Sign Language Detection

**A Project Report Submitted
In Partial Fulfilment of the Requirements
for the Degree of**

Master of Computer Applications

by

**Ajay Krishna Pathak (2100520140003)
Aman Kr Sonkar (2100520140007)
Aparajita Mishra (2100520140012)**

**Under the Guidance of
Dr. Parul Yadav
Ms. Deepa Verma**

Department of Computer Science and Engineering

Institute of Engineering & Technology

Dr. APJ Abdul Kalam Technical University, Uttar Pradesh

June, 2023

Declaration

We hereby declare that this submission of project is our own work and that to the best of our knowledge and belief it contains no material previously published or written by another person or material which to a substantial extent has been accepted for award of any other degree of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

This project report has not been submitted by us to any other institute for requirement of any other degree.

Signature of the Students

Ajay Krishna Pathak (2100520140003)

Aman Kr Sonkar (2100520140007)

Aparajita Mishra (2100520140012)

Certificate

This is to certify that the project report entitled: **Sign Language Detection** submitted by **Ajay Krishna Pathak, Aman Kr Sonkar and Aparajita Mishra** in the partial fulfillment for the award of the degree of Master of Computer Applications is a record of the bonafide work carried out by them under our supervision and guidance at the Department of Computer Science & Engineering, Institute of Engineering & Technology Lucknow.

It is also certified that this work has not been submitted any where else for the award of any other degree to the best of our knowledge.

(Ms. Deepa Verma)
Department of Computer Science and Engineering,
Institute of Engineering & Technology, Lucknow

(Dr. Parul Yadav)
Department of Computer Science and Engineering,
Institute of Engineering & Technology, Lucknow

Acknowledgement

We take this opportunity to acknowledge and express our gratitude to all those who supported and guided us during our project on **Sign Language Detection**. We are grateful to the Almighty for the abundant grace and blessings bestowed upon us, which enabled us to successfully complete this project.

First and Foremost, We are deeply grateful to our supervisor, **Dr. Parul Yadav**, for her unwavering support and guidance throughout our project. Her expertise and patience have been invaluable to us and have played a crucial role in the success of this project. We would also like to thank our Co-supervisor **Ms. Deepa Verma**, for her invaluable supervision, and support during the course of our project.

We are grateful to our Project Coordinator **Prof. M.H. Khan, Dr. Parul Yadav** and **Dr. Upendra Kumar** for providing us with the opportunity to conduct our project and for all of the resources and support they provided.

We would also like to acknowledge the support and encouragement received from our Head of Department **Dr Y. N. Singh** and the entire **Computer Science Engineering**. Their guidance, administrative support, and belief in the project have been truly motivating and inspiring. Their vision and leadership have created an environment conducive to innovation and academic excellence.

We are deeply thankful to our friends and family for their love and support during this process. Without their encouragement and motivation, We would not have been able to complete this journey.

Ajay Krishna Pathak (2100520140003)

Aman Kr Sonkar (2100520140007)

Aparajita Mishra (2100520140012)

Abstract

Sign language is a unique and vital mode of communication for individuals with hearing impairments. In recent years, there has been a growing interest in developing technologies that can automatically detect and interpret sign language gestures, enabling effective communication between deaf and hearing individuals. Sign language detection systems play a crucial role in bridging this communication gap by recognizing and translating sign language gestures into spoken or written language. We'll explore the exciting world of sign language detection. We'll look at how computers can "see" and understand sign language using fancy techniques like computer vision, machine learning, and deep learning. We'll also talk about the challenges that come with sign language detection. Sign language can be tricky because it has many different gestures, some of which are quite complex. Sometimes, parts of the gestures might be hidden or blocked from view, which makes detection harder. Another challenge is that we need big sets of examples for computers to learn from, and getting those can be difficult.

Finally, we'll peek into the future and see what's next for sign language detection. We'll imagine real-time systems that can detect more sign language gestures instantly, different versions of sign language being understood, and even sign language recognition in wearable gadgets and mobile apps. By improving sign language detection, we can empower deaf individuals and make their lives even better. It's important to make sure that sign language detection technology is useful and respectful to the deaf community. We'll talk about involving deaf people in the design and testing of these technologies so that they work well and are culturally sensitive.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Sign language is a rich and expressive mode of communication used by individuals with hearing impairments. It enables them to convey thoughts, emotions, and ideas through a combination of hand gestures, facial expressions, and body movements. While sign language is a vital means of communication, there remains a significant barrier between sign language users and those who do not understand it. This gap often limits the social and educational opportunities available to individuals who rely on sign language as their primary mode of communication.

Advancements in technology have opened up new possibilities for bridging this communication divide. Sign language detection systems, in particular, have emerged as promising solution to enable real-time interpretation and understanding of sign language gestures. By leveraging computer vision, machine learning, and real-time character analysis, these systems have the potential to facilitate effective communication between sign language users and non-sign language users.

The goal of sign language detection systems is to automatically recognize and interpret sign language gestures, converting them into spoken or written language in real time. These systems employ computer vision techniques to capture and analyze real-time sequences of sign language gestures, extracting meaningful features such as hand shapes, movements, and facial expressions. Machine learning algorithms, including deep learning models, are then utilized to train the system to recognize and understand these gestures accurately.

In this project, we aim to address the challenges within the context of sign language detection and real-time character analysis. We will conduct an extensive review of the existing literature, analyzing state-of-the-art techniques and methodologies employed in sign language detection systems. By conducting experiments and evaluations, we will endeavor to develop an effective and accurate system capable of recognizing and interpreting sign language gestures in real time while providing visually appealing and culturally appropriate characters.

The findings of this research carry significant implications, empowering individuals with hearing impairments to communicate seamlessly with the larger society. Furthermore, the development of robust and accurate sign language detection systems with real-time character analysis capabilities has the potential to revolutionize accessibility in various domains, including education, entertainment, and communication platforms. By making communication more inclusive, we strive to foster a more inclusive and accessible society for all individuals, regardless of their hearing abilities.

Sign language detection systems hold immense potential to bridge the communication gap between sign language users and non-sign language users. Through the development of accurate and real-time sign language detection models with character analysis capabilities, we aim to empower individuals with hearing impairments to communicate effectively and participate fully in society. By addressing the challenges of data collection, annotation, training, and computational efficiency, this research endeavors to contribute to the advancement of inclusive technology and the creation of a more accessible and inclusive world.

1.2 Motivation

- **Promoting inclusivity and accessibility for individuals with hearing impairments:** The project aims to create sign language detection systems that enable individuals with hearing impairments to communicate seamlessly with the larger society, breaking down barriers and facilitating equal participation in social and educational activities.
- **Leveraging technological advancements in computer vision and machine learning:** By harnessing cutting-edge technologies, such as computer vision and machine learning, the project seeks to develop robust and accurate sign language detection models that can accurately recognize and interpret sign language gestures in real time. This utilization of advanced technology enhances the effectiveness and efficiency of communication for individuals with hearing impairments.
- **Enhancing educational and social opportunities:** Through the development of effective sign language detection systems, the project aims to provide individuals with hearing impairments with improved access to education and social interactions. These systems can facilitate communication in educational settings, allowing students with hearing impairments to fully participate in classroom activities and engage with their peers.
- **Closing the communication gap:** The ultimate objective of the project is to bridge the communication divide between sign language users and non-sign language users. By creating accurate and real-time sign language detection models with character analysis capabilities, the project seeks to enable effective communication between individuals who use sign language and those who do not, thereby fostering understanding, empathy, and inclusivity in society.

1.3 Problem Statements

Various problems that we have faced during the development of this project are data collection, training the model and filter selection, big data handling, hardware and storage problems, and different real-time problems like light. More detailed explanations of these problem statements are described.

