO Works:

Unlike traditional object detection methods that analyze image regions multiple times, YOLO takes a single pass through the image.

1. **Image Division:** The input image is divided into a grid of cells.
2. **Bounding Box & Class Prediction:** For each cell, YOLO predicts the probability of an object existing within that cell and its bounding box coordinates (location and size). It also predicts the class of the object (e.g., car, person, dog).

4.2.9 RNN

RNNs are a type of artificial neural network specifically designed to handle sequential data. Unlike traditional neural networks that process each piece of data independently, RNNs can utilize their internal state (memory) to process information based on preceding elements in the sequence.

How RNN Works:

1. **Input & Hidden State:** The network receives an input from the sequence and combines it with the current hidden state.
2. **Activation Function:** The combined information is passed through an activation function that determines the output of the current step.
3. **Updated Hidden State:** The hidden state is updated based on the current input, previous hidden state, and the activation function's output.
4. **Sequence Processing:** These steps (1-3) are repeated for each element in the sequence, allowing the network to learn long-term dependencies.

4.2.10 DATASET DESCRIPTION

The dataset selected for training our model consists of approximately 1 GB of IAM dataset for handwritten text recognition. This dataset includes 165 handwritten prescriptions. The IAM dataset is well-suited for training Recurrent Neural Network (RNN) models to accurately read and interpret text from handwritten prescriptions.

Dataset Details:

1. Size: Approximately 1 GB.
2. Number of prescriptions: 165.

Why IAM Dataset…?

1. The dataset provides ample data to train RNN models effectively.
2. Annotations and labels within the dataset are meticulously organized.
3. The dataset is freely accessible, facilitating cost-effective research and development.

5. DESIGN REQUIREMENT ENGINEERING

Concept of uml:

The purpose of these UML-based diagrams is to visually depict the system, including its key components, roles, operations, objects, or interactions. This visual representation aims to enhance understanding, facilitate manipulation, and effectively document or manage system-related information.

UML DIAGRAMS:

The Unified Modeling Language (UML) serves as a standardized language for creating models across various domains. Its primary goal is to visually represent the structure of a system, akin to blueprints in engineering disciplines. In complex applications involving multiple teams, clear communication is crucial, especially to stakeholders who may not be familiar with programming code. UML facilitates this communication by illustrating essential system requirements, features, and processes in a visual manner. By depicting processes, user interactions, and the system's static structure, UML helps teams streamline collaboration and optimize efficiency.

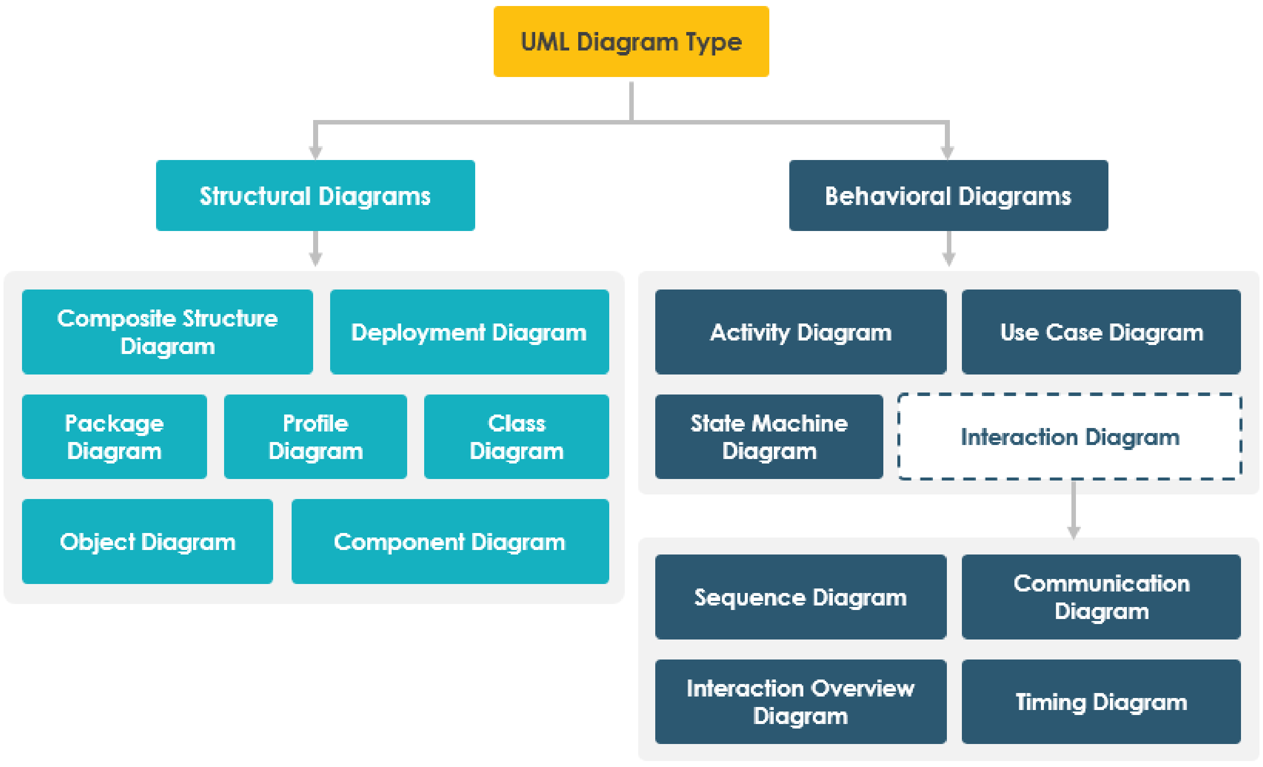


Fig no. 5 Concepts of uml

5.1 Use case Diagram:

The Use Case diagram for the Medical Prescription Recognition system illustrates the interactions between doctors, pharmacists, and the system. It highlights key functionalities such as uploading prescriptions, preprocessing images, recognizing and validating text, storing data, and retrieving and generating reports from the database.

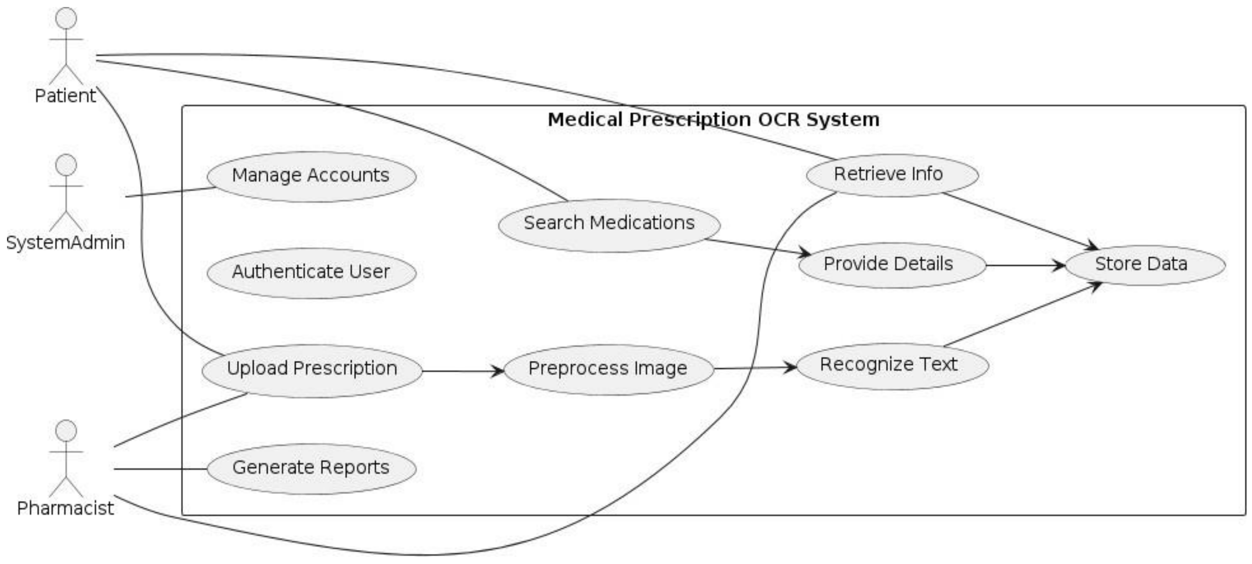


Fig no. 6 UseCase Diagram

5.2 Class Diagram

A class diagram is a static type of structural diagram that visually represents the architecture of a system by illustrating the connections among the system's classes, attributes, operations, and relationships. It provides a blueprint of how different components of the system interact and collaborate, showcasing the structure in a clear and organized manner.

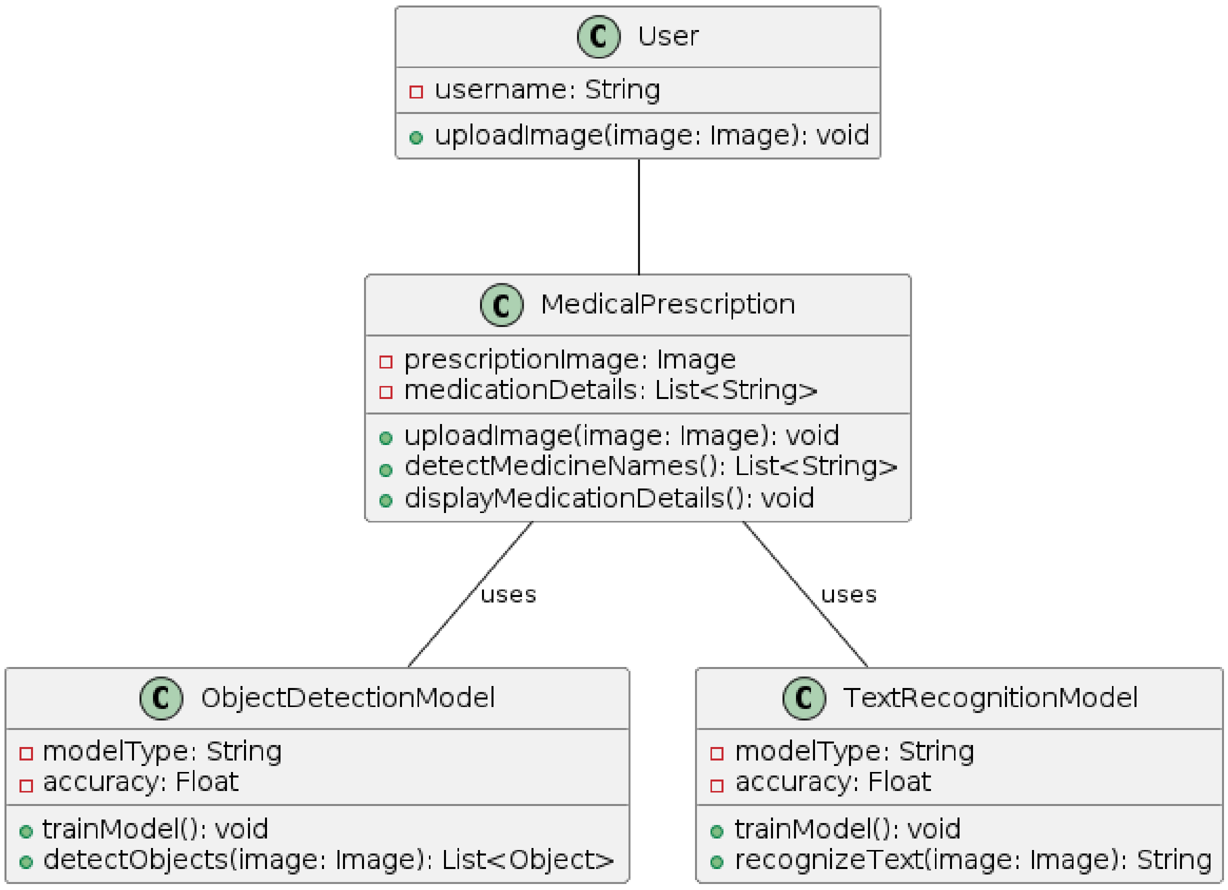

User
username: String
uploadImage(image: Image): void
MedicalPrescription
prescriptionImage: Image
medicationDetails: List<String>
uploadImage(image: Image): void
detectMedicineNames(): List<String>
displayMedicationDetails(): void
uses
uses
ObjectDetectionModel
modelType: String
accuracy: Float
trainModel(): void
detectObjects(image: Image): List<Object>
TextRecognitionModel
modelType: String
accuracy: Float
trainModel(): void
recognizeText(image: Image): String

Fig no. 7 Class Diagram

5.3 Activity diagram:

An activity diagram in UML illustrates the sequence of actions and decisions within a system or process. It shows how activities interact and flow from start to finish, making it easy to understand and analyze complex workflows and business processes.

