CIS611 Final Project
Muttaki I. Bismoy
G02518887

Real-Time Sign Language Detection using CNN

Motivation

The main purpose of the model is to create a webcam based real-time sign language detection model. This technology helps deaf and hard of hearing people communicate. A system like this is driven by improving communication. Sign language is the primary mode of communication for many deaf people, so a real-time system that interprets and translates gestures can simplify interactions. This technology makes everyday communication easier and creates more inclusive environments by allowing sign language users to talk to non-signers. In addition to communication, this system is educational. Sign language learning is difficult, so real-time feedback from such a system can be very helpful. It corrects gestures, improves proficiency, and deepens understanding of this form of communication. The effects on digital accessibility are significant. Real-time sign language detection makes video conferencing and virtual meetings accessible to deaf and hard-of-hearing users. Integration could change how we think about digital accessibility, making it more welcoming and inclusive. The immediate translation of sign language can help sign language users communicate better in these settings. Human-computer interaction also advances with innovation. Enabling computers to understand and interpret nonverbal communication expands our technological interactions. Finally, this technology's real-time element is crucial. This system can support live translation and interpretation without a human interpreter. Immediacy improves communication and gives users a sense of independence and autonomy. In essence, the creation of a real-time sign language detection system using a webcam is much more than a technological achievement. It represents a meaningful step towards social inclusion, breaking down barriers in communication, and opening up new avenues for interaction that cater to a diverse range of needs and preferences. However, only five sign language gestures—"Hello," "Yes," "No," "Thank you," and "I like you"—were shown in the early stages of this course project. The model can be used with any sign language, though.

Resources

A sign language model on GitHub served as inspiration for this project. Here's the link. The Particular Five sign language and the code for gathering images via the webcam of a computer are identical to the author's. In contrast, the author employed a pre-existing model for sign language detection, while I developed and trained my own model.

Software Engineering Model

This project's software process model was the Waterfall Model. The waterfall model is a good choice for making a real-time sign language detection system because it has features that work well with the project. It is helpful for some types of projects that the waterfall model is very linear and step-by-step. This is one of the best things about the Waterfall model: it is organized and careful. This model is made up of several steps, such as Requirements, Design, Implementation, Verification, and Maintenance. Having clear steps between stages can be very helpful for projects with clear needs that will not change much as they are being worked on, like sign language detection. Each step of Waterfall model has been completed one's then been moved to the next one. Prior to writing code, the model makes sure that everyone knows exactly what the system is supposed to do. Planning and writing things down carefully are also very important in the Waterfall model. Planning ahead is another good thing about the Waterfall model. Still, it is important to keep in mind that the Waterfall model is not perfect, especially when it comes to being able to change things quickly. The Waterfall model is set in its ways, while the Agile model is meant to be able to change and adapt to new situations. It can be hard and cost a lot to make changes after a phase is over. Because of this, it works best for projects where the needs will not change much, like my project to find Sign Language. Large, complicated projects with a lot of risks should use methods like the Spiral model, which combines design and prototyping in stages that build on each other. The Waterfall model's simple, step-by-step approach may work better for projects with stable, clear requirements and readily understandable technology. In conclusion, the Waterfall model is a good choice for this project because it is structured, focuses on careful planning and documentation, is only used once, and is predictable.

Use Case Diagram

The use-case diagram illustrates a "Real-Time Sign Language Detector" system with interactions involving the user (actor) and the system's use cases. The main use cases identified are "Perform a Gesture" and "Detect a Sign".