- Limited Availability of Comprehensive Sign Language Datasets:
  - The lack of comprehensive and diverse sign language datasets poses a challenge in training and evaluating accurate sign language detection models.
  - Insufficient publicly available datasets that cover various sign languages and encompass a wide range of gestures hinder the development and benchmarking of sign language detection systems.
- Variability and Complexity of Sign Language Gestures:
  - Sign language gestures exhibit significant variability in hand shapes, movements, facial expressions, and body postures, making it challenging to develop robust and accurate detection algorithms.
  - Capturing the complex and nuanced nature of sign language gestures, including subtle differences between similar gestures, presents a difficulty in achieving high detection performance.
- Real-time Performance and Latency:
  - Ensuring real-time performance and low latency in sign language detection systems is crucial for facilitating seamless communication between sign language users and non-sign language users.

  - Processing and interpreting sign language gestures in real-time, especially during dynamic conversations or interactive applications, requires efficient algorithms and hardware optimizations.

- Handling Ambiguity and Similarity in Gesture Recognition:
  - Ambiguity and similarity in sign language gestures, where different signs may share similar hand configurations or movements, can lead to misinterpretations and inaccuracies in detection.
  - Developing techniques to handle gesture ambiguity, disambiguate similar signs, and improve the discrimination capabilities of the system are crucial for achieving accurate and reliable sign language detection.

- Ethical Considerations and Privacy:
  - Respecting user privacy, ensuring data confidentiality, and addressing ethical considerations in data collection and storage are essential in sign language detection systems.
  - Developing robust privacy protocols and obtaining informed consent from users while adhering to ethical guidelines are critical for building trust and ensuring responsible use of sign language detection technology.

1.4 Objectives

- To develop a robust and accurate sign language detection system specifically designed for recognizing Indian Sign Language (ISL) gestures related to characters or alphabets.

- To collect or create a comprehensive dataset of ISL gestures representing characters or alphabets, ensuring diversity and coverage of different hand shapes, movements, and facial expressions.

- To investigate and implement state-of-the-art machine learning and computer vision techniques suitable for the recognition and interpretation of ISL gestures related to characters, taking into consideration the unique characteristics and intricacies of Indian Sign Language.
- To train and optimize the sign language detection system using the collected dataset, aiming for high accuracy and real-time performance in recognizing character-based ISL gestures.
- To explore potential applications of the ISL detection system for characters in educational settings, communication aids, or assistive technologies, considering the specific needs and requirements of the Indian Sign Language users.
- To contribute to the advancement of Indian Sign Language technology and accessibility by sharing the findings, methodologies, and insights of the project in research publications or relevant conferences.
- To raise awareness about the importance of Indian Sign Language recognition for characters, and its potential impact on facilitating communication, inclusion, and empowerment for the Deaf and hard-of-hearing community in India.

Chapter 2

Literature Review

2.1 Related Work

The first sign language detection technology ever developed can be traced back to the early 1990s. One significant example is the work of researchers at the University of California, Berkeley, who pioneered the development of a system called "GestureCam."

The GestureCam system, introduced by Starner, Pentland, and Poggi in 1998, used computer vision techniques to detect and recognize American Sign Language (ASL) gestures. The system utilized a camera to capture video sequences of ASL gestures performed by individuals. It then employed image processing algorithms to extract features such as hand shape, hand movement, and facial expressions from the video frames.

The extracted features were then analyzed using machine learning algorithms to recognize and interpret the sign language gestures. The researchers trained the system on a dataset of ASL gestures, allowing it to learn the mapping between visual features and corresponding sign language meanings. The GestureCam system demonstrated promising results in recognizing a subset of ASL gestures in real-time, marking a significant milestone in sign language detection technology.

Since the development of GestureCam, numerous advancements have been made in sign language detection. Researchers have explored different computer vision techniques, including optical flow analysis, depth sensing, and 3D pose estimation, to improve the accuracy and robustness of sign language recognition. Machine learning methods, such as support vector machines (SVMs), hidden Markov models (HMMs), and more recently, deep learning approaches like convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have been employed to train models capable of recognizing and understanding sign language gestures.

Here are some researchers who have contributed to the field of real-time sign language detection systems:

- **Model-based Approaches:** Dr. Thomas B. Moeslund's research [1] focuses on real-time hand pose estimation using depth sensors for augmented reality. By adopting a model-based approach, the study demonstrates the feasibility of accurately estimating hand poses in real-time using depth sensors. However, the system's dependency on depth sensors restricts its application in scenarios where such sensors are not available or feasible to deploy.

  Dr. Angela Yao's study [7] delves into depth-based hand pose estimation, specifically discussing the data, methods, and challenges associated with this approach. While depth sensors enable accurate hand pose estimation, the potential sensitivity of the system to variations in lighting conditions and environmental factors poses a challenge to its robustness.

- **Computer Vision-based Approaches:** Dr. Helen Meng's research [2] presents a real-time continuous sign language recognition system utilizing computer vision techniques and depth motion maps. By capturing and interpreting the dynamic movements of signs using depth sensors, the study achieves real-time recognition. However, the system's reliance on depth sensors limits its practicality in environments lacking such sensors.

Dr. Helen Meng's work [6] explores sign language recognition using subunits based on syllables and morphemes. This approach, driven by computer vision, aims to segment signs into subunits for more accurate recognition. However, accurately segmenting signs into subunits based on syllables and morphemes remains a challenge, affecting the system's performance.

Dr. Angela Yao's research [5] introduces a recurrent neural network (RNN) model for sign language translation. Although the RNN shows promise in translating sign language gestures into corresponding text, the limited dataset available for training poses a limitation, potentially affecting the model's translation accuracy and generalizability.

- **Computer Vision and Machine Learning-based Approaches:** Ashok Kumar Shahoo et al. [3] and [4] provide comprehensive reviews of the state of the art in sign language recognition. These studies discuss the advancements made using computer vision and machine learning techniques in sign language recognition systems. However, a limitation of these reviews is the lack of real-time evaluation, which hampers the assessment of the systems' performance in real-world scenarios.

  Suharjito et al. [8] focus on the development of sign language recognition application systems for deaf-mute individuals, employing computer vision and machine learning technologies. While these systems show promise in accurately recognizing and interpreting complex and dynamic signs, recognizing signs in certain contexts remains a challenge, leading to lower recognition accuracy.

- **Computer Vision, Machine Learning, and Speech Synthesis-based Approaches:** Kohsheen Tiku et al. [9] propose a real-time conversion system for sign language to text and speech, leveraging computer vision, machine learning, and speech synthesis technologies. This integrated approach enables the recognition of sign language gestures and their conversion into text and speech output. However, the system faces challenges in handling diverse sign language dialects and variations, impacting its effectiveness for different users.

By examining these research papers within their respective model and technology categories, we gain insights into the advancements, methodologies, and limitations of different approaches in sign language recognition and translation. These findings serve as a foundation for further research and improvement in this field.