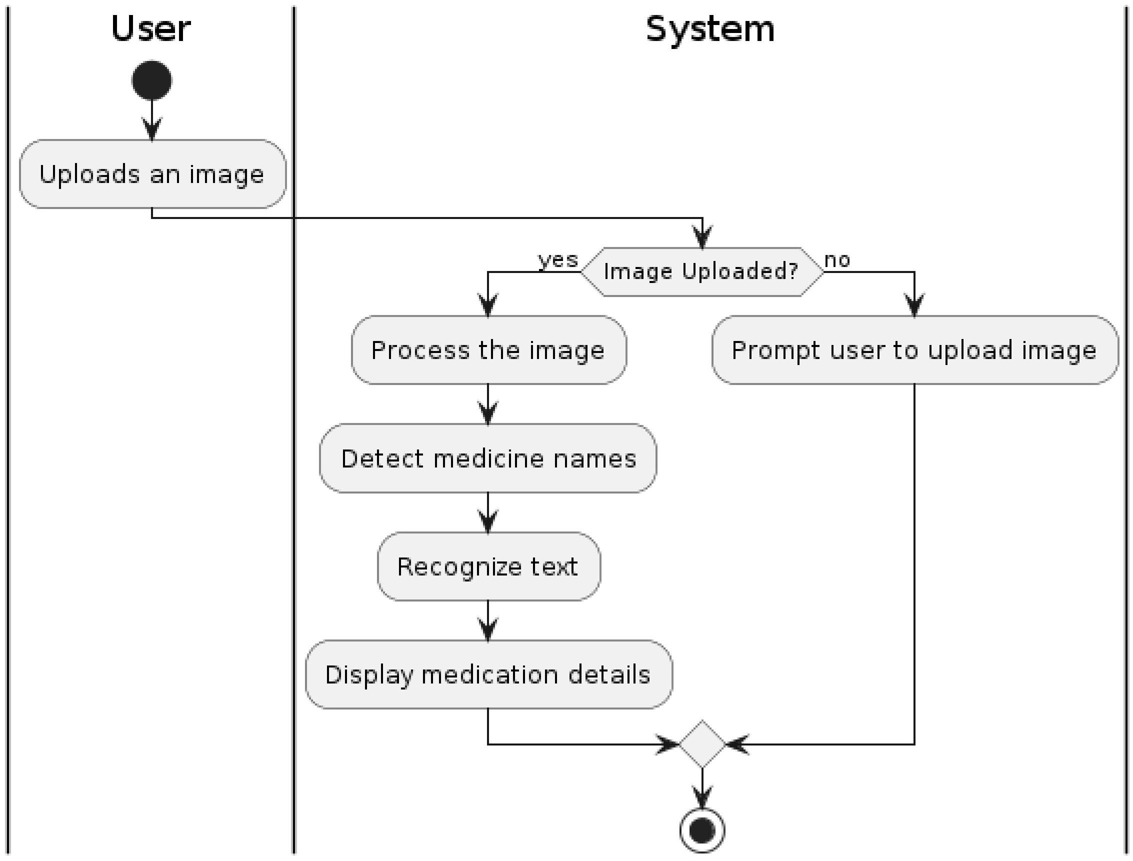
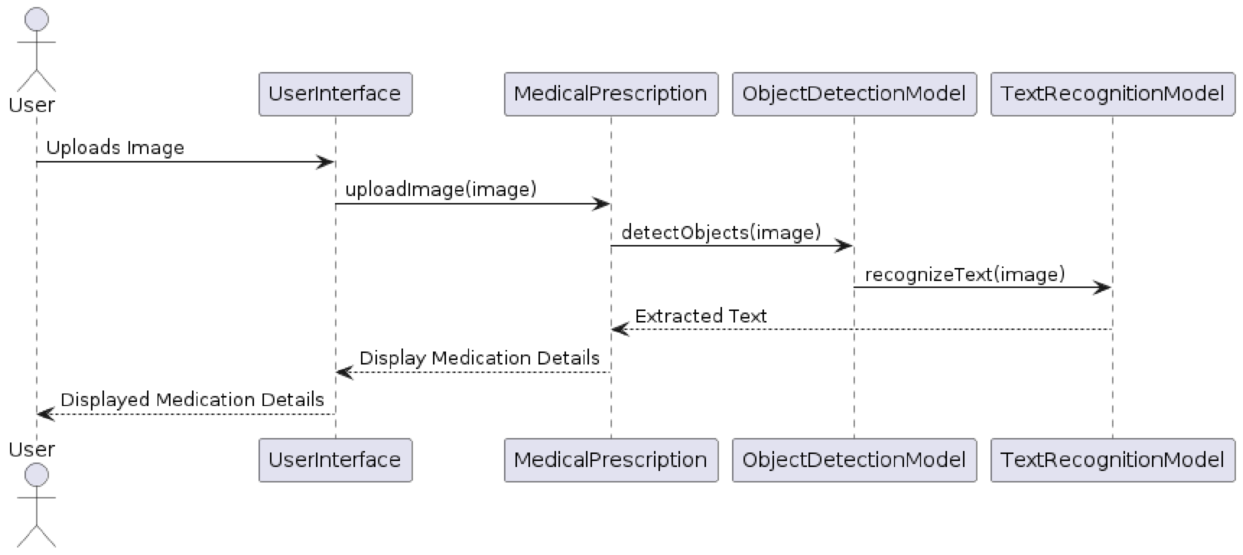


Fig no. 8 Activity Diagram

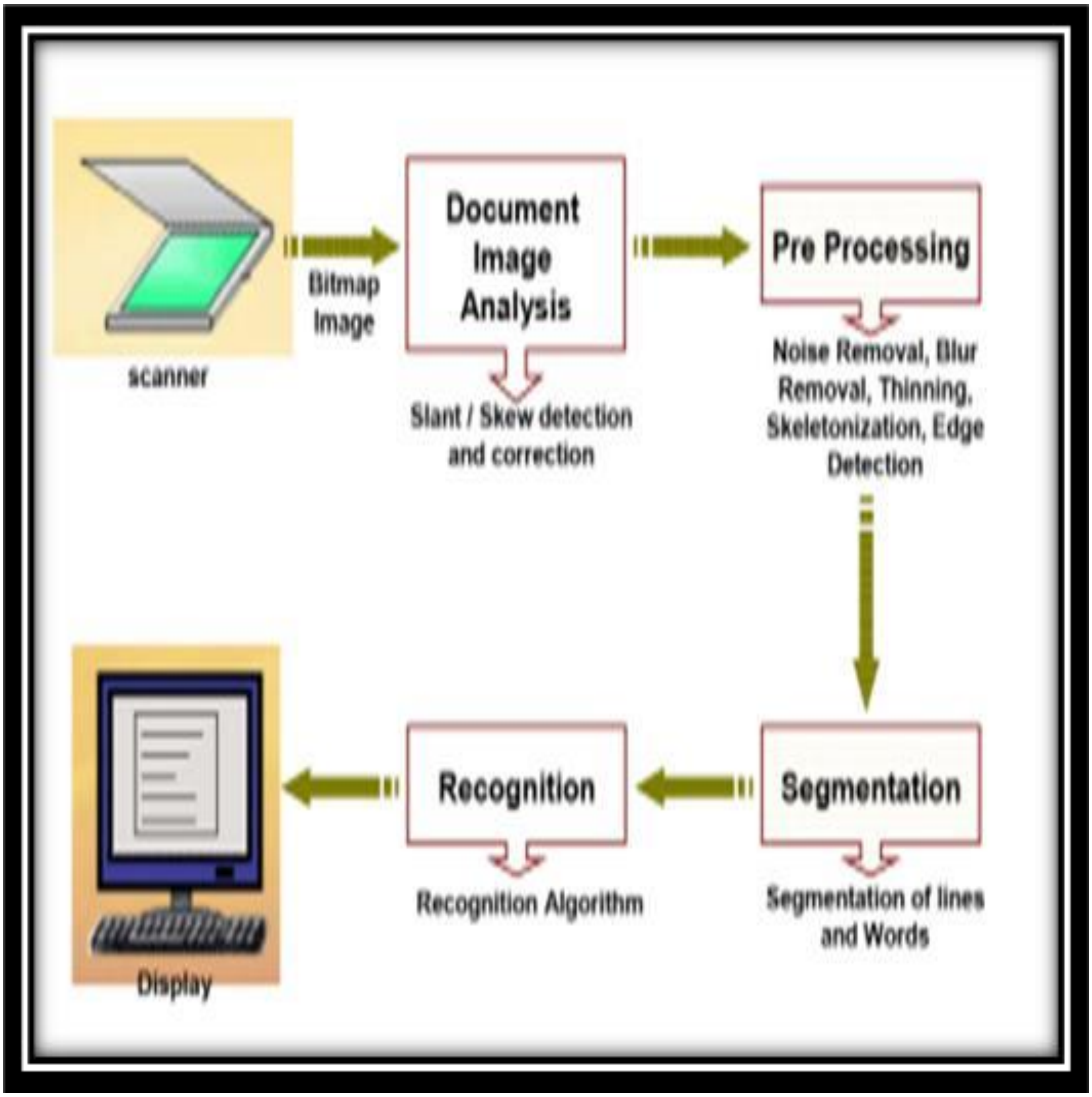
5.4 Sequence Diagram

A sequence diagram illustrates interactions between objects in a sequence over time, showing the messages exchanged between them. It visually represents the flow of control and data between objects in a system, emphasizing the order of events and the lifeline of each participating object.



<u>Fig no. 9 Sequence Diagram</u>

5.5 SYSTEM ARCHITECTURE



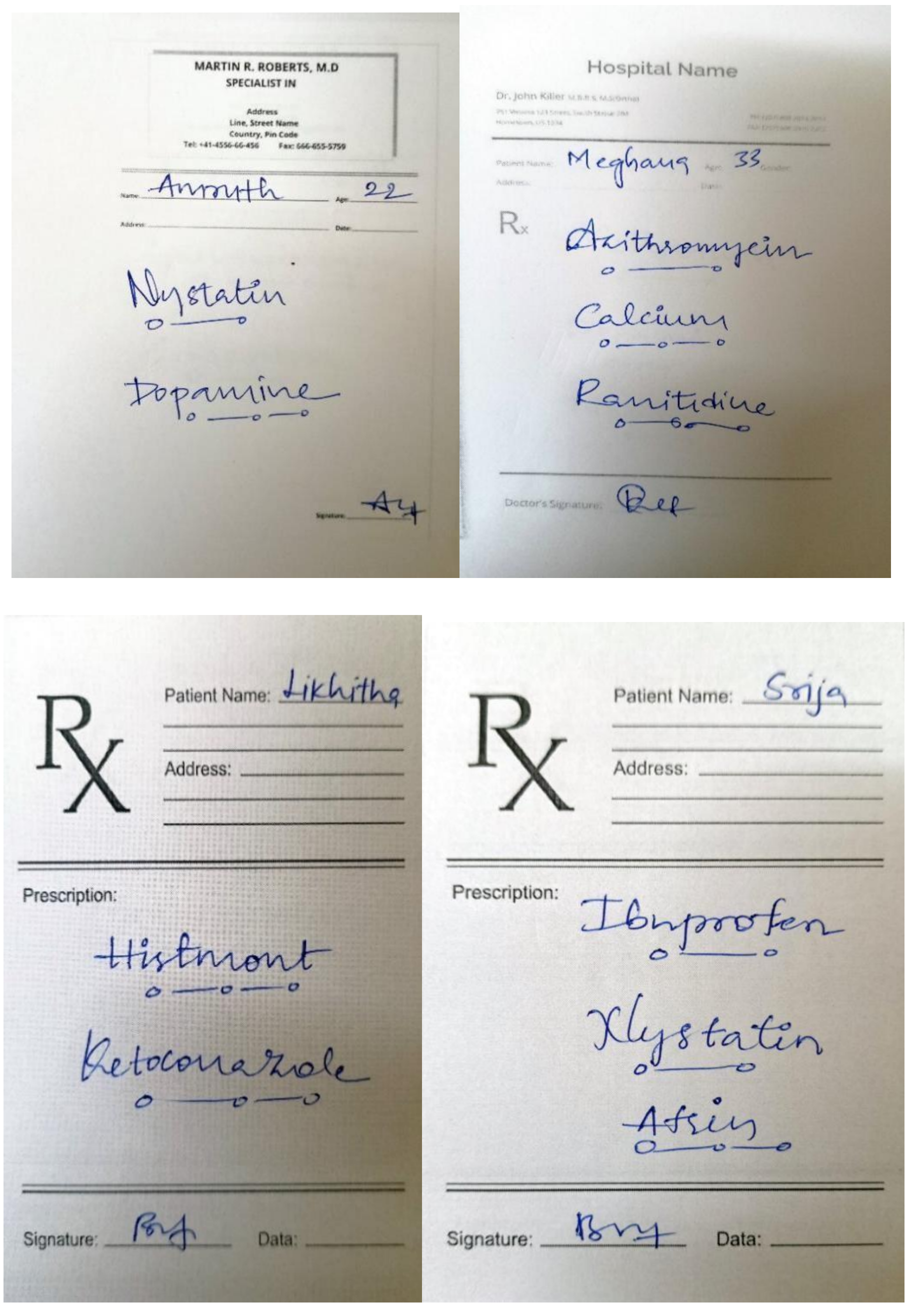
scanner
Bitmap Image
Document Image Analysis
Slant / Skew detection and correction
Pre Processing
Noise Removal, Blur Removal, Thinning, Skeletonization, Edge Detection
Segmentation
Segmentation of lines and Words
Recognition
Recognition Algorithm
Display

Fig no. 10 SystemArchitecture

6. IMPLEMENTATION

Sample Data for the Problem Statement:

Handwritten prescription images are included in the dataset to build an RNN model for text recognition. A few sample images from the dataset are shown below.



MARTIN R. ROBERTS, M.D
SPECIALIST IN

Address
Line, Street Name
Country, Pin Code
Tel: +41-4556-66-456 Fax: 666-655-5759

Name: Amruth Age: 22
Address: Date:

Nystatin

Dopamine

Signature: [illegible]

Hospital Name

Dr. John Killer [illegible]

Patient Name: Meghana Age: 33 Gender:
Address: Date:

Rx Azithromycin

Calcium

Ranitidine

Doctor's Signature: [illegible]

Rx Patient Name: Likhitha
Address:

Prescription:

Histmont

Ketoconazole

Signature: [illegible] Data:

Rx Patient Name: Srija
Address:

Prescription:

Ibuprofen

Nystatin

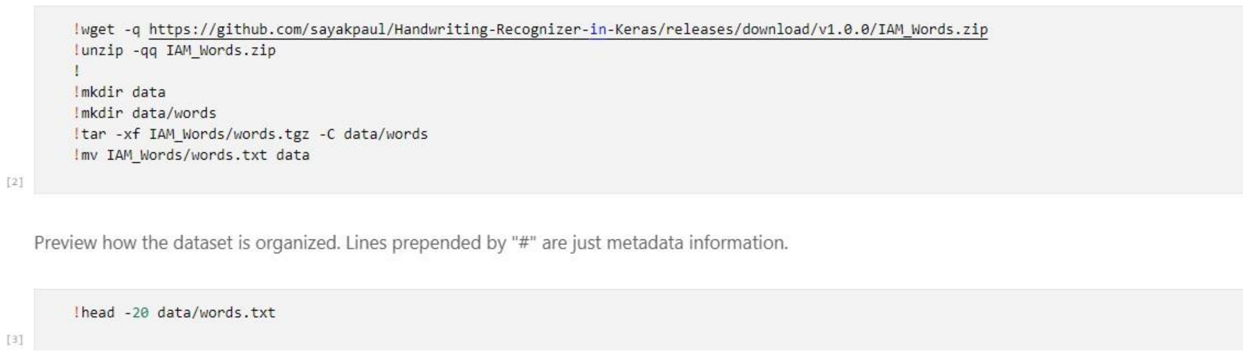
Afsin

Signature: [illegible] Data:

Fig no. 12 Sample data fron Dataset

Model Construction

Data Collection



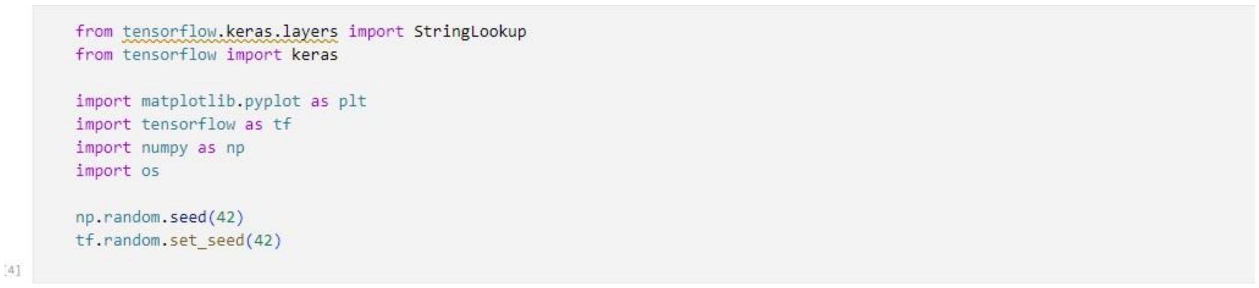
```
!wget -q https://github.com/sayakpaul/Handwriting-Recognizer-in-Keras/releases/download/v1.0.0/IAM_Words.zip
!unzip -qq IAM_Words.zip
!
!mkdir data
!mkdir data/words
!tar -xf IAM_Words/words.tgz -C data/words
!mv IAM_Words/words.txt data
```
[2]

