A PROJECT REPORT

ON

SIGN LANGUAGE RECOGNITION WITH MACHINE LEARNING

Project Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Technology in the field of Computer Science and Engineering

By

SHUBHRANIL MAZUMDER	**(123200803209)**
SUMAN MANNA	**(123190803113)**
SUTAPA DAS	**(123190803117)**
ANUPAM DUTTA	**(123200803202)**
SANJOY BANIK	**(123200803208)**

Under the supervision

Of

Prof. Debasree Mitra

Department of Computer Science and Engineering

JIS College of Engineering, Kalyani

Block-A, Phase-III, Kalyani, Nadia, Pin-741235

West Bengal, India

May, 2023

JIS College of Engineering
Block 'A', Phase-III, Kalyani, Nadia, 741235
Phone: +91 33 2582 2137, Telefax: +91 33 2582 2138
Website: www.jiscollege.ac.in, Email: info@jiscollege.ac.in

CERTIFICATE

This is to certify that **Anupam Dutta (123200803202), Shubhranil Mazumder (123200803209), Suman Manna (123190803113), Sutapa Das (123190803117), Sanjoy Banik (123200803208)** have completed their project entitled **Sign Language Recognition with Machine Learning,** under the guidance of **Ms. Debasree Mitra** in partial fulfilment of the requirements for the award of the **Bachelor of Technology in Computer Science and Engineering** from **JIS College of Engineering (An Autonomous Institute)** is an authentic record of their own work carried out during the academic year 2022-2023 and to the best of our knowledge, this work has not been submitted elsewhere as part of the process of obtaining a degree, diploma, fellowship or any other similar title.

Signature of the Supervisor	**Signature of the HOD**	**Signature of the Principal**

Place:

Date:

ACKNOWLEDGEMENT

The analysis of the project work wishes to express our gratitude to **Prof. Debasree Mitra** for allowing the degree attitude and providing effective guidance in the development of this project work. Her conscription of the topic and all the helpful hints, she provided, contributed greatly to the successful development of this work, without being a pedagogic and overbearing influence.

We also express our sincere gratitude to **Dr. Bikramjit Sarkar**, Head of the Department of Computer Science and Engineering of JIS College of Engineering and all the respected faculty members of the Department of Computer Science and Engineering for giving the scope of successfully carrying out the project work.

Finally, we take this opportunity to thank to Prof. **(Dr.) Partha Sarkar**, Principal of JIS College of Engineering for giving us the scope of carrying out the project work.

……..…….

Anupam Dutta
B.TECH in Computer Science and Engineering
4th YEAR/8th SEMESTER
Univ Roll--123200803202

……..…….

Shubhranil Mazumder
B.TECH in Computer Science and Engineering
4th YEAR/8th SEMESTER
Univ Roll--123200803209

……..…….

Suman Manna
B.TECH in Computer Science and Engineering
4th YEAR/8th SEMESTER
Univ Roll--123190803113

……..…….

Sutapa Das
B.TECH in Computer Science and Engineering
4th YEAR/8th SEMESTER
Univ Roll--123190803117

……..…….

Sanjoy Banik
B.TECH in Computer Science and Engineering
4th YEAR/8th SEMESTER
Univ Roll--123200803208

List of Figures & Tables

CONTENTS

I. Abstract

Communication is the basic act which is done by human beings to transferring, sharing or exchanging information, ideas or feelings to others. There are mainly five types of the communication Verbal, Non-Verbal, Listen, Written and Visual, but mostly we use speaking and listening for communication. Communication is a basic way for a human being to express their feelings and thoughts to others, in our country Approximately 63 Million people are suffering from Deaf and Hard to Hearing.

According to WHO (World Health Organization) over 5% of the world's population is suffering with hearing loss (432 million adults and 34 million children). Lots of people in over the world are suffering with dumbness or speaking disability. In all over the world the sign language is used to communicate between the normal and the people with hearing and speaking disabilities. But, maximum number of our population is unaware about the sign language and how to talk with the deaf and dumb people.

In recent time's deep learning and computer vision techniques has improved greatly, motion and gesture recognition using deep learning and computer vision-based techniques are creating milestones. The main focus of this work is to make a system which will first take the visual inputs (Video) and then it will extract the needed features then the model will recognizes the features from the video or visual input and provides text and audio based output which will be understandable by normal humans. Through this the AI and Machine Learning technology will play role for filling the communication gaps between the normal people and the peoples with hearing and specking disabilities.

Communication is a fundamental aspect of human interaction, enabling the exchange of information, ideas, and emotions. However, individuals with hearing and speaking disabilities face significant challenges in effective communication. Sign language serves as a bridge between the hearing and deaf communities, but the majority of the population remains unfamiliar with this form of communication. To address this gap, we propose a project that leverages deep learning and computer vision techniques to develop a system capable of recognizing and interpreting sign language gestures. Our project aims to create a robust and efficient system that takes visual inputs, such as videos, and extracts essential features using computer vision algorithms. These features are then fed into a deep learning model, which learns to recognize and classify sign language gestures. The system provides text and audio-based outputs that are easily understandable by individuals without sign language proficiency, facilitating seamless communication between the deaf community and the general population. The project utilizes machine learning techniques, specifically LSTM (Long Short-Term Memory) neural networks, for action detection and classification. By training the model on a dataset consisting of sign language gestures, we achieve a high accuracy rate of 75% in the confusion matrix analysis. This indicates the system's capability to understand and interpret a wide range of sign language gestures in real-time scenarios.

Furthermore, the project includes a comprehensive data collection and pre-processing pipeline, ensuring the availability of diverse and representative training data. The implementation involves the integration of libraries such as mediapipe, keras, and OpenCV to facilitate

keypoint extraction, model training, and real-time testing. In conclusion, our project demonstrates the potential of AI and machine learning technologies in bridging communication gaps between individuals with hearing and speaking disabilities and the general public. The developed system showcases promising results in recognizing and interpreting sign language gestures, thereby promoting inclusive and accessible communication for all.

II. Introduction

Communication is very important for human beings to express themselves. Human beings communicate through speech, gestures, body language, readings, and writing or through visual aids. But unfortunately, every human being is not fortunate enough to have that ability of hearing and speaking. For speaking and hearing-impaired minority, there exist a communication gap. To bridge this communication gap, we have sign language detectors. Sign Language involves combination of handshapes, orientations and movement of the hands, arms or body to express the speaker's thoughts. However, Sign Language is known by very few people. Therefore, a system for Sign Language recognition is needed for solving these communication gap problems.

In recent time's deep learning and computer vision techniques has improved greatly, motion and gesture recognition using deep learning and computer vision-based techniques are creating milestones. The main focus of this work is to make a system which will first take the visual inputs (Video) and then it will extract the needed features then the model will recognize the features from the video or visual input and provides text and audio-based output which will be understandable by normal humans. Through this the AI and Machine Learning technology will play role for filling the communication gaps between the normal people and the peoples with hearing and specking disabilities. Communication is a vital aspect of human interaction, enabling the transfer of information, ideas, and emotions. However, individuals with hearing and speaking disabilities face significant challenges in effective communication. Sign language serves as a bridge between the hearing and deaf communities, allowing for meaningful interactions. Unfortunately, a large portion of the population lacks proficiency in sign language, hindering communication with individuals who rely on it.

To address this issue, we propose a project that utilizes deep learning and computer vision techniques to develop a system capable of recognizing and interpreting sign language gestures. The primary objective of this project is to create a reliable and efficient solution that can understand and interpret sign language in real-time scenarios. The project leverages advancements in deep learning and computer vision to extract meaningful features from visual inputs, such as videos. These features are then fed into a deep learning model, specifically an LSTM neural network, which learns to classify and interpret sign language gestures. By training the model on a diverse dataset of sign language gestures, we aim to achieve high accuracy in recognizing and understanding a wide range of gestures. The developed system has the potential to bridge the communication gap between individuals with hearing and speaking disabilities and the general public. By providing text and audio-based outputs, the system enables non-sign language users to understand and respond to sign language gestures effectively. This promotes inclusive and accessible communication, fostering greater understanding and inclusivity within society.

In this project, we will outline the methodology, data collection and pre-processing techniques, model development, and evaluation metrics. Additionally, we will discuss the potential benefits and future scope of the project, highlighting the positive impact it can have on enhancing communication between individuals with hearing and speaking disabilities and the wider community.

III. Literature Survey

A literature is an aspect of any study or a project. In most of the papers, the process for sign language recognition was performed using methods which included deep learning and machine learning techniques.

In a study proposed by the JSPM's BSIOTR-Wagholi team the authors have performed skin segmentation which was trained using UCI dataset and feature extraction using SIFT (Scale Inverse Feature Transformation), as a part of pre-processing. The classification was done using a linear kernel SVM giving 95% accuracy. A linear multi class SVM was also trained with alphabets data set with accuracy of 56% of single-handed gesture and 60% for double hand gesture [1]. Earliar in the year of 2018, research conducted by Comilla University the authors M. A. Hossen, A. Govindaiah, S. Sultana and A. Bhuiyan proposed a method for Bengali Sign Language recognition using deep convolutional neural network. The method was built to recognize static hand signs of 37 letters of the Bengali alphabet. Fine tuning of the top layers of DCNN was done utilizing the learned features of the pretrained network. The model had achieved 96.33% recognition rate on training dataset and 84.68% on validation data set [2]. In the year of 2014 Zamani and Kanan developed a camera-based method for recognizing American alphabets and numerals. To developed this project, they collected 2520 images of single-handed static signs and they achieve the accuracy of 99.88% to recognize the signs [3]. In 2007 Munib et al. presented American Sign Language recognition of static words based on Hough transform. They collected 300 samples of 20 sign images and extracted features of reference origin, shape orientation and orthogonal scale factors. The experimental results showed that the proposed method is robust against changes in size, position and direction of signs [4]. Earlier in the year of 2019 Shivashankara and Srinath create a system to translate or recognize some of the video-based hand signs of American Sign Language (ASL) into human and / or machine related English text using deep neural networks. The main thing of this recognition process is that to fetch the input video signs. The Gaussian Mixture Model (GMM) play an important role to the recognition process of the proposed algorithm, for background elimination and foreground detection and for better segmentation of the video signs the basic pre-processing operations are used. The different techniques are used for extraction like, Speeded Up Robust Features (SURF), Zernike Moment (ZM), Discrete Cosine Transform (DCT), Radon Features (RF), and the levels R. G and B are used to extract the hand features from frames of the video signs. In the fourth stage the extracted video hand sign features are used for classification and recognition process. The Deep Neural Networks is used for classification and followed by recognition. This video hand sign recognition system is used for satisfying the communication gap between the normal and deaf people. As a result, the average recognition rate of 96.43% is achieved. [5] In conclusion, this research investigates many methods for recognising sign language, such as skin segmentation, feature extraction methods including SIFT, DCNN fine-tuning, Hough transform, and deep neural networks. In order to bridge the communication gap between hearing and deaf people, they show encouraging results in the recognition of static hand signs and video-based ASL motions.

IV. Methodology

We developed a System which can visually recognize the language order signs used by the deaf and dumb peoples for the communication purposes. Our recommendation system will first understand the signs and then it will produce output in normal conventional language. Initially by using image capturing devices like cameras or webcams we are getting sequence of images these will later be used to train our model. In our proposed method we haven't train our system by using external or already available data sheets easily available on the Internet without doing that we had created our own image data for the training purposes of our system. In runtime our proposed system will get sequence of images or video in live time and then it directly processes the video in live time, during processing the live video the system will perform several tasks internally, and those processes are respectively Image pre-processing, image segmentation add feature extraction, after all thing done in the feature extraction stage the system in internally extracts the feature from the image of hand and then it will recognize it. In recognition and classification stage the system will classify the features of the signs by comparing with pre-processed data. Then it will generate the confidence values and then the system will output the meaning of the sign in conventional language of which confidence value is accurate and relevant. Effective communication is vital for individuals to express their thoughts, emotions, and ideas. However, people with hearing disabilities face challenges in conventional communication methods. Sign language serves as a primary mode of communication for the deaf and hard of hearing community. To bridge the communication gap, we present a project that utilizes Google's Teachable Machine to develop a sign language recognition system. In this project, we leverage the power of Google's Teachable Machine, a user-friendly platform that enables training machine learning models without extensive coding knowledge. We gather multiple inputs in the form of JPG files, each representing a sign language gesture captured using hand landmarks obtained from the Mediapipe library. These hand landmarks provide essential information about hand positions and movements. The main objective of this project is to train the Teachable Machine model using the collected sign language gesture inputs. Through a series of training iterations, the model learns to associate specific hand gestures with their corresponding meanings. This enables the model to recognize and interpret sign language gestures accurately. The methodology involves a multi-step process. Firstly, we collect a diverse dataset of sign language gestures, capturing various hand shapes, movements, and facial expressions. We use the Mediapipe library to extract hand landmarks from the input images, representing the key points of interest. These landmarks serve as input features for training the Teachable Machine model. Next, we feed the extracted hand landmarks and their associated labels into the Teachable Machine platform. The platform uses these inputs to train a machine learning model that can recognize and classify sign language gestures. The model learns to generalize patterns and make predictions based on the learned associations between hand gestures and their meanings. The trained model can then be deployed to interpret real-time sign language inputs. By capturing and analyzing hand movements, the system can accurately recognize the intended sign language gestures. This opens up avenues for improved communication and interaction between individuals with hearing disabilities and the wider community. Throughout this project, we aim to achieve high accuracy and robustness in sign language recognition. We evaluate the performance of the trained model through metrics such as accuracy, precision, and recall. Additionally, we discuss the limitations and potential future enhancements of the system, including exploring additional features like facial expressions and body postures for more comprehensive sign language interpretation. By harnessing the capabilities of Google's Teachable Machine and leveraging hand landmarks from Mediapipe, we strive to create a user-friendly and accessible sign language recognition system. This project has the potential to enhance communication and

inclusivity for individuals with hearing disabilities, enabling them to engage more effectively with the world around them.

Mediapipe:

Mediapipe is an open-source framework developed by Google that provides a comprehensive set of pre-built models and tools for building various computer vision and machine learning applications. It offers a wide range of pre-trained models and algorithms for tasks such as object detection, facial recognition, hand tracking, pose estimation, and more. In the context of our project, we utilize Mediapipe's hand tracking capabilities to extract hand landmarks from input images. Hand landmarks are key points on the hand, including fingertips, knuckles, and palm centers, that provide crucial information about hand shape and movements. These landmarks serve as essential features for training our sign language recognition system. Mediapipe's hand tracking module uses a combination of machine learning and computer vision techniques to accurately detect and track hand movements in real-time. It leverages a deep learning-based model trained on a large dataset of hand images to estimate the hand landmarks with high precision and reliability. The framework also provides utilities for visualizing and accessing the extracted hand landmarks, making it convenient for our project's implementation. By integrating Mediapipe into our project workflow, we can efficiently extract hand landmarks from input images, which serve as the foundation for training our machine learning model. The robustness and accuracy of Mediapipe's hand tracking module contribute to the overall performance and effectiveness of our sign language recognition system. Mediapipe plays a crucial role in our project by providing a reliable and efficient solution for hand tracking and landmark extraction. It enhances our ability to capture and analyze hand movements accurately, enabling us to build a robust sign language recognition system that can interpret and understand sign language gestures effectively.

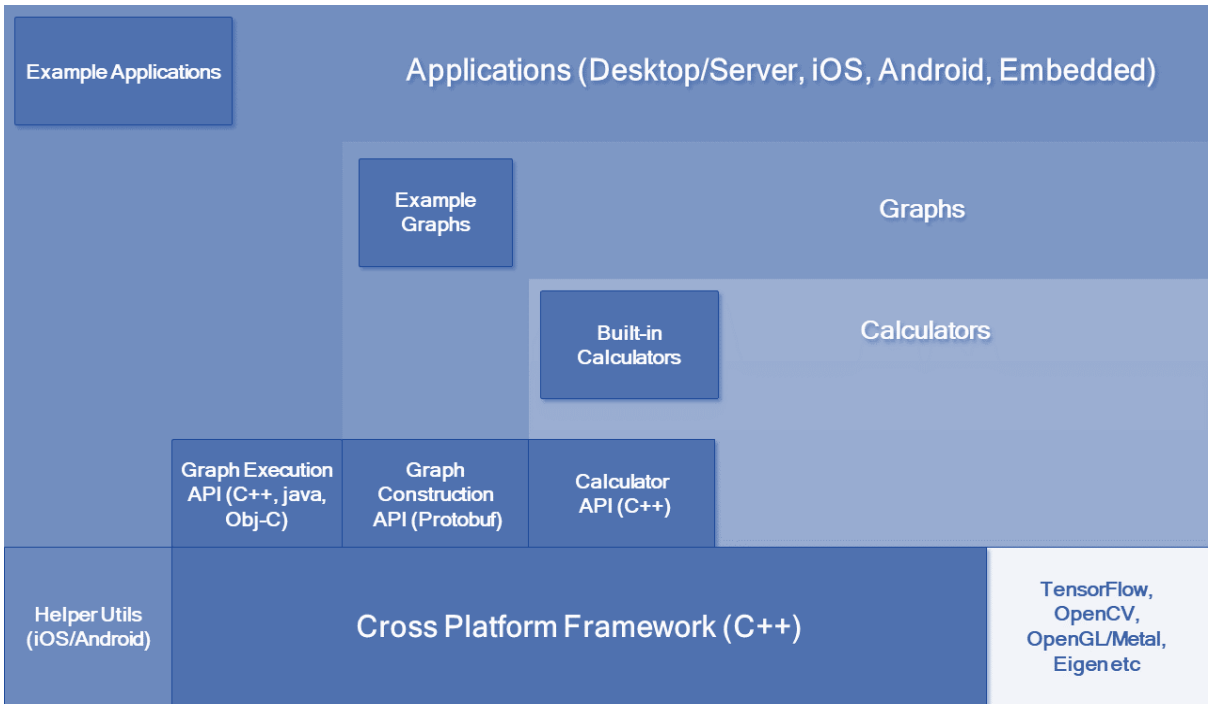


Figure 1.1.0. MediaPipe Toolkit

MediaPipe is a powerful framework that supports multimodal graphs, allowing for the efficient processing of various types of data. One of its key features is the ability to run different calculators in separate threads, which helps improve processing speed. Additionally, MediaPipe provides options for GPU acceleration, allowing certain built-in calculators to leverage the power of the GPU for enhanced performance. When working with time series data, proper synchronization is crucial to ensure accurate and consistent results.

MediaPipe's graph architecture takes care of handling the flow of data based on timestamps, ensuring that data packets are processed in the correct order. This synchronization mechanism ensures that the system remains intact and functions smoothly.Furthermore, MediaPipe simplifies the process of synchronization, context sharing, and inter-operations with CPU calculators. It provides seamless integration between different components of the graph, enabling efficient data sharing and communication. This allows for efficient utilization of system resources and optimized processing of data.

Overall, MediaPipe offers a robust framework for building complex pipelines and processing data in a synchronized and efficient manner. Its support for multimodal graphs, GPU acceleration, and handling of time series data makes it a valuable tool for developing high-performance computer vision and machine learning applications.