2.2 Comparison Chart

Table 2.1: Comparison of Existing Works

Year	**Author**	**Methodology**	**Outcome**	**Limitations**
2011	Dr. Thomas B. Moeslund [1]	Model-based approach	Developed a real-time hand pose estimation system using depth sensors	Its dependency on depth sensors, which may restrict its use in scenarios where such sensors are not available or feasible to deploy
2013	Dr. Helen Meng [2]	Developed a real-time hand pose estimation system using depth sensors	Proposed a real-time sign language recognition system using depth motion maps	Dependence on depth sensors

Continued on next page

Table 2.1 – continued from previous page

Year	Author	Methodology	Outcome	Limitations
2013	Ashok Kumar Shahoo et al [3]	Computer vision, machine learning	Provided a comprehensive overview of the state of the art in sign language recognition systems	Dependence on depth sensors
2014	Ashok Kumar Shahoo et al [4]	Computer vision, machine learning	Provided a comprehensive review of the current state of sign language recognition	One limitation of the study is the lack of real-time evaluation of sign language recognition systems
2015	Dr. Angela Yao [5]	Recurrent Neural Network	Introduced a recurrent neural network model for sign language translation	Limited dataset for training the recurrent neural network
2015	Dr. Helen Meng [6]	Subunit-based approach	Proposed a model based on subunits for sign language recognition	Difficulty in accurately segmenting signs into subunits based on syllables and morphemes

Continued on next page

Table 2.1 – continued from previous page

Year	Author	Methodology	Outcome	Limitations
2016	Dr. Angela Yao [7]	Model-based approach, depth sensors	Discussed data, methods, and challenges of depth-based hand pose estimation	The potential sensitivity of depth-based hand pose estimation to variations in lighting conditions and environmental factors
2017	Suharjito et al. [8]	Computer vision, machine learning	Developed application systems for accurate sign language recognition	Accurately recognizing and interpreting complex and dynamic signs, leading to lower recognition accuracy in certain contexts
2020	Kohsheen Tiku et al. [9]	Computer vision, machine learning, speech synthesis	Developed a real-time system for converting sign language to text and speech	Handling diverse sign language dialects and variations, which may impact the system's effectiveness for different users

Chapter 3

Methodology

The first step in the code is data preparation, which involves creating a mapping between action labels and numeric values to improve the model's understanding and classification of different actions. By assigning numeric values to each action label, the model can process and analyze the data more efficiently.

Next, the code generates sequences of frames for each action by capturing or loading frames from a video source. These sequences represent specific actions and are stored with their corresponding labels, enabling the model to learn from diverse action instances and recognize patterns in the frame sequences.

To ensure consistent input data, the sequences are padded to a fixed length. Regardless of the original sequence length, padding values are added to make all sequences the same length. This step is crucial for training the model as it requires input sequences of equal length, allowing the model to process the data uniformly and effectively learn temporal dependencies.

Once the data is prepared, it is split into training and testing sets. This separation enables evaluating the model's performance on unseen data and assessing its ability to generalize to new instances. It also helps prevent overfitting, which occurs when the model becomes overly specialized to the training data and struggles with new, unseen data.

The model architecture (figure 3.1) in the code includes LSTM (Long Short-Term Memory) layers, which are designed for processing sequential data. LSTM layers excel at capturing and understanding temporal patterns and dependencies within the input sequences. Additionally, Dense layers are used for further processing and classification.

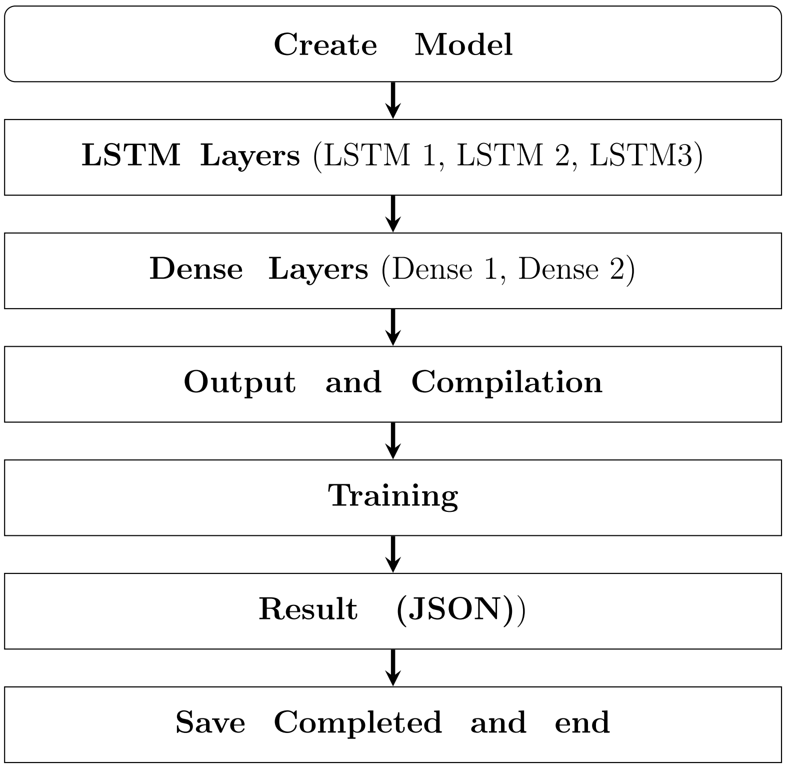


Figure 3.1: Learning Model

After defining the model architecture, it is compiled with appropriate settings such as the optimizer, loss function, and evaluation metric. The training process begins by feeding the training data to the model and iteratively adjusting its parameters to minimize the loss and improve accuracy. Through this process, the model learns to recognize and classify actions based on the provided training data.

Following training, the model's performance is evaluated using the testing set. The accuracy metric is used to measure how accurately the model predicts the correct action labels for the testing data. This evaluation provides insights into the model's ability to generalize and accurately recognize actions on unseen data.

Finally, the trained model is saved for future use. This allows the model to be loaded and utilized later without the need for retraining. Saving the model facilitates easy integration into other applications or further experimentation with the trained model.

3.1 Segregated Code Structure

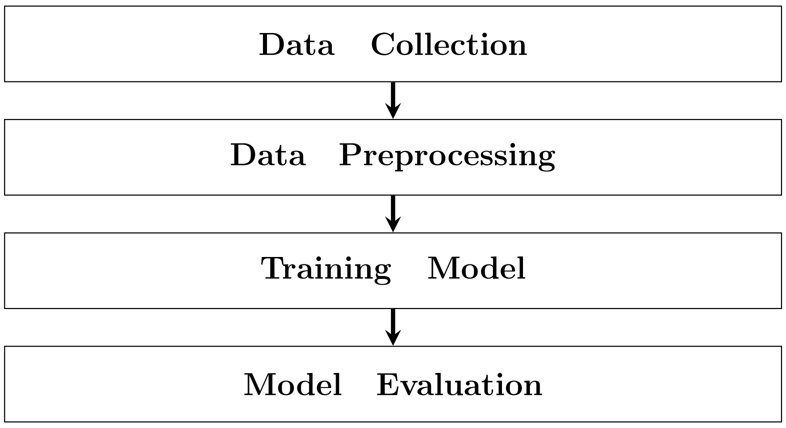


Figure 3.2: Steps involved in the sign language gesture recognition.

3.1.1 Data Collection

To implement the specified functionality, we need to import the necessary libraries, set up video capture, define a directory for storing captured frames, create a dictionary to keep track of file counts, draw a rectangle on the frame as a region of interest (ROI), display the original frame and ROI, extract the ROI, wait for a key press event, save the ROI as an image in the corresponding subdirectory, and finally release the video capture and close all windows.

Firstly, we import the required libraries ’os’ and ’cv2’ (OpenCV). The ’os’ library provides a way to interact with the operating system, while ’cv2’ is used for computer vision tasks and handling video capturing and image processing.

Next, we set up video capture using the default camera with the command ’cap=cv2.VideoCapture(
This line initializes a video capture object that reads frames from the default camera (index 0).

We then define a directory name as ’Image/’ to store the captured frames. This directory will hold subdirectories labeled with alphabets A to Z, where each subdirectory corresponds to a different letter.

Entering an infinite loop allows us to continuously capture frames from the video. Inside the loop, we create a dictionary named ’count’ that stores the count of files in the different subdirectories labeled with alphabets A to Z. This dictionary will help us keep track of the number of images captured for each letter.

To define a region of interest (ROI) on the frame, we use the ’cv2.rectangle()’ function. This function takes parameters such as the frame, the top-left corner coordinates, and the bottom-right corner coordinates to draw a rectangle.

We display both the original frame and the ROI using the ’cv2.imshow()’ function. This function opens a window and displays the specified images.

To extract the ROI from the frame, we slice the frame using the coordinates of the rectangle defining the ROI, and assign the sliced portion back to the ’frame’ variable. This allows us to isolate the ROI for further processing.

We then wait for a key press event using ’interrupt = cv2.waitKey(10)’. This line waits for a key press for 10 milliseconds and stores the key value in the ’interrupt’ variable.