Preview how the dataset is organized. Lines prepended by "#" are just metadata information.

```
!head -20 data/words.txt
```
[3]

Fig no. 13 Data Collection

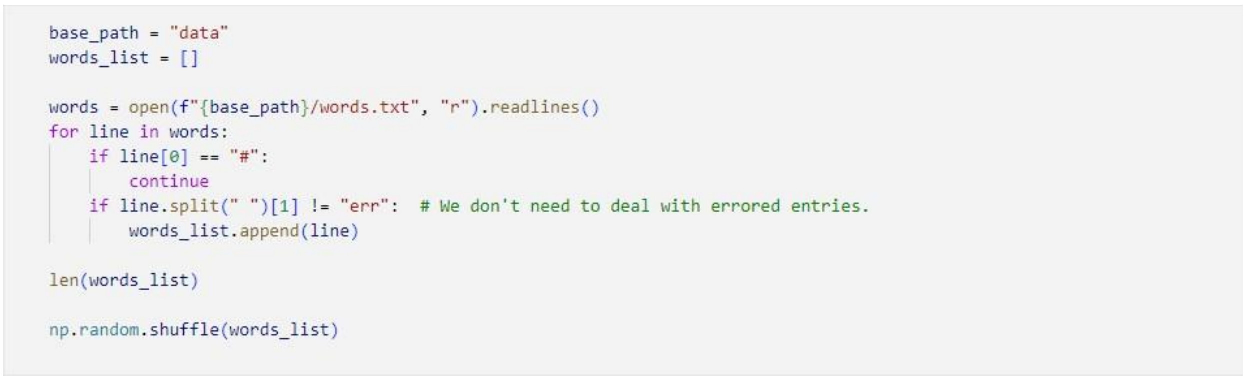
Importing Modules and Packages



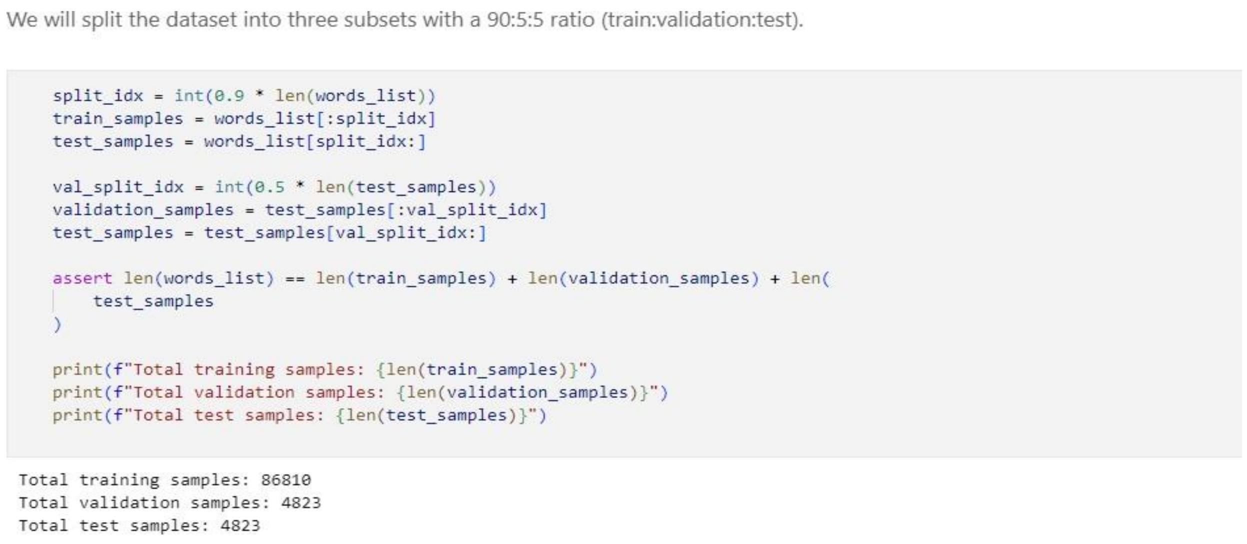
```
from tensorflow.keras.layers import StringLookup
from tensorflow import keras

import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import os

np.random.seed(42)
tf.random.set_seed(42)
```
[4]

Fig no. 14 Importing Packages

Dataset Splitting



```
base_path = "data"
words_list = []

words = open(f"{base_path}/words.txt", "r").readlines()
for line in words:
    if line[0] == "#":
        continue
    if line.split(" ")[1] != "err":  # We don't need to deal with errored entries.
        words_list.append(line)

len(words_list)

np.random.shuffle(words_list)
```



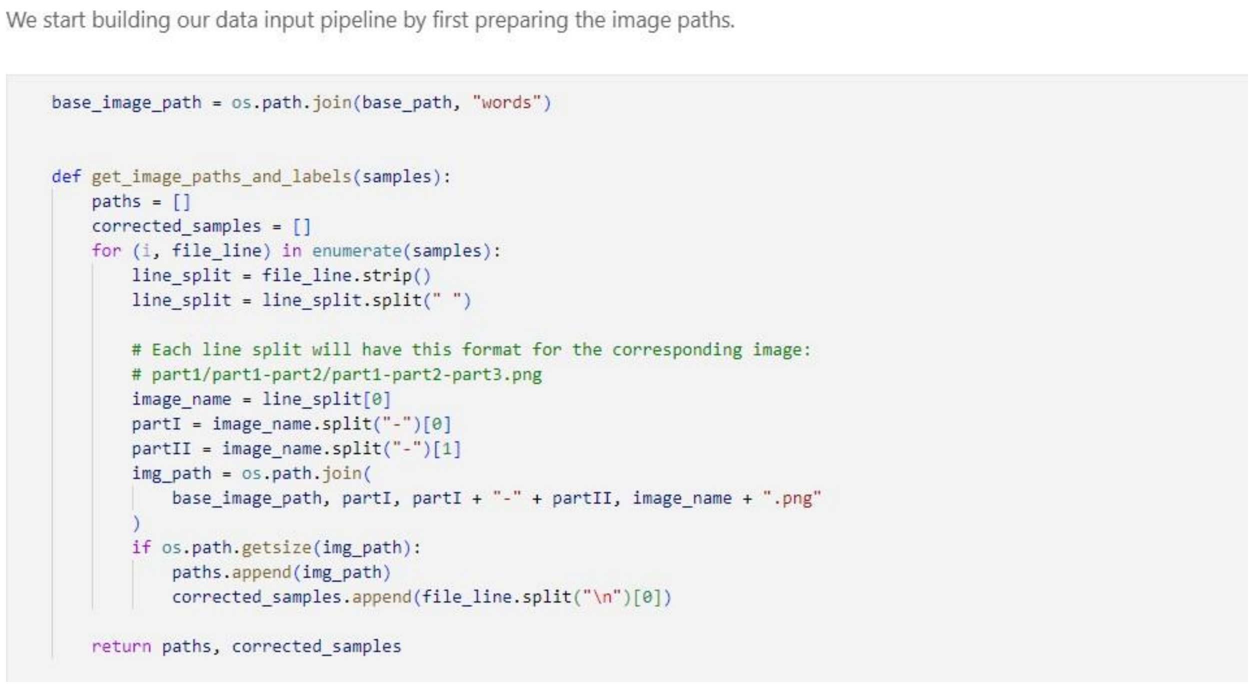
We will split the dataset into three subsets with a 90:5:5 ratio (train:validation:test).

```
split_idx = int(0.9 * len(words_list))
train_samples = words_list[:split_idx]
test_samples = words_list[split_idx:]

val_split_idx = int(0.5 * len(test_samples))
validation_samples = test_samples[:val_split_idx]
test_samples = test_samples[val_split_idx:]

assert len(words_list) == len(train_samples) + len(validation_samples) + len(
    test_samples
)

print(f"Total training samples: {len(train_samples)}")
print(f"Total validation samples: {len(validation_samples)}")
print(f"Total test samples: {len(test_samples)}")
```

```
Total training samples: 86810
Total validation samples: 4823
Total test samples: 4823
```

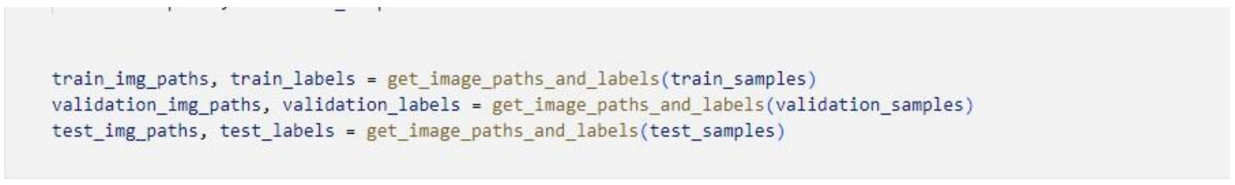
Fig no. 15 Splitting Dataset

Data Input Pipeline



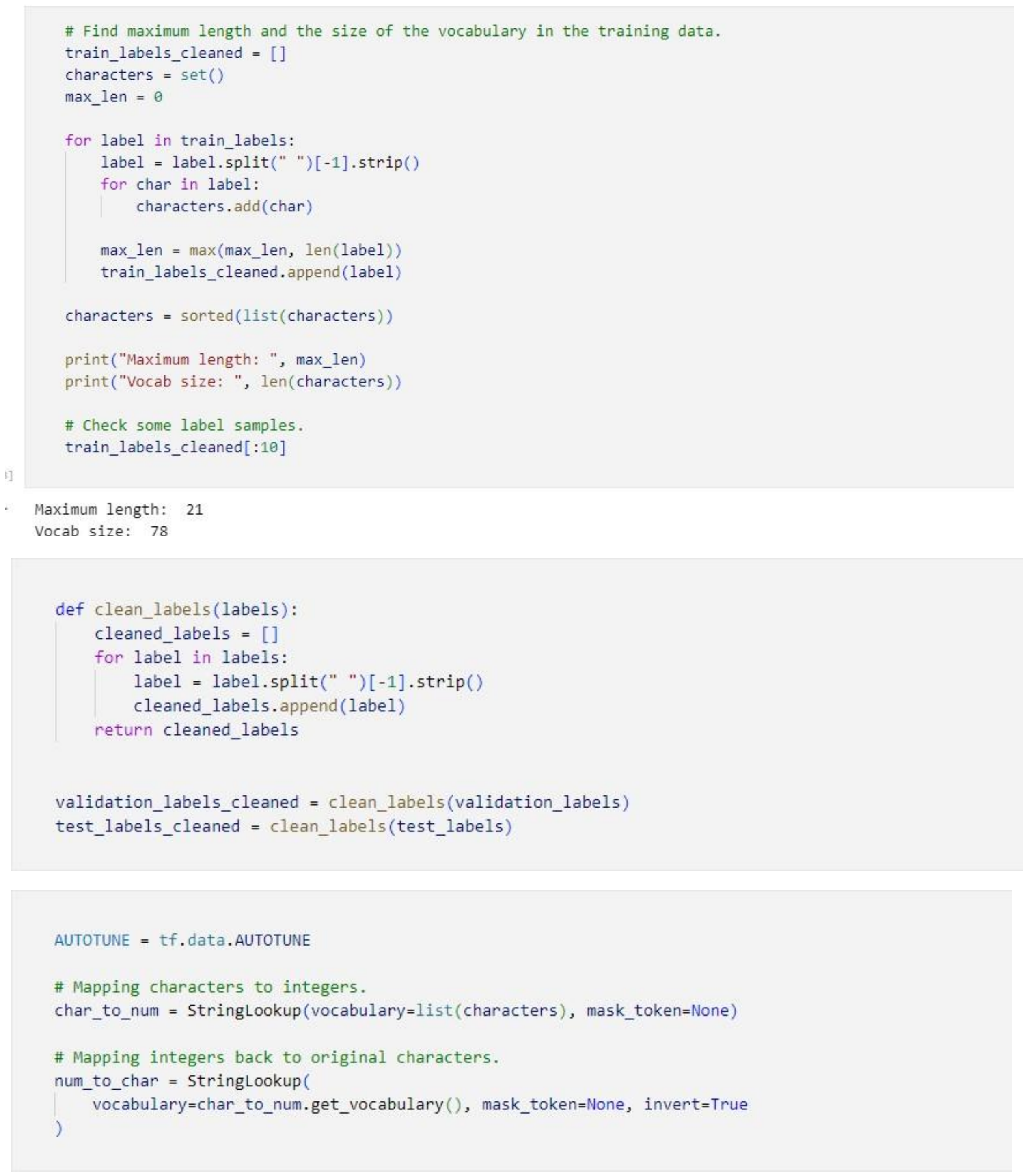
We start building our data input pipeline by first preparing the image paths.

```
base_image_path = os.path.join(base_path, "words")


def get_image_paths_and_labels(samples):
    paths = []
    corrected_samples = []
    for (i, file_line) in enumerate(samples):
        line_split = file_line.strip()
        line_split = line_split.split(" ")

        # Each line split will have this format for the corresponding image:
        # part1/part1-part2/part1-part2-part3.png
        image_name = line_split[0]
        partI = image_name.split("-")[0]
        partII = image_name.split("-")[1]
        img_path = os.path.join(
            base_image_path, partI, partI + "-" + partII, image_name + ".png"
        )
        if os.path.getsize(img_path):
            paths.append(img_path)
            corrected_samples.append(file_line.split("\n")[0])

    return paths, corrected_samples
```



```
train_img_paths, train_labels = get_image_paths_and_labels(train_samples)
validation_img_paths, validation_labels = get_image_paths_and_labels(validation_samples)
test_img_paths, test_labels = get_image_paths_and_labels(test_samples)
```

Then we prepare the ground-truth labels.



```python
# Find maximum length and the size of the vocabulary in the training data.
train_labels_cleaned = []
characters = set()
max_len = 0

for label in train_labels:
    label = label.split(" ")[-1].strip()
    for char in label:
        characters.add(char)

    max_len = max(max_len, len(label))
    train_labels_cleaned.append(label)

characters = sorted(list(characters))

print("Maximum length: ", max_len)
print("Vocab size: ", len(characters))

# Check some label samples.
train_labels_cleaned[:10]
```