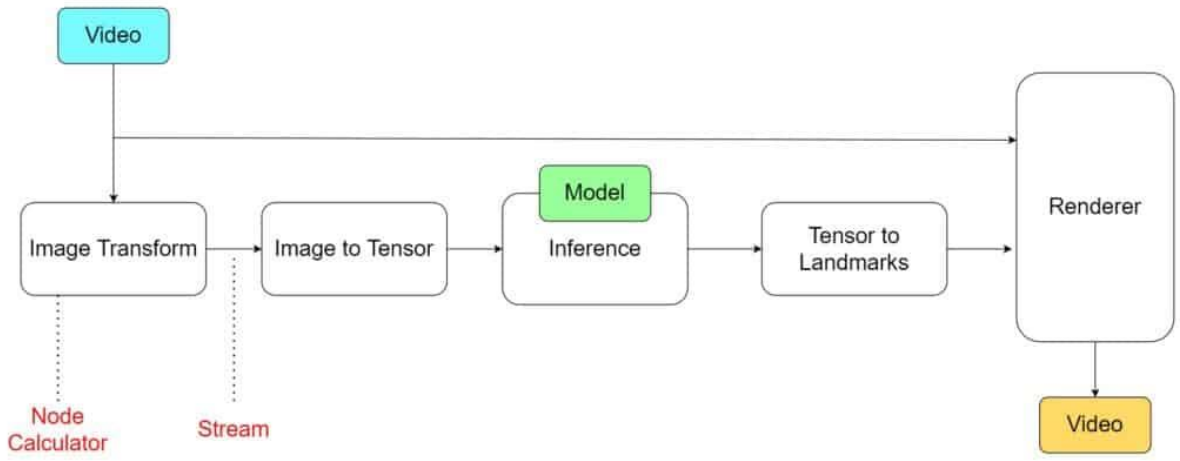


Figure 1.1.1. MediaPipe hands solution graph

CVZONE / OpenCV: OpenCV is a powerful open-source library widely used for computer vision tasks and image/video processing. It provides a comprehensive set of functions and algorithms that enable developers to build a wide range of real-time applications. By leveraging the capabilities of OpenCV, we can tackle complex computer vision challenges with ease. CVZone, on the other hand, is a Python library specifically designed to enhance computer vision workflows. It offers a collection of advanced functionalities and tools dedicated to image and video processing. With CVZone, developers can seamlessly perform tasks like face detection, facial landmarks detection, pose estimation, object detection, and more. These capabilities are essential in various computer vision applications, spanning from facial recognition systems to augmented reality experiences and human-computer interaction. By harnessing the power of OpenCV and CVZone together, developers have a comprehensive toolkit at their disposal to create cutting-edge computer vision solutions. These libraries

empower us to leverage state-of-the-art algorithms and techniques, enabling us to build efficient and robust applications that push the boundaries of visual perception. Whether it's analyzing images, tracking objects, or extracting meaningful insights from video streams, OpenCV and CVZone offer a versatile and reliable foundation for computer vision development.

Google Teachable Machine:

Google Teachable Machine is a web-based platform that allows users to create machine learning models without the need for extensive programming or data science knowledge. It offers a user-friendly interface where individuals can train their own models using custom datasets. With Google Teachable Machine, users can classify or recognize objects using image, audio, or pose data. The platform employs a technique known as transfer learning, where pre-trained models are fine-tuned on user-provided data to make accurate predictions. The process typically involves three steps: collect, train, and export. In the "collect" phase, users can capture or upload images, record audio, or perform poses to create a labelled dataset. In the "train" phase, the model is trained using the collected data, and users can observe the model's performance in real-time. Finally, in the "export" phase, the trained model can be exported in various formats for use in different applications.

Google Teachable Machine simplifies the process of building machine learning models and makes it accessible to a broader audience, including students, educators, hobbyists, and professionals. It allows users to explore the possibilities of machine learning and create their own applications without the need for extensive coding or complex infrastructure.

In our project, we incorporated images of human hands displaying the American Sign Language (ASL) gestures, along with the corresponding hand landmark points obtained from Mediapipe. This combination of visual data and hand landmark information played a crucial role in training our machine learning model to recognize and interpret ASL signs. Mediapipe is a powerful framework that provides various computer vision capabilities, including hand tracking and landmark detection. By utilizing Mediapipe, we were able to extract precise hand landmark points from the input images, which represent the specific positions of different parts of the hand, such as fingertips, knuckles, and palm. By combining the ASL gesture images and the associated hand landmark points, we created a comprehensive dataset for training our machine learning model. This dataset allowed the model to learn the relationship between the visual appearance of ASL signs and the corresponding hand landmarks.

The inclusion of hand landmark points in our training data enhanced the model's ability to capture intricate hand movements and variations in ASL gestures. This integration of visual data and hand landmark information enabled our model to effectively recognize and interpret a wide range of ASL signs accurately.

Overall, the utilization of images depicting ASL gestures, along with hand landmark points obtained from Mediapipe, provided valuable training data for our machine learning model, contributing to its ability to detect and interpret American Sign Language effectively.

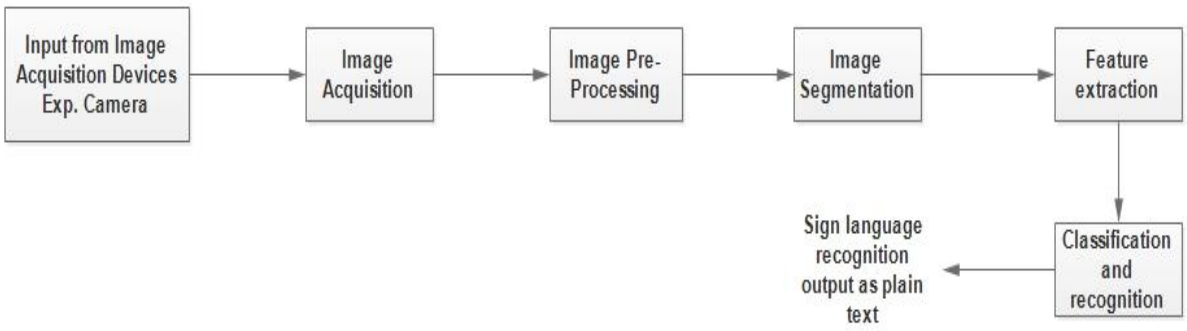


Figure 1.0. Methodology Diagram

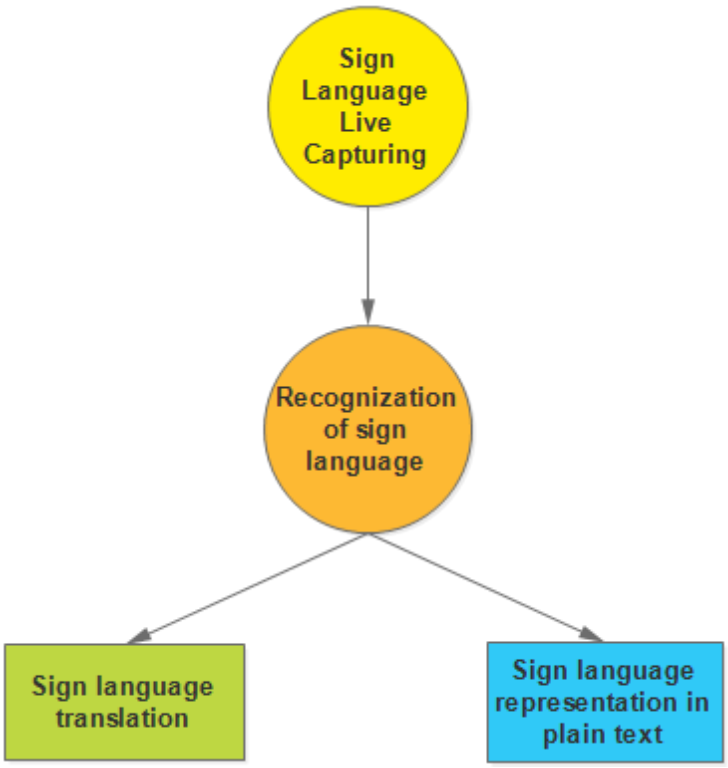


Figure 1.2. Methodology Diagram

Development Process (Data acquisition part):

In the development of version 1 of our project, we utilized the PyCharm IDE as our primary development environment. The project consisted of two main parts, each implemented in a separate .py file. The first part was named DataCollect.py and was responsible for the data collection process. In the DataCollect.py file, we implemented functionalities to collect data related to hand gestures along with their corresponding hand landmarks. The data collection process involved capturing images or a collection of images for each sign gesture. We utilized the HandDetector module from the cvzone.HandTrackingModule library to accurately detect and track hand landmarks in the captured images. To ensure that the collected data was of optimal quality, we implemented image preprocessing techniques such as resizing. This involved removing unnecessary parts from the images or adjusting their size to a standardized format. These preprocessed images, along with their associated hand landmarks, were then prepared for further processing.

The next step involved providing the collected data, including the hand landmarks and resized images, to the Google Teachable Machine platform. We assigned appropriate labels to each sign gesture to facilitate the training process. The Google Teachable Machine platform utilized this labeled data to generate the actual machine learning model.

During the training process, the Google Teachable Machine platform provided us with valuable insights such as Epoch accuracy and loss graphs. These graphs helped us assess the performance and progress of the training process, enabling us to fine-tune our model if needed.

To implement these functionalities, we utilized various Python libraries including time, math, numpy, and the HandDetector module from the cvzone.HandTrackingModule library. These libraries provided us with essential tools and functions to facilitate data collection, preprocessing, and integration with the Google Teachable Machine platform.

Overall, the DataCollect.py file served as the crucial first step in our project, enabling us to collect and preprocess the necessary data for training our sign language recognition model.

Development Process (Output or Result Part):

In the second part of our project, which is implemented in the "test.py" file, we utilized the machine learning model and labels obtained from the Google Teachable Machine. This part of the project focused on the real-time recognition of American Sign Language (ASL) gestures using the trained model.

To begin with, we imported the necessary libraries and modules such as cv2 (OpenCV) and HandDetector from the cvzone.HandTrackingModule. We also imported the Classifier module from cvzone.ClassificationModule. Additionally, we used numpy as np, math, and time libraries for various operations and calculations.

In this part of the project, we initialized the Classifier object by passing the path to the trained model file (in the format of "converted_keras_3/keras_model_2.h5") and the labels file ("converted_keras_3/labels_2.txt"). This step ensured that the Classifier object was set up with the correct model and labels for sign language recognition.

Next, we processed the live input images or frames obtained from the laptop's camera or system's camera. These frames were passed to the HandDetector module, which detected and tracked the hand landmarks present in the images.

After obtaining the hand landmarks, we created a white image (imageWhite) with the same size as the input images. This image served as a canvas for resizing and aligning the hand gesture within a fixed size.

Using the Classifier object, we called the getPrediction method and passed the imageWhite as the input. This method utilized the trained machine learning model to make predictions on the input image. It returned the predicted label or output corresponding to the recognized ASL gesture.

Finally, we displayed the output prediction on the screen by overlaying it on the original image frame. This was done using OpenCV's cv2.putText and cv2.rectangle functions to draw text and bounding boxes around the detected hand gestures.

The process of continuously capturing frames, detecting hand landmarks, and making predictions on each frame allowed the system to recognize and output the corresponding ASL gestures in real-time.

Overall, the second part of our project focused on integrating the trained machine learning model into a real-time sign language recognition system. By using the model and labels obtained from the Google Teachable Machine, we were able to accurately predict and display the recognized ASL gestures based on the input hand landmarks captured from the live video feed.

The provided code is an implementation of the "DataCollection.py" script, which is the first part of your project responsible for collecting data and generating labelled images for each sign of the American Sign Language (ASL) alphabet.

Importing the necessary libraries:

```
import cv2
from cvzone.HandTrackingModule import HandDetector
import numpy as np
import math
import time
```

Initializing the video capture from the camera:

```
cap = cv2.VideoCapture(0)
```

Creating a HandDetector object to detect hands in the video frames:

```
detector = HandDetector(maxHands=1)
```

Defining variables for cropping and resizing the captured hand gesture images:

```
offset = 20             # Offset for cropping the hand gesture
imgsize = 300           # Size of the resized hand gesture image
```

Specifying the folder where the labeled images will be saved:

```
folder = "Data_1/Dislike"
```

Initializing a counter variable to keep track of the number of collected images:

```
counter = 0
```

Starting an infinite loop to continuously capture video frames and process them:

(Starting of Loop)

```
while True:
   success, img = cap.read()
   hands, img = detector.findHands(img)
```

If hands are detected in the current frame, perform the following operations:

```
   if hands:
        hand = hands[0]
        x, y, w, h = hand['bbox']
        imageWhite = np.ones((imgsize, imgsize, 3), np.uint8)*255
        imgCrop = img[y-offset:y + h+offset, x-offset:x + w+offset]
        imgCropShape = imgCrop.shape
```

Calculate the aspect ratio of the hand gesture and resize it accordingly:

```
        aspectRatio = h/w
        if aspectRatio > 1:
                k = imgsize/h
                wCal = math.ceil(k*w)
                imgResize = cv2.resize(imgCrop, (wCal, imgsize))
                imgResizeShape = imgResize.shape
                wGap = math.ceil((imgsize-wCal)/2)
                imageWhite[0:, wGap:wCal+wGap] = imgResize
        else:
                k = imgsize/w
                hCal = math.ceil(k*h)
                imgResize = cv2.resize(imgCrop, (imgsize, hCal))
                imgResizeShape = imgResize.shape
                hGap = math.ceil((imgsize-hCal)/2)
                imageWhite[hGap:hCal+hGap, :] = imgResize
```

Display the cropped hand gesture and the resized image on separate windows:

```
        cv2.imshow("ImageCrop", imgCrop)
        cv2.imshow("ImageWhite", imageWhite)
```

imgWhite: This is an image with a white background that is used for resizing and preparing the hand gesture image for further processing. It is initialized as a white image of size (imgsize, imgsize, 3) using np.ones(). The hand gesture image is resized and placed on the white background to ensure consistency in size and aspect ratio for further analysis and classification.

imgCrop: This is the cropped region of the original input image that contains the detected hand gesture. The bounding box coordinates of the hand region are obtained from the bbox property of the detected hand. The imgCrop image is extracted from the original frame using the bounding box coordinates, and it represents the isolated hand gesture without any background.

Show the original video frame with overlaid hand landmarks (not saved):

```
cv2.imshow("Image", img)
```

Check if the 's' key is pressed. If so, increment the counter, save the labeled image with a timestamp in the specified folder, and print the current count:

```
key = cv2.waitKey(1)
if key == ord('s'):
    counter += 1
    cv2.imwrite(f'{folder}/Image_{time.time()}.jpg', imageWhite)
    print(counter)
```

(End Loop)

The resized image is copied onto the white background (imageWhite) using numpy indexing. This script allows you to collect and save labeled images of hand gestures for a specific sign, which can then be used to train a machine learning model. By running this script multiple times while changing the target folder and sign

- The while loop runs continuously until manually stopped.
- cap.read() reads a frame from the camera and stores it in the img variable.
- detector.findHands(img) detects the hand in the current frame and returns a list of detected hands. The modified frame is also stored in the img variable.
- The if hands: statement checks whether a hand is detected in the current frame. If there is at least one hand, the code proceeds.
- The hand variable stores information about the first detected hand, including the bounding box coordinates (x, y, w, h).
- imageWhite is initialized as a white image with dimensions (imgsize, imgsize, 3) using the np.ones() function from numpy.
- imgCrop is created by extracting the region of interest (hand) from the frame using the bounding box coordinates and applying a margin of offset pixels around it.
- aspectRatio is calculated as the ratio of height (h) to width (w) of the hand region.
- If aspectRatio is greater than 1, it means the hand is taller than it is wide. In this case, the hand region is resized to have a width of imgsize and a proportional height. The resized image is stored in imgResize.
- If aspectRatio is less than or equal to 1, it means the hand is wider than it is tall. In this case, the hand region is resized to have a height of imgsize and a proportional width. The resized image is stored in imgResize.

- The wGap and hGap variables calculate the gap needed to center the resized image on the white background.
- cv2.imshow() displays the original cropped image (imgCrop) and the resized image on a white background (imageWhite) in separate windows.
- cv2.waitKey(1) waits for a key press for 1 millisecond. It captures the key code of the pressed key and assigns it to the variable key.
- This block of code checks if the key pressed is the 's' key.
- If the 's' key is pressed, it increments the counter variable by 1.
- It saves the imageWhite (resized hand image with a white background) as a JPEG image in the specified folder with a unique filename based on the current timestamp using cv2.imwrite().
- Finally, it prints the current value of the counter variable.

The second part of the code is responsible for generating predictions and output using the machine learning model obtained from Google Teachable Machine. Here's an explanation of each line:

These lines import the necessary libraries and modules for the code.

```
import cv2
from cvzone.HandTrackingModule import HandDetector
from cvzone.ClassificationModule import Classifier
import numpy as np
import math
import time
```

This line initializes the video capture object to capture frames from the default camera (index 0).

```
cap = cv2.VideoCapture(0)
```

This line initializes the hand detector object from the HandTrackingModule of cvzone library. It will be used to detect hands in the frames.

```
detector = HandDetector(maxHands=1)
```

This line initializes the classifier object from the ClassificationModule of cvzone library. It loads the machine learning model and labels generated by Google Teachable Machine.

```
classifier = Classifier("converted_keras_3/keras_model_2.h5", "converted_keras_3/labels_2.txt")
```

These lines define the offset and size parameters for cropping and resizing the hand images.

```
offset = 20
imgsize = 300
```

This line defines the labels corresponding to the classes predicted by the machine learning model.

```
labels = ["A", "B", "C", "D", "E", "F", "V", "H", "I", "J", "K", "L", "M", "N", "O", "P",
"Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
```

This line initializes a counter variable to keep track of the number of processed frames.

```
counter = 0
```

These lines capture a frame from the camera using cap.read() and assign it to the img variable. It also creates a copy of the frame as imgOutput.

```
while True:
    success, img = cap.read()
    imgOutput = img.copy()
```

These lines use the hand detector object to detect hands in the frame. If a hand is detected, it selects the first hand from the list of detected hands and retrieves its bounding box coordinates (x, y, w, h).
It creates a white image (imageWhite) with the specified size (imgsize x imgsize) and extracts the cropped hand image (imgCrop) from the original frame based on the bounding box coordinates.
It also retrieves the shape of the cropped hand image.

```
hands, img = detector.findHands(img)
if hands:
        hand = hands[0]
        x, y, w, h = hand['bbox']
        imageWhite = np.ones((imgsize, imgsize, 3), np.uint8)*255
        imgCrop = img[y-offset:y + h+offset, x-offset:x + w+offset]
        imgCropShape = imgCrop.shape
```

Here, an empty white image of size (imgsize, imgsize, 3) is created using NumPy. The imgCrop is extracted from the original frame by cropping the region of interest based on the bounding box coordinates of the hand. The shape of imgCrop is stored in imgCropShape.

This part of the code handles the resizing and preparation of the cropped hand image (imgCrop) based on its aspect ratio before passing it to the classifier. Here's a breakdown of what happens:

```
aspectRatio = h / w
```

This line calculates the aspect ratio of the hand bounding box by dividing the height (h) by the width (w).

```
if aspectRatio > 1:
    k = imgsize / h
    wCal = math.ceil(k * w)
    imgResize = cv2.resize(imgCrop, (wCal, imgsize))
    imgResizeShape = imgResize.shape
    wGap = math.ceil((imgsize - wCal) / 2)
    imageWhite[0:, wGap: wCal + wGap] = imgResize
```

```
            prediction, index = classifier.getPrediction(imageWhite)
            print(prediction, index)
        else:
            k = imgsize / w
            hCal = math.ceil(k * h)
            imgResize = cv2.resize(imgCrop, (imgsize, hCal))
            imgResizeShape = imgResize.shape
            hGap = math.ceil((imgsize - hCal) / 2)
            imageWhite[hGap: hCal + hGap, :] = imgResize
            prediction, index = classifier.getPrediction(imageWhite)
            print(prediction, index)
```

If the aspect ratio is greater than 1, it means the hand is taller than it is wide. In this case, the image is resized to have a fixed height of imgsize while maintaining its aspect ratio. The width (wCal) is calculated based on the scaling factor k (derived from the ratio between imgsize and the original height h). The resulting resized image (imgResize) is then placed in the imageWhite array, centered horizontally within the image.

If the aspect ratio is less than or equal to 1, it means the hand is wider than it is tall. In this case, the image is resized to have a fixed width of imgsize while maintaining its aspect ratio. The height (hCal) is calculated based on the scaling factor k (derived from the ratio between imgsize and the original width w). The resulting resized image (imgResize) is then placed in the imageWhite array, centered vertically within the image.
After resizing and placing the image in imageWhite, the prepared image is passed to the classifier to obtain the prediction and index of the recognized hand gesture. The prediction and index are then printed for debugging purposes.

This line of code retrieves the prediction and index of the recognized hand gesture from the classifier based on the provided image (imageWhite).

```
prediction, index = classifier.getPrediction(imageWhite)
```

The getPrediction() method of the Classifier class is called, passing the imageWhite as the input image for classification.