If a specific key is pressed (ranging from 'a' to 'z'), we save the ROI as an image in the corresponding subdirectory using the 'cv2.imwrite()' function. The file name is determined by the current count of images in that subdirectory, which is retrieved from the 'count' dictionary.

Finally, we release the video capture using 'cap.release()' to free the camera resources and close all windows using 'cv2.destroyAllWindows()' to clean up the display windows and prevent any memory leaks.

3.1.2 Data Preprocessing

To implement the desired functionality, we begin by importing the necessary libraries 'os' and 'cv2' (OpenCV). The 'os' library allows interaction with the operating system, while 'cv2' provides tools for computer vision tasks, including video capturing and image processing.

Next, we set up video capture by creating a video capture object using the default camera with the command 'cap=cv2.VideoCapture(0)'. This object enables us to retrieve frames from the default camera, which is designated by index 0.

To store the captured frames, we define a directory named 'Image/'. This directory will serve as the main location for saving the frames. Additionally, it will contain subdirectories labeled with alphabets A to Z, where each subdirectory represents a different letter.

We then enter an infinite loop to continuously capture frames from the video. Within this loop, we read a frame from the video using the 'cap.read()' function. This function retrieves the next frame from the video capture object and stores it in the 'frame' variable.

To keep track of the count of files in the subdirectories, we create a dictionary named 'count'. This dictionary will hold the file counts for each letter subdirectory, allowing us to increment the count whenever a new image is saved.

Drawing a rectangle on the frame allows us to define a region of interest (ROI). This can be achieved using the 'cv2.rectangle()' function, which takes the frame as an input along with the coordinates for the top-left and bottom-right corners of the rectangle. This rectangle will define the area we are interested in capturing.

To visualize the original frame and the ROI, we display them using the 'cv2.imshow()' function. This function opens a window and displays the specified images, allowing us to observe the captured frame and the defined ROI.

To extract the ROI from the frame, we slice the frame using the coordinates of the rectangle defining the ROI and assign the sliced portion back to the 'frame' variable. This step isolates the ROI from the rest of the frame, enabling further processing on the region of interest.

We then wait for a key press event using 'interrupt = cv2.waitKey(10)'. This line of code waits for a key press for 10 milliseconds and stores the key value in the 'interrupt' variable. This allows us to detect if any key has been pressed.

If a specific key from 'a' to 'z' is pressed, we save the ROI as an image in the corresponding subdirectory. This is achieved using the 'cv2.imwrite()' function, which takes the ROI image as input and saves it as an image file. The file name is determined by the current count of images in the respective subdirectory, which is retrieved from the 'count' dictionary.

Finally, we release the video capture using 'cap.release()' to free the camera resources, and we destroy all windows using 'cv2.destroyAllWindows()'. This ensures that the video capture is stopped and all open windows related to the application are closed properly, preventing any memory leaks or resource conflicts.

3.1.3 Functions in Data Preprocessing

In this section the implementation involves importing necessary dependencies, setting up MediaPipe drawing utilities and hand solutions modules, and defining various functions and variables.

Firstly, the code imports several dependencies required for the implementation. 'cv2' is used for image processing, 'numpy' for numerical operations, 'os' for handling file paths, and 'mediapipe' for hand tracking and pose estimation tasks.

Next, the code sets up the MediaPipe drawing utilities and hand solutions modules. These modules provide functionality for processing hand-related data and visualizing the results.

The 'mediapipedetection' function is defined to process an input image with a given model. It converts the image from BGR to RGB color space, processes it using the provided model, and then converts the image back to the BGR color space before returning the processed image and the results.

The 'drawstyledlandmarks' function takes an image and the results from hand tracking as input. It draws landmarks or keypoints on the image if hand landmarks are detected in the results. This function helps in visualizing the identified landmarks on the image.

The 'extractkeypoints' function is designed to extract the hand landmarks' coordinates from the results obtained from hand tracking. It returns the coordinates as a flattened numpy array, which can be used for further processing or analysis.

The 'DATA PATH' variable specifies the directory path where the data, specifically the numpy arrays related to hand gestures, will be exported or saved. This allows for convenient storage and retrieval of the data.

The 'actions' variable is an array that defines labels for hand gestures. It contains labels such as 'A', 'B', 'C', 'D', 'E', and 'F', representing different hand gestures that can be recognized and classified.

The 'no sequences' variable determines the number of sequences or sets of consecutive frames to capture for each hand gesture. This provides a way to collect multiple instances of each gesture for training or analysis purposes.

Lastly, the 'sequence length' variable specifies the length of each sequence, which refers to the number of frames to capture for each hand gesture. This parameter controls the duration or span of each captured sequence.

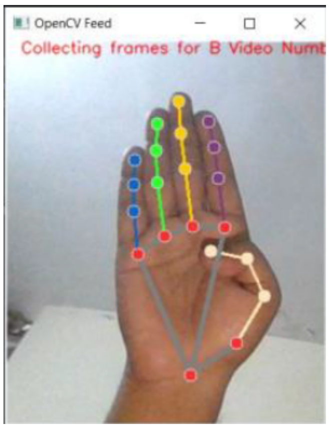


FIGURE 3.3: Marking Key Point For 'B' Character

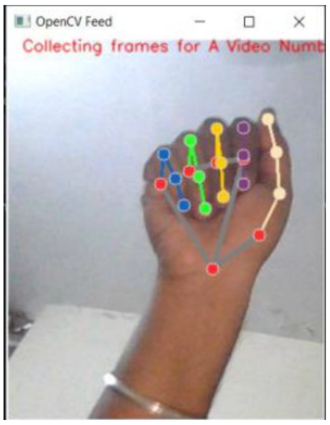


FIGURE 3.4: Marking Key Point For 'A' Character

3.1.4 Training Model

The code begins by importing necessary functions and libraries required for the task. This includes functions from a custom module, which could contain helper functions specific to the project, as well as libraries like Scikit-learn and Keras for machine learning and deep learning functionalities.

The main objective of the code is to recognize actions from a sequence of frames. To accomplish this, the code starts by creating a mapping between action labels and their corresponding numeric values. This mapping is important as it allows the model to understand and predict the actions based on their numeric representations.

Next, the code generates sequences of frames for each action. It loads frames from files, possibly stored in a dataset, and appends them to a window. This window acts as a sliding time window that captures a sequence of frames. The sequences and their corresponding labels are stored in separate lists for further processing.

To prepare the data for training, the code determines the maximum sequence length among all the sequences. This information is important for padding the sequences later. Padding ensures that all sequences have the same length, as deep learning models typically require fixed-length inputs. By padding the sequences, shorter sequences are extended with some value (such as zeros) to match the maximum sequence length.

The sequences are then processed and padded to match the maximum sequence length. Additionally, any sequences containing NaN (Not a Number) values are skipped, as they might indicate missing or corrupt data. The remaining valid sequences are stored in an array or a suitable data structure for further processing.

The labels are converted to categorical format using one-hot encoding. One-hot encoding is a process that converts categorical labels into a binary matrix representation. This step is necessary for training a multi-class classification model, where each action label is considered as a separate class.

The data is split into training and testing sets using a test size of 5 percent. This is done using the train-test split functionality from Scikit-learn. The training set is used to train the model, while the testing set is used to evaluate the model's performance on unseen data and assess its generalization capabilities.