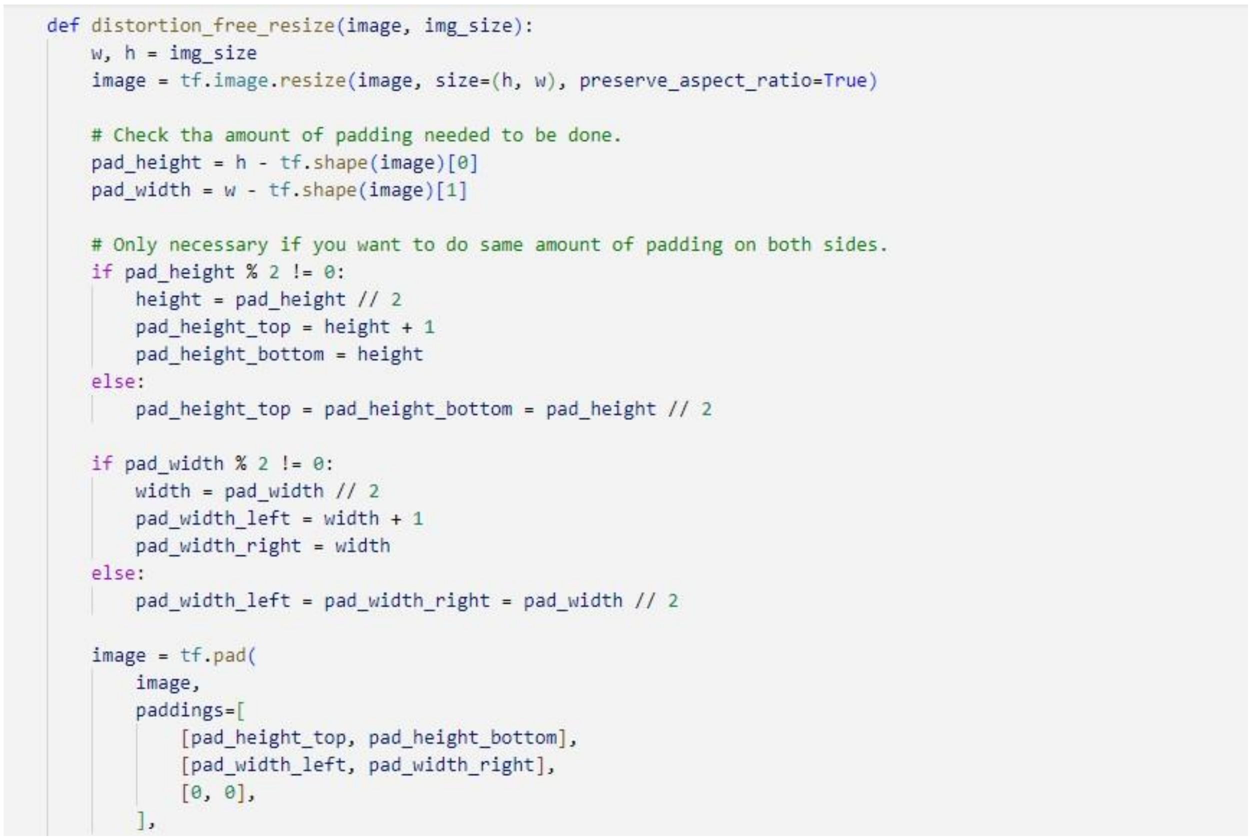
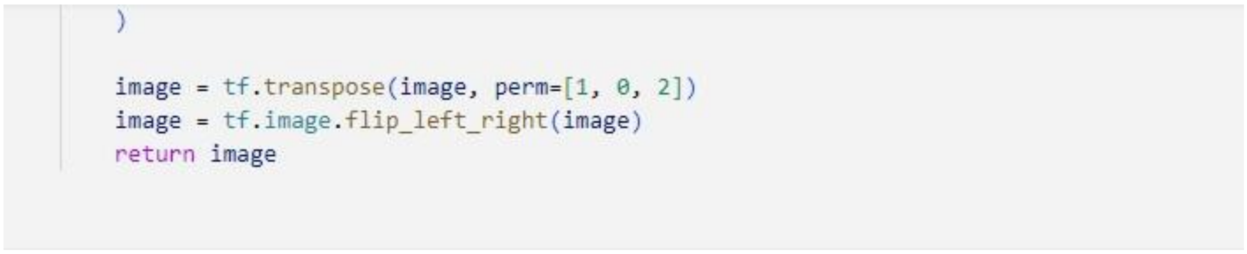
```
Maximum length:  21
Vocab size:  78
```

```python
def clean_labels(labels):
    cleaned_labels = []
    for label in labels:
        label = label.split(" ")[-1].strip()
        cleaned_labels.append(label)
    return cleaned_labels


validation_labels_cleaned = clean_labels(validation_labels)
test_labels_cleaned = clean_labels(test_labels)
```

```python
AUTOTUNE = tf.data.AUTOTUNE

# Mapping characters to integers.
char_to_num = StringLookup(vocabulary=list(characters), mask_token=None)

# Mapping integers back to original characters.
num_to_char = StringLookup(
    vocabulary=char_to_num.get_vocabulary(), mask_token=None, invert=True
)
```

<u>Fig no. 16 Data Input Pipeline</u>

Resizing Images without Disortion

```python
def distortion_free_resize(image, img_size):
    w, h = img_size
    image = tf.image.resize(image, size=(h, w), preserve_aspect_ratio=True)

    # Check tha amount of padding needed to be done.
    pad_height = h - tf.shape(image)[0]
    pad_width = w - tf.shape(image)[1]

    # Only necessary if you want to do same amount of padding on both sides.
    if pad_height % 2 != 0:
        height = pad_height // 2
        pad_height_top = height + 1
        pad_height_bottom = height
    else:
        pad_height_top = pad_height_bottom = pad_height // 2

    if pad_width % 2 != 0:
        width = pad_width // 2
        pad_width_left = width + 1
        pad_width_right = width
    else:
        pad_width_left = pad_width_right = pad_width // 2

    image = tf.pad(
        image,
        paddings=[
            [pad_height_top, pad_height_bottom],
            [pad_width_left, pad_width_right],
            [0, 0],
        ],
    )

    image = tf.transpose(image, perm=[1, 0, 2])
    image = tf.image.flip_left_right(image)
    return image
```

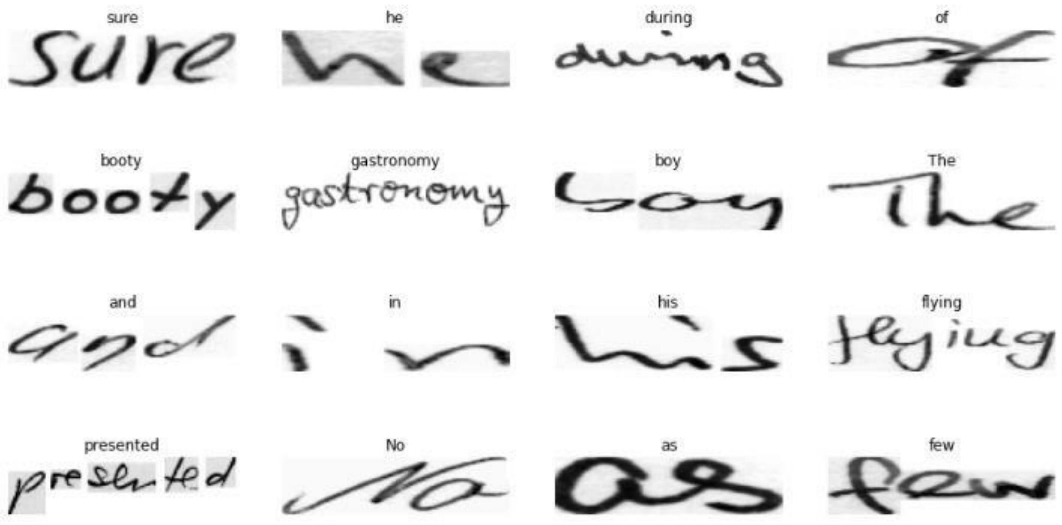


Fig no. 17 Resizing Images

Prepare Dataset Objects

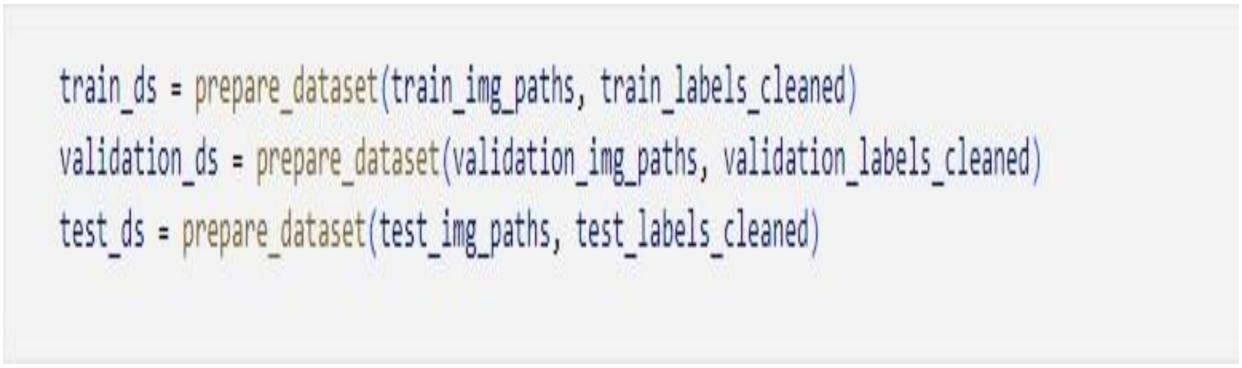
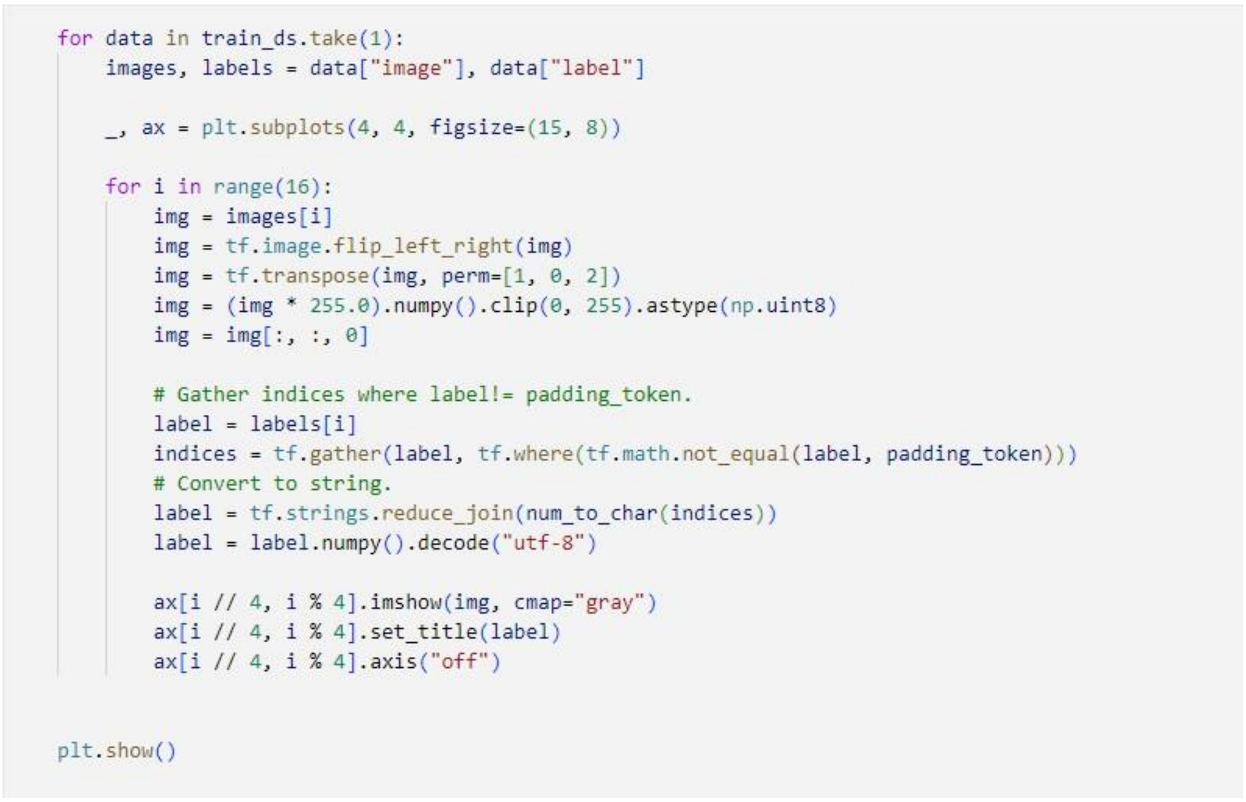

```
train_ds = prepare_dataset(train_img_paths, train_labels_cleaned)
validation_ds = prepare_dataset(validation_img_paths, validation_labels_cleaned)
test_ds = prepare_dataset(test_img_paths, test_labels_cleaned)
```

Fig no. 18 Preparing Dataset Objects

Visualize few Samples


```
for data in train_ds.take(1):
    images, labels = data["image"], data["label"]

    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    for i in range(16):
        img = images[i]
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]

        # Gather indices where label!= padding_token.
        label = labels[i]
        indices = tf.gather(label, tf.where(tf.math.not_equal(label, padding_token)))
        # Convert to string.
        label = tf.strings.reduce_join(num_to_char(indices))
        label = label.numpy().decode("utf-8")

        ax[i // 4, i % 4].imshow(img, cmap="gray")
        ax[i // 4, i % 4].set_title(label)
        ax[i // 4, i % 4].axis("off")


plt.show()
```