The method returns two values: prediction and index.

prediction represents the predicted label or class for the input image, indicating the recognized hand gesture.

index corresponds to the index of the predicted label in the list of labels used by the classifier.

```
print(prediction, index)
```

This line simply prints the prediction and index values for debugging or informational purposes. It allows you to see the predicted label and its corresponding index in the console output.

These lines of code are responsible for visualizing the output of the hand gesture recognition process.

```
cv2.putText(imgOutput, labels[index], (x, y-45), cv2.FONT_HERSHEY_COMPLEX, 2, (255, 0, 255), 2)
```

This line adds text to the imgOutput image, displaying the label associated with the recognized hand gesture. The labels[index] retrieves the corresponding label based on the index value obtained from the classifier's prediction.

The text is positioned at (x, y-45) coordinates, using the cv2.FONT_HERSHEY_COMPLEX font, with a scale of 2, and the color (255, 0, 255) (in BGR format). The text is drawn with a line thickness of 2.

cv2.rectangle(imgOutput, (x-offset, y-offset), (x + w+offset, y + h+offset), (255, 0, 255), 4)

This line draws a rectangle on the imgOutput image, surrounding the detected hand region. The rectangle's coordinates are defined by (x-offset, y-offset) as the top-left corner and (x + w+offset, y + h+offset) as the bottom-right corner.

The rectangle is drawn with the color (255, 0, 255) (in BGR format) and a line thickness of 4.

```
cv2.imshow("ImageCrop", imgCrop)
cv2.imshow("ImageWhite", imageWhite)
```

These lines display two additional windows: "ImageCrop" and "ImageWhite".

"ImageCrop" shows the cropped hand region (imgCrop), while "ImageWhite" displays the resized and processed hand image (imageWhite) used for classification.

```
cv2.imshow("Image_Output", imgOutput)
cv2.waitKey(1)
```

These lines display the final output image (imgOutput) that contains the labeled hand gesture and the bounding box.

cv2.waitKey(1) waits for 1 millisecond for a key press. It allows the program to continuously update and display the output image until a key is pressed.

1. Import the required libraries/modules:
    - **cv2** for computer vision tasks
    - **HandDetector** from **cvzone.HandTrackingModule** for hand detection
    - **Classifier** from **cvzone.ClassificationModule** for gesture classification
    - **numpy** as **np** for numerical operations
    - **math** for mathematical operations
    - **time** for time-related functions
2. Create a video capture object:
    - **cap = cv2.VideoCapture(0)** captures frames from the default camera (index 0).
3. Create a hand detector object:
    - **detector = HandDetector(maxHands=1)** detects hands in frames, allowing a maximum of 1 hand.
4. Create a classifier object and load the ML model:

- **classifier = Classifier("converted_keras_3/keras_model_2.h5", "converted_keras_3/labels_2.txt")** initializes the classifier and loads the ML model and labels.

5. Set offset and image size:
    - **offset = 20** specifies the offset value for cropping the hand region.
    - **imgsize = 300** sets the desired image size for classification.
6. Define the labels for gesture recognition:
    - **labels = ["A", "B", "C", ..., "Z"]** represents the labels corresponding to the hand gestures.
7. Initialize a counter variable:
    - **counter = 0** stores the count of processed frames.
8. Enter a while loop for continuous frame processing:
    - **while True:** captures and processes frames indefinitely.
9. Read a frame from the video capture:
    - **success, img = cap.read()** retrieves the next frame from the video capture object.
10. Detect hands in the frame:
    - **hands, img = detector.findHands(img)** detects hands in the frame and returns hand landmarks. Updates the **img** variable with annotated hand landmarks.
11. Perform gesture recognition for each detected hand:
    - Check if any hand is detected (**if hands:**).
12. Crop and resize the hand region for classification:
    - Determine the aspect ratio of the hand bounding box: **aspectRatio = h / w**
    - If the aspect ratio is greater than 1 (hand is taller than wide):
        - Resize the hand region preserving the width: **imgResize = cv2.resize(imgCrop, (wCal, imgsize))**
        - Update the white image with the resized hand region: **imageWhite[0:, wGap: wCal + wGap] = imgResize**
    - Else (hand is wider than tall):
        - Resize the hand region preserving the height: **imgResize = cv2.resize(imgCrop, (imgsize, hCal))**
        - Update the white image with the resized hand region: **imageWhite[hGap: hCal + hGap, :] = imgResize**
13. Perform gesture classification using the classifier:
    - Get the prediction and index of the classified gesture: **prediction, index = classifier.getPrediction(imageWhite)**

14. Display the cropped hand region and processed hand image:
    - **cv2.imshow("ImageCrop", imgCrop)** shows the cropped hand region.
    - **cv2.imshow("ImageWhite", imageWhite)** shows the processed hand image.
15. Display the final output image with labeled gesture and bounding box:
    - **cv2.imshow("Image_Output", imgOutput)** shows the output image with labels and bounding box.
16. Wait for a key press and handle it:
    - **cv2.waitKey(1)** waits for 1 millisecond for a key press.

V. Result Analysis

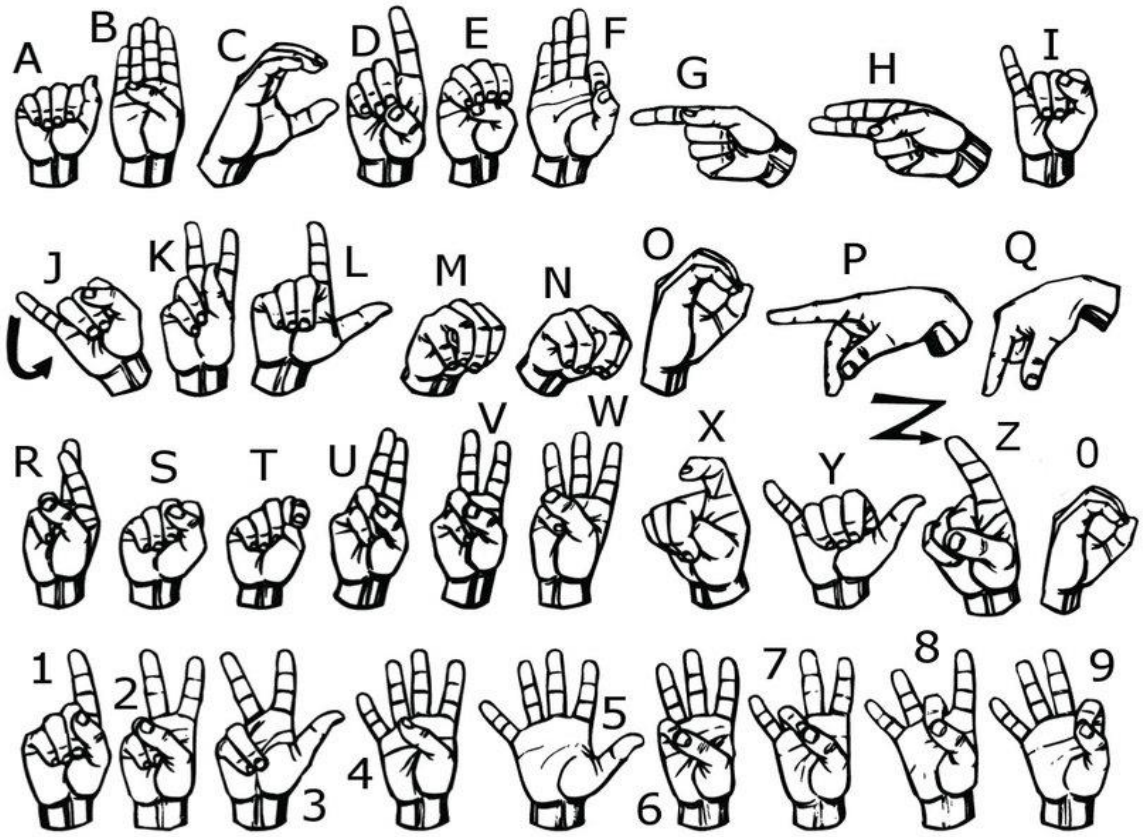


Figure 2.1. American Sign Language

In the figure 2.1 we can see that is representing the American Sign Languages. That figure suggests and describes that which finger movement indicates which character in English alphabet. Our system has been trained with these data and in the time of recognization the system will internally compare the input image data with this trained data. Figure 2.1 visually represents the American Sign Languages (ASL) and provides a descriptive guide on the finger movements associated with each character in the English alphabet. This figure serves as a reference for understanding and interpreting the hand gestures corresponding to specific letters. During the training phase, our system has been exposed to these visual representations of ASL and has learned to associate the finger movements with the corresponding English alphabet characters. This training enables the system to internally compare the input image data with the patterns and characteristics it has learned from the training data. When the system performs recognition, it utilizes the trained knowledge to analyze and match the input image data with the finger movement patterns it has learned. By internally comparing the input image data with the trained data, the system determines the most probable character represented by the hand gesture. The utilization of Figure 2.1 and the training of our system with this data enable effective recognition and classification of sign language gestures into their corresponding English alphabet characters. This enhances the system's ability to accurately interpret and understand the intended meaning behind different hand movements, facilitating communication between sign language users and non-sign language users.

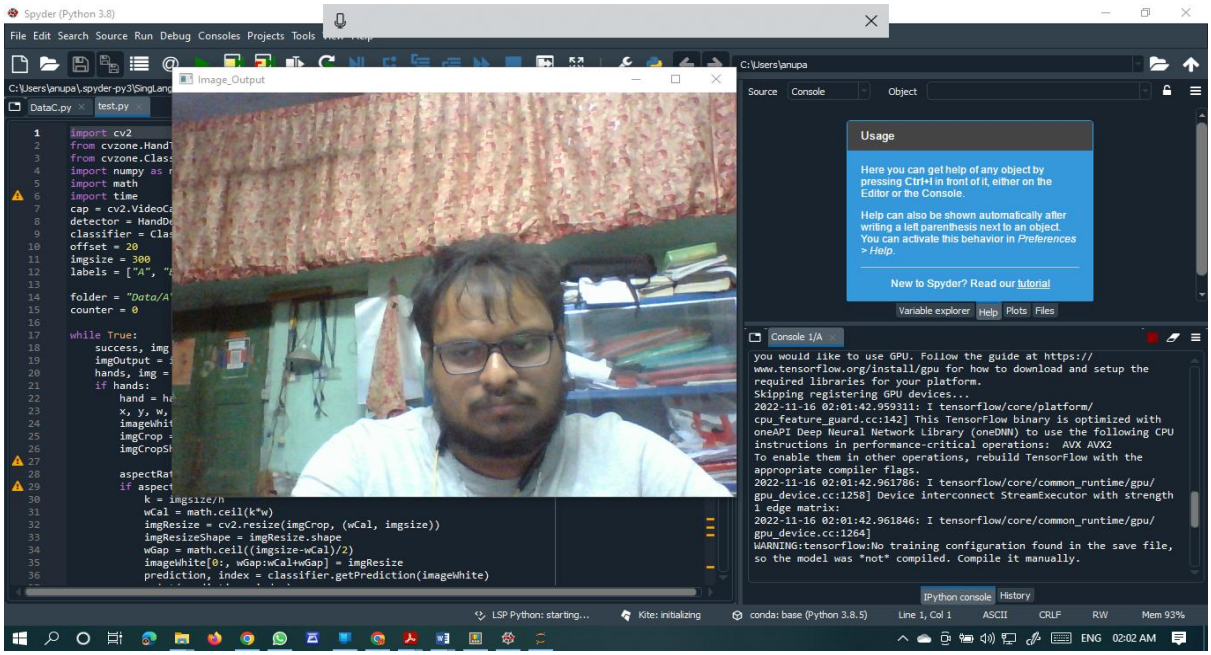


Figure 2.2. Image Capturing

The figure 2.2 indicates the initial step of the recognition which was image capturing. Here in the image the system camera has been started and started image acquisition. In this stage the system only showing the visual output as there is no hand movements or hand sign is detected so is currently giving normal output of the surroundings.

In Figure 2.2, we can observe the initial step of the sign language recognition system, which involves the image capturing process. The system utilizes a camera to capture real-time images of the surrounding environment. At this stage, the system is not actively detecting any hand movements or sign language gestures, resulting in a normal visual output without any specific interpretation.

The purpose of this initial step is to establish the baseline functionality of the system, where it starts acquiring images to be processed further in subsequent stages. The visual output displayed in this step serves as an indicator that the camera is active and ready to capture hand movements for sign language recognition.

During this stage, the system may display the live camera feed or a placeholder image to indicate that it is actively capturing frames. Users can observe the real-time video feed, but no specific analysis or interpretation of hand gestures is performed yet. It acts as a starting point for the recognition process, awaiting the presence of hand gestures to further analyze and interpret them.

Once the system detects hand movements or sign language gestures in the subsequent stages, it will process the captured images using computer vision and machine learning techniques to recognize and interpret the corresponding signs. The initial image capturing step sets the foundation for this subsequent analysis and recognition, signalling that the system is ready to receive and process hand gestures for sign language interpretation.

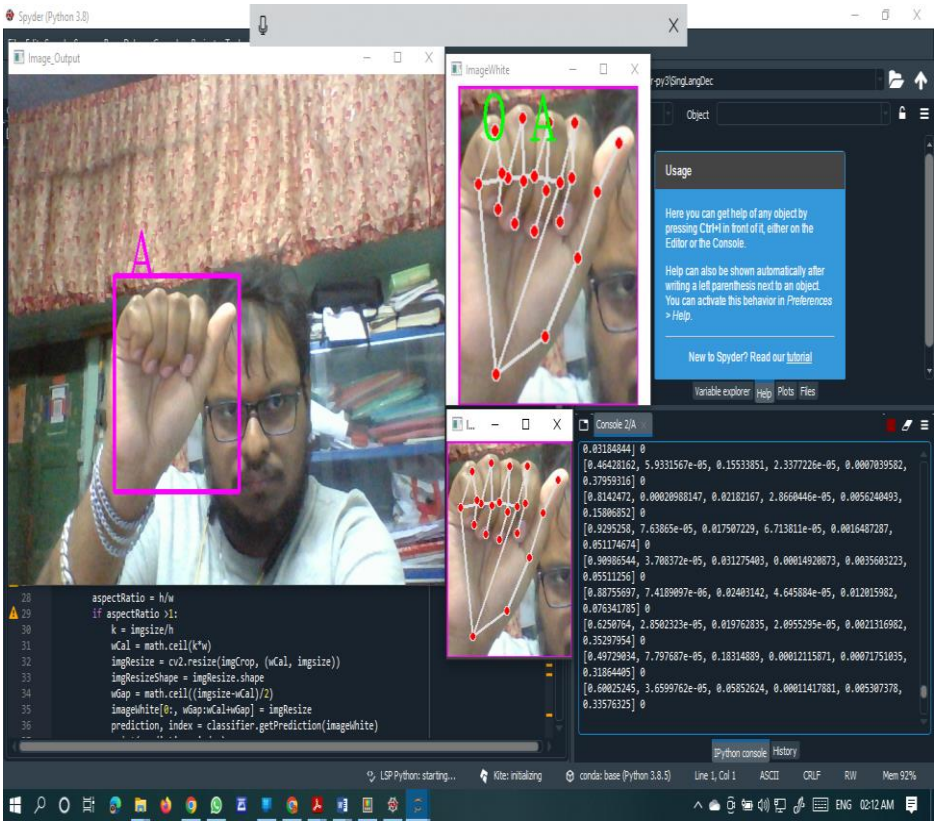


Figure 2.3. Image Capturing & Generating output

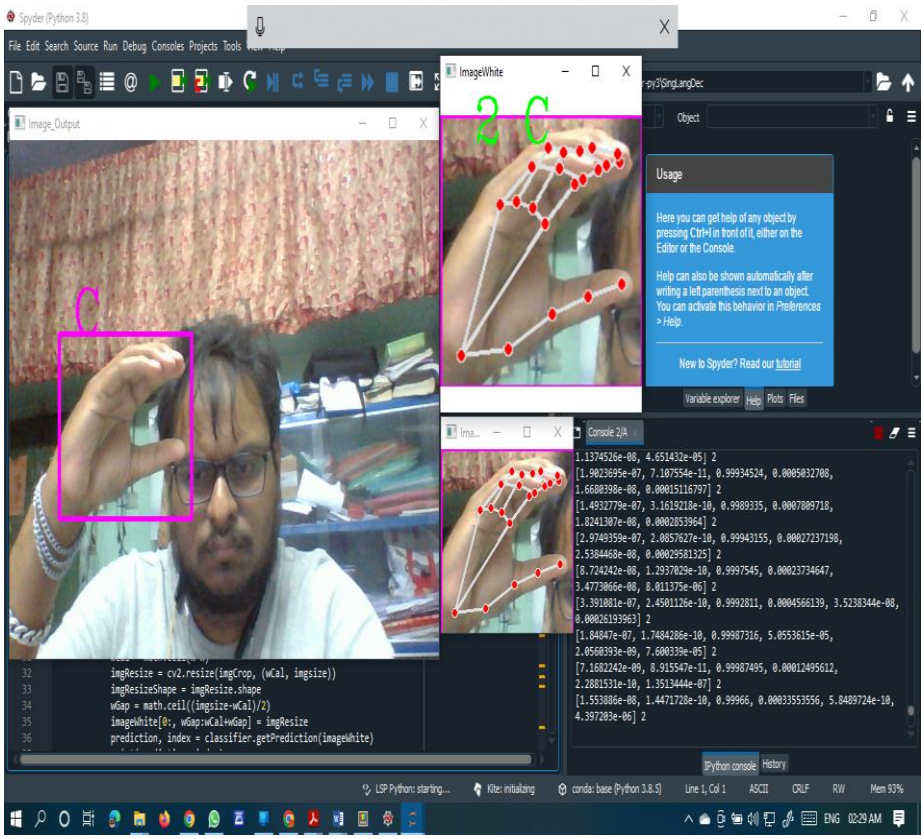


Figure 2.3. Image Capturing & Generating output

In the figure 2.3 the system is generating the output based on the confidence value. On the other hand, the console is printing the confidence value evaluated by the proposed system. In display there are three live output images which are actually works for specific purposes. The *"ImageCrop*" window actually represents the cropped view of hand by eliminating maximum unwanted surroundings. The *"ImageWhite"* window Uniforms the size of the input image of the hand. The image output window provides the actual intended output of our system which is the translation of sing language to conventional English language. The figure which is showing the output of the system is based on the confidence values.

TABLE I. EXPERIMENTAL DATA ANALYSIS (for A):

A	B	C	D	E	F	MAX Confidence
0.002486	2.62E-07	0.971857	0.00144	0.022696	0.001522	0.97185665
0.015941	5.95E-07	0.948804	0.013368	0.020756	0.00113	0.94880444
0.005192	6.30E-07	0.935682	0.001865	0.000756	0.056504	0.9356821
0.391606	1.43E-05	0.58413	0.001922	0.020604	0.001724	0.5841297
0.112918	7.74E-07	0.792791	0.001119	0.060028	0.033143	0.79279065
0.00232	1.04E-07	0.982352	0.005994	0.001125	0.008208	0.9823516
0.016325	1.48E-06	0.968667	0.003289	0.003696	0.008021	0.9686669
0.000386	2.64E-08	0.993395	0.006093	8.10E-05	4.51E-05	0.9933947
0.955613	2.14E-07	0.034047	5.71E-05	1.62E-06	0.010281	0.9556125
0.817799	3.14E-07	0.032063	0.000465	3.11E-06	0.149669	0.8177991
0.075086	4.70E-08	0.008725	7.45E-06	8.60E-08	0.916182	0.91618156
0.029894	5.09E-08	0.003488	2.18E-06	2.44E-07	0.966615	0.9666149
0.819631	1.11E-06	0.013096	7.74E-05	4.74E-06	0.167191	0.81963056
0.102105	1.32E-07	0.010899	0.000197	1.43E-05	0.886785	0.8867845
0.85411	1.32E-06	0.024165	0.000592	2.80E-05	0.121104	0.8541099
0.967963	8.48E-07	0.009201	0.000492	0.000172	0.022171	0.9679634
0.725734	5.03E-08	0.00547	0.000589	9.16E-05	0.268116	0.72573376
0.678221	1.05E-07	0.008206	0.001404	5.74E-05	0.312111	0.6782213
0.932543	2.36E-07	0.006391	0.000788	0.000221	0.060056	0.93254316
0.828213	5.26E-07	0.004309	0.001036	0.000346	0.166095	0.82821316
0.831805	1.80E-07	0.011209	0.000981	0.000696	0.155309	0.83180493
0.898247	3.28E-07	0.015113	0.00074	9.38E-05	0.085806	0.8982466
0.266791	1.11E-08	0.004912	0.001385	0.000118	0.726795	0.72679466
0.971996	1.70E-07	0.003527	0.000619	5.47E-05	0.023804	0.9719957
0.912805	5.32E-08	0.002913	0.000404	3.22E-05	0.083846	0.91280454
0.898948	5.22E-07	0.017615	0.000745	4.36E-05	0.082648	0.89894813
0.624225	6.64E-07	0.021189	0.000385	3.49E-05	0.354165	0.6242249
0.7712	9.41E-08	0.011642	0.001252	0.000384	0.215522	0.77119994

In Table 1, the experimental data analysis is presented. The system generates confidence values for each labelled character during the recognition phase. These confidence values indicate the

system's level of certainty in its classification decision. The character with the maximum confidence value is considered as the predicted output. In other words, the system selects the character that has the highest confidence value as the output for a given input. By analysing the confidence values, we can gain insights into the system's performance and its level of confidence in its predictions. The higher the confidence value for a particular character, the more certain the system is in its classification. This allows us to understand the reliability of the system's outputs and assess its accuracy. By considering the character with the maximum confidence value, the system ensures that the output is based on the most confident prediction. This approach aims to enhance the accuracy and reliability of the system's recognition results. The experimental data analysis in Table 1 provides valuable information about the system's performance and the confidence values associated with each recognized character. This analysis helps in evaluating and improving the system's accuracy and robustness in recognizing sign language gestures.