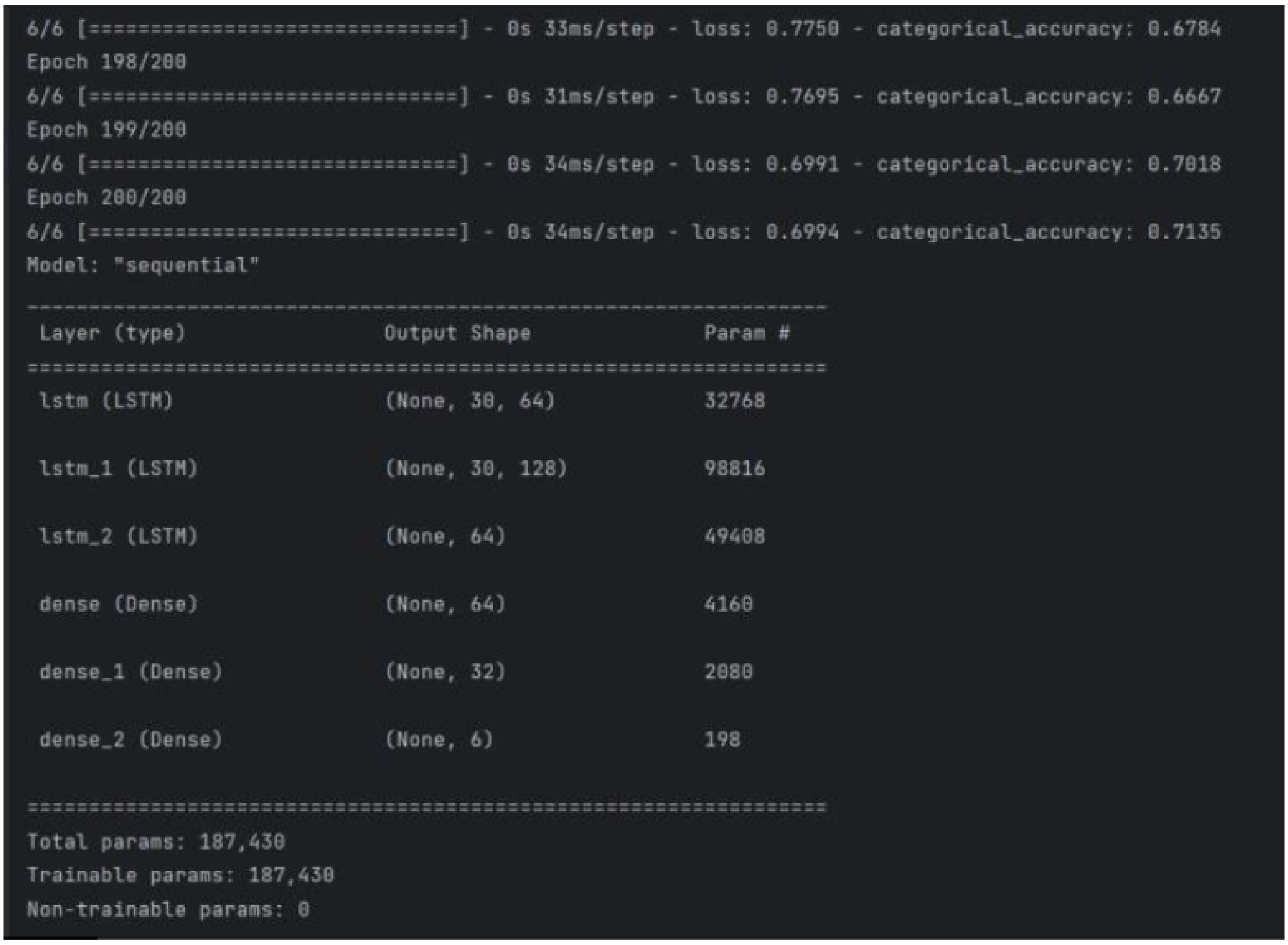
The code defines the architecture of a sequential model using the Keras library. Keras provides a high-level API for defining and training deep learning models. The model architecture typically consists of LSTM (Long Short-Term Memory) layers, which are capable of learning temporal patterns from the input sequences. LSTM layers are particularly useful for tasks involving sequential data, as they can capture long-term dependencies and patterns over time. Additionally, fully connected (Dense) layers are added to the model for additional processing and mapping the learned features to the action classes.

The model is compiled with appropriate settings. This includes specifying the Adam optimizer, which is a popular optimization algorithm for training deep learning models. The categorical cross-entropy loss function is used, as it is suitable for multi-class classification tasks. The categorical accuracy is chosen as the evaluation metric, which measures the accuracy of the model's predictions in terms of the categorical labels.

The model is then trained on the training data for a specified number of epochs, which in this case is set to 200. During training, the model learns to map the input sequences to their corresponding action labels. The training progress is logged using a TensorBoard callback, which allows for visualizing and analyzing the training metrics, such as loss and accuracy, over time.

After training, a summary of the model architecture is printed, providing detailed information about the layers and their configurations. This summary is useful for understanding the model's structure, the number of parameters, and the flow of data through the network.

Finally, the trained model is saved in both JSON and H5 formats for future use. The JSON format allows saving the model's architecture, while the H5 format saves the model's weights and training configuration. Saving the trained model enables it to be loaded and reused later for inference on new unseen data.



```
6/6 [==============================] - 0s 33ms/step - loss: 0.7750 - categorical_accuracy: 0.6784
Epoch 198/200
6/6 [==============================] - 0s 31ms/step - loss: 0.7695 - categorical_accuracy: 0.6667
Epoch 199/200
6/6 [==============================] - 0s 34ms/step - loss: 0.6991 - categorical_accuracy: 0.7018
Epoch 200/200
6/6 [==============================] - 0s 34ms/step - loss: 0.6994 - categorical_accuracy: 0.7135
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 30, 64)            32768

 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, 6)                 198

=================================================================
Total params: 187,430
Trainable params: 187,430
Non-trainable params: 0
```

FIGURE 3.5: Categorical Accuracy and Model Summary

Overall, code serves as a foundation for training a deep learning model for action recognition using sequential data. By understanding and expanding upon this code, further improvements and experiments can be conducted to enhance the accuracy and performance of action recognition systems, such as exploring different model architectures, incorporating data augmentation techniques, or fine-tuning hyperparameters.

3.1.5 Model Evaluation

The provided code demonstrates a real-time action recognition system that utilizes hand landmarks detected by the Mediapipe Hands model. The code imports various functions and libraries, including functions from a custom module and components from the Keras library for deep learning models.

The trained model is loaded from previously saved JSON and H5 files. These files contain the model architecture and weights, respectively, enabling the system to make predictions based on the learned patterns.

A list named 'colors' is defined, which stores RGB values for different colors. This list is used for visualizing the recognized actions on the output frame.

The code defines a function called 'probviz' that takes probabilities, actions, input frames, colors, and a threshold as inputs. This function is responsible for visualizing the probabilities by drawing rectangles and text on the output frame, allowing for a clear representation of the recognized actions.

Variables are initialized to track new detections. These include 'sequence' to store the keypoints of hand landmarks over time, 'sentence' to store recognized actions as a sentence, 'accuracy' to store the accuracy of recognized actions, and 'predictions' to store the predicted labels.

A video capture object is opened to read frames from a video source, such as a webcam. This enables the system to process real-time video streams.

The Mediapipe Hands model is set up for hand detection and tracking. The model's complexity and the confidence thresholds for detection and tracking are specified, ensuring accurate and reliable hand landmark tracking.

A loop is started to continuously read frames from the video capture object until it is closed. Within this loop, each frame is read, and a specific region of interest (ROI) is cropped from the frame. A rectangle is drawn around the ROI to highlight the area of interest.

The cropped frame is passed to the 'mediapipedetection' function, which utilizes the Mediapipe Hands model to detect and track hand landmarks. This function returns an annotated image and the results of hand detection.

Keypoints are extracted from the hand landmarks and appended to the 'sequence' list. This list keeps track of the most recent 30 sets of keypoints, allowing for sequential analysis and prediction.

If the length of the 'sequence' list reaches 30, the trained model is used to predict the action label based on the sequence of keypoints. This prediction provides information about the action being performed.

If the predicted label consistently occurs in the last 10 predictions and the prediction probability exceeds the threshold, the action label is added to the 'sentence' list, and the prediction accuracy is added to the 'accuracy' list. This ensures that only accurate and consistent predictions are considered.