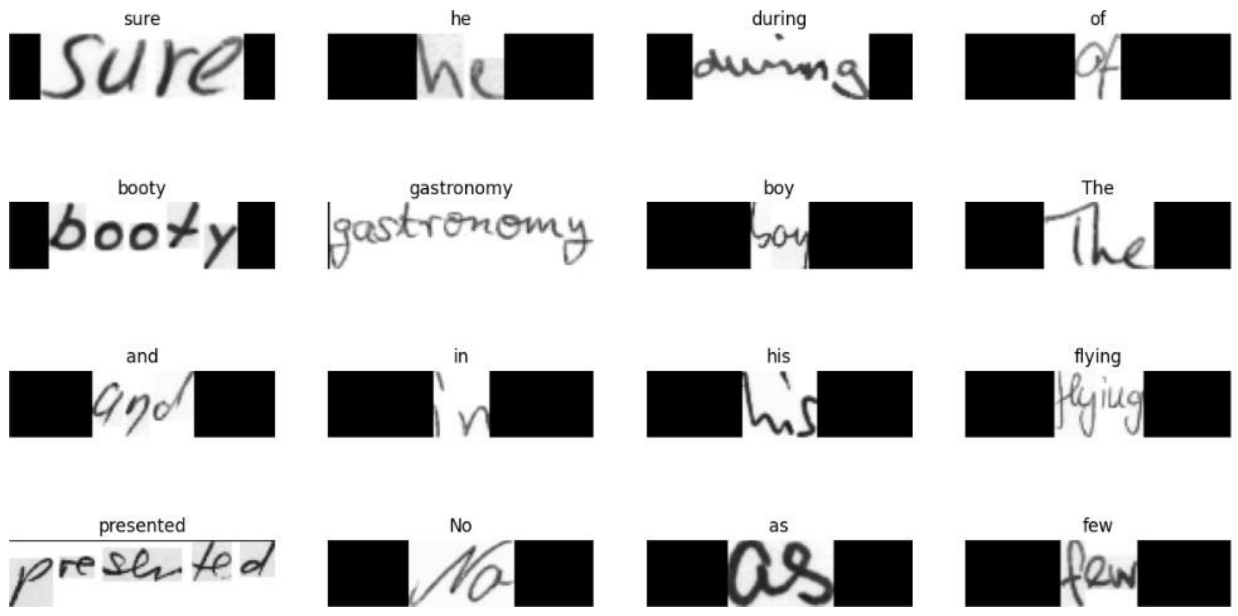
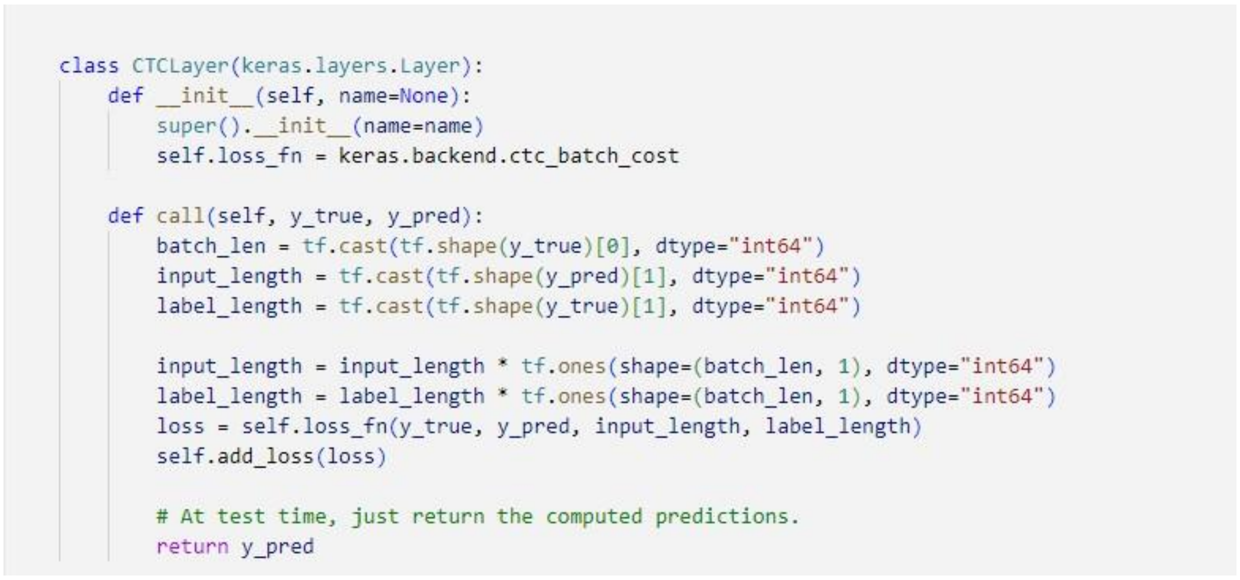


Fig no. 19 Visualizing Samples

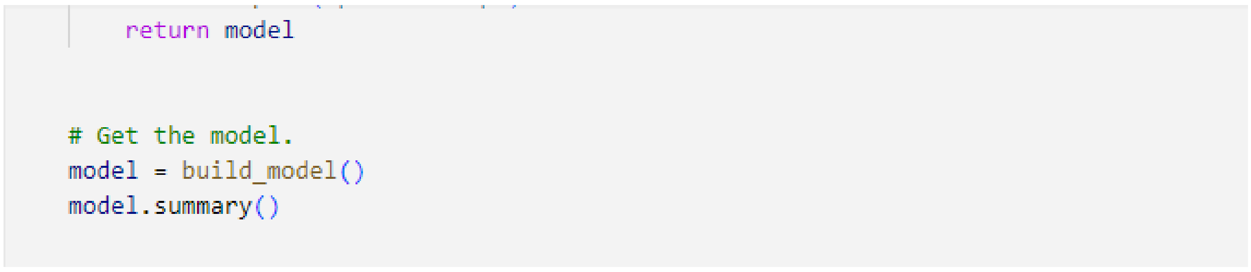
Model



```
class CTCLayer(keras.layers.Layer):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.loss_fn = keras.backend.ctc_batch_cost

    def call(self, y_true, y_pred):
        batch_len = tf.cast(tf.shape(y_true)[0], dtype="int64")
        input_length = tf.cast(tf.shape(y_pred)[1], dtype="int64")
        label_length = tf.cast(tf.shape(y_true)[1], dtype="int64")

        input_length = input_length * tf.ones(shape=(batch_len, 1), dtype="int64")
        label_length = label_length * tf.ones(shape=(batch_len, 1), dtype="int64")
        loss = self.loss_fn(y_true, y_pred, input_length, label_length)
        self.add_loss(loss)

        # At test time, just return the computed predictions.
        return y_pred
```

```python
def build_model():
    # Inputs to the model
    input_img = keras.Input(shape=(image_width, image_height, 1), name="image")
    labels = keras.layers.Input(name="label", shape=(None,))

    # First conv block.
    x = keras.layers.Conv2D(
        32,
        (3, 3),
        activation="relu",
        kernel_initializer="he_normal",
        padding="same",
        name="Conv1",
    )(input_img)
    x = keras.layers.MaxPooling2D((2, 2), name="pool1")(x)

    # Second conv block.
    x = keras.layers.Conv2D(
        64,
        (3, 3),
        activation="relu",
        kernel_initializer="he_normal",
        padding="same",
        name="Conv2",
    )(x)
    x = keras.layers.MaxPooling2D((2, 2), name="pool2")(x)
```

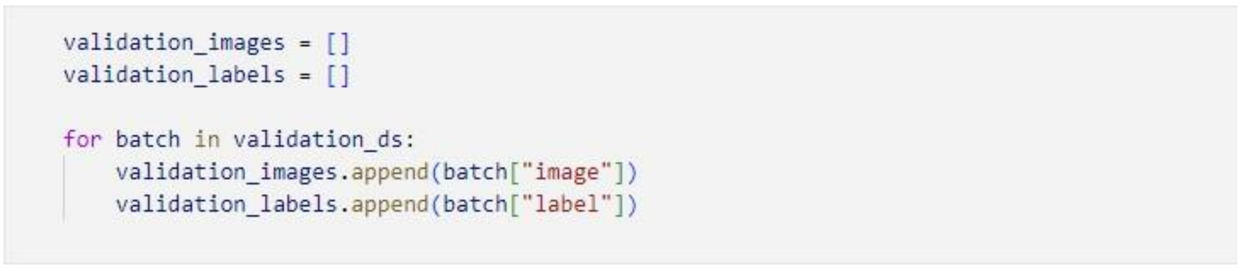
```python
    new_shape = ((image_width // 4), (image_height // 4) * 64)
    x = keras.layers.Reshape(target_shape=new_shape, name="reshape")(x)
    x = keras.layers.Dense(64, activation="relu", name="dense1")(x)
    x = keras.layers.Dropout(0.2)(x)

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

    # +2 is to account for the two special tokens introduced by the CTC loss.
    # The recommendation comes here: https://git.io/J0eXP.
    x = keras.layers.Dense(
        len(char_to_num.get_vocabulary()) + 2, activation="softmax", name="dense2"
    )(x)

    # Add CTC layer for calculating CTC loss at each step.
    output = CTCLayer(name="ctc_loss")(labels, x)

    # Define the model.
    model = keras.models.Model(
        inputs=[input_img, labels], outputs=output, name="handwriting_recognizer"
    )
    # Optimizer.
    opt = keras.optimizers.Adam()
    # Compile the model and return.
    model.compile(optimizer=opt)
```



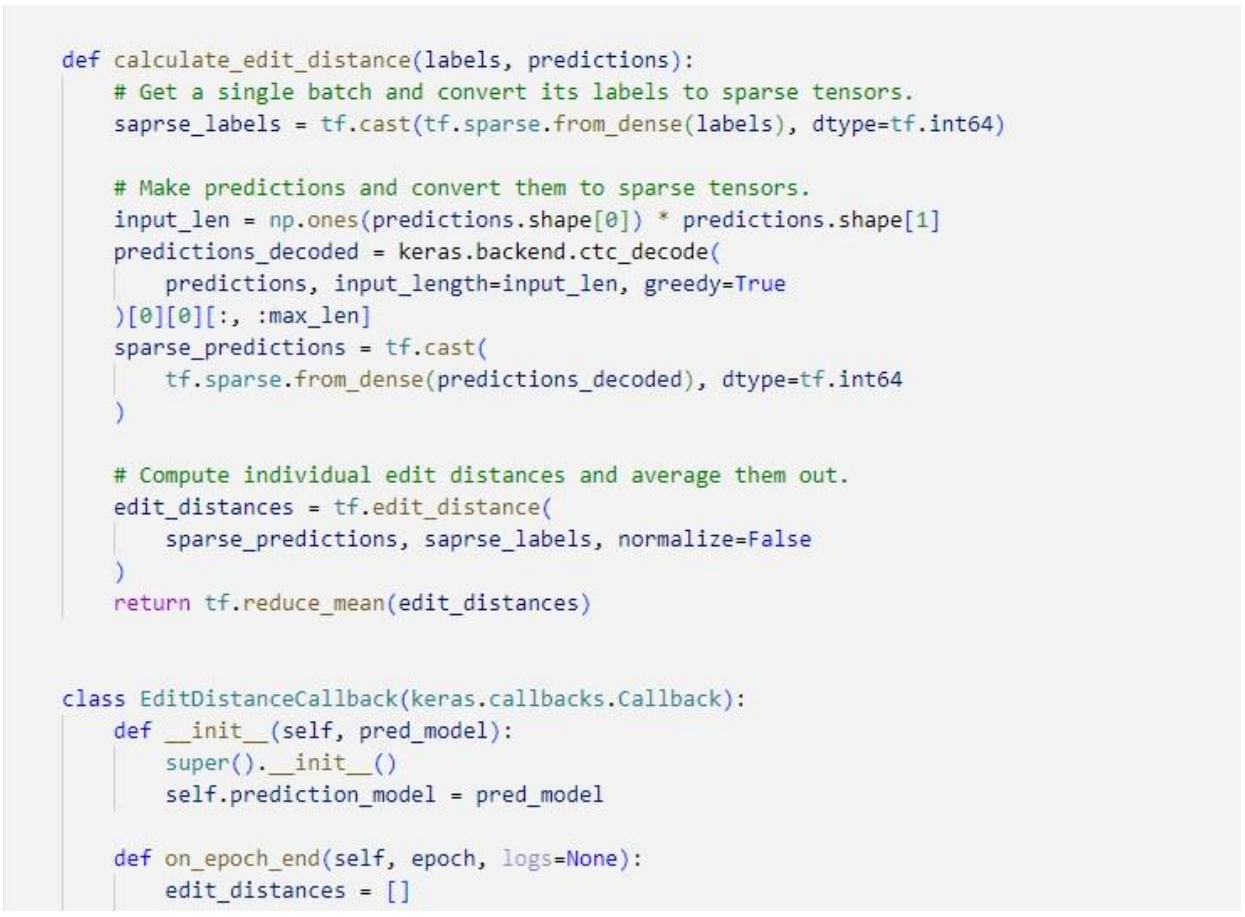
```python
    return model


# Get the model.
model = build_model()
model.summary()
```

Fig no. 20 Model

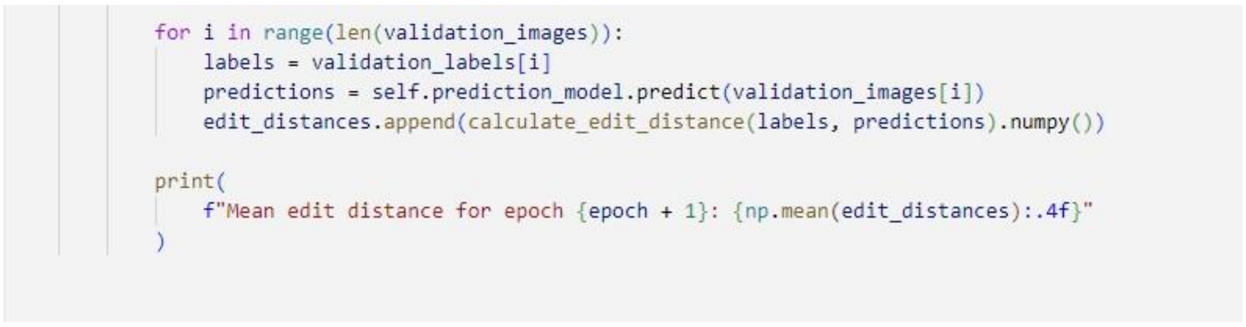
Evaluation Metric



```python
validation_images = []
validation_labels = []

for batch in validation_ds:
    validation_images.append(batch["image"])
    validation_labels.append(batch["label"])
```