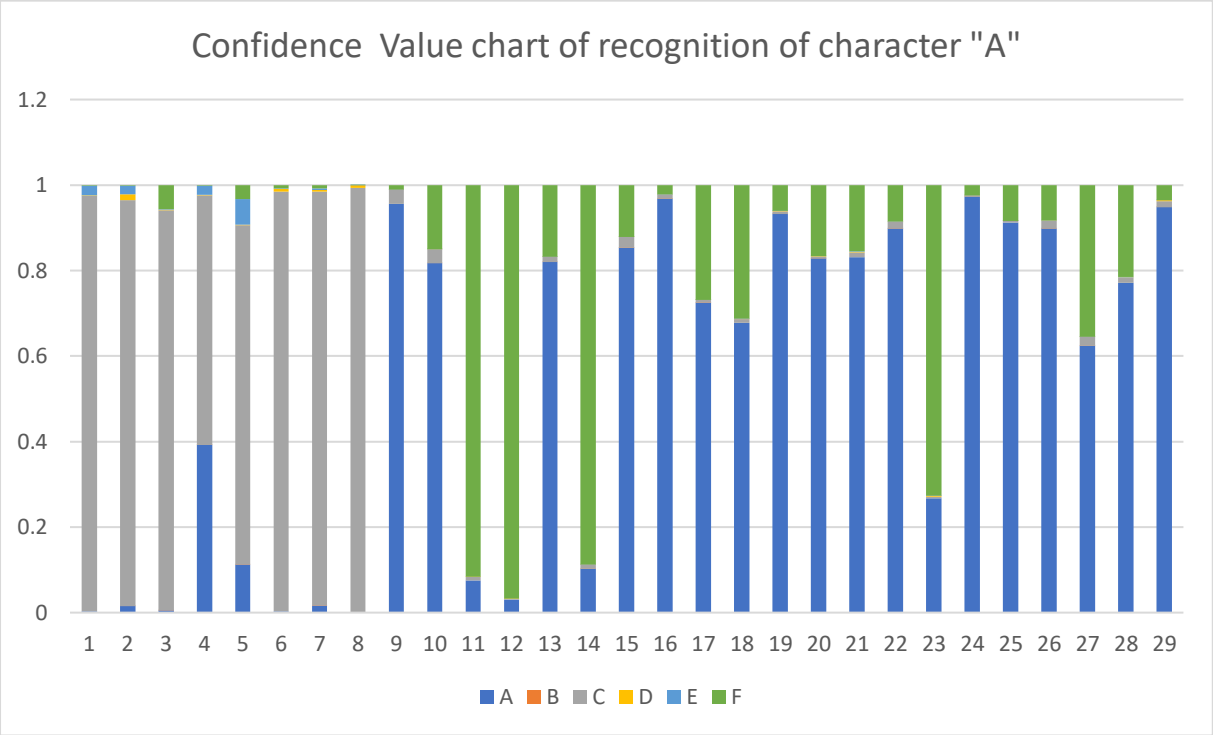


Figure 3.1. Confidence Value chart of recognition of character "A"

The confidence value chart of character “A” Represents the confidence value in a graphical manner. From that chart we can evaluate that most of the time our system is generating the intended correct output although sometimes due to some environmental noises it provides false output. The accuracy of the Experimental data analysis for recognition of character “A” calculated as 62.02%. The confidence value chart for character "A" visually represents the confidence values generated by the system during the recognition process. This chart allows us to assess the system's performance in correctly identifying the character "A" based on its confidence levels. By analyzing the chart, we can observe that the system generally produces high confidence values for character "A," indicating a correct classification. However, there are instances where the system encounters environmental noises or other factors that result in lower confidence values, leading to incorrect outputs. To quantitatively evaluate the accuracy of the experimental data analysis for the recognition of character "A," the accuracy is calculated

as 62.02%. This accuracy value represents the percentage of correctly classified instances of character "A" out of the total instances considered in the analysis. The accuracy value helps us understand the overall performance of the system in recognizing character "A." It indicates that, on average, the system achieves a 62.02% accuracy rate in correctly identifying and classifying character "A." By analyzing the confidence value chart and evaluating the accuracy, we can gain insights into the system's performance and identify areas for improvement to enhance the accuracy and robustness of character recognition in the sign language detection system.

TABLE II. EXPERIMENTAL DATA ANALYSIS (for C):

A	B	C	D	E	F	MAX Confidence
4.61E-06	9.83E-08	0.98650485	0.004179969	2.71E-05	0.009283303	9.87E-01
1.32E-07	9.38E-11	0.99934894	5.59E-07	1.22E-07	0.000650187	9.99E-01
8.68E-08	9.47E-11	0.9989893	0.000446067	1.66E-07	0.000564291	9.99E-01
9.80E-07	6.65E-09	0.9980221	0.000307356	1.52E-06	0.001668174	9.98E-01
3.87E-09	7.79E-11	0.9996468	0.000343273	1.30E-08	9.85E-06	1.00E+00
3.05E-08	4.47E-10	0.99913883	0.00085778	2.85E-08	3.32E-06	9.99E-01
7.77E-07	3.09E-09	0.9997832	4.41E-05	3.65E-07	0.000171498	1.00E+00
3.68E-07	1.67E-09	0.99983096	2.66E-05	1.77E-07	0.000141893	1.00E+00
1.98E-08	6.73E-10	0.9999734	1.20E-05	5.60E-09	1.47E-05	1.00E+00
3.05E-07	4.77E-09	0.9997336	0.000141824	1.08E-06	0.000123122	1.00E+00
1.39E-06	3.23E-09	0.99916196	0.000501654	1.64E-06	0.000333332	9.99E-01
1.05E-07	4.99E-10	0.999746	4.60E-05	1.74E-08	0.000207883	1.00E+00
4.74E-07	4.70E-09	0.9992341	0.000475317	2.67E-06	0.000287492	9.99E-01
3.49E-08	4.08E-10	0.9996866	0.00031033	3.20E-08	3.10E-06	1.00E+00
2.78E-07	1.69E-09	0.99864966	0.001261452	1.78E-07	8.86E-05	9.99E-01
8.63E-08	1.30E-09	0.9997495	0.000223214	7.67E-08	2.71E-05	1.00E+00
1.37E-06	3.18E-09	0.9990551	0.000306692	5.87E-07	0.000636308	9.99E-01
2.28E-06	1.84E-09	0.99938965	0.000225255	4.10E-06	0.000378688	9.99E-01
5.46E-07	1.55E-09	0.999747	0.000158609	3.01E-07	9.35E-05	1.00E+00
1.48E-05	4.41E-09	0.9993593	3.60E-05	6.14E-06	0.00058383	9.99E-01
2.46E-07	6.60E-10	0.9998863	4.25E-05	4.42E-07	7.04E-05	1.00E+00
5.25E-07	1.03E-09	0.9987255	0.001247809	9.53E-07	2.52E-05	9.99E-01
3.86E-07	6.84E-10	0.9999393	4.76E-05	4.92E-08	1.26E-05	1.00E+00
2.41E-07	1.20E-09	0.9998945	3.76E-05	1.21E-07	6.75E-05	1.00E+00
3.30E-07	7.70E-10	0.9998878	0.00011037	9.57E-08	1.37E-06	1.00E+00
7.81E-07	4.46E-09	0.9998004	0.00015294	1.14E-06	4.48E-05	1.00E+00
3.93E-06	1.83E-09	0.9996573	0.000302175	3.96E-06	3.26E-05	1.00E+00
6.48E-07	2.99E-10	0.99975723	0.000176058	2.98E-07	6.59E-05	1.00E+00
2.94E-06	1.55E-09	0.99891937	0.001059461	2.54E-06	1.57E-05	9.99E-01

In the table two it shows the experimental data analysis of the character “C”. This system is generating some confidence values for each labelled characters. The maximum confidence

value is responsible to produce output in the simple words the system classifieds and generates output based on the Maximum confidence value. The system provides the output of the character which game maximum confidence value while recognization phase.

In Table 2, the experimental data analysis is presented for character "C" in the sign language detection system. Similar to Table 1, this analysis involves generating confidence values for each labeled character, where the maximum confidence value determines the system's output. The system utilizes the maximum confidence value obtained during the recognition phase to classify and generate the output for character "C." By selecting the character with the highest confidence value, the system aims to provide accurate classification and recognition results. The experimental data analysis in Table 2 allows us to examine the confidence values assigned to character "C" during various instances.

By analyzing these values, we can evaluate how well the system performs in correctly identifying and classifying character "C." The information provided in Table 2 enables us to assess the system's recognition capabilities for character "C" and understand the distribution of confidence values. This analysis helps in identifying trends and patterns in the system's performance, including cases where the system may encounter challenges or produce incorrect outputs. By analyzing the confidence values and examining the system's performance for character "C," we can gain insights into the accuracy and reliability of the sign language detection system for this specific character. This information can guide further improvements and refinements to enhance the system's overall performance and accuracy in recognizing character "C" and other sign language gestures.

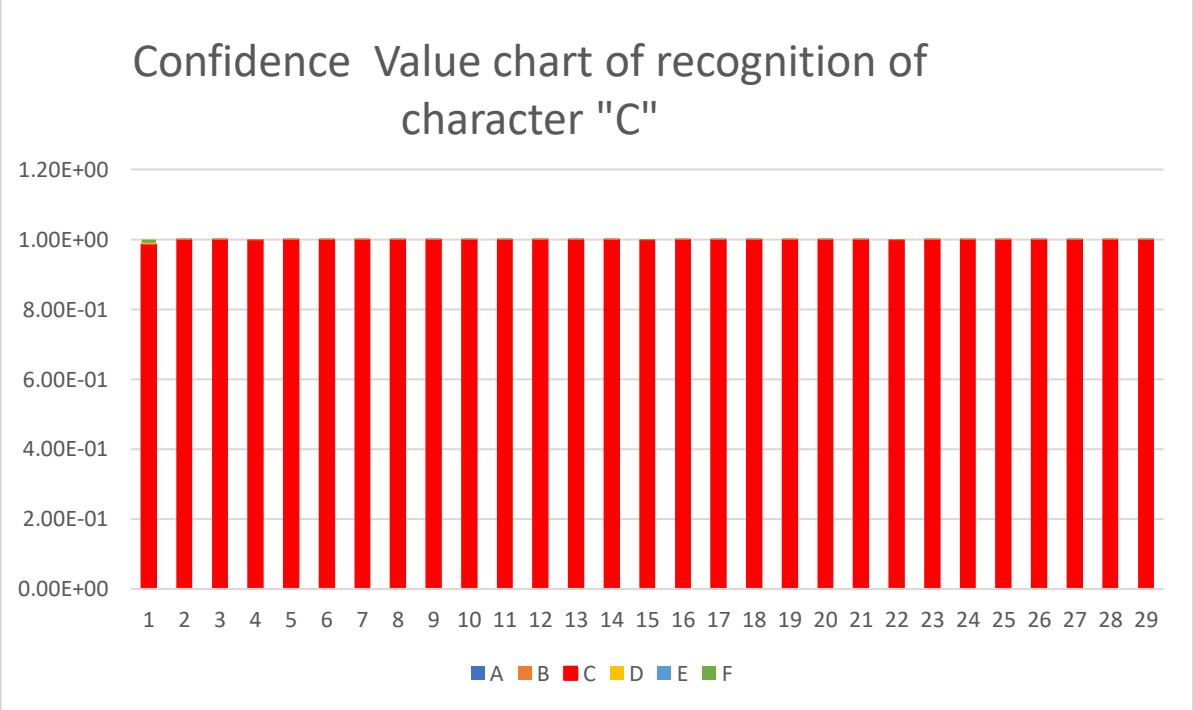


Figure 3.2. Confidence Value chart of recognition of character "C"

The confidence value chart of character “C” Represents the confidence value in a graphical manner. From that chart we can evaluate that most of the time our system is generating the intended correct output although sometimes due to some environmental noises it provides false

output, however in case of Recognition of character “C”, this time the number of errors is very less and in most cases our model or our system has recognized the correct answer. The accuracy of the Experimental data analysis for recognition of character “C” calculated as 99.9%. The confidence value chart of character "C" visually represents the confidence values obtained during the recognition process. This chart allows us to evaluate the system's performance and accuracy in classifying character "C" based on the assigned confidence values.

Upon analyzing the chart, we can observe that the system consistently generates high confidence values for character "C," indicating that it often produces the correct output. However, it is important to note that in some instances, the presence of environmental noises may lead to false outputs, resulting in lower confidence values. In the case of character "C," the experimental data analysis reveals a significantly low number of errors. The system demonstrates a high level of accuracy and successfully recognizes character "C" in the majority of cases.

The accuracy of the experimental data analysis for recognition of character "C" is calculated as 99.9%. This high accuracy rate suggests that the system is proficient in identifying and classifying character "C" based on the input data and the assigned confidence values. It indicates the effectiveness of the implemented model and the success of the system in recognizing character "C" accurately.

The exceptional accuracy achieved in the experimental data analysis for character "C" highlights the potential of the sign language detection system and its capability to recognize specific sign language gestures with high precision. It showcases the system's reliability and usefulness in facilitating communication for individuals using sign language, further emphasizing its practical value and potential applications.

VI. Conclusions of the Previous Work:

Based on the current version of the project, we have successfully developed a system that can recognize individual hand gestures in American Sign Language (ASL) using computer vision and machine learning techniques. The project consists of two main parts: data collection and gesture recognition.

In the data collection phase, we utilized the capabilities of the CVzone library, specifically the HandDetector module, to capture hand gestures with hand landmarks. The captured images were then pre-processed by resizing them to a standardized size and removing unnecessary parts. These pre-processed images, along with appropriate labels, were used to train a machine learning model using Google Teachable Machine. The model was generated with the help of the provided Epoch accuracy and loss graphs.

In the gesture recognition phase, we utilized the trained model to predict ASL gestures in real-time. The system continuously captures video frames from the camera and processes them using the HandDetector module to detect and track the hand. The detected hand region is then cropped and resized to match the input requirements of the model. The pre-processed image is passed through the model, and the predicted gesture is obtained. The corresponding label for the predicted gesture is retrieved from the labels list, and it is displayed on the output frame using OpenCV's putText and rectangle functions. The output frame, along with the cropped hand region and resized image, is displayed in separate windows for visualization.

VII. Future Scope of The Previous Work:

To further enhance the capabilities of the system, the second version of the project will focus on implementing more advanced machine learning techniques. Specifically, we plan to incorporate LSTM (Long Short-Term Memory) and Dense layers into the model architecture. By integrating LSTM layers, the model will be able to capture the temporal dependencies present in a sequence of ASL gestures & other hand & pose gestures. This enhancement will enable the system to understand complete sentences in ASL & other hand & pose gestures, rather than just individual signs.

To implement the upgraded model, modifications will be made to the model architecture, including the addition of LSTM layers and appropriate data pre-processing techniques. The dataset used for training will also be expanded to include a wider range of ASL gestures, ensuring that the model learns and recognizes a diverse set of signs accurately.

Additionally, the user interface of the system can be improved to provide a more intuitive and user-friendly experience. This may involve displaying translated text or providing spoken output alongside the recognized gestures, making it easier for users to communicate with the system.

In conclusion, the future scope of the project involves advancing the machine learning capabilities by incorporating LSTM layers, expanding the dataset, and improving the user interface. These enhancements will enable the system to understand complete ASL sentences, leading to improved usability and functionality in real-world scenarios.

Information about the Neural Network used in Improved Model:

1. Dense Neural Network:

- A Dense neural network, also referred to as a fully connected neural network, is a fundamental type of artificial neural network where each neuron in a layer is connected to every neuron in the subsequent layer.

- In a Dense layer, the output of each neuron is calculated by applying a weighted sum of inputs from the previous layer, followed by the application of an activation function.

- The activation function introduces non-linearity into the network, enabling it to learn complex relationships between inputs and outputs.

- The number of neurons in a Dense layer determines the dimensionality of the learned features. More neurons allow the network to capture more intricate patterns, but also increase computational complexity.

- Dense layers are commonly employed for tasks such as image classification, natural language processing, and various other machine learning applications.

- In this particular project, Dense layers are utilized to learn high-level representations of the input data (i.e., keypoints extracted from video frames) and perform the final classification based on these learned features.

- By leveraging the expressive power of Dense layers, the model can discern intricate spatial patterns in the keypoints and make accurate predictions about the actions being performed.

2. **LSTM (Long Short-Term Memory) Neural Network**:

- LSTM is a type of recurrent neural network (RNN) designed to overcome the limitations of traditional RNNs when dealing with long sequences and capturing long-term dependencies.

- RNNs are well-suited for processing sequential data because they maintain internal memory to retain information from previous time steps.

- LSTM networks introduce memory cells and gates that regulate the flow of information within the network, allowing it to selectively remember or forget information as needed.

- The LSTM architecture comprises several components, including input gates, output gates, forget gates, and a cell state that preserves long-term memory.

- The input gate controls the flow of new information into the cell state, the forget gate determines which information to discard, and the output gate combines the current input with the cell state to produce the output at each time step.

- LSTM networks excel at capturing dependencies and long-term patterns in sequential data, making them particularly valuable for action recognition tasks where temporal information plays a crucial role.

- In this project, LSTM layers are employed to capture the temporal dynamics of keypoints over a sequence of video frames. This allows the model to learn meaningful representations of actions by considering the sequential nature of the data.

Technical Information:

- The Dense layers in the model have varying numbers of units (neurons), such as 64, 32, and the number of actions in the dataset. The number of units determines the dimensionality of the learned features and the complexity of the model.
- The LSTM layers have different numbers of LSTM units, such as 64, 128, and 64. The number of units determines the capacity of the LSTM to capture temporal dependencies and learn meaningful representations.
- Both Dense and LSTM layers use the rectified linear unit (ReLU) activation function. ReLU introduces non-linearity, allowing the model to learn complex patterns and enabling efficient training.

- The model is compiled with the Adam optimizer, an adaptive optimization algorithm widely used for training deep neural networks. Adam adjusts the learning rate dynamically during training, leading to faster convergence.
- The loss function used is categorical cross-entropy, suitable for multi-class classification tasks where each action label is mutually exclusive.
- During training, the model is fit to the training data using the **fit** function, specifying the number of training epochs. Training progress is monitored using TensorBoard, which provides visualizations and insights into the model's performance and training dynamics.