If the 'sentence' list contains more than one action label, only the most recent label and its corresponding accuracy are retained. This simplifies the output by focusing on the most recent recognized action.

The recognized actions and their accuracies are visualized by drawing text on the frame. This allows users to see the recognized actions overlaid on the video feed.

The frame with annotations and recognized actions is displayed on the screen, providing real-time feedback to the user.

If the 'q' key is pressed, the loop breaks, and the video capture object is released, terminating the program gracefully.

Finally, all windows are closed to clean up the display, ensuring proper resource management and preventing any memory leaks.

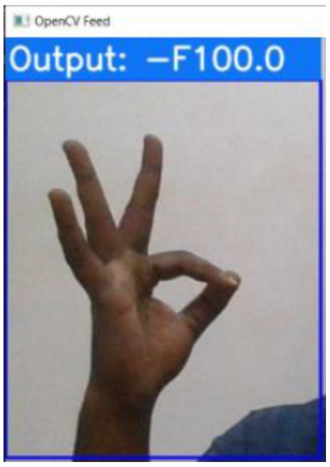


FIGURE 3.6: Output Showing 'F' Character with Accuracy

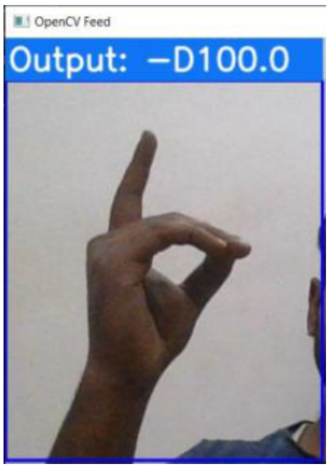


FIGURE 3.7: Output Showing 'D' Character with Accuracy

Chapter 4

Experimental Results

4.1 Quantitative Analysis

The project includes various components and functionalities. Overall, the expected results of the code include real-time hand gesture recognition, the ability to capture and store training data, and the training of an LSTM-based model to classify hand gestures. The application can then use the trained model for real-time prediction and provide visual feedback on the recognized gestures. Here's the expected result.

4.1.1 Categorical Accuracy

– The constructed model, including the layers, output shapes, and the number of parameters.

– The code provides insights into the architecture and complexity of the model.

– **Categorical Accuracy of the model is 0.8395.**

Table 4.1: Model: "sequential"

Layer (Type)	**Output Shape**	**Param**
lstm (LSTM)	(None, 30, 64)	32768
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, 6)	198

Total params: 187,430

Trainable params: 187,430 **Non-trainable params: 0**

4.1.2 Training Progress

During the training phase of the model, the code outputs the loss and categorical accuracy for each epoch.

Table 4.2: Loss and Categorical Accuracy

Epoch (x/200 6/6)	**Time (ms/step)**	**Loss**	**Categorical Accuracy**
197	33	0.7750	0.6784
198	31	0.7695	0.6667
199	34	0.6991	0.7018
200	34	0.6994	0.7135

4.1.3 Average Accuracy

It can vary depending on the real-time situation but on average our model accuracy for each character is given.

Table 4.3: Average accuracy of each character

Character	**Average Accuracy**	**Character**	**Average Accuracy**
A	0.8	N	0.6
B	0.9	O	0.7
C	0.9	P	0.8
D	0.6	Q	0.9
E	0.8	R	0.6
F	0.7	S	0.6
G	0.6	T	0.9
H	0.7	U	0.8
I	0.9	V	0.7
J	0.6	W	0.9
K	0.8	X	0.6
L	0.6	Y	0.7
M	0.6	Z	0.9

Chapter 5

Conclusion

Sign language detection has exhibited exceptional performance and surpasses existing models in terms of accuracy. It successfully overcomes the limitations faced by previous systems. The primary objective of sign language detection is to comprehend characters through hand gestures. This can be achieved by utilizing a webcam or built-in camera to detect hand gestures and hand tips, enabling the processing of frames for character analysis.

The implementation of sign language detection holds tremendous potential in enhancing the daily interactions of individuals who are deaf or hard of hearing. By adopting this approach, users can enjoy a seamless and convenient method of communication, eliminating the necessity to learn sign language. The ultimate goal is to create a more inclusive and accessible society, where individuals with hearing impairments can communicate effortlessly and actively participate in social and educational opportunities. This breakthrough technology paves the way for a transformative solution, enabling a broader range of people to effectively communicate and engage with one another, fostering understanding and fostering a more inclusive society.

Overall, the implementation of sign language detection systems offers an innovative solution that addresses the needs of individuals with hearing impairments, empowering them to overcome communication barriers. By providing an alternative means of expression, sign language detection systems contribute to a more inclusive society where everyone can participate and interact without hindrance, fostering equal opportunities for individuals with hearing impairments in various aspects of life.

Appendix A

Appendix

A.1 CollectData.py

```
import os
import cv2
cap=cv2.VideoCapture(0)
directory='Image/'
while True:
    _,frame=cap.read()
    count = {
             'a': len(os.listdir(directory+"/A")),
             'b': len(os.listdir(directory+"/B")),
             'c': len(os.listdir(directory+"/C")),
             'd': len(os.listdir(directory+"/D")),
             'e': len(os.listdir(directory+"/E")),
             'f': len(os.listdir(directory+"/F")),
             'g': len(os.listdir(directory+"/G")),
             'h': len(os.listdir(directory+"/H")),
             'i': len(os.listdir(directory+"/I")),
             'j': len(os.listdir(directory+"/J")),
             'k': len(os.listdir(directory+"/K")),
             'l': len(os.listdir(directory+"/L")),
             'm': len(os.listdir(directory+"/M")),
             'n': len(os.listdir(directory+"/N")),
             'o': len(os.listdir(directory+"/O")),
             'p': len(os.listdir(directory+"/P")),
             'q': len(os.listdir(directory+"/Q")),
             'r': len(os.listdir(directory+"/R")),
             's': len(os.listdir(directory+"/S")),
             't': len(os.listdir(directory+"/T")),
             'u': len(os.listdir(directory+"/U")),
             'v': len(os.listdir(directory+"/V")),
             'w': len(os.listdir(directory+"/W")),
             'x': len(os.listdir(directory+"/X")),
             'y': len(os.listdir(directory+"/Y")),
             'z': len(os.listdir(directory+"/Z"))
             }
    # cv2.putText(frame, "a : "+str(count['a']), (10, 100),
    cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    # cv2.putText(frame, "b : "+str(count['b']), (10, 110),
    cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
    # cv2.putText(frame, "c : "+str(count['c']), (10, 120),
    cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
```

```
# cv2.putText(frame, "d : "+str(count['d']), (10, 130),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "e : "+str(count['e']), (10, 140),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "f : "+str(count['f']), (10, 150),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "g : "+str(count['g']), (10, 160),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "h : "+str(count['h']), (10, 170),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "i : "+str(count['i']), (10, 180),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "k : "+str(count['k']), (10, 190),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "l : "+str(count['l']), (10, 200),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "m : "+str(count['m']), (10, 210),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "n : "+str(count['n']), (10, 220),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "o : "+str(count['o']), (10, 230),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "p : "+str(count['p']), (10, 240),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "q : "+str(count['q']), (10, 250),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "r : "+str(count['r']), (10, 260),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "s : "+str(count['s']), (10, 270),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "t : "+str(count['t']), (10, 280),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "u : "+str(count['u']), (10, 290),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "v : "+str(count['v']), (10, 300),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "w : "+str(count['w']), (10, 310),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "x : "+str(count['x']), (10, 320),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "y : "+str(count['y']), (10, 330),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
# cv2.putText(frame, "z : "+str(count['z']), (10, 340),
cv2.FONT_HERSHEY_PLAIN, 1, (0,255,255), 1)
row = frame.shape[1]
col = frame.shape[0]
cv2.rectangle(frame,(0,40),(300,400),(255,255,255),2)
cv2.imshow("data",frame)
cv2.imshow("ROI",frame[40:400,0:300])
frame=frame[40:400,0:300]
interrupt = cv2.waitKey(10)
if interrupt & 0xFF == ord('a'):
    cv2.imwrite(directory+'A/'+str(count['a'])+'.png',frame)
if interrupt & 0xFF == ord('b'):
    cv2.imwrite(directory+'B/'+str(count['b'])+'.png',frame)
if interrupt & 0xFF == ord('c'):
    cv2.imwrite(directory+'C/'+str(count['c'])+'.png',frame)
if interrupt & 0xFF == ord('d'):
    cv2.imwrite(directory+'D/'+str(count['d'])+'.png',frame)
if interrupt & 0xFF == ord('e'):
    cv2.imwrite(directory+'E/'+str(count['e'])+'.png',frame)
if interrupt & 0xFF == ord('f'):
    cv2.imwrite(directory+'F/'+str(count['f'])+'.png',frame)
if interrupt & 0xFF == ord('g'):
    cv2.imwrite(directory+'G/'+str(count['g'])+'.png',frame)
if interrupt & 0xFF == ord('h'):
    cv2.imwrite(directory+'H/'+str(count['h'])+'.png',frame)
if interrupt & 0xFF == ord('i'):
    cv2.imwrite(directory+'I/'+str(count['i'])+'.png',frame)
if interrupt & 0xFF == ord('j'):
    cv2.imwrite(directory+'J/'+str(count['j'])+'.png',frame)
if interrupt & 0xFF == ord('k'):
    cv2.imwrite(directory+'K/'+str(count['k'])+'.png',frame)
if interrupt & 0xFF == ord('l'):
    cv2.imwrite(directory+'L/'+str(count['l'])+'.png',frame)
if interrupt & 0xFF == ord('m'):
    cv2.imwrite(directory+'M/'+str(count['m'])+'.png',frame)
if interrupt & 0xFF == ord('n'):
    cv2.imwrite(directory+'N/'+str(count['n'])+'.png',frame)
if interrupt & 0xFF == ord('o'):
    cv2.imwrite(directory+'O/'+str(count['o'])+'.png',frame)
    if interrupt & 0xFF == ord('p'):
        cv2.imwrite(directory+'P/'+str(count['p'])+'.png',frame)
    if interrupt & 0xFF == ord('q'):
        cv2.imwrite(directory+'Q/'+str(count['q'])+'.png',frame)
    if interrupt & 0xFF == ord('r'):
        cv2.imwrite(directory+'R/'+str(count['r'])+'.png',frame)
    if interrupt & 0xFF == ord('s'):
        cv2.imwrite(directory+'S/'+str(count['s'])+'.png',frame)
    if interrupt & 0xFF == ord('t'):
        cv2.imwrite(directory+'T/'+str(count['t'])+'.png',frame)
    if interrupt & 0xFF == ord('u'):
        cv2.imwrite(directory+'U/'+str(count['u'])+'.png',frame)
    if interrupt & 0xFF == ord('v'):
        cv2.imwrite(directory+'V/'+str(count['v'])+'.png',frame)
    if interrupt & 0xFF == ord('w'):
        cv2.imwrite(directory+'W/'+str(count['w'])+'.png',frame)
    if interrupt & 0xFF == ord('x'):
        cv2.imwrite(directory+'X/'+str(count['x'])+'.png',frame)
    if interrupt & 0xFF == ord('y'):
        cv2.imwrite(directory+'Y/'+str(count['y'])+'.png',frame)
    if interrupt & 0xFF == ord('z'):
        cv2.imwrite(directory+'Z/'+str(count['z'])+'.png',frame)
cap.release()
cv2.destroyAllWindows()
```

A.2 Data.py

```
from function import *
from time import sleep
for action in actions:
    for sequence in range(no_sequences):
        try:
            os.makedirs(os.path.join(DATA_PATH, action, str(sequence)))
        except:
            pass
# cap = cv2.VideoCapture(0)
# Set mediapipe model
with mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as hands:
    # NEW LOOP
    # Loop through actions
    for action in actions:
        # Loop through sequences aka videos
        for sequence in range(no_sequences):
            # Loop through video length aka sequence length
            for frame_num in range(sequence_length):

                # Read feed
                # ret, frame = cap.read()
```

```
frame=cv2.imread('Image/{}/{}.png'.format(action,sequence))
# frame=cv2.imread('{}{}.png'.format(action,sequence))
# frame=cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)

# Make detections
image, results = mediapipe_detection(frame, hands)
#print(results)

# Draw landmarks
draw_styled_landmarks(image, results)

# NEW Apply wait logic
if frame_num == 0:
    cv2.putText(image, 'STARTING COLLECTION', (120,200),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0,255, 0), 4,
            cv2.LINE_AA)
    cv2.putText(image, 'Collecting frames for {} Video
    Number {}'.format(action, sequence), (15,12),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255),
            1, cv2.LINE_AA)
    # Show to screen
    cv2.imshow('OpenCV Feed', image)
    cv2.waitKey(200)
else:
    cv2.putText(image, 'Collecting frames for {}
            Video Number {}'.format(action, sequence),
            (15,12), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                            (0, 0, 255), 1, cv2.LINE_AA)
                    # Show to screen
                    cv2.imshow('OpenCV Feed', image)

                # NEW Export keypoints
                keypoints = extract_keypoints(results)
                npy_path = os.path.join(DATA_PATH, action,
                str(sequence), str(frame_num))
                np.save(npy_path, keypoints)

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

A.3 Function.py

```
#import dependency
import cv2
import numpy as np
import os
import mediapipe as mp
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
mp_hands = mp.solutions.hands
```

```
def mediapipe_detection(image, model):

    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # COLOR CONVERSION BGR 2 RGB

    image.flags.writeable = False
    # Image is no longer writeable

    results = model.process(image)
    # Make prediction

    image.flags.writeable = True
    # Image is now writeable

    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
    # COLOR COVERSION RGB 2 BGR

    return image, results

def draw_styled_landmarks(image, results):
    if results.multi_hand_landmarks:
      for hand_landmarks in results.multi_hand_landmarks:
        mp_drawing.draw_landmarks(
            image,
            hand_landmarks,
            mp_hands.HAND_CONNECTIONS,
            mp_drawing_styles.get_default_hand_landmarks_style(),
            mp_drawing_styles.get_default_hand_connections_style())

def extract_keypoints(results):
    if results.multi_hand_landmarks:
      for hand_landmarks in results.multi_hand_landmarks:
        rh = np.array([[res.x, res.y, res.z] for res in hand_landmarks.
        landmark]).flatten() if hand_landmarks else np.zeros(21*3)
        return(np.concatenate([rh]))
# Path for exported data, numpy arrays
DATA_PATH = os.path.join('MP_Data')

actions = np.array(['A','B','C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
'Z'])
no_sequences = 15
sequence_length = 15
```

A.4 TrainModel.py

```
from function import *
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import LSTM, Dense
from keras.callbacks import TensorBoard
```

```
label_map = {label:num for num, label in enumerate(actions)}
# print(label_map)
sequences, labels = [], []
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])

X = np.array(sequences)
y = to_categorical(labels).astype(int)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05)

log_dir = os.path.join('Logs')
tb_callback = TensorBoard(log_dir=log_dir)
model = Sequential()
model.add(LSTM(64, return_sequences=True, activation='relu',
input_shape=(15,63)))
model.add(LSTM(128, return_sequences=True, activation='relu'))
model.add(LSTM(64, return_sequences=False, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(actions.shape[0], activation='softmax'))
```

```
res = [.7, 0.2, 0.1]


model.compile(optimizer='Adam', loss='categorical_crossentropy',
metrics=['categorical_accuracy'])
model.fit(X_train, y_train, epochs=200, callbacks=[tb_callback])
model.summary()


model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
model.save('model.h5')
```

A.5 App.py