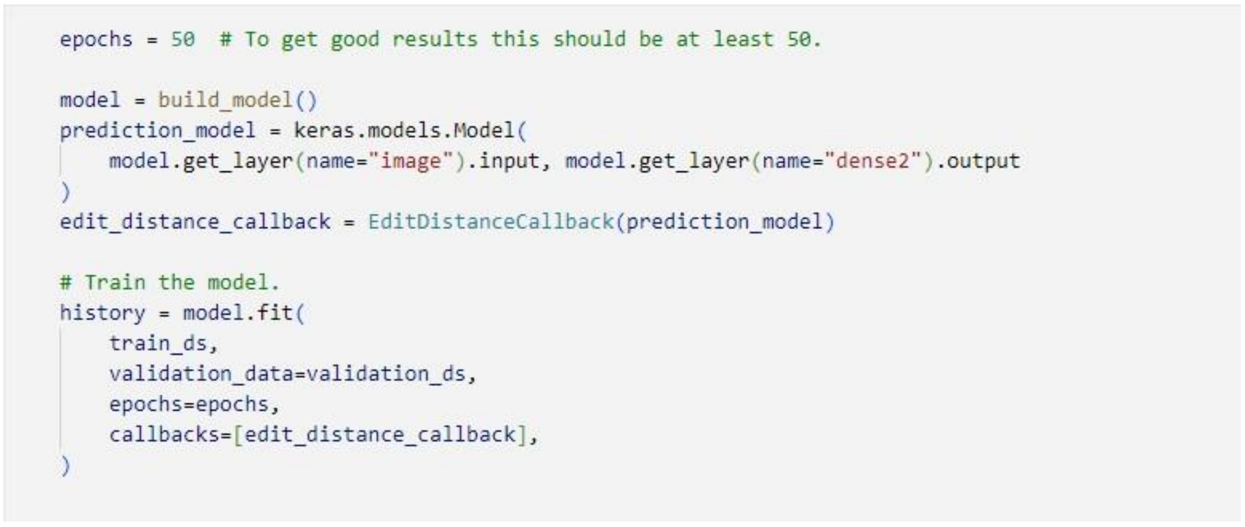
```python
def calculate_edit_distance(labels, predictions):
    # Get a single batch and convert its labels to sparse tensors.
    saprse_labels = tf.cast(tf.sparse.from_dense(labels), dtype=tf.int64)

    # Make predictions and convert them to sparse tensors.
    input_len = np.ones(predictions.shape[0]) * predictions.shape[1]
    predictions_decoded = keras.backend.ctc_decode(
        predictions, input_length=input_len, greedy=True
    )[0][0][:, :max_len]
    sparse_predictions = tf.cast(
        tf.sparse.from_dense(predictions_decoded), dtype=tf.int64
    )

    # Compute individual edit distances and average them out.
    edit_distances = tf.edit_distance(
        sparse_predictions, saprse_labels, normalize=False
    )
    return tf.reduce_mean(edit_distances)


class EditDistanceCallback(keras.callbacks.Callback):
    def __init__(self, pred_model):
        super().__init__()
        self.prediction_model = pred_model

    def on_epoch_end(self, epoch, logs=None):
        edit_distances = []
```



```
        for i in range(len(validation_images)):
            labels = validation_labels[i]
            predictions = self.prediction_model.predict(validation_images[i])
            edit_distances.append(calculate_edit_distance(labels, predictions).numpy())

        print(
            f"Mean edit distance for epoch {epoch + 1}: {np.mean(edit_distances):.4f}"
        )
```

Fig no. 21 Evaluation Metrics

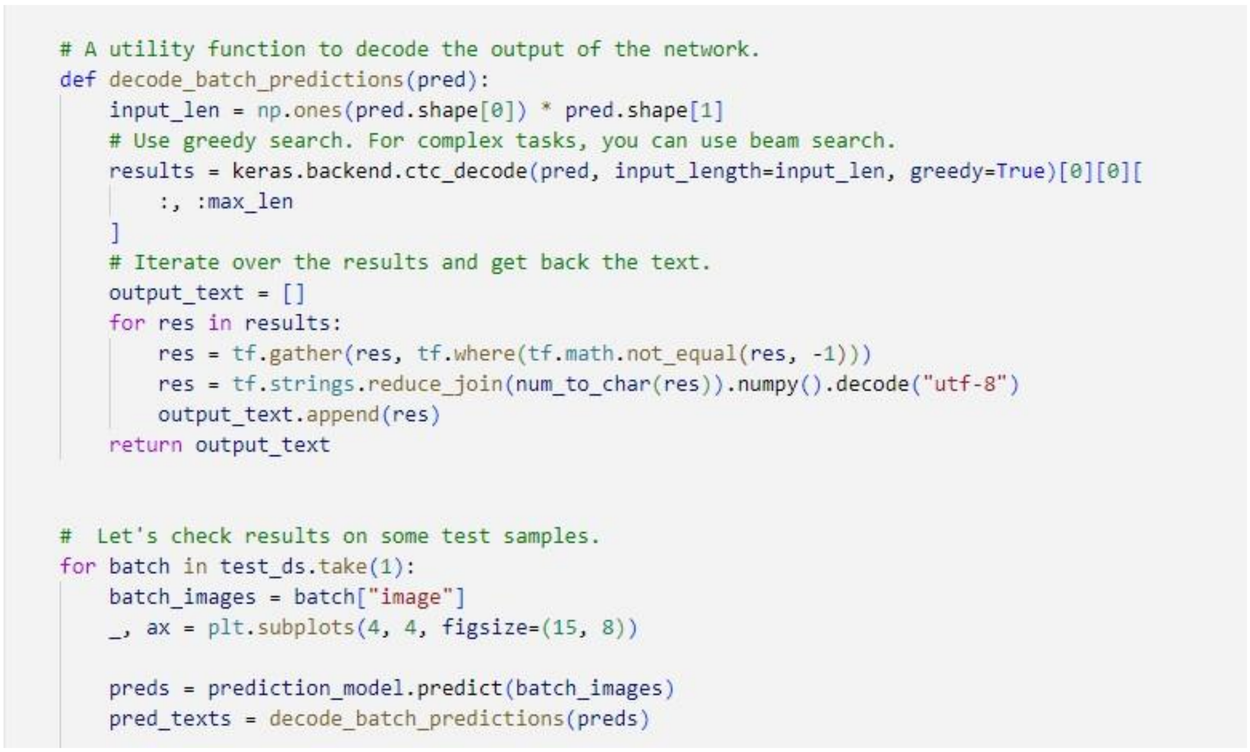
Training



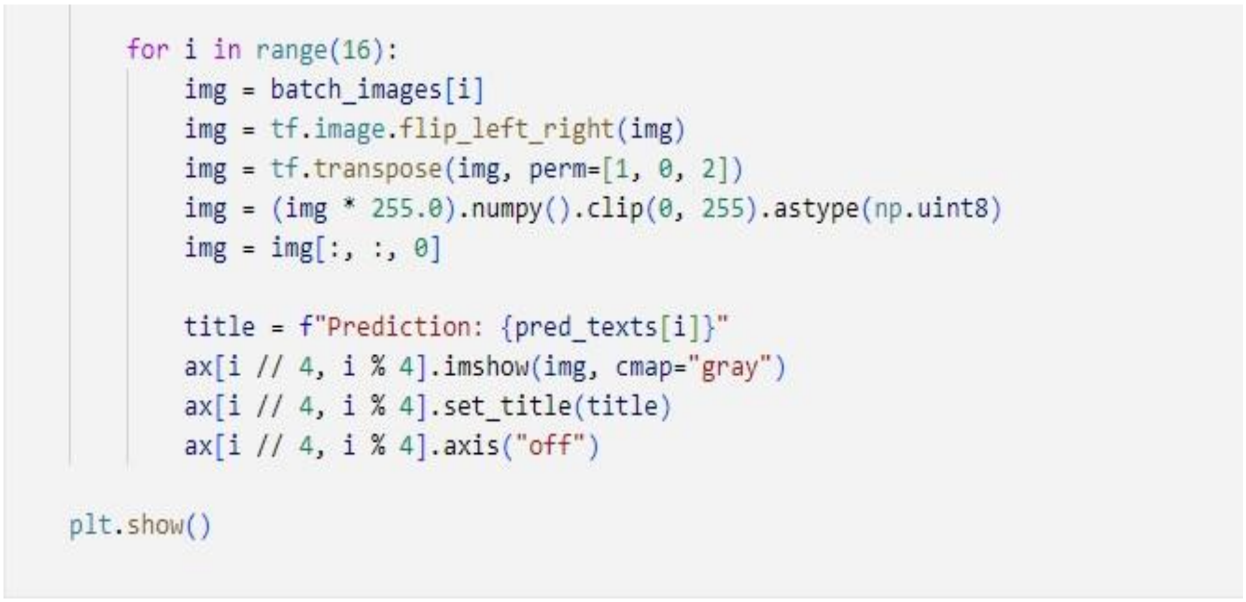
```
epochs = 50  # To get good results this should be at least 50.

model = build_model()
prediction_model = keras.models.Model(
    model.get_layer(name="image").input, model.get_layer(name="dense2").output
)
edit_distance_callback = EditDistanceCallback(prediction_model)

# Train the model.
history = model.fit(
    train_ds,
    validation_data=validation_ds,
    epochs=epochs,
    callbacks=[edit_distance_callback],
)
```

Fig no. 22 Training

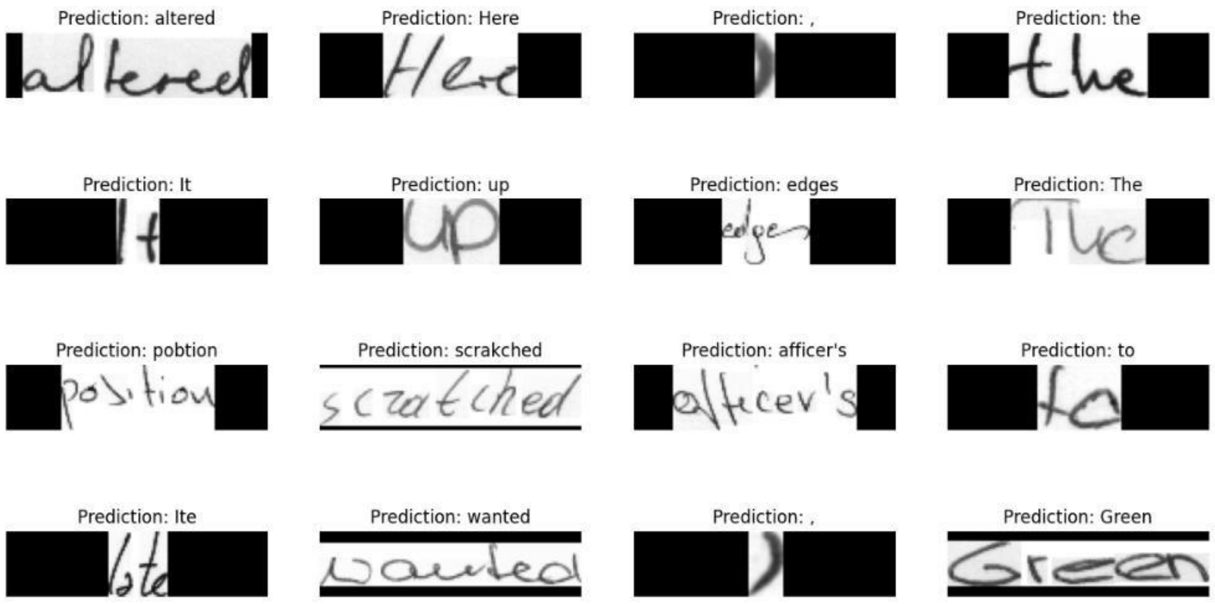
Inference



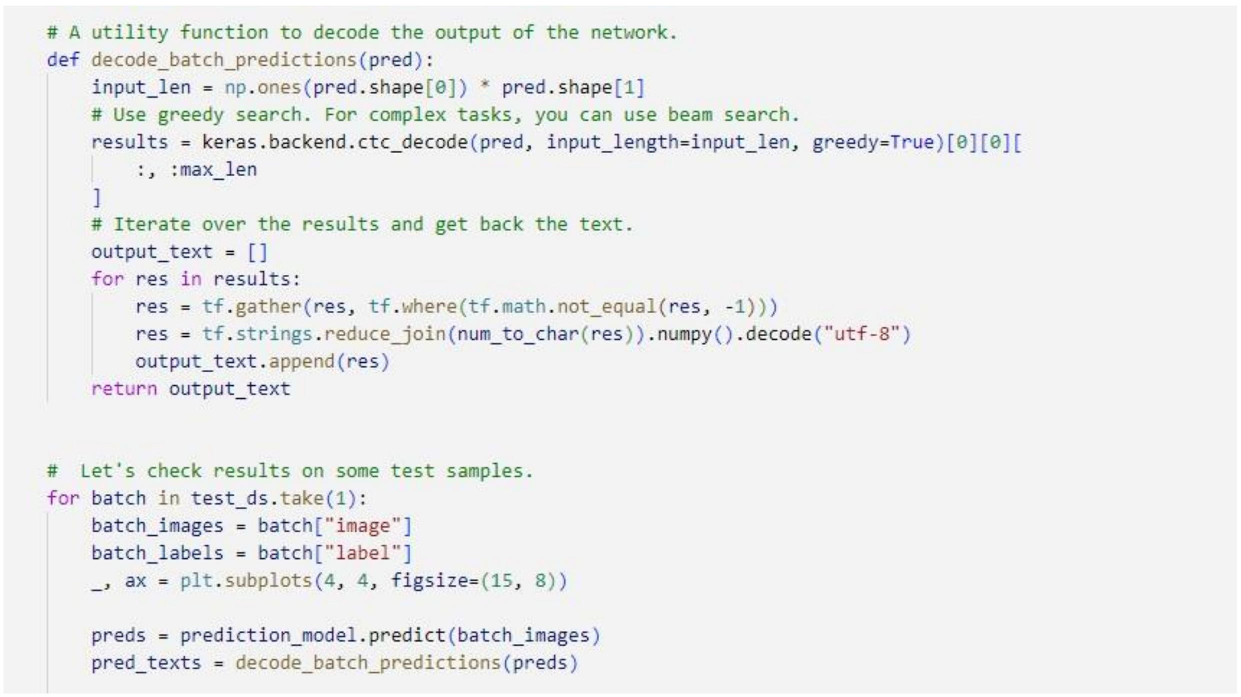
```python
# A utility function to decode the output of the network.
def decode_batch_predictions(pred):
    input_len = np.ones(pred.shape[0]) * pred.shape[1]
    # Use greedy search. For complex tasks, you can use beam search.
    results = keras.backend.ctc_decode(pred, input_length=input_len, greedy=True)[0][0][
        :, :max_len
    ]
    # Iterate over the results and get back the text.
    output_text = []
    for res in results:
        res = tf.gather(res, tf.where(tf.math.not_equal(res, -1)))
        res = tf.strings.reduce_join(num_to_char(res)).numpy().decode("utf-8")
        output_text.append(res)
    return output_text


#  Let's check results on some test samples.
for batch in test_ds.take(1):
    batch_images = batch["image"]
    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    preds = prediction_model.predict(batch_images)
    pred_texts = decode_batch_predictions(preds)
```