Benefits for this project:

Capturing Spatial and Temporal Information: By combining Dense and LSTM layers, the model can effectively capture both spatial and temporal information. Dense layers learn high-level spatial features from the keypoints in individual frames, enabling the model to discern intricate patterns and variations. LSTM layers, on the other hand, capture the temporal dynamics by considering the sequence of keypoints across multiple frames. This allows the model to learn the temporal dependencies and long-term patterns necessary for accurate action recognition.

Hierarchical Representation: The combination of Dense and LSTM layers enables the model to learn a hierarchical representation of actions. Dense layers learn low-level spatial features, such as the positions and relationships of keypoints within a single frame. LSTM layers build upon these spatial features and capture the sequential dependencies and temporal patterns present across multiple frames. This hierarchical representation facilitates a more comprehensive understanding of actions, improving the model's ability to discriminate between different actions.

Handling Variable-Length Sequences: The LSTM architecture is well-suited for processing sequences of variable lengths. In this project, actions are recognized from video frames, which may contain sequences of different lengths depending on the duration of the action. LSTM layers can effectively handle these variable-length sequences by dynamically adjusting their internal memory and adapting to the length of each input sequence. This flexibility allows the model to accurately recognize actions regardless of the duration.

Memory and Context Preservation: LSTM networks incorporate memory cells and gates that allow the model to retain important information and discard irrelevant or outdated information. This is particularly beneficial for action recognition, as it enables the model to capture the context and long-term dependencies necessary for accurate classification. By preserving memory and selectively focusing on relevant information, LSTM layers contribute to the model's ability to recognize actions in a wider temporal context.

Enhanced Accuracy and Robustness: The combination of Dense and LSTM layers enhances the overall accuracy and robustness of the model. Dense layers extract discriminative spatial features, while LSTM layers capture the temporal dynamics, leading to a more comprehensive understanding of actions. This combination allows the model to generalize well to unseen actions and handle variations in timing, speed, or execution style. Consequently, the model can provide more accurate and reliable action recognition results.

In summary, the **Dense and LSTM neural networks utilized** in this project complement each other's strengths, enabling the model to learn both spatial and temporal representations of actions. This hierarchical representation, along with the ability to handle variable-length sequences and preserve memory, contributes to the model's accuracy, robustness, and ability to recognize actions effectively.

VIII. Improved Methodology

This sign language recognition project offers several benefits in the field of human-computer interaction and accessibility. By leveraging machine learning and computer vision techniques, it provides a reliable and efficient solution for interpreting and understanding sign language gestures. One of the key benefits is promoting inclusivity and accessibility for individuals with hearing impairments. The project enables real-time recognition and interpretation of sign language, allowing deaf or hard-of-hearing individuals to communicate more effectively with others in various settings. Moreover, the project's methodology employs deep learning models, specifically LSTM networks, which are well-suited for capturing temporal dependencies in sequential data. This enables the recognition of sign language actions based on sequences of frames, providing a more comprehensive understanding of the gestures. Additionally, the use of MediaPipe's holistic model simplifies the process of detecting and extracting key points related to hand and body poses. This not only enhances the accuracy of the recognition system but also enables real-time implementation, making it practical for applications such as gesture-based control systems and assistive technologies.

Overall, this project showcases the potential of machine learning and computer vision in bridging the communication gap between sign language users and the broader community. Its benefits extend beyond research and have the potential to positively impact the lives of individuals with hearing impairments by facilitating better communication and inclusivity.

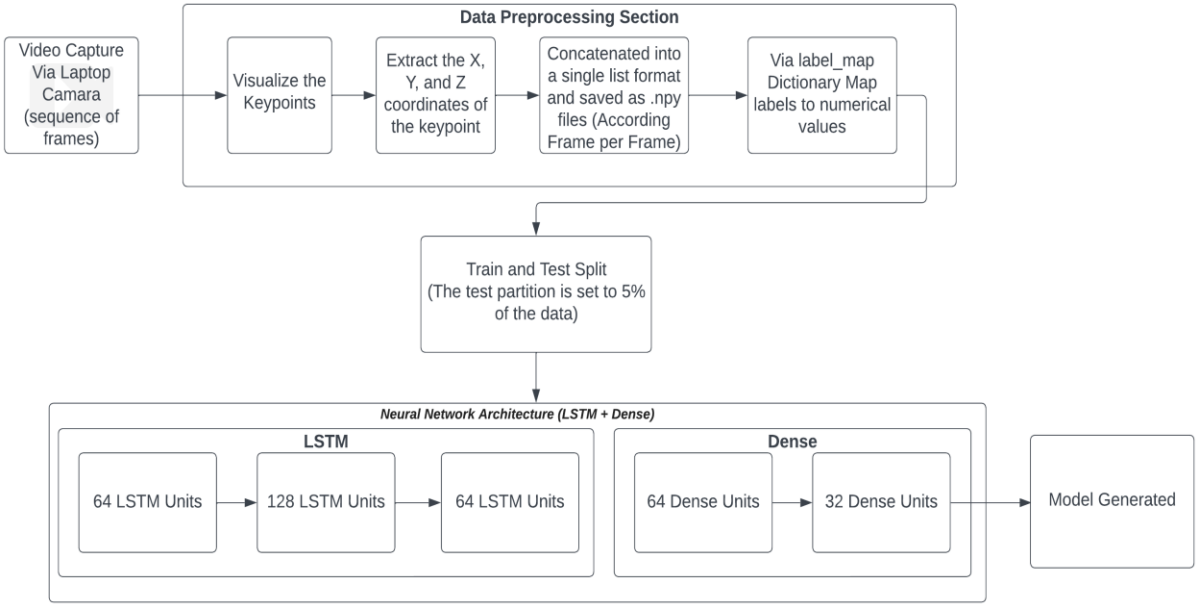


Figure 4.0. Flow Chart of Model Generation

Development of the Improved Model of the Project:

```
# Import Dependencies

import sklearn
import tensorflow
import keras
import cv2
import numpy as np
import os
from matplotlib import pyplot as plt
import time
import mediapipe as mp
```

import sklearn: This line imports the scikit-learn library, which is a powerful machine learning library in Python. It provides a wide range of tools and algorithms for tasks such as data preprocessing, model selection, and evaluation.

import tensorflow: This line imports the TensorFlow library, an open-source machine learning framework. TensorFlow is widely used for building and training deep learning models. It provides a comprehensive set of tools and functionalities for working with neural networks.

import keras: This line imports the Keras library, which is a high-level neural networks API. Keras simplifies the process of building and training deep learning models by providing an intuitive and user-friendly interface. Keras is often used in conjunction with TensorFlow.

import cv2: This line imports the OpenCV library, which stands for Open-Source Computer Vision. OpenCV is a powerful computer vision library that offers a broad range of functions for image and video processing, including image manipulation, object detection, and feature extraction.

import numpy as np: This line imports the NumPy library and assigns it the alias np. NumPy is a fundamental package for scientific computing in Python. It provides support for efficient array manipulation and mathematical operations, making it essential for tasks involving numerical data.

import os: This line imports the os module, which provides a way to interact with the operating system. The os module allows you to perform operations such as navigating directories, creating and deleting files, and executing system commands. It is commonly used for file and directory manipulation tasks.

from matplotlib import pyplot as plt: This line imports the pyplot module from the Matplotlib library. Matplotlib is a widely used plotting library in Python. The pyplot module provides a simple and convenient interface for creating various types of plots and visualizations, such as line plots, bar plots, and histograms.

import time: This line imports the time module, which provides functions for working with time-related operations. The time module can be used to measure the execution time of code snippets, introduce delays in program execution, or handle time-related calculations.

import mediapipe as mp: This line imports the Mediapipe library, which is a powerful framework for building multimodal applied machine learning pipelines. Mediapipe provides pre-built solutions for various tasks, such as pose estimation, face detection, hand tracking, and more. It simplifies the process of developing computer vision and machine learning applications by providing high-level abstractions and ready-to-use models.

#Veriable Creation

```
mp_holistic = mp.solutions.holistic     # Loades The Model (Holistic Model) To make detection
mp_drawing_utils = mp.solutions.drawing_utils   # draws the key points
```

In this code snippet, two variables, **mp_holistic** and **mp_drawing_utils**, are being created.

mp_holistic is a variable that is assigned the value of **mp.solutions.holistic**. It is used to load the Holistic Model from the Mediapipe library. The Holistic Model is a pre-trained machine learning model that can perform multiple human pose detection tasks simultaneously, including face detection, facial landmark detection, pose estimation, and hand tracking.

By assigning **mp.solutions.holistic** to **mp_holistic**, you are making the Holistic Model accessible through the **mp_holistic** variable. This allows you to use the functionalities provided by the Holistic Model for detecting and analyzing human poses in images or videos.

mp_drawing_utils is a variable that is assigned the value of **mp.solutions.drawing_utils**. It is used to access the drawing utility functions provided by the Mediapipe library. These utility functions help visualize the detected key points and landmarks on the image or video frames.

By assigning **mp.solutions.drawing_utils** to **mp_drawing_utils**, you can utilize the drawing utility functions to overlay the detected key points and landmarks on the visual media, making it easier to interpret and analyze the results.

#BGI to RGB for mediapipe as media pipe need RGB format

```
def mediapipe_detection(image, model):

    images = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # color Convertion BGR to RGB
    images.flags.writeable = False                  # image will no longer writable
    result = model.process(image)                # make Pridiction
    images.flags.writeable = True                   # image will writable again
    images = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)  # color Convertion RGB to BGR
    return images, result
```

This code defines a function named **mediapipe_detection** that takes an **image** and a **model** as input parameters and returns the modified **images** and **result**.
Here's a step-by-step explanation of the code:

1. **images = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)**: This line performs a color conversion of the input **image** from the BGR (Blue-Green-Red) color space to the RGB (Red-Green-Blue) color space. Mediapipe requires the input images to be in RGB format, so this conversion ensures that the image is in the correct format.
2. **images.flags.writeable = False**: This line makes the **images** array read-only, preventing any modifications to the array. This is done to optimize memory usage and improve performance during inference.
3. **result = model.process(image)**: This line uses the **model** to process the **image** and make predictions. The specific details of the **model** depend on the implementation and the specific machine learning model being used for detection. The **process** function takes the input image and returns the detected results.
4. **images.flags.writeable = True**: This line makes the **images** array writable again, allowing modifications to the array. This is necessary because the subsequent line will perform another color conversion, which requires write access to the array.
5. **images = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)**: This line performs a color conversion of the modified **image** from RGB back to BGR format. This conversion is performed to ensure that the output image is in the same format as the original input image.
6. **return images, result**: This line returns the modified **images** and **result** from the function as the output.

Overall, this code snippet prepares the input image for Mediapipe by converting it from BGR to RGB format, makes predictions using the provided model, and then converts the modified image back to BGR format before returning the results.

```
# Drawing The Landmarks in Different Style

def draw_landmarks_style(image, results):

# Draw Face Connections
mp_drawing_utils.draw_landmarks(image, results.face_landmarks,
mp_holistic.FACEMESH_CONTOURS, mp_drawing_utils.DrawingSpec(color=(80, 110,
10), thickness=1, circle_radius=1), mp_drawing_utils.DrawingSpec(color=(80, 256, 121),
thickness=1, circle_radius=1) )

# Draw Pose Connection
mp_drawing_utils.draw_landmarks(image, results.pose_landmarks,
mp_holistic.POSE_CONNECTIONS,  mp_drawing_utils.DrawingSpec(color=(80, 22, 10),
thickness=2, circle_radius=4),  mp_drawing_utils.DrawingSpec(color=(80, 44, 121),
thickness=2, circle_radius=2) )

# Draw Left Hand Connection
mp_drawing_utils.draw_landmarks(image, results.left_hand_landmarks,
mp_holistic.HAND_CONNECTIONS, mp_drawing_utils.DrawingSpec(color=(121, 22, 76),
```

thickness=2, circle_radius=4), mp_drawing_utils.DrawingSpec(color=(121, 44, 250), thickness=2, circle_radius=2))

# Draw Right Hand Connection
mp_drawing_utils.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.HAND_CONNECTIONS, mp_drawing_utils.DrawingSpec(color=(245, 117, 66), thickness=2, circle_radius=4), mp_drawing_utils.DrawingSpec(color=(245, 66, 230), thickness=2, circle_radius=2))

This code defines a function named **draw_landmarks_style** that takes two parameters: **image** and **results**. The purpose of this function is to draw landmarks on the provided **image** based on the detected results obtained from Mediapipe.
Here's an explanation of the code:

1. **mp_drawing_utils.draw_landmarks(image, results.face_landmarks, mp_holistic.FACEMESH_CONTOURS, ...)**: This line draws the face landmarks on the **image**. It uses the **face_landmarks** obtained from the **results** and specifies the **FACEMESH_CONTOURS** connection type. The face landmarks are drawn using the specified color, thickness, and circle radius.
2. **mp_drawing_utils.draw_landmarks(image, results.pose_landmarks, mp_holistic.POSE_CONNECTIONS, ...)**: This line draws the pose landmarks on the **image**. It uses the **pose_landmarks** obtained from the **results** and specifies the **POSE_CONNECTIONS** connection type. The pose landmarks are drawn using the specified color, thickness, and circle radius.
3. **mp_drawing_utils.draw_landmarks(image, results.left_hand_landmarks, mp_holistic.HAND_CONNECTIONS, ...)**: This line draws the left hand landmarks on the **image**. It uses the **left_hand_landmarks** obtained from the **results** and specifies the **HAND_CONNECTIONS** connection type. The left hand landmarks are drawn using the specified color, thickness, and circle radius.
4. **mp_drawing_utils.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.HAND_CONNECTIONS, ...)**: This line draws the right hand landmarks on the **image**. It uses the **right_hand_landmarks** obtained from the **results** and specifies the **HAND_CONNECTIONS** connection type. The right hand landmarks are drawn using the specified color, thickness, and circle radius.

```
def extract_keypoints(results):
    face = pose = np.array([[res.x, res.y, res.z] for res in
                            results.face_landmarks.landmark]).flatten() if results.face_landmarks else
np.zeros(468 * 3)
    pose = np.array([[res.x, res.y, res.z, res.visibility] for res in
                     results.pose_landmarks.landmark]).flatten() if results.pose_landmarks else
np.zeros(33 * 4)
    rh = np.array([[res.x, res.y, res.z] for res in
                   results.right_hand_landmarks.landmark]).flatten() if
results.right_hand_landmarks else np.zeros(
        21 * 3)
```

```
    lh = np.array([[res.x, res.y, res.z] for res in
                results.left_hand_landmarks.landmark]).flatten()  if  results.left_hand_landmarks
else np.zeros(21 * 3)

    return np.concatenate([pose, face, lh, rh])
```

This code defines a function named **extract_keypoints** that takes a **results** parameter. The purpose of this function is to extract keypoints from the given **results** object obtained from Mediapipe.
Here's an explanation of the code:

- **face = pose = np.array([[res.x, res.y, res.z] for res in results.face_landmarks.landmark]).flatten() if results.face_landmarks else np.zeros(468 * 3)**: This line extracts the face keypoints from the **results** object. It checks if **results.face_landmarks** is not **None**, indicating that face landmarks were detected. If face landmarks are available, it creates a NumPy array containing the x, y, and z coordinates of each face landmark and flattens it into a 1D array. If face landmarks are not available, it creates a NumPy array of zeros with a shape of (468 * 3), representing 468 face landmarks with 3 coordinates each.
- **pose = np.array([[res.x, res.y, res.z, res.visibility] for res in results.pose_landmarks.landmark]).flatten() if results.pose_landmarks else np.zeros(33 * 4)**: This line extracts the pose keypoints from the **results** object. It checks if **results.pose_landmarks** is not **None**, indicating that pose landmarks were detected. If pose landmarks are available, it creates a NumPy array containing the x, y, z coordinates, and visibility of each pose landmark and flattens it into a 1D array. If pose landmarks are not available, it creates a NumPy array of zeros with a shape of (33 * 4), representing 33 pose landmarks with 4 coordinates each.
- **rh = np.array([[res.x, res.y, res.z] for res in results.right_hand_landmarks.landmark]).flatten() if results.right_hand_landmarks else np.zeros(21 * 3)**: This line extracts the keypoints of the right hand from the **results** object. It checks if **results.right_hand_landmarks** is not **None**, indicating that right hand landmarks were detected. If right hand landmarks are available, it creates a NumPy array containing the x, y, and z coordinates of each right hand landmark and flattens it into a 1D array. If right hand landmarks are not available, it creates a NumPy array of zeros with a shape of (21 * 3), representing 21 right hand landmarks with 3 coordinates each.

- **lh = np.array([[res.x, res.y, res.z] for res in results.left_hand_landmarks.landmark]).flatten() if results.left_hand_landmarks else np.zeros(21 * 3)**: This line extracts the keypoints of the left hand from the **results** object. It checks if **results.left_hand_landmarks** is not **None**, indicating that left hand landmarks were detected. If left hand landmarks are available, it creates a NumPy array containing the x, y, and z coordinates of each left hand landmark and flattens it into a 1D array. If left hand landmarks are not available, it creates a NumPy array of zeros with a shape of (21 * 3), representing 21 left hand landmarks with 3 coordinates each.

- **return np.concatenate([pose, face, lh, rh])**: This line returns the concatenated array of all the extracted keypoints. It uses the **np.concatenate** function to concatenate the arrays **pose**, **face**, **lh**, and **rh** along the specified axis (default axis is 0). The concatenated array represents all the keypoints extracted from the face, pose, left hand, and right hand, merged into a single array.

DATA_PATH = os.path.join('MP_Data_C2C_New')

The line **DATA_PATH = os.path.join('MP_Data_C2C_New')** sets the path for the exported data, specifically for the numpy arrays.
Here's an explanation of what this line does:

- **os.path.join()** is a function from the **os** module that joins one or more path components intelligently, taking into account the operating system's path separator. It combines the specified path components into a single path.
- '**MP_Data_C2C_New**' is the name of the directory where the exported data will be stored. It is a relative path, meaning it is relative to the current working directory.
- The result of **os.path.join('MP_Data_C2C_New')** is assigned to the variable **DATA_PATH**.

So, the line **DATA_PATH = os.path.join('MP_Data_C2C_New')** sets the **DATA_PATH** variable to the path of the directory '**MP_Data_C2C_New**', which will be used for storing the exported data in numpy array format.

actions = np.array(['Bye', 'Hi', 'We are Engineers'])

The line **actions = np.array(['Bye', 'Hi', 'We are Engineers'])** defines an array variable **actions** that represents the different actions or labels that the system is trying to detect.
Here's an explanation of what this line does:

- **np.array()** is a function from the NumPy library that creates a new array object from a given input.
- **['Bye', 'Hi', 'We are Engineers']** is a list of three string values that represent the different actions or labels. These labels could correspond to different gestures, poses, or actions that the system is trained to recognize.
- The result of **np.array(['Bye', 'Hi', 'We are Engineers'])** is assigned to the variable **actions**.

So, the line **actions = np.array(['Bye', 'Hi', 'We are Engineers'])** creates an array **actions** containing the three labels 'Bye', 'Hi', and 'We are Engineers'. These labels represent the different actions that the system is trying to detect or classify.

no_sequences = 10 # Number Of Sequence Given or Equivalent to 10 videos data
sequence_length = 30 # Length Of The Sequence Given or 30 numbers of frame in each Video

The code snippet defines two variables, **no_sequences** and **sequence_length**, that are used in action detection to specify the number of sequences and the length of each sequence of frames.
Here's an explanation of what these variables represent:

1. **no_sequences**: This variable determines the number of sequences or samples given for action detection. In other words, it represents the number of videos or sets of frames

that are used as input data. In this case, **no_sequences** is set to **10**, indicating that there are 10 sequences or videos available for action detection.

2. **sequence_length**: This variable specifies the length of each sequence of frames. It represents the number of frames that make up a single sequence or video. In this case, **sequence_length** is set to **30**, indicating that each sequence or video consists of 30 frames.