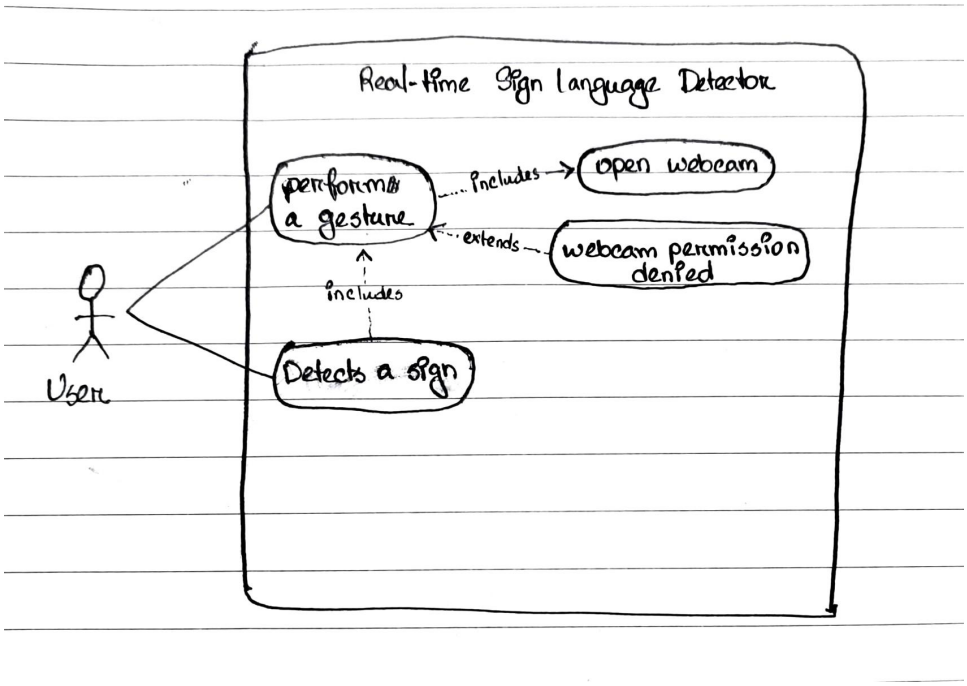

Fig. 01: Use Case Diagram

Use Case: Perform a Gesture
Actor: User
Description: This use case delineates the scenario in which the user interacts with the system using a sign language gesture. Due to the fact that the system is incapable of detecting "no input" or any other gesture besides the five specified, it will not produce any output if any of the five gestures are outside the webcam's frame. One of the predetermined gestures that the system possesses the ability to identify is the gesture. The system processes the visual input from the webcam.
Pre-condition: the webcam must be functional, and the sign language detection model must be active in order to interpret user gestures. The "Open Webcam" use case is included in the "Perform a Gesture" use case, indicating that opening the webcam is a step that must occur as part of the gesture performance. Additionally, there's an "extends" relationship to a "Webcam Permission Denied" use case, indicating an alternative path that can occur during the "Open Webcam" use case. This likely represents an exception or error handling path if the system fails to obtain the necessary permissions to access the webcam.

Use Case: Detects a Sign
Actor: User
Description: Upon the successful execution of the gesture, the system initiates an analysis of the input in order to discern the specific sign. The indicated sign language's meaning will be displayed in the upper left corner of the webcam frame. The system will continue to detect gestures in real-time and display the

output in the upper left corner of the frame so long as the user does not enter "q" on his keyboard while displaying various gestures. This use case is included in the "Perform a Gesture" use case, signifying that detection is part of the gesture performance process.
Pre-condition: The user must have completed the "Perform a Gesture" use case, resulting in a sign that the system can detect and interpret.

Code Explanation

The entire project has been conducted in parts:

1. Write and Run Code to **create a dataset** (Adapted from the author)
2. Run built-in labeling tool to **label the dataset** (Adapted from GitHub)
3. **Creating the model** and generating metrics
4. **Test:** Write code to check real-time input via Webcam

Platform

1. Jupyter Notebook
2. PyCharm
3. Google Colab
4. GitHub

Create the dataset:
The first step for the project was to create a dataset. The provided code snippet [Fig. 02] appears to be part of a Python script intended to collect images for a sign language detection system using a webcam. The code was sourced from GitHub by the same author who inspired me to build this project. The script uses OpenCV for capturing images, os for file path handling, time for adding delays, and uuid for generating unique image filenames. The script defines a path to save the images and a list of labels corresponding to the sign language gestures to be detected. It sets a goal to capture a specific number (15) of images for each label. For each label, the script attempts to create a directory where the captured images will be stored. It then starts the webcam, and for a set number of iterations, it captures a frame from the webcam, generates a unique filename for each image, and saves the image to the designated directory. After displaying the frame for a brief period, it waits for a key press. If the 'q' key is pressed, it exits the loop and releases the webcam. The output messages indicate that when attempting to create directories for each label, the script finds that these directories already exist. This suggests that the image collection process has been attempted previously and that the necessary folders have already been

created. The labels' names served as a guide for saving the images and organizing them into folders. Thus, the dataset is created.

```
In [1]:
import cv2 #openCV
import os #helps with file paths
import time #help with delaying
import uuid #name the image files

In [2]:
IMAGES_PATH = "Tensorflow/workspace/images/collectedimages"

In [3]:
labels = ['hello', 'thanks', 'yes', 'no', 'ilikeyou']
number_imgs = 15

In [4]:
for label in labels:
    !mkdir {'Tensorflow\workspace\images\collectedimages\\'+label}
    cap = cv2.VideoCapture(0)
    print("Collecting images for {}".format(label))
    time.sleep(5)
    for imgnum in range(number_imgs):
        ret, frame = cap.read()
        imgname = os.path.join(IMAGES_PATH, label, label+'.'+'{}.jpg'.format(str(uuid.uuid1())))
        cv2.imwrite(imgname, frame)
        cv2.imshow('frame', frame)
        time.sleep(2)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cap.release()

A subdirectory or file Tensorflow\workspace\images\collectedimages\hello already exists.
Collecting images for hello
A subdirectory or file Tensorflow\workspace\images\collectedimages\thanks already exists.
Collecting images for thanks
A subdirectory or file Tensorflow\workspace\images\collectedimages\yes already exists.
Collecting images for yes
A subdirectory or file Tensorflow\workspace\images\collectedimages\no already exists.
Collecting images for no
A subdirectory or file Tensorflow\workspace\images\collectedimages\ilikeyou already exists.
Collecting images for ilikeyou
```

Fig. 02: Code Snippet to Create a Dataset

Label the dataset:

An open-source program known as labelImg is responsible for generating labels for the dataset. I downloaded the code from GitHub and executed it using the PyCharm platform on my system. The code generates a user interface that requires manual labeling of each image. I designated specific portions of my hand gestures for each sign language with their respective meanings. The labelling is included in an XML file generated by the system. After placing the XML files for each sign language in their respective folders, I proceeded to upload the dataset to my Google Drive account. Subsequently, the dataset was annotated and uploaded to the cloud (Google Drive). Fig. 03 demonstrates how the dataset has been organized. Dataset < Label [Folder] < Images [Unique Names] + Labeling [.xml files for each image].

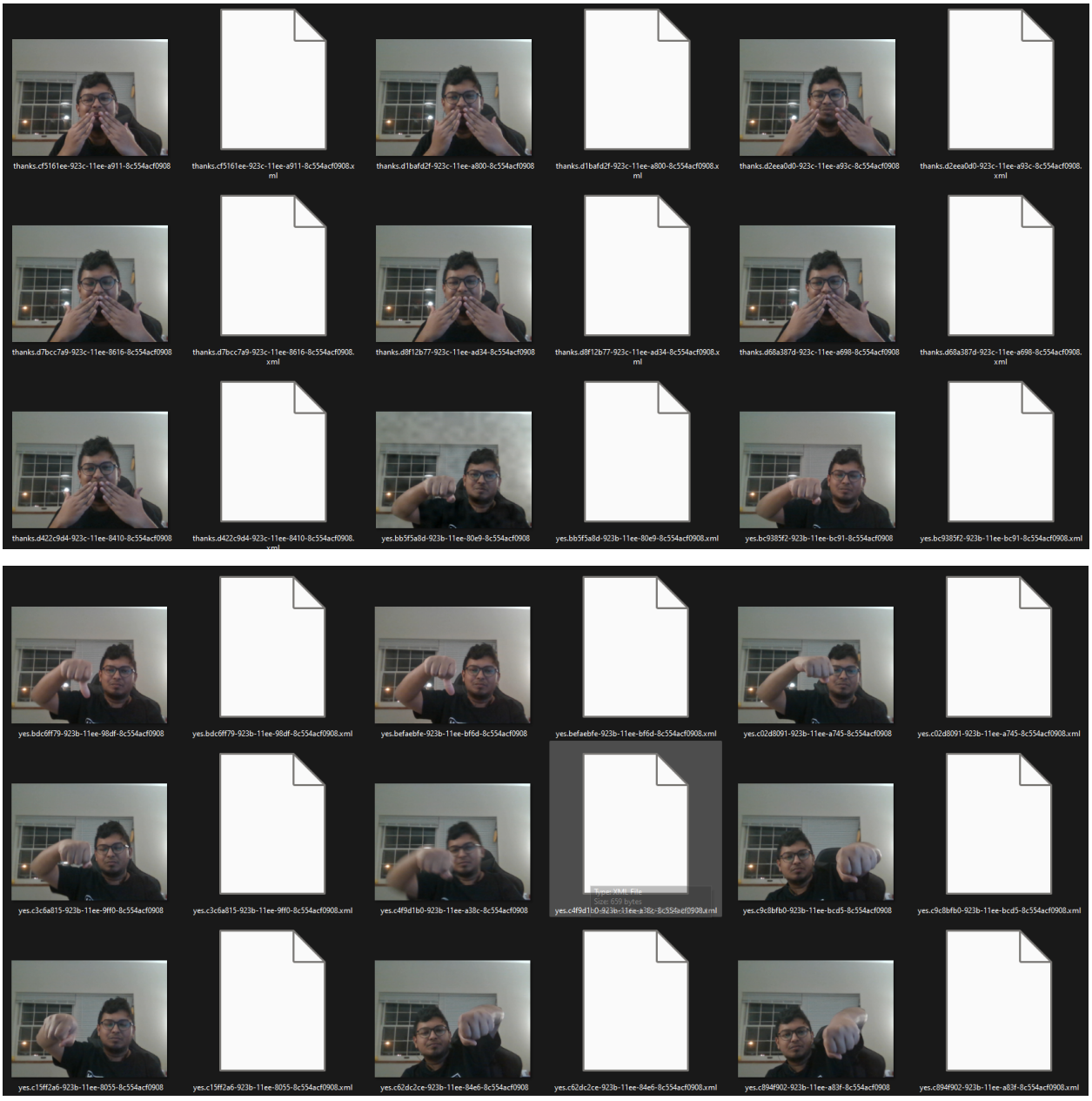

Fig. 03: Dataset of gestures alongside the .xml files of the labeling

Creating the Model:

The code for creating the model has been run in Google Colab, used for building a convolutional neural network (CNN) to recognize sign language from images. It starts by importing the necessary libraries (os, numpy, opencv, sklearn, and keras) and mounting Google Drive to access the dataset. The dataset, expected to be organized with a separate folder for each sign ('Hello', 'Yes', 'No', 'ThankYou', 'ILikeYou'), is loaded and preprocessed. Images are read from the dataset, resized to 64x64 pixels, normalized, and split into training and testing sets. Then, Keras is used to create a CNN model. This

model has two convolutional layers (each with a max-pooling layer following it), a flattening layer, a fully connected layer, and a dropout layer to stop the model from fitting too well.

Real-Time.ipynb

File Edit View Insert Runtime Tools Help Last saved at 1:05 AM

+ Code + Text

```
import os
import numpy as np
import cv2
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator
```

```
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# Accessing your dataset (replace 'path_to_your_dataset' with the actual path)
dataset_path = '/content/drive/My Drive/Real-Time_Dataset'

```

```
# Assuming the dataset is organized with a folder for each sign
#dataset_path = '/path/to/dataset'
labels = ['Hello', 'Yes', 'No', 'ThankYou', 'ILikeYou']
label_map = {label: idx for idx, label in enumerate(labels)}
```

```
# Load and preprocess data
def load_data(dataset_path):
    images = []
    image_labels = []

    for label in labels:
        sign_folder = os.path.join(dataset_path, label)
        for image_file in os.listdir(sign_folder):
            image_path = os.path.join(sign_folder, image_file)
            image = cv2.imread(image_path)

            if image is None:
                print(f"Warning: Unable to read image at {image_path}")
                continue

            image = cv2.resize(image, (64, 64))
            images.append(image)
            image_labels.append(label_map[label])

    images = np.array(images, dtype='float32') / 255.0
    image_labels = to_categorical(np.array(image_labels), num_classes=len(labels))

    return train_test_split(images, image_labels, test_size=0.2, random_state=42)

```

Fig. 04: Import Libraries and Load Data

A softmax activation function is employed in the final layer to perform a multi-class classification of the five signs. The Adam optimizer and the categorical crossentropy loss function are utilized in the compilation of the model. It is trained with the designated epochs (32) and batch sizes (16) on the training data, with the validation set also being utilized. The trained model is saved to Google Drive, and its architecture and performance summary are displayed following training. This model, once saved, can be utilized for sign language detection in real time.

The code shown in Fig. 04 is intended for processing and preparing data for the model. The first block of code imports various libraries, each serving a distinct purpose. The `os` library is for interacting with the operating system, such as navigating directories and file paths. `numpy`, commonly imported as `np`, is a fundamental package for scientific computing in Python, offering support for arrays and matrices. `cv2` is the OpenCV library used for computer vision tasks, including working with images and videos. The `sklearn.model_selection` provides utilities for splitting datasets into training and test sets, and `sklearn.metrics` includes performance metrics such as accuracy and confusion matrix. The `keras` library, which is a part of TensorFlow 2, includes modules for building neural networks, with layers like `Conv2D` for convolutional operations, `MaxPooling2D` for downsampling, `Flatten` for converting matrices to vectors, `Dense` for fully connected layers, and `Dropout` for regularization. The `ImageDataGenerator` is used for real-time data augmentation during the training of neural networks. In the subsequent code blocks, the notebook demonstrates mounting my Google Drive to access datasets stored there. The code proceeds to define the dataset's path, which is structured with separate folders for each sign language gesture. It then presents a function `load_data()` designed to load images and their corresponding labels from the specified path, process them (resize and normalize), and split them into training and test sets. This is essential for training a machine learning model where processed data is needed in a consistent format for both learning and evaluation. The purpose of this code is to set up a pipeline for feeding processed data into a neural network for the task of sign language recognition.

There are several layers in the model, starting with convolutional layers that use ReLU activation and max pooling for downsampling, a flattening layer that turns the input data into a fully connected network, a dense layer that also uses ReLU activation, a dropout layer that stops the model from overfitting, and finally a dense layer that uses a softmax activation function for classification [shown in Fig. 05]. The Adam optimizer is used to build the model, and categorical crossentropy is used as the loss function and accuracy as the metric. The architecture is shown in the model summary, along with the output shapes and parameter counts for each layer.

```
[25]  1 # Define CNN model
      2 model = Sequential()
      3 model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)))
      4 model.add(MaxPooling2D((2, 2)))
      5 model.add(Conv2D(64, (3, 3), activation='relu'))
      6 model.add(MaxPooling2D((2, 2)))
      7 model.add(Flatten())
      8 model.add(Dense(64, activation='relu'))
      9 model.add(Dropout(0.5))
     10 model.add(Dense(len(labels), activation='softmax'))
```

```
[26]  1 # Compile the model
      2 model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
      1 model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 62, 62, 32)        896

 max_pooling2d_2 (MaxPoolin  (None, 31, 31, 32)        0
 g2D)

 conv2d_3 (Conv2D)           (None, 29, 29, 64)        18496

 max_pooling2d_3 (MaxPoolin  (None, 14, 14, 64)        0
 g2D)

 flatten_1 (Flatten)         (None, 12544)             0

 dense_2 (Dense)             (None, 64)                802880

 dropout_1 (Dropout)         (None, 64)                0

 dense_3 (Dense)             (None, 5)                 325

=================================================================
Total params: 822597 (3.14 MB)
Trainable params: 822597 (3.14 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Fig. 05: Model & its summary

Precision is the number of true positives compared to all predicted positives. Recall is the number of true positives compared to all actual positives. The classification report [Fig. 06] also has support, which is the number of actual instances of the class in the dataset, and f1-score, which is the harmonic mean of precision and recall. For each class, the report shows high precision, recall, and f1-scores, which means the model does well in all of them. The overall accuracy is 97%, which is very good and shows that the model is very good at sorting the data given. A confusion matrix [Fig. 07] is shown as an array below the report. It shows how well the model did by showing how many correct and incorrect predictions it made for each class. There are two plots at the bottom [Fig. 06]. The one on the left shows how accurate the model was over epochs for both training (accuracy) and validation (val_accuracy). The right one shows how much the model has lost over time, this time for both training (loss) and validation (val_loss).

These graphs show how the model is learning over time. The accuracy graph shows how well the model fits the data, and the loss graph shows how much error the model makes during training and validation.

```
[39]  1 from sklearn.metrics import classification_report, confusion_matrix
```

```
 1 # Print classification report and confusion matrix
 2 import numpy as np
 3
 4 # Convert one-hot encoded labels to single integer labels
 5 y_true_single = np.argmax(y_true, axis=1)
 6 y_pred_single = np.argmax(y_pred, axis=1)
 7
 8 print("Classification Report:\n", classification_report(y_true_single, y_pred_single))
 9 print("Confusion Matrix:\n", confusion_matrix(y_true_single, y_pred_single))
10
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       0.89      1.00      0.94         8
           2       1.00      0.83      0.91         6
           3       1.00      1.00      1.00         6
           4       1.00      1.00      1.00         5

    accuracy                           0.97        30
   macro avg       0.98      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30

Confusion Matrix:
 [[5 0 0 0 0]
 [0 8 0 0 0]
 [0 1 5 0 0]
 [0 0 0 6 0]
 [0 0 0 0 5]]
```

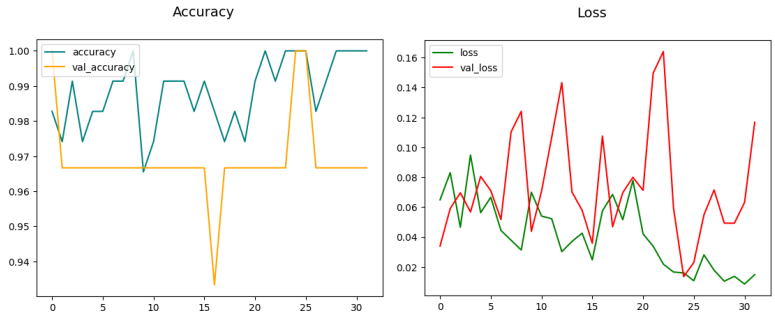

Fig. 06: Classification Reports and Graph plots

```
[41]  1 #import numpy as np
      2 #import matplotlib.pyplot as plt
      3 #from sklearn.metrics import confusion_matrix
      4 import seaborn as sns
      5
      6 # Assuming y_true_single and y_pred_single are your true and predicted labels
      7 cm = confusion_matrix(y_true_single, y_pred_single)
      8
      9 # Plotting
     10 plt.figure(figsize=(10, 7))
     11 sns.heatmap(cm, annot=True, fmt='g', cmap='Blues')
     12 plt.xlabel('Predicted')
     13 plt.ylabel('True')
     14 plt.title('Confusion Matrix')
     15 plt.show()
```

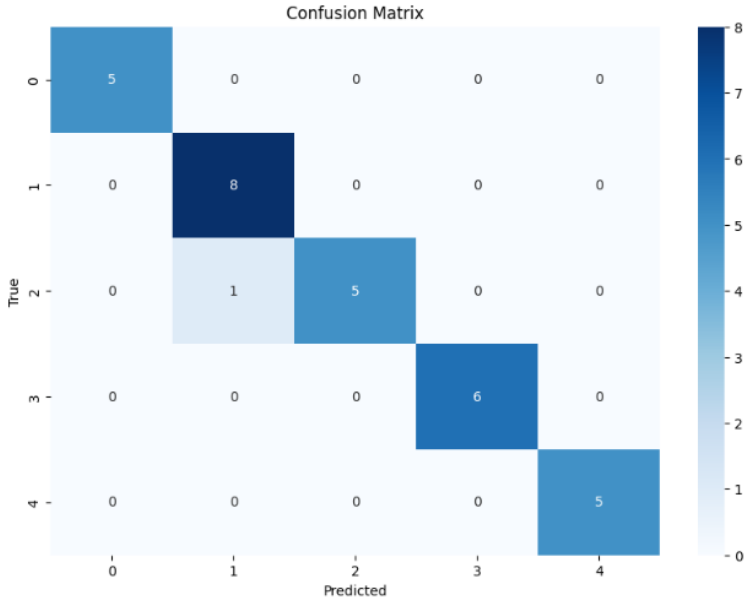

Fig. 07: Confusion Matrix

Test

The code [Fig. 08] for the test case defines a class `SignLanguageDetector` that contains methods for initializing the model, predicting gestures, and starting a webcam stream to detect sign language in real-time.In the `__init__` method, the model path is set, the model is loaded using Keras' `load_model` function, and a dictionary `class_labels` is created to map indices to label names. The `predict_gesture` method resizes the input frame to 64x64 pixels, normalizes it by dividing by 255, reshapes it to fit the model's input requirements, and then passes it to the model to make a prediction. The predicted class index is obtained by finding the argument with the maximum probability in the prediction. The corresponding label is then returned. The `start_webcam` method initializes the webcam and enters a loop where it continually reads frames from the webcam. If a frame is successfully read, it calls `predict_gesture` to get the sign language prediction, which it then displays on the frame using OpenCV's `putText` method. Pressing "q" will release the webcam and close the OpenCV windows,

which will end the loop. This script is set up to run a sign language detection model in real-time, taking input from a webcam, making predictions, and displaying the results live on the screen. The real-time demonstration of the code has been shown in Fig. 09.

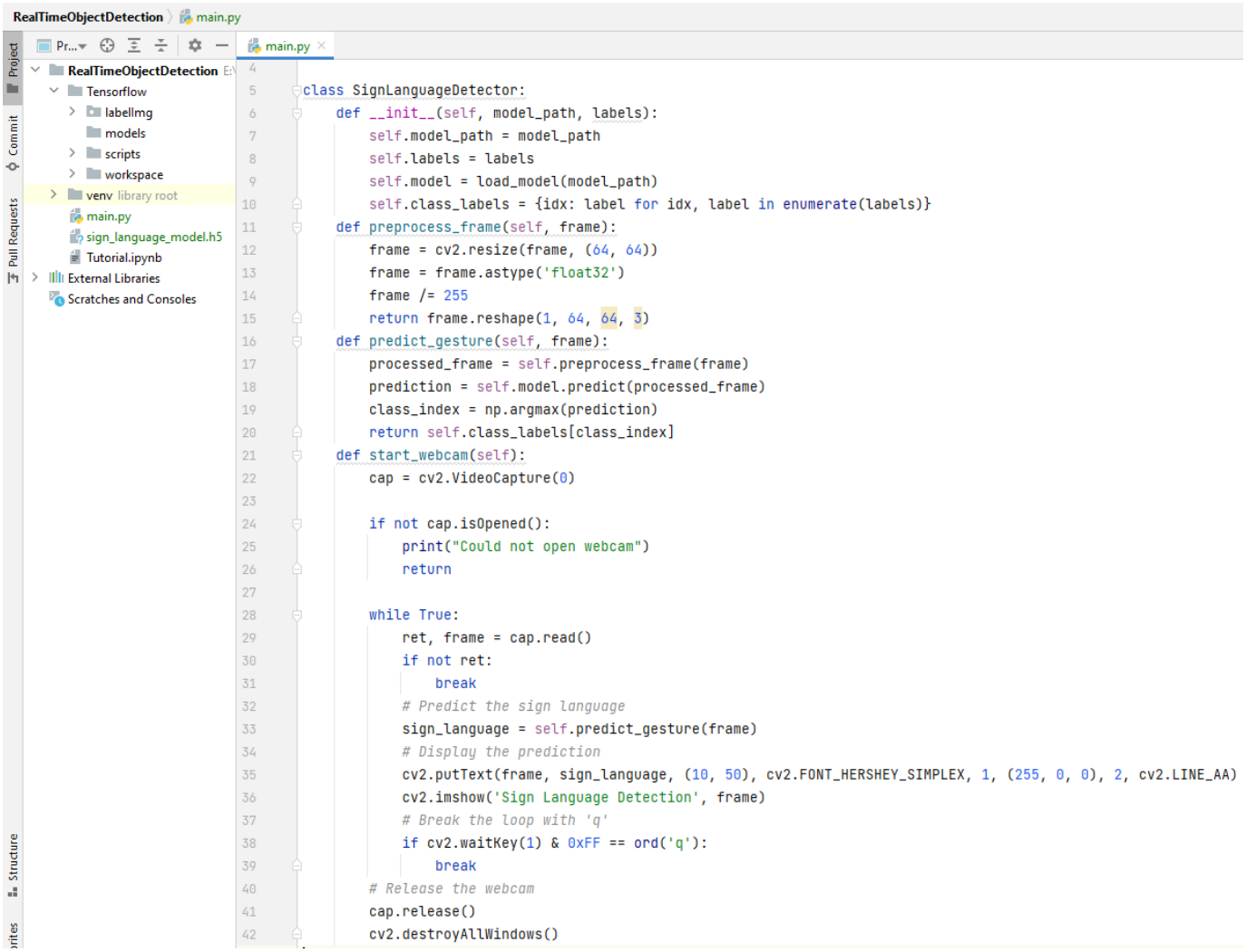

Fig. 08: Code for Testing in real-time

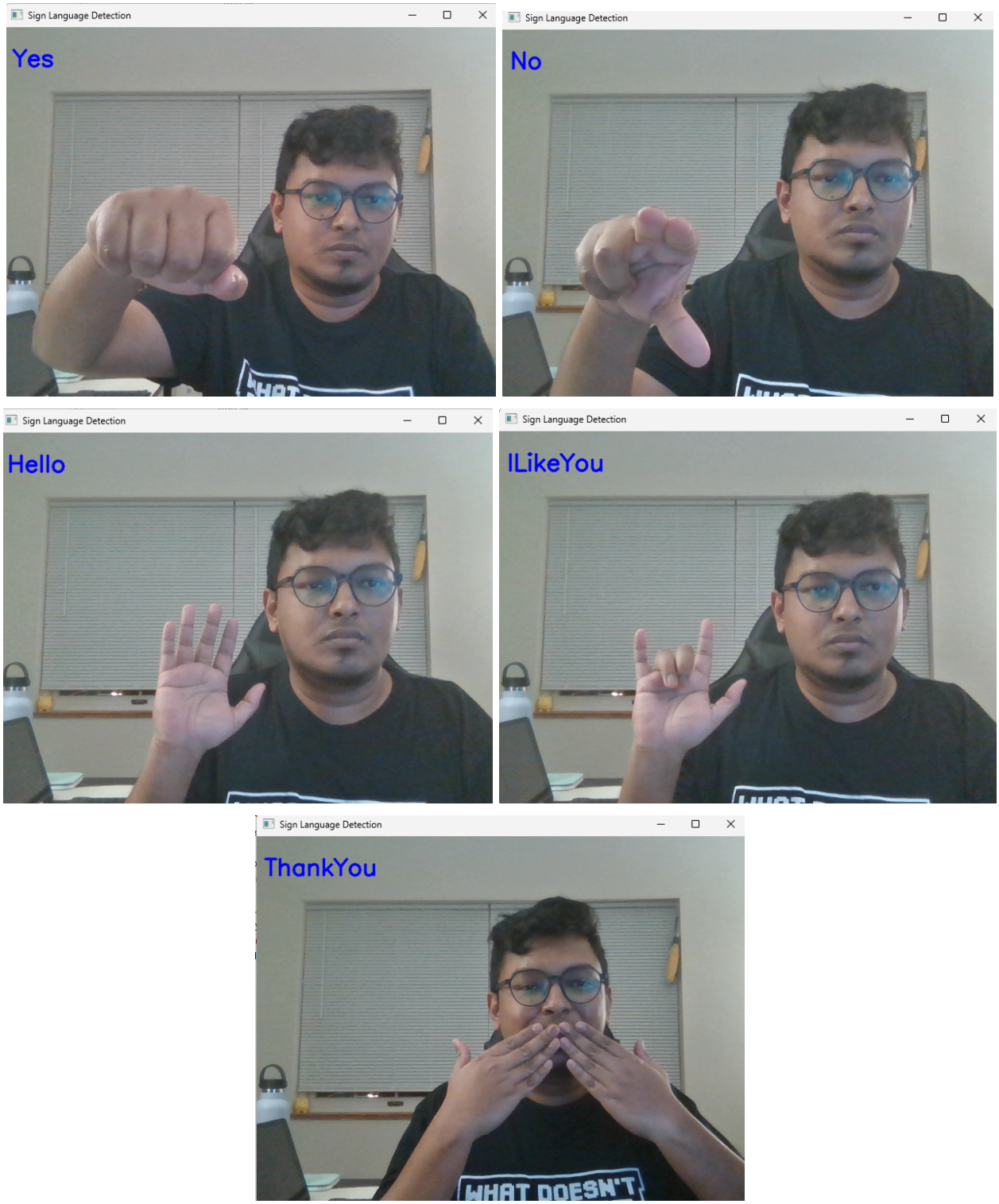

Fig. 09: Test Case Demonstration

The Source Code of the entire project will be found [here]

Software Engineering Phases

For the development of a real-time sign language detection system, the project can be structured into distinct phases following classic software engineering principles. These phases include Definition, Development, Verification, Maintenance, along with overarching Umbrella Activities that span across all these phases. Here's a detailed breakdown:

1. Definition Phase

- Requirement Analysis: This initial stage involves gathering and analyzing the requirements for the sign language detection system. It includes understanding the specific signs ('Hello', 'Yes', 'No', 'Thank You', and 'I Like You') to be detected, the operational environment, user needs, and hardware constraints (like webcam specifications).
- System and Software Design: Based on the requirements, the system's overall architecture and model design are planned. This involves deciding on the technological stack, the machine learning model to be used for detection, and the integration with the webcam for real-time processing.

2. Development Phase

- Implementation and Unit Testing: During this phase, the actual coding of the system takes place. The development of the machine learning model, the user interface, and the integration with the webcam are the key activities. Alongside coding, unit testing is performed to ensure that individual units or components of the software work correctly.
- Integration and System Testing: After unit development, the components are integrated to form a complete system. System testing is then performed to verify that the integrated system functions as intended and meets the specified requirements.

3. Verification Phase

- Validation: This step involves ensuring that the system meets all user needs and that it performs accurately in real-time sign language detection. It includes testing the software in real-world scenarios and checking its performance, accuracy, and usability.

4. Maintenance Phase

- Operational Maintenance: After deployment, the system enters the maintenance phase, where it is updated, modified, and corrected as required over time. This includes fixing any bugs that arise, updating the system for compatibility with new hardware or software, and adding new features or enhancements. During the deployment, I found a few bugs regarding the detection of gestures in real time, and I fixed them later.

5. Umbrella Activities

- Project Management: Throughout all phases, effective project management is essential. This includes planning, scheduling, risk management, and resource allocation.
- Quality Assurance: Ensuring the quality of the software is an ongoing process. It involves setting and adhering to standards, conducting formal technical reviews, and ensuring compliance with regulatory requirements.
- Documentation: Comprehensive documentation is maintained throughout the project lifecycle. This includes documenting requirements, design, code, test cases, and user manuals.
- Configuration Management: As the software evolves, configuration management ensures control over changes to the system's configuration. This includes version control and maintaining the integrity of the system's performance over time.

Each phase is crucial for the successful development and deployment of the real-time sign language detection system, and the umbrella activities provide a supportive framework to ensure the project's overall success.