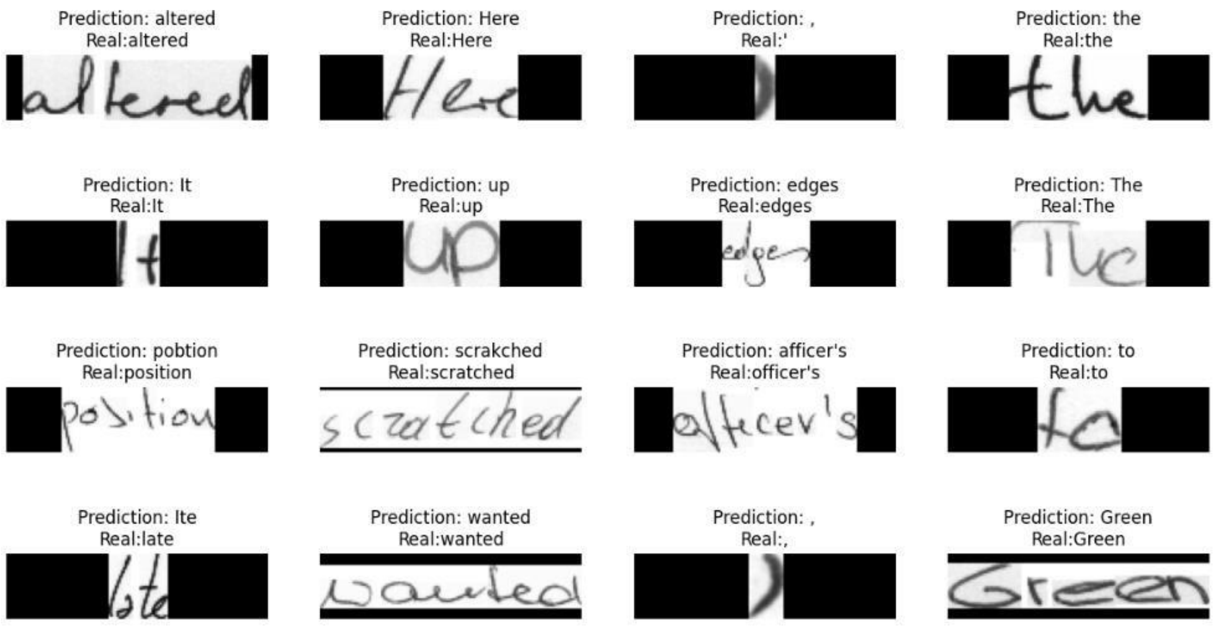
```python
    for i in range(16):
        img = batch_images[i]
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]

        title = f"Prediction: {pred_texts[i]}"
        ax[i // 4, i % 4].imshow(img, cmap="gray")
        ax[i // 4, i % 4].set_title(title)
        ax[i // 4, i % 4].axis("off")

plt.show()
```



<u>Fig no. 23 Inference</u>

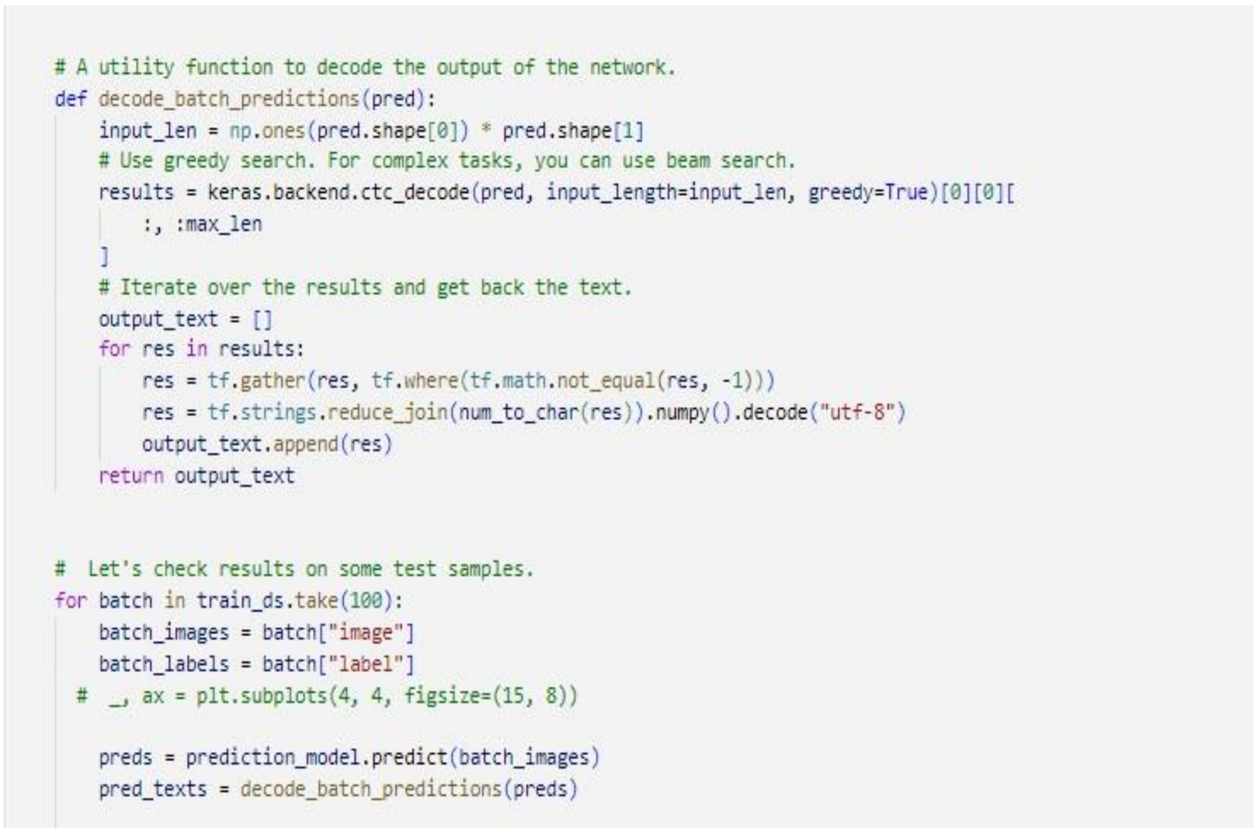


```python
# A utility function to decode the output of the network.
def decode_batch_predictions(pred):
    input_len = np.ones(pred.shape[0]) * pred.shape[1]
    # Use greedy search. For complex tasks, you can use beam search.
    results = keras.backend.ctc_decode(pred, input_length=input_len, greedy=True)[0][0][
        :, :max_len
    ]
    # Iterate over the results and get back the text.
    output_text = []
    for res in results:
        res = tf.gather(res, tf.where(tf.math.not_equal(res, -1)))
        res = tf.strings.reduce_join(num_to_char(res)).numpy().decode("utf-8")
        output_text.append(res)
    return output_text


#  Let's check results on some test samples.
for batch in test_ds.take(1):
    batch_images = batch["image"]
    batch_labels = batch["label"]
    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    preds = prediction_model.predict(batch_images)
    pred_texts = decode_batch_predictions(preds)
    for i in range(16):
        img = batch_images[i]
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]
        # Gather indices where label!= padding_token.
        label = batch_labels[i]
        indices = tf.gather(label, tf.where(tf.math.not_equal(label, padding_token)))
        # Convert to string.
        label = tf.strings.reduce_join(num_to_char(indices))
        label = label.numpy().decode("utf-8")
        title = f"Prediction: {pred_texts[i]}"
        new_label = title + "\nReal:" + str(label)

        real_labels_list.append(pred_texts[i])
        predicted_labels_list.append(str(label))


        ax[i // 4, i % 4].imshow(img, cmap="gray")
        ax[i // 4, i % 4].set_title(new_label)
        ax[i // 4, i % 4].axis("off")

plt.show()
```

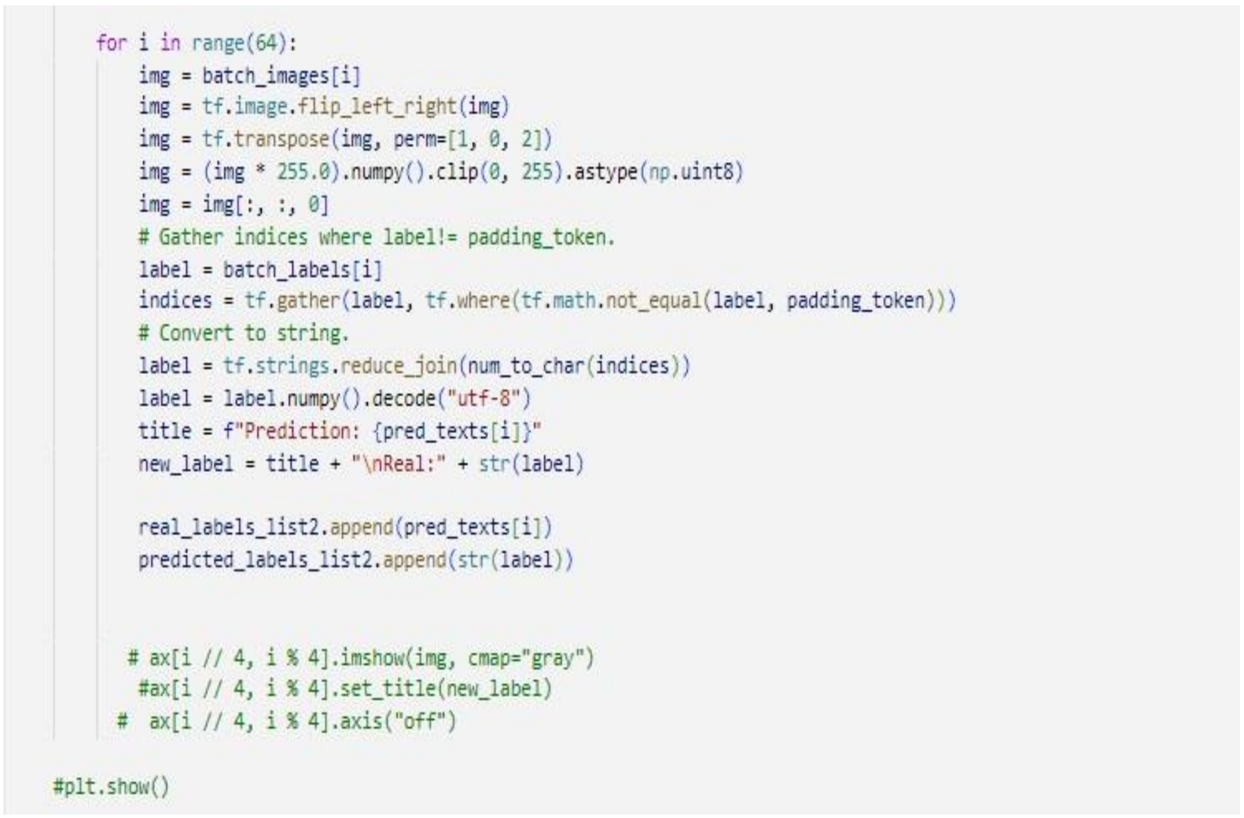


```python
def character_error_rate(predicted_labels_list, real_labels_list):

  if len(predicted_labels_list) != len(real_labels_list):
    raise ValueError("Predicted and real labels lists must have the same length.")

  total_errors = 0
  total_chars = 0
  for predicted_label, real_label in zip(predicted_labels_list, real_labels_list):
    total_errors += sum(a != b for a, b in zip(predicted_label, real_label))
    total_chars += len(real_label)

  cer = total_errors / total_chars
  return cer

cer = character_error_rate(predicted_labels_list, real_labels_list)
print("word Error Rate (CER):", cer)
```

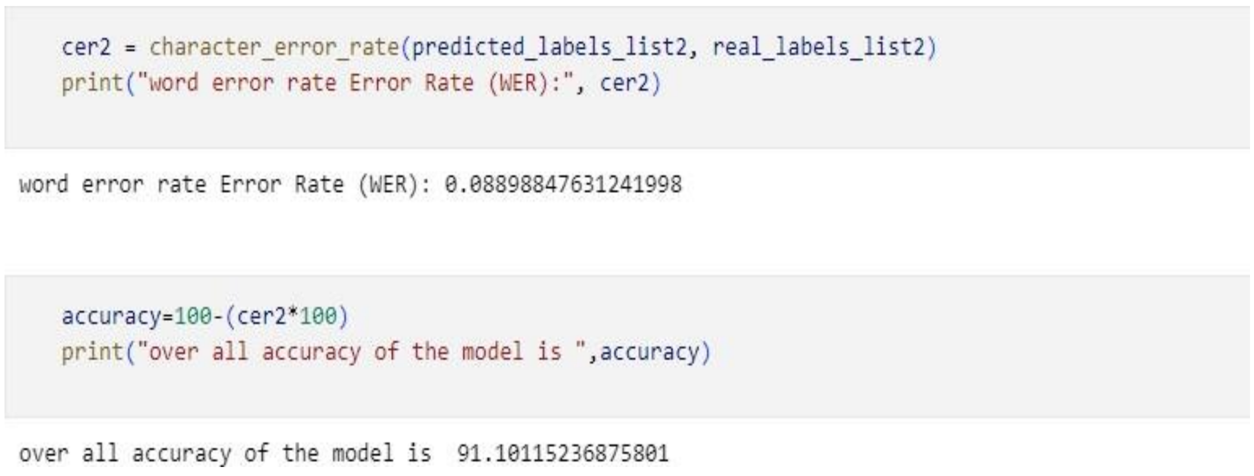
```
word Error Rate (CER): 0.15942028985507245
```



```python
# A utility function to decode the output of the network.
def decode_batch_predictions(pred):
    input_len = np.ones(pred.shape[0]) * pred.shape[1]
    # Use greedy search. For complex tasks, you can use beam search.
    results = keras.backend.ctc_decode(pred, input_length=input_len, greedy=True)[0][0][
        :, :max_len
    ]
    # Iterate over the results and get back the text.
    output_text = []
    for res in results:
        res = tf.gather(res, tf.where(tf.math.not_equal(res, -1)))
        res = tf.strings.reduce_join(num_to_char(res)).numpy().decode("utf-8")
        output_text.append(res)
    return output_text


#  Let's check results on some test samples.
for batch in train_ds.take(100):
    batch_images = batch["image"]
    batch_labels = batch["label"]
  #  _, ax = plt.subplots(4, 4, figsize=(15, 8))

    preds = prediction_model.predict(batch_images)
    pred_texts = decode_batch_predictions(preds)
```



```python
    for i in range(64):
        img = batch_images[i]
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]
        # Gather indices where label!= padding_token.
        label = batch_labels[i]
        indices = tf.gather(label, tf.where(tf.math.not_equal(label, padding_token)))
        # Convert to string.
        label = tf.strings.reduce_join(num_to_char(indices))
        label = label.numpy().decode("utf-8")
        title = f"Prediction: {pred_texts[i]}"
        new_label = title + "\nReal:" + str(label)

        real_labels_list2.append(pred_texts[i])
        predicted_labels_list2.append(str(label))


       # ax[i // 4, i % 4].imshow(img, cmap="gray")
        #ax[i // 4, i % 4].set_title(new_label)
      #  ax[i // 4, i % 4].axis("off")

#plt.show()
```



```
cer2 = character_error_rate(predicted_labels_list2, real_labels_list2)
print("word error rate Error Rate (WER):", cer2)
```

```
word error rate Error Rate (WER): 0.08898847631241998
```

```
accuracy=100-(cer2*100)
print("over all accuracy of the model is ",accuracy)
```

```
over all accuracy of the model is  91.10115236875801
```

Fig no. 24 Final Result

Accuracy measures are crucial for evaluating our Medical Prescription OCR system. Key metrics include accuracy, precision, recall, F1 score, and AUC-ROC. Accuracy indicates the proportion of correctly interpreted prescriptions. Precision measures the true positives among predicted positives, while recall assesses the system's ability to identify all relevant medicine names. The F1 score balances precision and recall, and AUC-ROC evaluates the model's class differentiation ability. These metrics ensure the OCR system's effectiveness in accurately interpreting and managing medical prescriptions.

7. SOFTWARE TESTING

Software testing is the process of testing before the real software is run through to the end. Ensuring that the expected output is free from mistakes and faults is the primary goal of software testing.

7.1 Unit Testing:

Unit testing is the initial stage in the testing process, focusing on each individual unit or method within a module to ensure it produces the expected output. This approach helps identify and fix defects and errors at the unit level. In the context of ship detection using remote sensing images, unit testing can be applied to various parts or operations that constitute the ship detection algorithm. Here are some examples of unit testing in this scenario:

To ensure that medical prescription images are properly loaded, normalized, and ready for further processing, test the image loading and preprocessing functions. Verify that preprocessing operations such as resizing, color space conversion, and noise removal are performed correctly and that the images are read in the expected format.

Implement and test the evaluation metrics for the prescription recognition algorithm to determine its performance. Test functions that compute accuracy, precision, recall, F1 score, and AUC-ROC to measure the effectiveness of the algorithm in identifying and extracting prescription details.

Use sample test cases that cover a range of scenarios, such as different handwriting styles, prescription formats, and image conditions, when conducting unit testing. By thoroughly testing each individual component before integrating them into the overall prescription recognition system, you can ensure their accuracy and functionality.

7.2 Integration Testing:

After unit testing, integration testing is carried out. The output of unit testing serves as the input for integration testing. Functional requirements are taken as input. Individual units of code in a module are gathered or integrated for testing in this method.

In the case of the Medical Prescription Recognition system, integration testing focuses on the interaction and integration between the various parts or modules. This includes validating the input and output mechanisms, testing the integration with evaluation and performance metrics, and ensuring the smooth integration of preprocessing and text recognition components.

It also includes ensuring the correct integration of deep learning models or rule-based algorithms with other components. The objective is to guarantee that the parts function in harmony, that data is accurately transferred across modules, and that the desired results are produced.

By performing integration testing, any problems or discrepancies in the integration process may be found and fixed, resulting in a reliable and effective medical prescription recognition system.

7.3 Acceptance Testing:

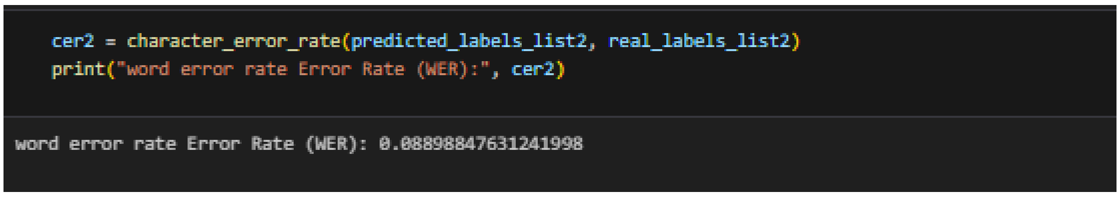
Acceptance testing is conducted using the results of system testing as a starting point. This is done to verify that the expected and assumed requirements are met.

Acceptance testing for the Medical Prescription Recognition system ensures it meets its intended requirements and criteria. This involves evaluating its overall performance, accuracy, reliability, and robustness in interpreting medical prescriptions. The testing simulates real-world scenarios to confirm the system accurately extracts medicine names, dosages, and other crucial details across different prescription formats and handwriting styles. The goal is to ensure the system effectively supports healthcare professionals in medication management while enhancing patient safety and satisfaction.

7.4 Testing on our System:

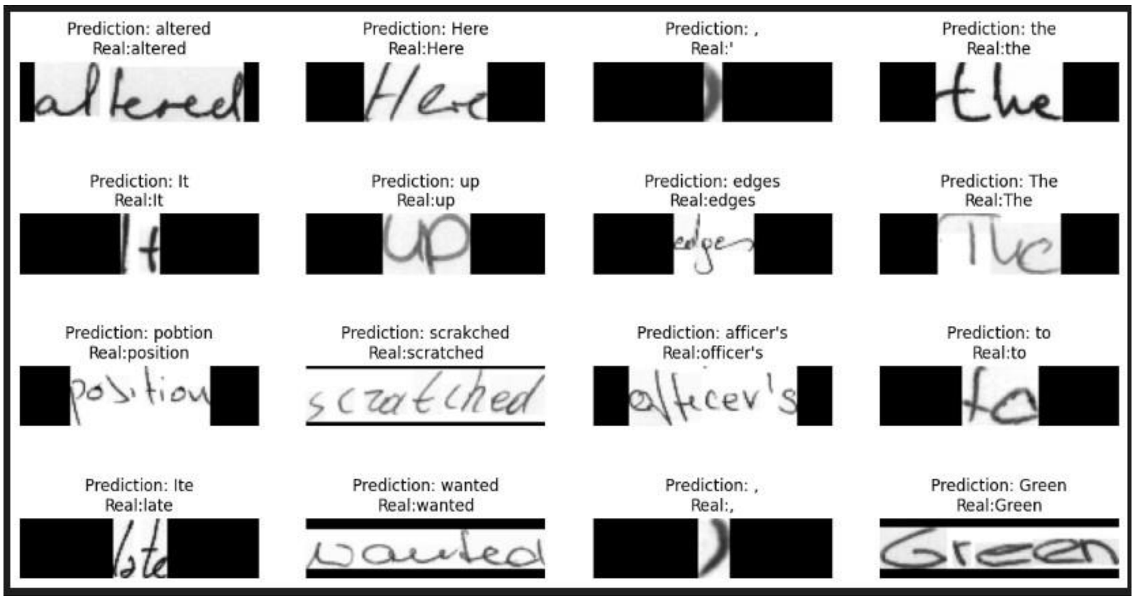
Following integrated testing, machine testing is conducted for the Medical Prescription Recognition system. This phase incorporates both functional and non-functional testing aspects based on the integrated testing results. System testing evaluates the system's design and behavior in interpreting medical prescriptions accurately and reliably. The objective is to validate that the system functions as intended, meeting performance benchmarks and ensuring robust operation across various prescription formats and user interactions.

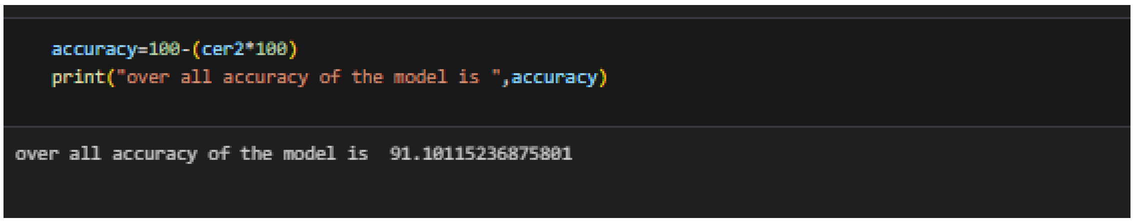
8. RESULTS



```
cer2 = character_error_rate(predicted_labels_list2, real_labels_list2)
print("word error rate Error Rate (WER):", cer2)
```

```
word error rate Error Rate (WER): 0.08898847631241998
```





```
accuracy=100-(cer2*100)
print("over all accuracy of the model is ",accuracy)
```

```
over all accuracy of the model is  91.10115236875801
```

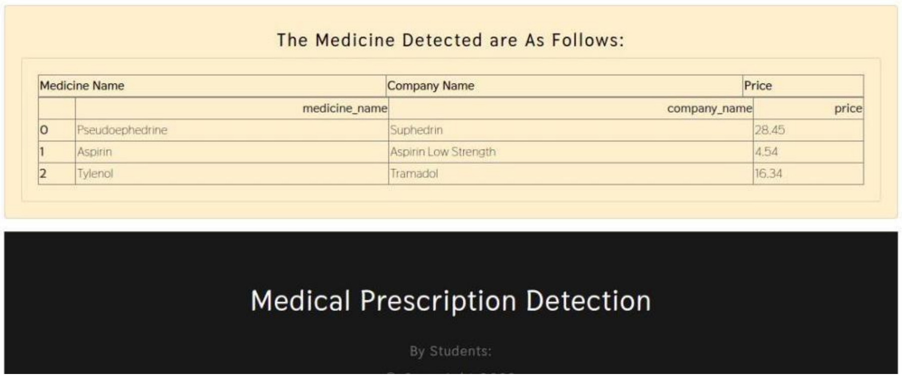
Number of Epochs: 50

Loss Function: Connectionist Temporal Classification

Optimizer: Adam

Accuracy: 91.10%

Word Error Rate (WER): 8.90%


The Medicine Detected are As Follows:

Medicine Name		Company Name	Price
	medicine_name	company_name	price
0	Pseudoephedrine	Suphedrin	28.45
1	Aspirin	Aspirin Low Strength	4.54
2	Tylenol	Tramadol	16.34

Medical Prescription Detection

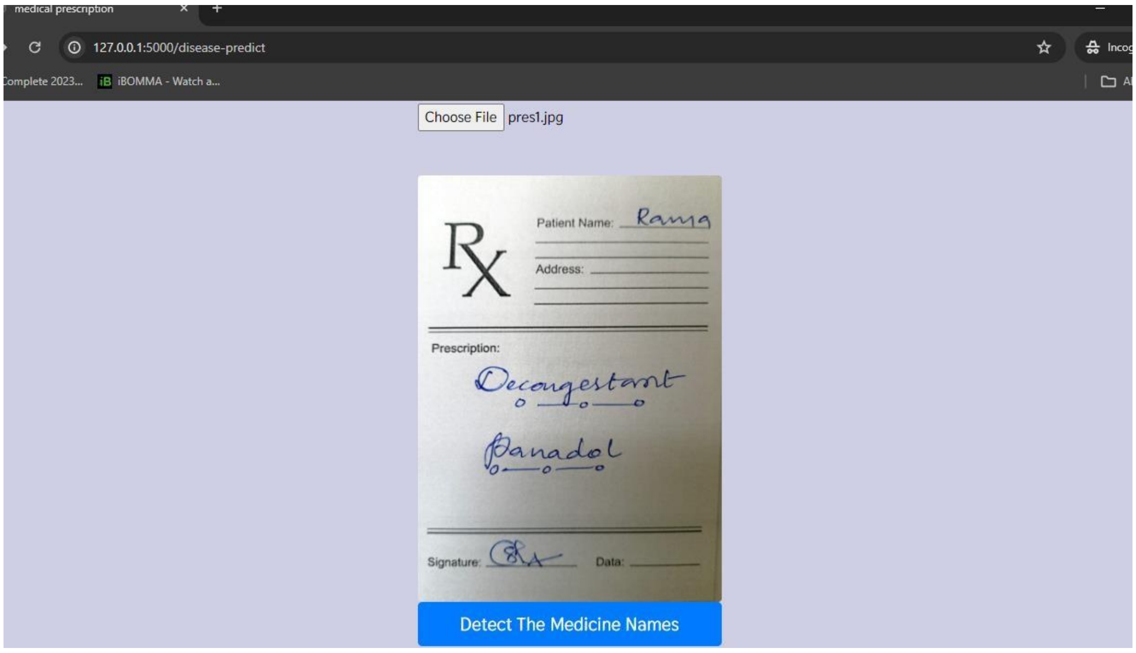
By Students:

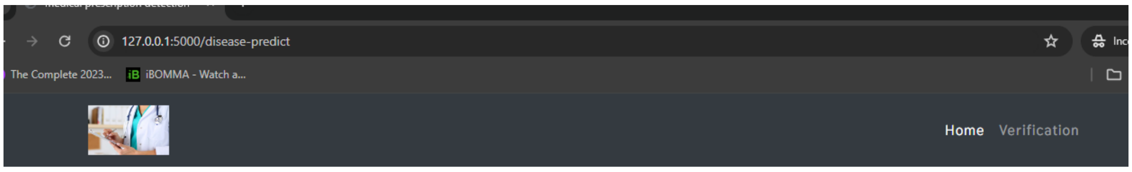
Fig no. 25 Final Output

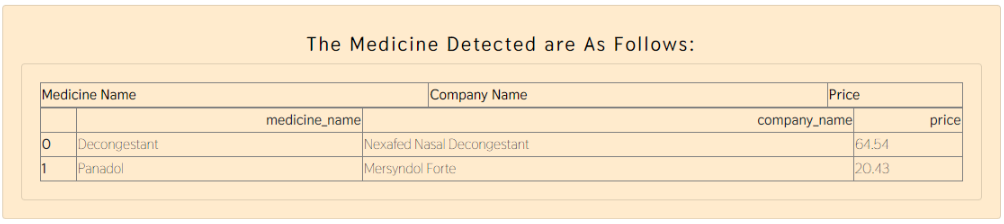
<u>9. CONCLUSION AND FUTURE ENHANCE MENTS</u>

The Medical Prescription OCR system successfully automated prescription interpretation with high accuracy using YOLOv5 and deep learning models.

Future enhancements include refining accuracy, integrating NLP for complex instructions, enabling real-time processing, integrating with EHR systems, personalizing medication recommendations, and establishing a user feedback mechanism.







The Medicine Detected are As Follows:

	Medicine Name	Company Name	Price
	medicine_name	company_name	price
0	Decongestant	Nexafed Nasal Decongestant	64.54
1	Panadol	Mersyndol Forte	20.43

10. BIBLIOGRAPHY

[1] https://ieeexplore.ieee.org/document/9609390

[2] https://www.semanticscholar.org/paper/Medical-Handwritten-Prescription-Recognition-Using-Achkar-Ghayad/3fff5b7c44431728c6cd36b7a164697c02d83acc

[3] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9509260/

[4] https://colab.ws/articles/10.1109%2FCCWC51732.2021.9376141?utm_source

[5] https://arxiv.org/abs/2210.11666?utm_source

[6] https://www.researchgate.net/publication/350150845_Medical_Prescription_Recognition_using_Machine_Learning?utm_source