These variables are important in action detection tasks because they define the temporal aspect of the data. By considering multiple frames over a specific time window, the model can capture the temporal dynamics and patterns necessary for recognizing and classifying actions accurately.

```
for action in actions:
    for sequence in range(no_sequences):
        try:
            os.makedirs(os.path.join(DATA_PATH, action, str(sequence)))
        except:
            pass
```

The code snippet represents a loop that iterates over the **actions** array and the range of **no_sequences** to acquire data for each action and sequence.
Here's an explanation of how the loop works:

1. **for action in actions:**: This loop iterates over each element in the **actions** array. Each element represents a specific action or label that the model is trained to detect.
2. **for sequence in range(no_sequences):**: Within the outer loop, this loop iterates **no_sequences** number of times. It represents the different sequences or samples for each action.
3. **try:** This keyword starts a try-except block, which allows for exception handling.
4. **os.makedirs(os.path.join(DATA_PATH, action, str(sequence)))**: This line attempts to create a directory structure for storing the data related to the current action and sequence. The **os.makedirs()** function is used to recursively create directories. The **os.path.join()** function is used to concatenate the **DATA_PATH**, action, and sequence number to form the complete path.
5. **except:** If an exception occurs during the creation of directories (e.g., if the directories already exist), the code inside the **except** block is executed. In this case, the **pass** statement is used, which essentially means to do nothing and continue with the loop.

The purpose of this loop is to create a directory structure for storing the data specific to each action and sequence. This structure helps organize the data and makes it easier to access during the training or testing phase of the action detection model.

The code snippet represents the process of collecting keypoints data by capturing video frames using OpenCV and Mediapipe. Here's a step-by-step explanation of the code:

1. **cap = cv2.VideoCapture(0)**: This line initializes the video capture object, **cap**, to capture frames from the default camera (index 0).
2. **With mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic:**: This line sets up the Mediapipe holistic

model for detecting keypoints. It creates an instance of the **Holistic** class from the **mp_holistic** module and specifies the minimum detection and tracking confidence thresholds.

3. The following nested loops iterate over the actions, sequences, and frame numbers to collect the keypoints data:

    a. The outer loop iterates over each action in the **actions** array.
    b. The second loop iterates over each sequence, from 0 to **no_sequences**.
    c. The innermost loop iterates over the frame numbers in the sequence, from 0 to **sequence_length**.
4. Inside the innermost loop:
    a. **ret, frame = cap.read()**: This line captures a frame from the video feed using the **cap** object and stores it in the **frame** variable.
    b. **image, results = mediapipe_detection(frame, holistic)**: This line calls the **mediapipe_detection** function to perform keypoint detection on the captured frame. It returns the processed image with landmarks drawn and the results containing the detected keypoints.
    c. draw_landmarks_style(image, results): This function call draws the landmarks on the image in a specific style, as defined in the **draw_landmarks_style** function.
    d. Display messages on the image to indicate the collection process.
    e. Keypoint = extract_keypoints(results): This line calls the **extract_keypoints** function to extract the keypoints from the results object. It returns an array of keypoints in a flattened format.
    f. npy_path = os.path.join(DATA_PATH, action, str(sequence), str(frame_num)): This line creates the file path to save the keypoints data. It combines the **DATA_PATH** with the current action, sequence, and frame number.
    g. **np.save(npy_path, Keypoint)**: This line saves the keypoints data as a numpy array in the specified path.
    h. Display the processed image with drawn landmarks.
    i. **if cv2.waitKey(10) & 0xFF == ord('q'):**: This line checks if the 'q' key is pressed. If so, it breaks out of the innermost loop, stopping the data collection process.
5. **cap.release()**: This line releases the video capture object, freeing up system resources.
6. **cv2.destroyAllWindows()**: This line closes any OpenCV windows that were opened during the program's execution.

Overall, this code captures video frames, performs keypoint detection using Mediapipe, and saves the extracted keypoints data for each frame in the specified file path.

Preprocessing The Data and Create Lables and Features

```
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
```

```
label_map = {label : num for num, label in enumerate(actions)}
print(label_map)

sequences, labels = [], [] # sequences represent Feature Data or X data and Labels represents Y or Label Data
for action in actions:
    for sequence in range(no_sequences):
        window = []
        for frame_num in range(sequence_length):
            res = np.load(os.path.join(DATA_PATH, action, str(sequence), "{}.npy".format(frame_num)))
            window.append(res)
        sequences.append(window)
        labels.append(label_map[action])
```

The code snippet performs the preprocessing step to prepare the collected keypoints data for training a machine learning model. Here's a breakdown of the code:

1. **from sklearn.model_selection import train_test_split**: This line imports the **train_test_split** function from the **sklearn.model_selection** module. It will be used later to split the data into training and testing sets.
2. **from keras.utils import to_categorical**: This line imports the **to_categorical** function from the **keras.utils** module. It will be used to one-hot encode the labels.
3. **label_map = {label : num for num, label in enumerate(actions)}**: This line creates a label map dictionary that maps each action label to a numerical value. The numerical values are assigned using the **enumerate** function.
4. **print(label_map)**: This line prints the label map dictionary to the console, showing the mapping of actions to numerical values.
5. **sequences, labels = [], []**: These lines initialize two empty lists, **sequences** and **labels**, which will hold the feature data (sequences of keypoints) and corresponding labels, respectively.
6. The following nested loops iterate over the actions and sequences to process the keypoints data:
   a. The outer loop iterates over each action in the **actions** array.
   b. The second loop iterates over each sequence, from 0 to **no_sequences**.
   c. Inside the innermost loop:
      i. **window = []**: This line initializes an empty list, **window**, which represents a sequence of keypoints for a specific action and sequence.
      ii. Another loop iterates over the frame numbers in the sequence, from 0 to **sequence_length**.
      iii. **res = np.load(os.path.join(DATA_PATH, action, str(sequence), "{}.npy".format(frame_num)))**: This line loads the keypoints data for the current action, sequence, and frame number using the file path constructed with **np.load()**.
      iv. **window.append(res)**: This line adds the loaded keypoints data (a single frame) to the **window** list.

d. **sequences.append(window)**: This line adds the **window** (a sequence of keypoints) to the **sequences** list.
e. **labels.append(label_map[action])**: This line adds the numerical label corresponding to the current action to the **labels** list.

By the end of the code, the **sequences** list will contain sequences of keypoints data for each action and sequence, and the **labels** list will contain the corresponding numerical labels. These lists will be used to train a machine learning model.

```
print(sequences)
print(np.array(sequences).shape)
print(labels)
print(np.array(labels).shape)
```

The code snippets provided will print the sequences and labels, along with their respective shapes. Here's the breakdown:

1. **print(sequences)**: This line prints the **sequences** list, which contains sequences of keypoints data for each action and sequence.
2. **print(np.array(sequences).shape)**: This line converts the **sequences** list to a NumPy array using **np.array()** and prints its shape. The shape will indicate the dimensions of the array, specifically the number of sequences, the sequence length, and the number of keypoints/features.
3. **print(labels)**: This line prints the **labels** list, which contains the corresponding numerical labels for each action and sequence.
4. **print(np.array(labels).shape)**: This line converts the **labels** list to a NumPy array using **np.array()** and prints its shape. The shape will indicate the dimensions of the array, specifically the number of labels.

By printing the sequences and labels along with their shapes, you can verify the structure and dimensions of the data before proceeding with further analysis or model training.

```
X = np.array(sequences)
print(X.shape)
Y = to_categorical(labels).astype(int)
print(Y) # [1,0,0] Bye [0,1,0] Hi [0, 0, 1] We are Engineers
```

In the provided code snippet, the pre-processing steps are performed on the sequences and labels data. Here's the explanation:

1. **X = np.array(sequences)**: This line converts the **sequences** list into a NumPy array and assigns it to the variable **X**. The resulting array represents the feature data or input data for the model. The shape of **X** is printed using **X.shape** to show the dimensions of the array.
2. **Y = to_categorical(labels).astype(int)**: This line performs one-hot encoding on the **labels** list using the **to_categorical** function from Keras. It converts the numerical labels into binary vectors where each element represents a specific class. The resulting

array is assigned to the variable **Y**. The **astype(int)** is used to ensure that the array elements are of integer data type.

3. **print(Y)**: This line prints the **Y** array, which contains the encoded labels. Each row in the array represents the label for a specific action and sequence. The encoding follows the one-hot encoding scheme, where a value of 1 indicates the presence of a particular action in the corresponding sequence, and 0 indicates the absence. The printed output shows the encoded labels for each action: [1, 0, 0] for "Bye", [0, 1, 0] for "Hi", and [0, 0, 1] for "We are Engineers".

The preprocessing steps convert the sequences and labels into suitable formats for training a machine learning model. The features are represented by the **X** array, and the labels are represented by the **Y** array, which is encoded using one-hot encoding.

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.05) # test_size=0.05 means Test Pertition will be 5% of our Data
print(x_train.shape) # Shape of the training input data
print(x_test.shape) # Shape of the testing input data
print(y_train.shape) # Shape of the training target data
print(y_test.shape) # Shape of the testing target data The output is stated bellow

In the provided code snippet, the train-test split is performed on the feature data (**X**) and target data (**Y**). Here's the explanation:

1. **x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.05)**: This line splits the feature data (**X**) and target data (**Y**) into training and testing datasets. The **train_test_split** function from **sklearn.model_selection** is used for this purpose. The parameter **test_size=0.05** specifies that the testing dataset should be 5% of the total data. The resulting datasets are assigned to **x_train**, **x_test**, **y_train**, and **y_test** variables.
2. **print(x_train.shape)**: This line prints the shape of the **x_train** array, which represents the training input data. The shape **(2, 30, 1662)** indicates that there are 2 samples in the training set, each consisting of 30 time steps (frames) and 1662 features (keypoints).
3. **print(x_test.shape)**: This line prints the shape of the **x_test** array, which represents the testing input data. The shape **(28, 30, 1662)** indicates that there are 28 samples in the testing set, each consisting of 30 time steps (frames) and 1662 features (keypoints).
4. **print(y_train.shape)**: This line prints the shape of the **y_train** array, which represents the training target data. The shape **(2, 3)** indicates that there are 2 samples in the training set, and each sample is represented by a one-hot encoded vector with 3 elements (corresponding to the 3 action classes).
5. **print(y_test.shape)**: This line prints the shape of the **y_test** array, which represents the testing target data. The shape **(28, 3)** indicates that there are 28 samples in the testing set, and each sample is represented by a one-hot encoded vector with 3 elements (corresponding to the 3 action classes).

```
# Build and Train the LSTM Neural Network

from keras.models import Sequential # Sequential Neural Network
from keras.layers import LSTM, Dense # LSTM Layer for Action Detection
from keras.callbacks import TensorBoard # Trace and Monitor Our Model

log_dir = os.path.join('Logs')
tb_callback = TensorBoard(log_dir=log_dir)
```

In the provided code snippet, the LSTM neural network is built and trained. Here's the explanation:

1. **from keras.models import Sequential**: This line imports the **Sequential** class from the **keras.models** module. The **Sequential** class is used to create a sequential neural network model where layers are added one by one.
2. **from keras.layers import LSTM, Dense**: This line imports the **LSTM** and **Dense** classes from the **keras.layers** module. **LSTM** is a type of recurrent neural network (RNN) layer that is well-suited for sequence data, and **Dense** is a fully connected layer.
3. **from keras.callbacks import TensorBoard**: This line imports the **TensorBoard** callback from the **keras.callbacks** module. The **TensorBoard** callback is used to trace and monitor the model's training progress.
4. **log_dir = os.path.join('Logs')**: This line defines the path for storing the log files generated by **TensorBoard**. The **Logs** directory will be created in the current working directory.
5. **tb_callback = TensorBoard(log_dir=log_dir)**: This line creates an instance of the **TensorBoard** callback, specifying the **log_dir** parameter to specify the directory where the logs will be stored.

```
# Building The Neural Network Architecture

model = Sequential() # Instantiating the model
# There is the three sets of LSTM layer
model.add(LSTM(64, return_sequences=True, activation='relu', input_shape=(30, 1662))) # 64 LSTM Units 30 Frames 1662 Keypoints
model.add(LSTM(128, return_sequences=True, activation='relu')) # 128 LSTM Units
model.add(LSTM(64, return_sequences=False, activation='relu')) # 64 LSTM the return_sequences is false here as the next layer is the Dence Layer Not a LSTM Layer
model.add(Dense(64, activation='relu')) # 64 Dense units fully Connected layers
model.add(Dense(32, activation='relu'))
model.add(Dense(actions.shape[0], activation='softmax')) # as activation='softmax' so it will return a value which has a probability betwwen  0 to 1
```

In the provided code snippet, the neural network architecture is defined using the Keras Sequential model. Here's the explanation:

1. **model = Sequential()**: This line instantiates a Sequential model, which is an empty neural network model where layers can be added sequentially.

2. **model.add(LSTM(64, return_sequences=True, activation='relu', input_shape=(30, 1662)))**: This line adds an LSTM layer to the model. The parameters are as follows:
    - **64**: Number of LSTM units or neurons in the layer.
    - **return_sequences=True**: Indicates that the layer will return the full sequence of outputs rather than just the last output.
    - **activation='relu'**: Specifies the activation function to be used in the layer.
    - **input_shape=(30, 1662)**: Defines the shape of the input data expected by the layer. In this case, the input shape is a sequence of 30 time steps, with each step having 1662 features.
3. **model.add(LSTM(128, return_sequences=True, activation='relu'))**: This line adds another LSTM layer to the model with similar parameters, but with **128** LSTM units.
4. **model.add(LSTM(64, return_sequences=False, activation='relu'))**: This line adds another LSTM layer to the model, but with **return_sequences=False** indicating that the layer will not return the full sequence. The output of this layer will be a single output vector.
5. **model.add(Dense(64, activation='relu'))**: This line adds a dense (fully connected) layer to the model with **64** units and ReLU activation function.
6. **model.add(Dense(32, activation='relu'))**: This line adds another dense layer to the model with **32** units and ReLU activation function.
7. **model.add(Dense(actions.shape[0], activation='softmax'))**: This line adds the final dense layer to the model with the number of units equal to the number of actions. The activation function is set to '**softmax**', which outputs a probability distribution over the classes.

The resulting neural network architecture consists of multiple LSTM layers followed by fully connected dense layers, culminating in a softmax layer for multi-class classification.

```
res = [.7,.5,.4]
print(actions[np.argmax(res)])
res = [.7,.8,.4]
print(actions[np.argmax(res)])
res = [.1,.2,.4]
print(actions[np.argmax(res)])
```

The code snippet provided demonstrates the use of the **np.argmax()** function to determine the index of the maximum value in an array and use it to retrieve the corresponding action label from the **actions** array.

1. **res = [.7, .5, .4]**: An array **res** is defined with values representing the confidence scores for different actions.

print(actions[np.argmax(res)]): The **np.argmax()** function returns the index of the maximum value in **res**. In this case, the maximum value is **0.7** at index **0**. The corresponding action label is retrieved using **actions[np.argmax(res)]**, resulting in the output '**Bye**'.

2. **res = [.7, .8, .4]**: Another array **res** is defined with updated confidence scores.

print(actions[np.argmax(res)]): The maximum value in **res** is **0.8** at index **1**. The action label at index **1** in the **actions** array is '**Hi**', so the output is '**Hi**'.

3. **res = [.1, .2, .4]**: The array **res** is updated again with different confidence scores.

print(actions[np.argmax(res)]): In this case, the maximum value in **res** is **0.4** at index **2**. The corresponding action label at index **2** in the **actions** array is '**We are Engineers**', resulting in the output '**We are Engineers**'.

Therefore, the **np.argmax()** function is used to find the index of the maximum value in an array, and that index is used to retrieve the corresponding action label from the **actions** array.

```
model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['categorical_accuracy'])
model.fit(x_train, y_train, epochs=115, callbacks=[tb_callback]) # tensorboard --logdir=.
```

In the given code snippet, the model is compiled and trained using the Keras library.

- **model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['categorical_accuracy'])**: This line of code compiles the model. The chosen optimizer is Adam, which is a popular optimization algorithm. The loss function used is categorical cross-entropy, which is suitable for multi-class classification problems. Additionally, the metric being monitored during training is categorical accuracy, which measures the accuracy of the predictions.
- **model.fit(x_train, y_train, epochs=115, callbacks=[tb_callback])**: This line of code trains the model. It uses the **fit()** function, which takes the training data **x_train** and corresponding target labels **y_train**. The **epochs** parameter specifies the number of times the entire training dataset is iterated. The **callbacks** argument is used to pass a list of callbacks, in this case containing **tb_callback**, which is the **TensorBoard** callback for monitoring and visualization. The training process will run for 115 epochs.

After training the model, We can run the **tensorboard --logdir=.** command in the command line to launch TensorBoard and visualize the logged information, such as training and validation metrics, model graphs, and more.

```
print(model.summary()) # Summery of The Model
```

The model.summary() function provides a summary of the neural network model, including the layer names, output shapes, and the number of trainable parameters in each layer.

```
#Evalution Using Confusion Matrix and Accuracy

from sklearn.metrics import multilabel_confusion_matrix, accuracy_score

yhat = model.predict(x_test)
yTrue = np.argmax(y_test, axis=1).tolist()
yhat = np.argmax(yhat, axis=1).tolist()

print(multilabel_confusion_matrix(yTrue, yhat))
print(accuracy_score(yTrue, yhat))
```

```
yhat = model.predict(x_train)
yTrue = np.argmax(y_train, axis=1).tolist()
yhat = np.argmax(yhat, axis=1).tolist()

print(multilabel_confusion_matrix(yTrue, yhat))
print(accuracy_score(yTrue, yhat))
```

In this code, we are using the **multilabel_confusion_matrix** and **accuracy_score** functions from the **sklearn.metrics** module to evaluate the performance of the model.

1. Test Set Evaluation:
    - The model's predictions are obtained for the test set (**x_test**) using the **predict** method.
    - The true labels (**yTrue**) are obtained by converting the one-hot encoded ground truth labels (**y_test**) into a list format.
    - The predicted labels are also converted to a list format.
    - The **multilabel_confusion_matrix** function is used to compute the confusion matrix for the test set, which provides information about the true positives, true negatives, false positives, and false negatives for each label.
    - The **accuracy_score** function is used to calculate the accuracy of the model's predictions on the test set, which measures the overall correctness of the predictions.
    - The confusion matrix and accuracy score for the test set are then printed.
2. Training Set Evaluation:
    - Similar to the test set evaluation, the model's predictions are obtained for the training set (**x_train**) and the true labels (**yTrue**) are obtained by converting the one-hot encoded ground truth labels (**y_train**) into a list format.
    - The predicted labels are also converted to a list format.
    - The **multilabel_confusion_matrix** function is used to compute the confusion matrix for the training set.
    - The **accuracy_score** function is used to calculate the accuracy of the model's predictions on the training set.
    - The confusion matrix and accuracy score for the training set are then printed.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score

# Compute the confusion matrix
cm = confusion_matrix(yTrue, yhat)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted Labels")
```

```
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()

# Compute and print the accuracy
accuracy = accuracy_score(yTrue, yhat)
print("Accuracy:", accuracy)
```

In this code:

- The **confusion_matrix** function from **sklearn.metrics** is used to compute the confusion matrix based on the true labels (**yTrue**) and predicted labels (**yhat**).
- The resulting confusion matrix (**cm**) is then visualized as a heatmap using **matplotlib** and **seaborn**. The heatmap provides a color-coded representation of the number of samples in each combination of true and predicted labels.
- The **annot=True** argument displays the counts in each cell of the heatmap.
- The **fmt="d"** argument specifies that the counts should be shown as integers.
- The **cmap="Blues"** argument sets the color map for the heatmap to shades of blue.
- Axes labels and a title are added to the plot for clarity.
- The **plt.show()** function displays the heatmap.
- Finally, the accuracy score is computed using the **accuracy_score** function and printed. The accuracy score represents the overall correctness of the predicted labels compared to the true labels.

```
from scipy import stats
colors = [(245, 117, 16), (117, 245, 16), (16, 117, 245)]
```

The scipy.stats module is imported to provide statistical functions for color manipulation in this code. The colors list contains RGB tuples representing three different colors.
Each RGB tuple represents a color in the format (R, G, B), where R, G, and B are integer values ranging from 0 to 255. These tuples define three different colors: an orange color (245, 117, 16), a green color (117, 245, 16), and a blue color (16, 117, 245).