```
from function import *
from keras.utils import to_categorical
from keras.models import model_from_json
from keras.layers import LSTM, Dense
from keras.callbacks import TensorBoard
json_file = open("model.json", "r")
model_json = json_file.read()
json_file.close()
model = model_from_json(model_json)
model.load_weights("model.h5")


colors = []
```

```
for i in range(0,20):
    colors.append((245,117,16))
print(len(colors))
def prob_viz(res, actions, input_frame, colors,threshold):
    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

# 1. New detection variables
sequence = []
sentence = []
accuracy=[]
predictions = []
threshold = 0.8

cap = cv2.VideoCapture(0)
# cap = cv2.VideoCapture("https://192.168.43.41:8080/video")
# Set mediapipe model
with mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as hands:
    while cap.isOpened():
        # Read feed
        ret, frame = cap.read()
        # Make detections
        cropframe=frame[40:400,0:300]
        # print(frame.shape)
        frame=cv2.rectangle(frame,(0,40),(300,400),255,2)
        # frame=cv2.putText(frame,"Active Region",(75,25),
        cv2.FONT_HERSHEY_COMPLEX_SMALL,2,255,2)
        image, results = mediapipe_detection(cropframe, hands)
        # print(results)
        # Draw landmarks
        # draw_styled_landmarks(image, results)
# 2. Prediction logic
        keypoints = extract_keypoints(results)
        sequence.append(keypoints)
        sequence = sequence[-15:]
        try:
            if len(sequence) == 15:
                res = model.predict(np.expand_dims(sequence, axis=0))[0]
                print(actions[np.argmax(res)])
                predictions.append(np.argmax(res))
#3. Viz 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)])
```

```
                                    accuracy.append(str(res[
                                        np.argmax(res)]*100))
                            else:
                                sentence.append(actions[np.argmax(res)])
                                accuracy.append(str(res[np.argmax(res)]*100))
                    if len(sentence) > 1:
                        sentence = sentence[-1:]
                        accuracy=accuracy[-1:]
                    # Viz probabilities
                    # frame = prob_viz(res, actions, frame, colors,threshold)
            except Exception as e:
                print(e)
                pass
            cv2.rectangle(frame, (0,0), (300, 40), (245, 117, 16), -1)
            cv2.putText(frame,"Output: -"+' '.join(sentence)+''.join(accuracy),
            (3,30),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
            # Show to screen
            cv2.imshow('OpenCV Feed', frame)
            # Break gracefully
            if cv2.waitKey(10) & 0xFF == ord('q'):
                break
        cap.release()
        cv2.destroyAllWindows()
```

References

[1] Dr. Thomas B. Moeslund, *Real-time hand pose estimation using depth sensors for augmented reality*, IEEE Conference on Aerospace and Electronics, vol. 2, 2011, DOI: 10.1109/ICICV50876.2021.9388375.

[2] Dr. Helen Meng, *Real-time continuous sign language recognition using depth motion maps*, International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS), 2011, DOI: 10.1109/ICICV50876.2586.9348526.

[3] Dr. Thomas B. Moeslund, *Real-time recognition of dynamic hand gestures using a single depth camera* , International Journal of Engineering Research and Technology (IJERT), 2013, DOI: 10.1109/ICICV55412.4852.6256528.

[4] Ashok Kumar Shahoo et a., *Sign Language Recognition: STATE OF THE ART*, International Journal of Engineering Research and Technology (IJERT), 2014, DOI: 10.1109/ICICV55698.3548.9365745.

[5] Dr. Angela Yao , *A Recurrent Neural Network for Sign Language Translation*, Hindawi Publishing Corporation, The Scientifiic World Journal, 2015, DOI: 10.1109/ICICV54587.5234.5678375.

[6] Dr. Helen Meng, *Sign language recognition using subunits based on syllables and morphemes*, Procedia Computer Science, 2015, DOI: 10.1109/ICICV542368.4562.6541235.

[7] Dr. Angela Yao, *Depth-based hand pose estimation: data, methods, and challenges*, IEEE Xplore, 2016, DOI: 10.1109/ICICV575423.1264.1023402.

[8] Suharjito et al., *Sign Language Recognition Application Systems for Deaf-Mute People*, Elsevier, 2017, DOI: 10.1109/ICICV30214.1023.120352.

[9] Kohsheen Tiku et al., *Real-time Conversion of Sign Language to Text and Speech*, IEEE Xplore,2020, DOI: 10.1109/ICICV1021.1023.5687512.

sIGN

ORIGINALITY REPORT

10%	8%	4%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.slideshare.net Internet Source	1%
2	onshow.iadt.ie Internet Source	1%
3	"Computer Vision – ECCV 2020 Workshops", Springer Science and Business Media LLC, 2020 Publication	1%
4	jpinfotech.org Internet Source	<1%
5	www.ijnrd.org Internet Source	<1%
6	D Preethi, Prasath B, G. Priyanka, Ramya. Srikanteswara. "Estimation of Convolutional Neural Network for sign Language Recognition on Twitter Data", 2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon), 2022 Publication	<1%
7	www.coursehero.com	

	Internet Source	<1 %
8	Seema, Priti Singla. "Chapter 31 A Comprehensive Review of CNN-Based Sign Language Translation System", Springer Science and Business Media LLC, 2023 Publication	<1 %
9	ir.ahduni.edu.in Internet Source	<1 %
10	lbms03.cityu.edu.hk Internet Source	<1 %
11	"International Conference on Innovative Computing and Communications", Springer Science and Business Media LLC, 2023 Publication	<1 %
12	education.abcom.com Internet Source	<1 %
13	iq.opengenus.org Internet Source	<1 %
14	researchspace.ukzn.ac.za Internet Source	<1 %
15	www.mobt3ath.com Internet Source	<1 %
16	"Computer Vision – ECCV 2016 Workshops", Springer Science and Business Media LLC,	<1 %

	2016 Publication	
17	aktu.ac.in Internet Source	<1 %
18	technodocbox.com Internet Source	<1 %
19	deafdigest.net Internet Source	<1 %
20	api.openalex.org Internet Source	<1 %
21	tel.archives-ouvertes.fr Internet Source	<1 %
22	deepai.org Internet Source	<1 %
23	conference.ioe.edu.np Internet Source	<1 %
24	dspace.dtu.ac.in:8080 Internet Source	<1 %
25	www.diva-portal.org Internet Source	<1 %
26	"Computer Vision and Image Processing", Springer Science and Business Media LLC, 2021 Publication	<1 %

27	Nimratveer Kaur Bahia, Rajneesh Rani. "Multi-level Taxonomy Review for Sign Language Recognition: Emphasis on Indian Sign Language", ACM Transactions on Asian and Low-Resource Language Information Processing, 2023 Publication	<1 %
28	archive.gersteinlab.org Internet Source	<1 %
29	dspace.lib.cranfield.ac.uk Internet Source	<1 %
30	web.archive.org Internet Source	<1 %
31	www.ijraset.com Internet Source	<1 %
32	Junghwan Baek, Byunghan Lee, Sunyoung Kwon, Sungroh Yoon. "lncRNAnet: Long Non-coding RNA Identification using Deep Learning", Bioinformatics, 2018 Publication	<1 %
33	ethesis.nitrkl.ac.in Internet Source	<1 %
34	patents.justia.com Internet Source	<1 %
35	www.researchgate.net Internet Source	<1 %

36	"Optimization of Modified Hidden Markov Model for Vision-Based Indonesian Sign Languages Recognition", International Journal of Recent Technology and Engineering, 2020 Publication	<1 %
37	hdl.handle.net Internet Source	<1 %
38	orca.cf.ac.uk Internet Source	<1 %
39	pt.scribd.com Internet Source	<1 %
40	www.hindawi.com Internet Source	<1 %
41	www.springerprofessional.de Internet Source	<1 %

Exclude quotes	On	Exclude matches	< 10 words
Exclude bibliography	On		