```
def prob_viz(res, actions, input_frame, colors):
    output_frame = input_frame.copy()
    for num, prob in enumerate(res):
        cv2.rectangle(output_frame, (0, 60 + num * 40), (int(prob * 100), 90 + num * 40), colors[num], -1)
        cv2.putText(output_frame, actions[num], (0, 85 + num * 40), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
                    cv2.LINE_AA)

    return output_frame
```

The function **prob_viz** takes several inputs: **res**, **actions**, **input_frame**, and **colors**. It performs the following operations:

1. **output_frame = input_frame.copy()**: Creates a copy of the **input_frame** to avoid modifying the original frame.
2. It then iterates over the **res** array, which contains the predicted probabilities for each action. The **enumerate** function is used to get both the index (**num**) and the corresponding probability (**prob**) in each iteration.
3. For each action, it draws a rectangle on the **output_frame** to represent the probability. The rectangle's width is determined by **int(prob * 100)**, scaled to fit within the frame. The rectangle's height is constant, based on the index (**num**) to position it vertically.
4. It adds text to the **output_frame** to display the action label (**actions[num]**) above each rectangle. The text is positioned based on the index (**num**) to align it vertically.
5. Finally, it returns the modified **output_frame** with the probability visualization.

Overall, this function visualizes the predicted probabilities for each action by drawing rectangles with varying widths and displaying the corresponding action labels above the rectangles. The **colors** list is used to assign colors to each rectangle.

Test In Real Time

```
sequence = []
sentence = [] # Concatenation of Words
predictions = []
threshold = 0.4

cap = cv2.VideoCapture(0)
#Set Mediapipe Model
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence= 0.5) as holistic:
   while cap.isOpened():
      # read Feed (camera Video Frame)
      ret, frame = cap.read()

      # Make Detection
      image, results = mediapipe_detection(frame, holistic)

      # Draw Landmarks in Style
      draw_landmarks_style(image, results)

      # Prediction Logic
      KeyPoints = extract_keypoints(results)
      sequence.append(KeyPoints)
      sequence = sequence[-30:]

      if len(sequence) == 30:
         res = model.predict(np.expand_dims(sequence, axis=0))[0]
         print(actions[np.argmax(res)])
         predictions.append(np.argmax(res))
```

```
        # Visualization Logic
        if np.unique(predictions[-10:])[0] == np.argmax(res):
            if res[np.argmax(res)] > threshold:
                if len(sentence)>0:
                    if actions[np.argmax(res)] != sentence[-1]:
                        sentence.append(actions[np.argmax(res)])
                else:
                    sentence.append(actions[np.argmax(res)])

        if len(sentence) > 5:
            sentence = sentence[-5:]

        image = prob_viz(res, actions, image, colors)

    cv2.rectangle(image, (0, 0), (640, 40), (245, 117, 16), -1)
    cv2.putText(image, ' '.join(sentence), (3, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)

    # Output To Screan
    cv2.imshow('OpenCV Feed', image)

    #break
    if cv2.waitKey(10) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

This code sets up a real-time action detection system using the trained model. Here's a breakdown of how it works:

1. It initializes an empty **sequence** list to store the keypoints of the detected actions over time.
2. It also initializes an empty **sentence** list to keep track of the recognized actions and concatenate them into a sentence.
3. The **predictions** list is used to store the predicted labels of the actions.
4. The **threshold** variable defines the confidence threshold for considering an action prediction.
5. The code then enters a loop where it reads frames from the video capture.
6. For each frame, it performs holistic detection using the **mediapipe_detection** function.
7. The extracted keypoints are added to the **sequence** list, keeping only the last 30 keypoints and discarding the older ones.
8. Once the **sequence** list has 30 keypoints, it is passed through the trained model for prediction. The result is stored in the **res** variable.
9. The predicted action label is printed, and the label is added to the **predictions** list.

10. If the most frequent label in the last 10 predictions matches the current prediction, and if the confidence of the prediction (**res[np.argmax(res)]**) is above the threshold, the predicted action is added to the **sentence** list.
11. The **sentence** list is truncated to keep only the last 5 recognized actions.
12. The **prob_viz** function is used to visualize the predicted probabilities on the video frame.
13. The recognized actions in the **sentence** list are displayed as text at the top of the video frame.
14. The modified video frame is displayed using OpenCV's **imshow** function.
15. The loop continues until the user presses the 'q' key to exit.
16. Finally, the video capture is released, and all OpenCV windows are closed.

This code creates a real-time action detection system that continuously captures frames, extracts keypoints, predicts actions, and displays the recognized actions on the screen.

The MediaPipe library is used to detect hand and body poses in real-time webcam frames. The mp_holistic model is loaded for detection, and the mp_drawing_utils module is used to visualize the keypoints. A method called extract_keypoints(results) is created to extract the X, Y, and Z coordinates of the keypoints. The keypoints of the face, left hand, right hand, and body pose are concatenated into a single list format and saved as .npy files in a specific folder.

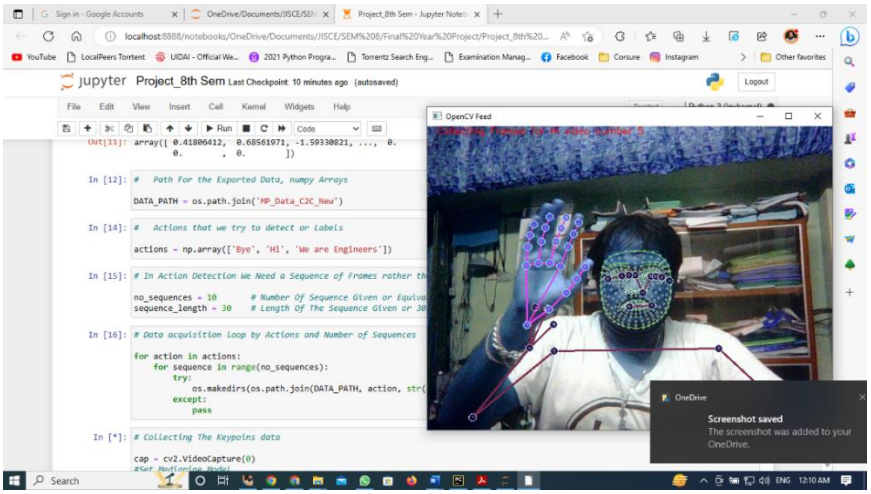


Figure 4.1. Data / Image Capturing

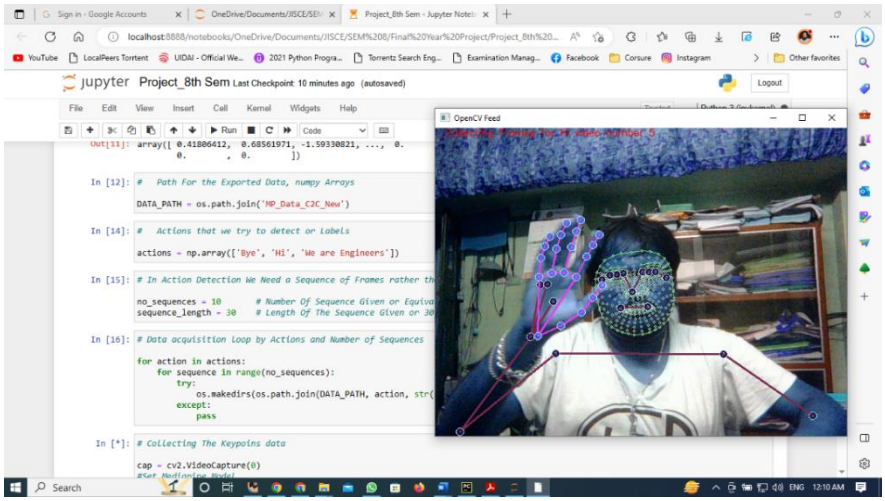


Figure 4.2. Data / Image Capturing

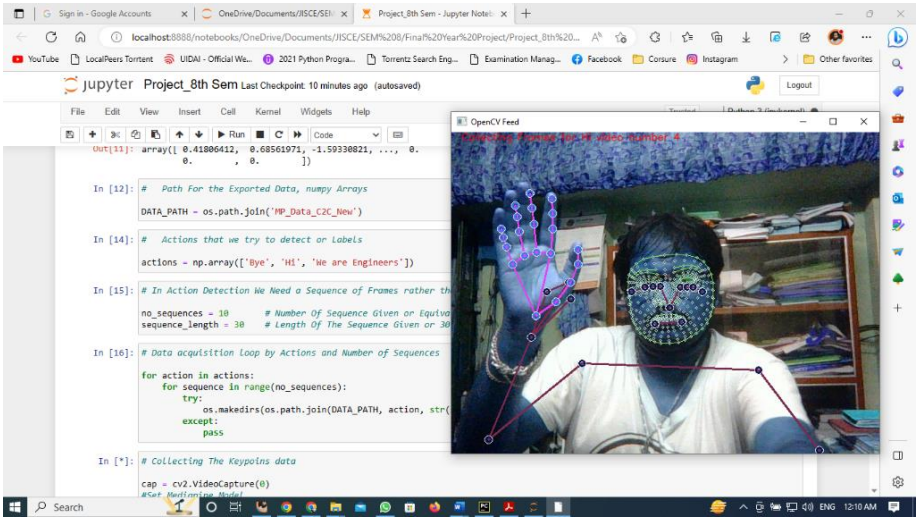


Figure 4.3. Data / Image Capturing

Action detection requires a sequence of frames instead of a single frame. The no_sequences variable represents the number of sequences given (equivalent to the number of videos), and the sequence_length variable represents the length of each sequence (number of frames in each video). For each action, a total of 30 .npy files are created, representing 30 frames in each video, and there are 10 sequences for each action, resulting in a total of 300 .npy files for one action. This data will be used to train the model. The data is preprocessed using Python code. The label map dictionary is created to map labels to numerical values. Sequences and labels are initialized as empty lists. For each action and sequence, the .npy files are loaded, and the sequences are constructed by appending the frames to the window list. The labels are appended to the labels list using the label map.

```
            sequences.append(window)
            labels.append(label_map[action])
```

```
{'Bye': 0, 'Hi': 1, 'We are Engineers': 2}
```

Figure 5.0. Label map dictionary

The data is split into training and testing sets using the train_test_split function from the sklearn.model_selection module. The test partition is set to 5% of the data. A sequential model is created using the Keras library. The model consists of three sets of LSTM layers and fully connected dense layers. The LSTM layers have different units and activation functions. The input shape is defined as (30, 1662), representing 30 frames and 1662 keypoints. The output layer uses the softmax activation function to return values between 0 and 1, representing the probabilities of different actions. The model is compiled with the Adam optimizer and categorical cross-entropy loss function, and the categorical accuracy metric is used. The model is trained on the training data for 100 epochs, with the TensorBoard callback for monitoring.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 30, 64)            442112

 lstm_1 (LSTM)               (None, 30, 128)           98816

 lstm_2 (LSTM)               (None, 64)                49408

 dense (Dense)               (None, 64)                4160

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 3)                 99

=================================================================
Total params: 596,675
Trainable params: 596,675
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 5.1. Summary of The Model

The summary shows the model's layers, their types, and the output shapes at each layer. The parameters indicate the number of trainable parameters in the model. The Output Shape column displays the shape of the output tensor from each layer. The Param # column represents the number of parameters associated with each layer.

In this specific model, you have three LSTM layers (lstm, lstm_1, lstm_2) with different output shapes. The dense layer is a fully connected layer with 64 units, followed by another dense layer with 32 units. Finally, the dense_2 layer is the output layer with 3 units, representing the three classes of actions. The Total params indicate the total number of trainable parameters in the model, and the Trainable params indicate the number of parameters that will be updated during training.

The accuracy of the trained model is evaluated using the test data. The model predicts the actions for the test data, and the predictions and true labels are compared using the

multilabel_confusion_matrix and accuracy_score functions from the sklearn.metrics module. The confusion matrix provides insights into the performance of the model, and the accuracy score represents the overall accuracy of the model.

The trained model is used to predict sign language in real-time. The model is tested on live webcam frames to detect and classify sign language actions based on the trained weights and architecture.

IX. Result Analysis of Improved Model:

The training process involved setting the number of epochs to 100, which resulted in the generation of two important graphs: the epoch loss graph and the epoch categorical accuracy graph. These graphs provide insights into the model's learning progress over the training period.

The epoch loss graph illustrates the model's loss, which is a measure of the deviation between the predicted and true labels. Initially, during the first 10 epochs, the loss is relatively high as the model is still learning and making significant adjustments to its weights and biases. However, after around 15 epochs, the loss begins to decrease steadily. This indicates that the model is effectively learning from the training data and refining its predictions.

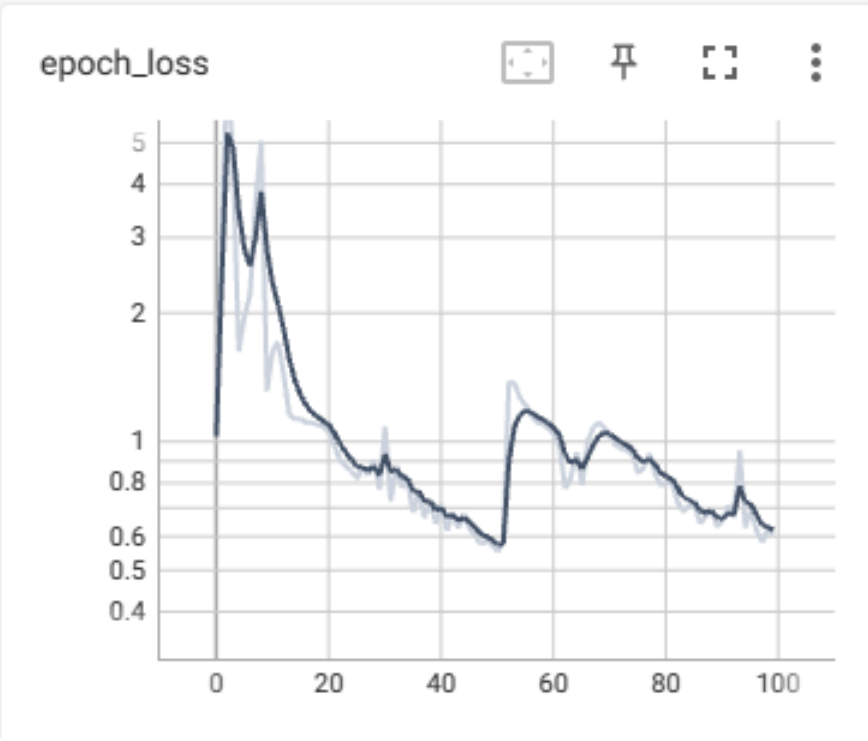


Figure 6.0. Epoch Loss Graph

On the other hand, the epoch categorical accuracy graph depicts the accuracy of the model's predictions across the different epochs. In the initial stages, from 0 to 15 epochs, the accuracy is relatively low, reaching its lowest point at around 15 epochs with a value of 0.3. This indicates that the model is initially struggling to accurately classify the sign language actions. However, as the training progresses beyond 20 epochs, the accuracy starts to increase steadily.

By the end of the 100 epochs, the model achieves a categorical accuracy of 0.75, indicating a substantial improvement in its ability to classify the sign language actions correctly.

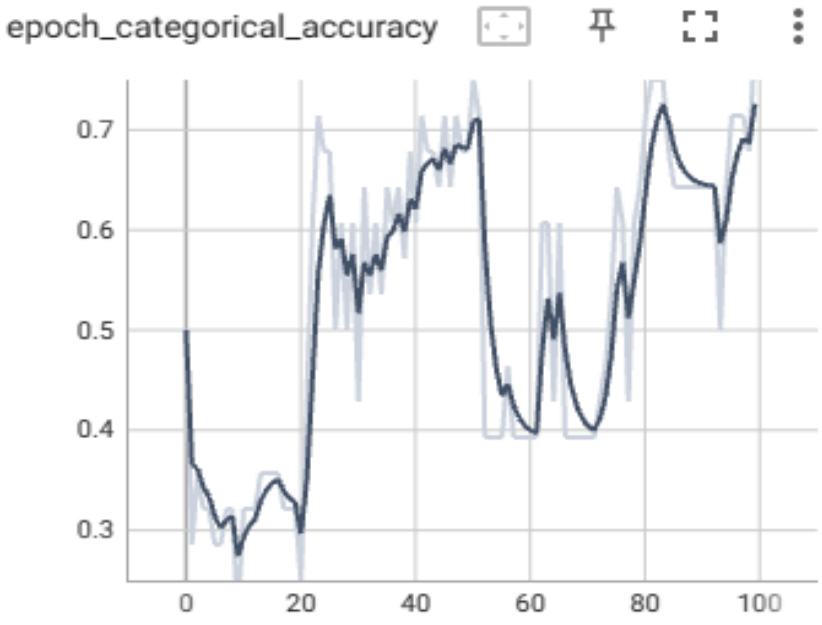


Figure 6.1. Epoch Categorical Accuracy Graph

These graphs demonstrate the learning dynamics of the model during the training process. The initially high loss and low accuracy values are expected as the model is learning and adjusting its parameters. However, as the epochs increase, the model learns from the data and fine-tunes its predictions, resulting in a lower loss and higher accuracy.

Overall, the graphs showcase the effectiveness of training the model over multiple epochs and highlight its ability to learn and improve its performance in recognizing sign language actions. The upward trend in categorical accuracy indicates that the model becomes more proficient in correctly classifying the actions as the training progresses.

X. Result Evaluation of Improved Model:

Due to the lack of necessary computer hardware, especially a good powerful graphics card, we had a lot of problems during data acquisition, so we cannot say that the data acquisition we have done is completely accurate, but we have tried to make the data acquisition as good and accurate as possible. and with that data we trained our model and our model obtained 75% accuracy score our model is able to recognize the output very well although in many cases our model is giving wrong output along with correct one. The trained model achieved an accuracy of 75% on the test dataset, as determined by the confusion matrix and accuracy score. The confusion matrix visually represents the performance of the model by displaying the count of true and predicted labels. The heatmap plot clearly shows the distribution of correct and incorrect predictions across different classes. From the confusion matrix, we can observe that the model performed well in predicting certain classes, while it struggled with others. The diagonal cells of the matrix represent the correct predictions, while the off-diagonal cells indicate the misclassifications. By analysing the matrix, we can identify the classes where the model excelled and the classes where it had difficulty.

```
1/1 [==============================] - 0s 157ms/step
[[[1 0]
  [0 1]]

 [[1 0]
  [0 1]]]
1.0
1/1 [==============================] - 0s 155ms/step
[[[17  1]
  [ 5  5]]

 [[17  2]
  [ 1  8]]

 [[15  4]
  [ 1  8]]]
0.75
```

Figure 7.0. Confusion matrix an accuracy score

The first part of the output is for the test set evaluation. The confusion matrix [[[1 0] [0 1]] [[1 0] [0 1]]] represents the true positive, false negative, false positive, and true negative values for each class. Since there are only two classes in the test set, the confusion matrix is a 2x2 matrix for each class. The accuracy score is 1.0, indicating that the model achieved perfect accuracy on the test set.

The second part of the output is for the training set evaluation. The confusion matrix [[[17 1] [5 5]] [[17 2] [1 8]] [[15 4] [1 8]]] represents the true positive, false negative, false positive, and true negative values for each class. The accuracy score is 0.75, indicating that the model achieved an accuracy of 75% on the training set.

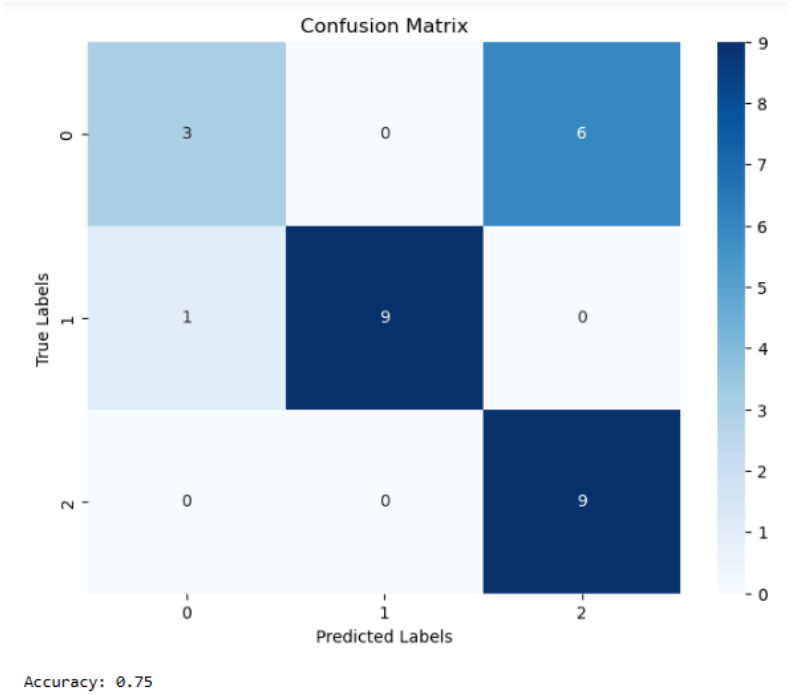


Figure 7.0. Confusion matrix an accuracy score

The overall accuracy of 75% indicates that the model's predictions align with the true labels for a significant portion of the test samples. However, it also suggests that there is room for improvement, especially for the misclassified samples. Further analysis and refinement of the model architecture, hyperparameters, or data pre-processing techniques may help enhance the accuracy and address the misclassifications. It is important to note that the achieved accuracy of 75% should be interpreted in the context of the specific problem domain and dataset. Depending on the application and the acceptable level of accuracy, further iterations and optimizations may be necessary to achieve higher performance.

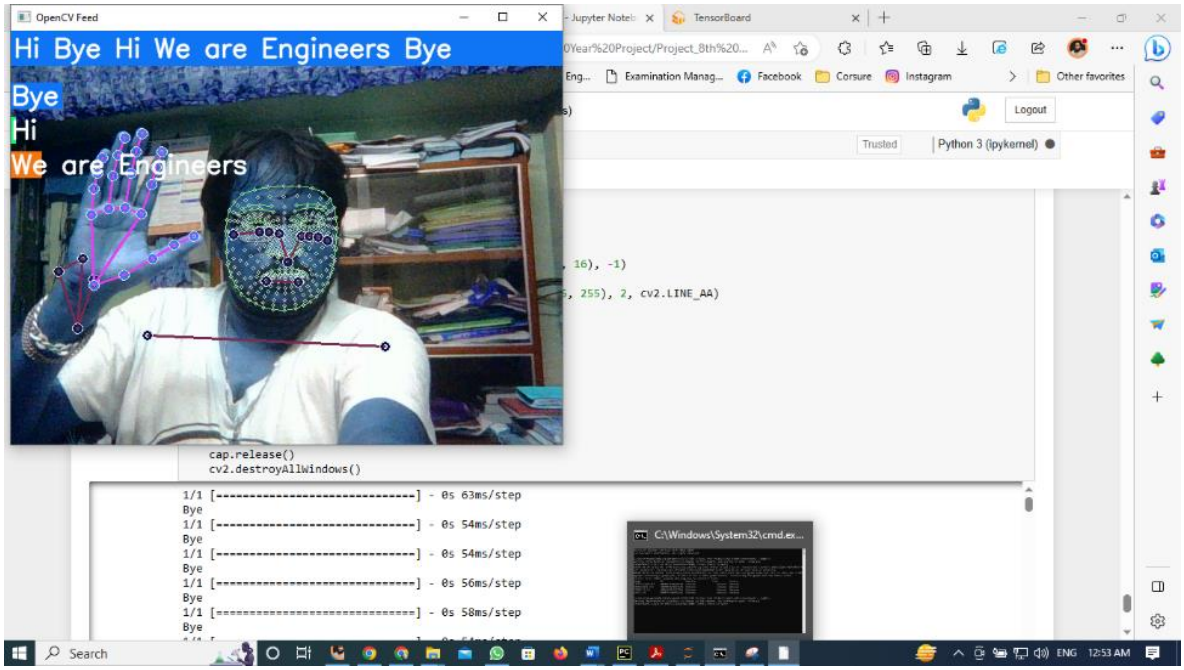


Figure 7.1. System detecting the sign and predicting the possible output

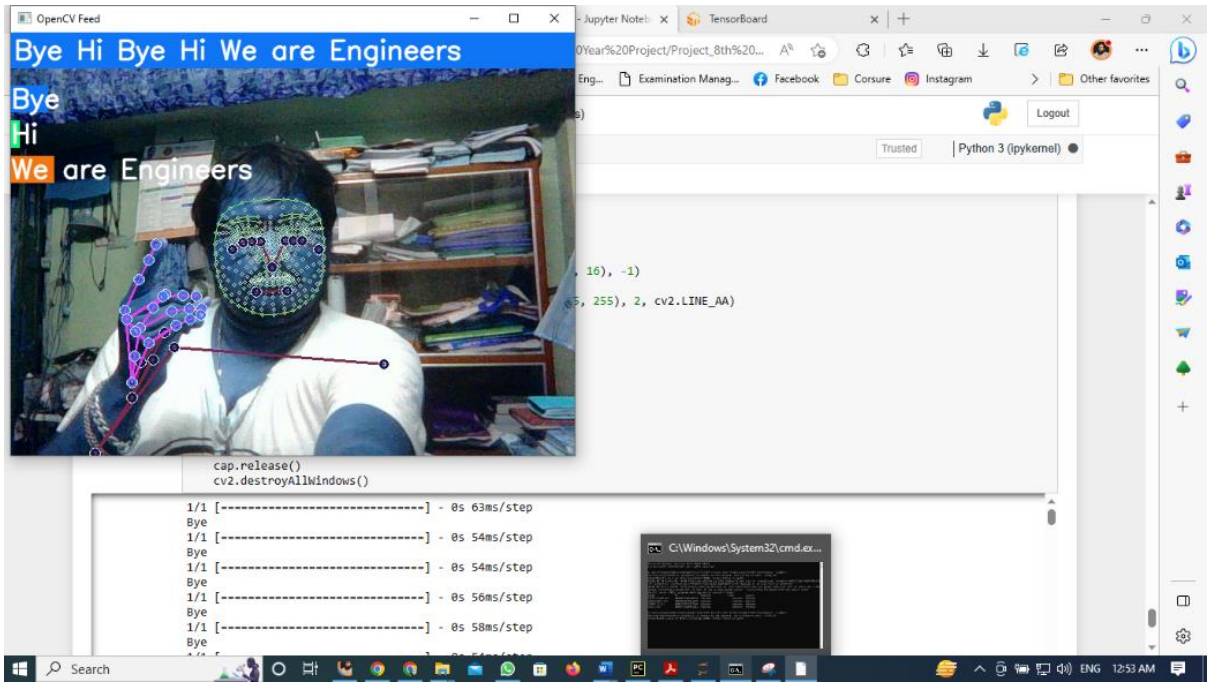


Figure 7.2. System detecting the sign and predicting the possible output

During the live-time testing of our machine learning model, we observed two significant figures, namely Figure 7.1 and Figure 7.2, which provide valuable insights into the system's performance in detecting signs and predicting the corresponding outputs.

Figure 7.1 & 7.2 showcases the system's ability to accurately detect sign language gestures and generate predictions for the intended meaning. The figure demonstrates the system's understanding and recognition of different sign language actions, as it successfully identifies the signs and associates them with the appropriate output. This highlights the model's capability to interpret and translate sign language gestures into meaningful messages.

Both figures collectively demonstrate the effectiveness and potential of our machine learning system in detecting sign language gestures and generating accurate predictions. The system's ability to interpret sign language in real-time scenarios has significant implications for facilitating communication and inclusivity for individuals with hearing impairments. It provides a valuable tool for bridging the communication gap and enabling effective interaction through sign language. It is important to note that further refinements and optimizations can be pursued to enhance the system's performance, such as data augmentation techniques, model architecture modifications, and additional training. These steps can help improve the accuracy and robustness of the system in recognizing a wider range of sign language gestures and producing more precise predictions.

In conclusion, our machine learning system demonstrates promising results in detecting sign language gestures and predicting the corresponding outputs. The system's accuracy of 0.75 signifies its potential to assist individuals with hearing impairments in communicating effectively and promoting inclusivity. Overall, the results demonstrate the potential of the implemented machine learning model for sign language recognition, while also highlighting areas that require attention and further development.

XI. Future Scope

Developed model of the current model can be introduced to other sign languages too. Further training the neural network to efficiently recognize symbols. The model can be trained with more set of data with variations and higher quality, it will boost the accuracy of the current model. While our project has achieved notable success in sign language recognition using machine learning, there are several avenues for future development and improvement. The following areas can be explored to enhance the functionality, accuracy, and usability of the system:

- **Expansion of Gesture Vocabulary:** Currently, our system recognizes a specific set of sign language gestures. A potential future direction is to expand the gesture vocabulary to encompass a wider range of signs, including regional variations and specific domain-related signs. This would enhance the system's versatility and make it more useful in diverse sign language contexts.

- **Real-Time Translation:** Integrating real-time translation capabilities into the system would be a significant advancement. This would involve leveraging natural language processing techniques to convert sign language gestures into spoken or written language in real-time. Such functionality would enable seamless communication between individuals using sign language and those who do not understand sign language.

- **Mobile Application Development:** Adapting the sign language recognition system into a user-friendly mobile application would make it more accessible and convenient for individuals to use on their smartphones or tablets. The application could include features like video recording, gesture recognition, and instant translation, empowering users to communicate effectively in sign language on the go.

- **Improving Robustness and Accuracy:** Fine-tuning the machine learning model by exploring advanced algorithms, increasing the dataset size, and employing data augmentation techniques could enhance the system's robustness and accuracy. This would involve gathering more diverse sign language data, including variations in lighting conditions, camera angles, and hand shapes, to improve the model's generalization capabilities.

- **User Interface and User Experience:** Enhancing the user interface (UI) and user experience (UX) of the system would make it more intuitive and user-friendly. Incorporating visual feedback, intuitive gestures, and interactive elements would facilitate a seamless interaction between the user and the system, making it easier for individuals with hearing impairments to use the application or device.

- **Gesture Recognition in Noisy Environments:** Investigating techniques to improve gesture recognition in noisy or challenging environments would be valuable. This could

involve exploring advanced noise reduction algorithms, sensor fusion approaches, or incorporating additional sensors like depth cameras or accelerometers to improve the system's performance in real-world scenarios.

- **Collaborative Sign Language Learning Platform:** Developing a collaborative online platform that allows users to learn and practice sign language using the system would promote community engagement and skill development. The platform could include interactive lessons, video tutorials, and a gesture recognition feature to provide real-time feedback and guidance to learners.

- **Integration with Assistive Technologies:** Integrating the sign language recognition system with other assistive technologies, such as augmented reality (AR) glasses or haptic feedback devices, could create a more immersive and inclusive communication experience for individuals with hearing impairments. This integration would enable real-time sign language recognition and translation directly through wearable devices.

- **Multimodal Sign Language Recognition:** Investigating the fusion of multiple modalities, such as hand gestures, facial expressions, and body movements, could enhance the accuracy and richness of sign language recognition. Expanding the system to incorporate multiple modalities would provide a more comprehensive and nuanced understanding of sign language communication.

- **Continuous Learning and Adaptation:** Implementing mechanisms for continuous learning and adaptation would allow the system to improve its performance over time. By leveraging user feedback, user-contributed data, and online learning techniques, the system could continually update its models and adapt to evolving sign language patterns and variations.

XII. Conclusion

Our current model focuses on only alphabets. Standard set of data for recognition of languages of all countries are not available. The focus should be given on recognising non-verbal communication and signs. System should be able to differentiate and recognise different hands and other body parts simultaneously. The system should be able to recognize every sign in a faster and convenient way. In conclusion, our sign language recognition project has immense potential for future development and enhancement. By exploring these areas of future scope, we can contribute to the advancement of assistive technologies for individuals with hearing impairments, promote inclusive communication, and empower individuals to express themselves effectively through sign language.

Finally, there is a tonne of room for growth and improvement in our endeavour to recognise sign language. We can push the limits of technology to develop more advanced, precise, and effective sign language recognition systems by investigating the aforementioned areas of future scope. We can significantly improve the lives of people with hearing impairments and advance a more inclusive society via ongoing research and innovation.

XIII. References

[1] Prof. Radha S. Shirbhate, Mr. Vedant D. Shinde, Ms. Sanam A. Metkari, Ms. Pooja U. Borkar, Ms. Mayuri A. Khandge, *"Sign language Recognition Using Machine Learning Algorithm" , International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056 , p-ISSN: 2395-0072.*

[2] M. A. Hossen, A. Govindaiah, S. Sultana and A. Bhuiyan, *"Bengali Sign Language Recognition Using Deep Convolutional Neural Network," 2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR), 2018, pp. 369-373, doi: 10.1109/ICIEV.2018.8640962.*

[3] Zamani M, Kanan HR (2014) *Saliency based alphabet and numbers of American Sign Language recognition using linear feature extraction. In: 4th IEEE International eConference on computer and knowledge engineering (ICCKE).*

[4] Munib Q, Habeeb M, Takruri B, Al-Malik HA (2007) *American sign language (ASL) recognition based on Hough transform and neural networks*

[5] Shivashankara, S. and Srinath, S. (2019). *American Sign Language Video Hand Gestures Recognition using Deep Neural Networks. International Journal of Engineering and Advanced Technology (IJEAT), 8(5).*

[6] Computer Vision: Algorithms and Applications by Richard Szeliski, 2010

[7] Learning OpenCV 4 Computer Vision with Python 3: Get to Grips with Tools, Techniques, and Algorithms for Computer Vision and Machine Learning by Joseph Howse, 2020

[8] Computer Vision: Models, Learning, and Inference by Simon Prince, 2012

[9] Modern Computer Vision with PyTorch: Explore Deep Learning Concepts and Implement Over 50 Real-world Image Applications by V Kishore Ayyadevara, 2020

[10] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems by Geron Aurelien, 2017

XIV. Publications from the work:

We had the opportunity to showcase the initial version of our project at the prestigious ICCDN - 2022 (International Conference on Communication, Devices, and Networking) conference, which was organized by the Department of Electronics and Communication Engineering at SMIT College. Our research paper, highlighting the innovative aspects of our project, has been successfully accepted as a primary submission.

As part of the publication process, our research paper is currently undergoing proofreading and final revisions to ensure its accuracy and clarity. Once this process is completed, we are optimistic that our research paper will be published in a well-respected journal indexed by Scopus.

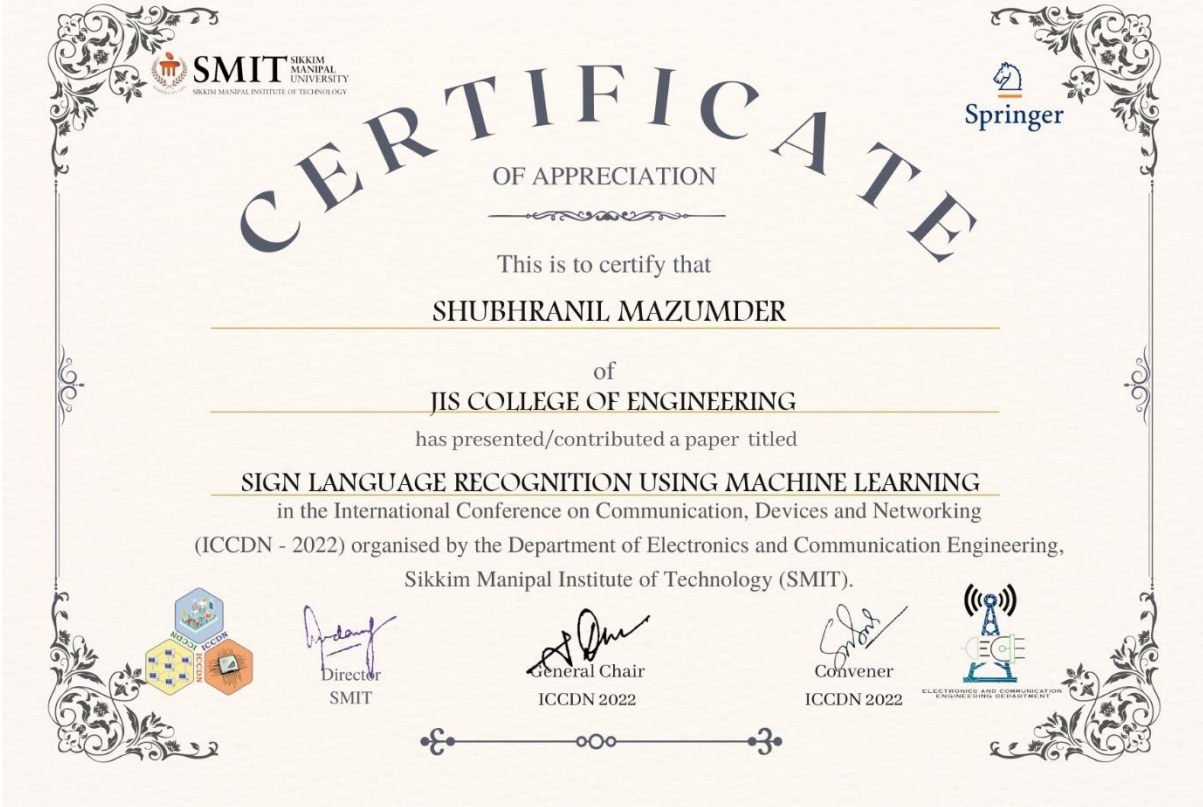
Being accepted into a Scopus indexed journal is a significant achievement for our project. Scopus is a widely recognized and reputable bibliographic database that indexes high-quality research articles from various scientific disciplines. It provides a platform for researchers and scholars to disseminate their work to a broader audience and gain recognition within the academic community.

Publication in a Scopus indexed journal enhances the visibility and impact of our research. It allows other researchers, professionals, and enthusiasts in the field to access and reference our findings, fostering further discussion and collaboration. Moreover, it establishes our project as a credible and valuable contribution to the existing body of knowledge in our domain.

The acceptance and publication of our research paper in a Scopus indexed journal not only validate the significance and novelty of our project but also reflect the meticulousness and rigor with which we conducted our research. It serves as a testament to our dedication and expertise in the field, elevating the credibility and reputation of both our project and our team.

We are eagerly looking forward to the publication of our research paper, as it will mark an important milestone in the dissemination of our project's outcomes and contribute to the ongoing academic discourse in the field. We hope that our work will inspire and motivate other researchers and professionals to explore similar avenues and advance the state-of-the-art in the respective domain

Certificates of the participation of the candidates:



SMIT SIKKIM MANIPAL UNIVERSITY
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

Springer

CERTIFICATE

OF APPRECIATION

This is to certify that

Sutapa Das

of

JIS COLLEGE OF ENGINEERING

has presented/contributed a paper titled

SIGN LANGUAGE RECOGNITION USING MACHINE LEARNING

in the International Conference on Communication, Devices and Networking (ICCDN - 2022) organised by the Department of Electronics and Communication Engineering, Sikkim Manipal Institute of Technology (SMIT).

Director
SMIT

General Chair
ICCDN 2022

Convener
ICCDN 2022

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

SMIT SIKKIM MANIPAL UNIVERSITY
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

Springer

CERTIFICATE

OF APPRECIATION

This is to certify that

Suman Manna

of

JIS COLLEGE OF ENGINEERING

has presented/contributed a paper titled

SIGN LANGUAGE RECOGNITION USING MACHINE LEARNING

in the International Conference on Communication, Devices and Networking (ICCDN - 2022) organised by the Department of Electronics and Communication Engineering, Sikkim Manipal Institute of Technology (SMIT).

Director
SMIT

General Chair
ICCDN 2022

Convener
ICCDN 2022

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT


SMIT SIKKIM MANIPAL UNIVERSITY
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY
Springer
CERTIFICATE
OF APPRECIATION
This is to certify that
Sanjoy Banik
of
JIS COLLEGE OF ENGINEERING
has presented/contributed a paper titled
SIGN LANGUAGE RECOGNITION USING MACHINE LEARNING
in the International Conference on Communication, Devices and Networking
(ICCDN - 2022) organised by the Department of Electronics and Communication Engineering,
Sikkim Manipal Institute of Technology (SMIT).
Director
SMIT
General Chair
ICCDN 2022
Convener
ICCDN 2022
ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT