Real Time Sign Language Recognition System Using Machine Learning

Submitted in fulfillment for the J Component of ITA6004 – Soft Computing

*in*

**MCA**

*by*

**Manvendra Gosain 22MCA0321**

*Under the guidance of*

**Dr. Anitha A Associate Professor Sr.**

**SITE**



# ABSTRACT:

Speech impairment affects an individual's ability to communicate using speech and hearing. In this case, people use other forms of communication, such as sign language. It remains challenging for non-sign language speakers to communicate with sign language speakers or signers, despite the prevalence of sign language in recent times. Recent advances in deep learning and computer vision have made it possible to recognize motion and gestures using deep learning and computer vision. This paper proposes a machine learning based model that detects and recognizes letters from a person’s gestures. This project focuses on real-time sign language recognition using Python, incorporating machine learning algorithms from scikit-learn, namely Random Forest, Support Vector Machines (SVM), and Multilayer Perceptron (MLP). Each model is trained on a labeled dataset of hand gestures, where each gesture corresponds to a specific sign in sign language. The project leverages the OpenCV library to capture real-time video input and employs Mediapipe's hand tracking solution to detect and track hand gestures. The goal of the project is to develop a system that can accurately recognize sign language gestures in real time. The implementation pipeline involves capturing video input through a webcam using OpenCV. The captured frames are then processed using Mediapipe's hand tracking solution to detect and track the hand landmarks.

**Keywords:** Sign Language, Support Vector Machine, Random Forest, Multilayer Perceptron, scikit-learn, OpenCV, Mediapipe

# INTRODUCTION WITH RELATED WORK:

Sign language is an important mode of communication for the deaf and hard-of-hearing community, but it can be difficult for non-signers to understand. In such a situation, a system that converts symbols in sign languages to plain text may enable real-time communication between the two parties involved. Sign language is not a new computer vision problem. Over the past two decades, researchers have addressed it in various ways. The development of sign language detection systems involves several technical challenges, such as recognizing a wide range of gestures, accounting for variations in signing style, and dealing with background noise and lighting conditions. Overall, sign language detection systems have the potential to enhance accessibility and inclusivity for individuals who use sign language, and to promote more effective communication and understanding across different communities. The objective of this research paper is to present a novel real-time sign language detection system that leverages deep learning techniques to enable seamless communication between sign language users and non-signers. By capturing video input in real-time and employing state-of-the-art computer vision algorithms, the system analyzes hand gestures and translates them into corresponding textual or spoken representations. The idea to leverage computer vision to aid in communication between people with such needs is not new. Some of the previously done work is mentioned here; [1]*"Real-time Dynamic Sign Language Recognition Using Temporal Convolutional Networks",* Mingrui Xia, Siyuan Ma, Jiebo Luo, Zicheng Liu(2021). This paper introduces a real-time dynamic sign language recognition system based on Temporal Convolutional Networks (TCNs). The proposed TCN architecture effectively captures temporal dynamics in sign language videos, enabling accurate and efficient recognition in real-time scenarios. To evaluate their approach, the authors conducted experiments on publicly available sign language datasets, including ChaLearn LAP and RWTH-PHOENIX-Weather. The results demonstrate that the proposed TCN-based method achieves state-of-the-art performance in terms of both accuracy and real-time processing speed. The system can recognize dynamic sign language gestures in real-time, making it suitable for applications such as sign language interpretation and communication. Another such work was carried out in, [2]*"Sign Language Recognition Using Deep Convolutional Neural Networks",* V. P. Tatiya, S. S. Sherekar. This conference paper proposes a sign language recognition system using deep convolutional neural networks. The authors leverage a CNN architecture to learn spatial features from sign language images, followed by a classifier for recognition. The system achieves promising results in recognizing different sign language gestures. The proposed method involves several stages, starting with the preprocessing of sign language images to enhance their quality and remove noise. The authors employ techniques such as image normalization, background subtraction, and segmentation to isolate the hand region, which is crucial for gesture recognition. Next, a deep CNN architecture is designed and trained to classify the hand images into different sign language classes. The network consists of multiple convolutional layers, followed by pooling and fully connected layers. The authors experiment with various CNN architectures, including LeNet-5, AlexNet, and VGGNet, and evaluate their performance on sign language datasets*.* [3]*"A Real-Time Deep Learning-Based Sign Language Recognition System for Mobile Devices"* by D. J. Kim (2021). This paper presents a real-time sign language recognition system designed specifically for mobile devices. The authors utilize deep learning techniques, such as convolutional neural networks and long short-term memory networks, to achieve accurate and efficient recognition on mobile platforms.

In conclusion, this research paper aims to advance the field of real-time sign language detection systems by leveraging machine learning techniques and formulate a system that is not only accurate but also efficient and fast in predicting and can be used on an everyday system with limited performing capabilities. By combining computer vision, machine learning, and the principles of inclusivity, the proposed system has the potential to enhance communication accessibility and empower individuals with hearing impairments.

# BACKGROUND FUNDAMENTALS:

* 1. Sign Language:

Sign language is a visual and gestural mode of communication used by deaf individuals and those with hearing impairments. It utilizes hand movements, facial expressions, and body language to convey meaning and express thoughts, just like spoken languages do. Sign languages are unique and distinct in different regions and countries, with their own grammatical rules and vocabulary. Sign language provides a means of communication for individuals who are deaf or hard of hearing. It allows them to express their thoughts, emotions, and ideas effectively, enabling full participation in social, educational, and professional interactions. It also ensures accessibility and inclusivity for individuals with hearing impairments. By recognizing and using sign language, society can break down communication barriers and provide equal opportunities for deaf individuals in education, employment, healthcare, and other aspects of life. It even has its own grammatical structures and linguistic features.

* 1. Machine Learning in Education:

Machine learning plays a crucial role in the education system, particularly for individuals with special needs. By leveraging the power of data analysis and pattern recognition, machine learning algorithms can aid in personalized and adaptive learning experiences for students with diverse learning requirements. These algorithms can analyze and interpret data about individual students, including their strengths, weaknesses, preferences, and progress, to develop tailored learning paths and interventions. Machine learning models can identify specific learning disabilities or challenges and offer targeted interventions, such as personalized exercises or content recommendations. Additionally, these algorithms can assist educators in identifying trends and patterns in student data to improve instructional strategies and develop effective interventions. Overall, machine learning has the potential to transform education for individuals with special needs by providing tailored support, fostering inclusivity, and enhancing the overall learning experience.

* 1. OpenCV:

The OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning library that offers a wide range of tools and algorithms for image and video processing. Developed by Intel, OpenCV has become one of the most widely used libraries in the field of computer vision. It provides functionalities for reading, writing, resizing, and manipulating images, as well as performing tasks such as filtering, edge detection, and image enhancement. OpenCV supports video capturing, playback, and conversion, making it suitable for applications involving real-time video streams.One of the significant strengths of OpenCV lies in its comprehensive collection of computer vision algorithms. It includes pre-implemented algorithms for tasks like object detection, feature extraction, image segmentation, optical flow, and camera calibration. These algorithms empower developers to build applications in areas such as robotics, surveillance, augmented reality, and autonomous vehicles. OpenCV provides camera calibration tools and APIs for camera access, facilitating the integration of cameras into computer vision applications. It also supports depth sensors, enabling depth estimation and 3D vision applications. Moreover, OpenCV includes features for GUI development and user interaction, allowing developers to create graphical overlays, draw shapes, and handle user inputs.

* 1. Scikit-learn:

Scikit-learn, also known as sklearn, is a popular machine learning library in Python that provides a wide range of tools and algorithms for various tasks related to data preprocessing, feature selection, model training, and evaluation. It is built on top of other scientific computing libraries like NumPy, SciPy, and matplotlib, making it easy to integrate with existing Python workflows. One of the key strengths of scikit-learn is its extensive collection of supervised and unsupervised learning algorithms. Among the many algorithms available, scikit-learn includes implementations of three widely used machine learning models: Random Forest, Support Vector Machines (SVM), and Multi-Layer Perceptron (MLP).

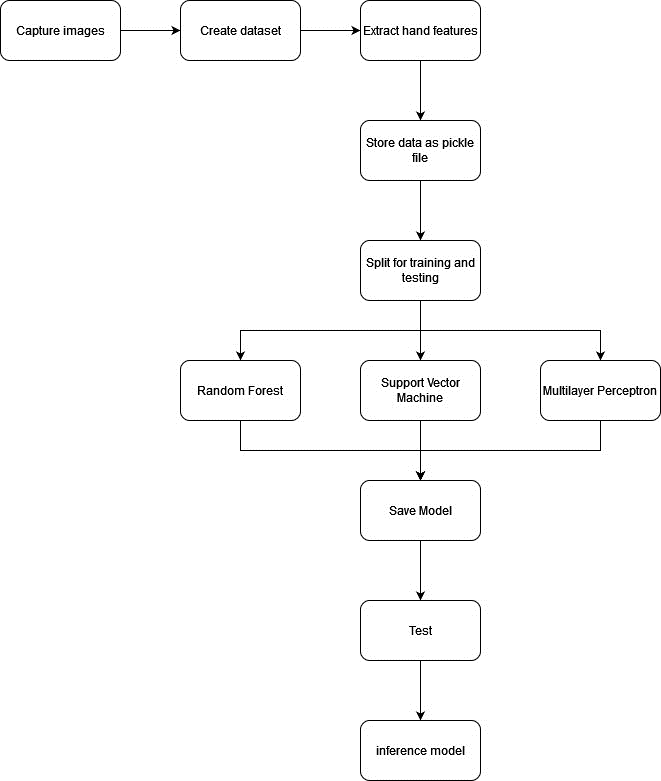
* 1. Mediapipe :

Mediapipe is an open-source framework developed by Google that offers a comprehensive set of pre-built tools and components for building machine learning (ML) and computer vision (CV) applications. It provides a high-level, intuitive interface and a collection of powerful algorithms that simplify the development of complex CV and ML pipelines. One of the key features of Mediapipe is its support for real-time multimedia processing. It offers a variety of pre-trained models and building blocks for tasks such as face detection, hand tracking, pose estimation, object tracking, and more. These models are designed to work seamlessly with real-time video streams, making it easy to integrate Mediapipe into applications that require live visual analysis and tracking. Mediapipe also includes a range of customizable ML and CV components, allowing developers to tailor the algorithms to their specific needs. It provides a graphical pipeline editor, where developers can visually connect and configure different processing blocks, making it easy to prototype and iterate on complex CV and ML pipelines.

* 1. Algorithms Used:

1. Random Forest : It is primarily used for both classification and regression tasks. Random Forest combines multiple decision trees to create a robust and accurate model. The basic idea behind Random Forest is to create an ensemble of decision trees, where each tree is trained on a random subset of the training data. The randomness is introduced in two ways: random sampling of training data and random feature selection. These randomization techniques make the Random Forest model more robust, less prone to overfitting, and capable of handling high-dimensional datasets.
2. Support Vector Machine (SVM): The key principle behind SVM is to find an optimal hyperplane that separates different classes in a dataset or predicts the continuous target variable with the largest margin. SVM works by transforming the input data into a higher-dimensional feature space using a kernel function. This transformation allows SVM to find a linear decision boundary that is non-linear in the original feature space. The selection of an appropriate kernel function is crucial, as it determines the shape of the decision boundary. The goal of SVM is to maximize the margin, which is the distance between the decision boundary and the nearest data points from each class. By maximizing the margin, SVM aims to achieve better generalization and robustness to unseen data. The data points that lie closest to the decision boundary, called support vectors, play a significant role in defining the hyperplane.
3. Multilayer Perceptron: Multilayer Perceptron (MLP) is a type of artificial neural network that is widely used for various machine learning tasks, including classification and regression. It is composed of multiple layers of interconnected nodes called neurons, organized in a feedforward manner. Each neuron applies an activation function to its inputs and passes the result to the next layer. MLP is known for its ability to learn complex patterns and relationships in data. However, it may suffer from overfitting if the model is too large or the training data is insufficient. Regularization techniques such as dropout and weight decay can be applied to mitigate overfitting. Hence, making it ideal for such a project.

# PROPOSED METHODOLOGY:



*Figure 1. Architecture Diagram*



*Figure 2. Flow Chart*

## Capture images -

* 1. **Setting up directory and variables:** The code sets up a directory path (**DATA\_DIR**) where the captured images will be stored. If the directory doesn't exist, it is created using the **os.makedirs()** function. Additionally, two variables are initialized: **number\_of\_classes** and **dataset\_size**. These variables specify the number of classes (in this case, 26) and the desired size of the dataset (in this case, 500 images per class).
  2. **Initializing the video capture:** The code creates a video capture object using **cv2.VideoCapture(0)**. The argument **0** indicates that the default camera will be used for capturing video.
  3. **Looping over each class:** The code enters a loop that iterates over each class (from 0 to **number\_of\_classes** - 1). This loop is responsible for capturing images for each class.
  4. **Creating class directories:** For each class, the code checks if a directory corresponding to that class exists within the **DATA\_DIR**. If the directory doesn't exist, it is created using **os.makedirs()**.
  5. **Prompting the user:** The code displays a frame from the video capture, and overlays a text instruction on the frame using **cv2.putText()**. The instruction asks the user to press the "s" key to capture images.
  6. **Capturing images:** Once the "s" key is pressed, the code enters a nested loop. This loop is responsible for capturing the desired number of images for the current class.
  7. **Capturing frames:** Inside the nested loop, the code continuously reads frames from the video capture using **cap.read()**. The frames are displayed using **cv2.imshow()**.
  8. **Saving images:** When a frame is being displayed, the code waits for a short period (25 milliseconds) using **cv2.waitKey(25)**. If the user does not press any key, the loop continues, and the next frame is read and displayed. If the user presses any key, the code saves the current frame as an image using **cv2.imwrite()**. The image is saved in the corresponding class directory with a filename based on the counter value.
  9. **Updating the counter:** After saving an image, the counter is incremented.
  10. **Releasing resources:** Once the desired number of images for the current class is captured, the nested loop ends. The code releases the video capture resources using **cap.release()** and closes all OpenCV windows using **cv2.destroyAllWindows()**.

1. **Create Dataset -**
   1. **Initializes the hand tracking model:** It initializes the hand tracking model from the **mp.solutions.hands** module with specific configurations such as static image mode and minimum detection confidence.
   2. **Defines the data directory:** It sets the **DATA\_DIR** variable to the directory path where the hand images are stored.
   3. **Initializes data and label lists**: It initializes two empty lists, **data** and **labels**, which will store the extracted hand data and corresponding labels.
   4. **Iterates over the data directory:** The code loops over each directory within the **DATA\_DIR**, representing different classes or categories of hand images.
   5. **Iterates over the image files:** Within each class directory, the code iterates over each image file.
   6. **Extracts hand landmark data:** For each image, the code reads the image, converts it to RGB format, and processes it using the hand tracking model. The hand landmarks (coordinates) are extracted for each detected hand.
   7. **Processes hand landmark data:** The code processes the hand landmarks by normalizing the coordinates and appending them to the **data\_aux** list. Each hand's normalized coordinates are calculated by subtracting the minimum x and y values from all the hand landmarks.
   8. **Appends data and labels:** The **data\_aux** list, containing the normalized hand landmark data, is appended to the **data** list. The corresponding label (class) is appended to the **labels** list.
   9. **Serializes data:** Finally, the code opens a file named 'data.pickle' in write binary mode, serializes the **data** and **labels** lists into a dictionary, and writes it to the file using **pickle.dump()**. The file is then closed.
2. **Train Model -**
   1. **Loads the data:** The code loads the serialized data from the 'data.pickle' file using pickle.load() and assigns it to the data\_dict variable.
   2. **Extracts data and labels:** It retrieves the 'data' and 'labels' arrays from the data\_dict dictionary.
   3. **Splits the data into training and testing sets:** The code uses train\_test\_split() to split the data and labels into training and testing sets. The testing set size is set to 30% of the total data, and stratified sampling is used to maintain the same class distribution in both sets.
   4. **Initializes and trains the model:** The code initializes an MLPClassifier model with a hidden layer configuration of (100, 100, 100) neurons. This configuration means the perceptron will have 3 hidden layers and each layer will have 100 neurons. The model is then trained on the training data using model.fit().
   5. **Makes predictions:** The code uses the trained model to predict the labels for the testing data using model.predict() and assigns the predicted labels to y\_predict.
   6. **Evaluates the model:** The code calculates the accuracy score by comparing the predicted labels (y\_predict) with the true labels (y\_test) using accuracy\_score().
   7. **Prints the accuracy:** The code prints the accuracy score, indicating the percentage of samples that were classified correctly.
   8. **Saves the model:** The code opens a file named 'model.p' in write binary mode, serializes the trained model into a dictionary, and writes it to the file using pickle.dump(). The file is then closed.
3. **Inference Model -**
   1. **Sets up video capture:** The code initializes the video capture from the default camera using cv2.VideoCapture() and assigns it to the cap variable.
   2. **Initializes hand tracking:** The code sets up the hand tracking module from mediapipe by initializing the mp\_hands.Hands() class. It configures the module for static image mode and specifies the minimum detection confidence for hand detection.
   3. **Defines label mapping:** The code creates a dictionary labels\_dict that maps the numerical labels of the model's output to corresponding alphabetical characters.
   4. **Main loop for real-time gesture recognition:** The code enters a while loop to continuously read frames from the camera feed.
   5. **Preprocessing the frame:** For each frame, the code converts it from BGR to RGB format using cv2.cvtColor().
   6. **Hand detection and landmark extraction:** The code processes the frame using the hand tracking module's hands.process() function. It retrieves the hand landmarks if hands are detected. It then visualizes the hand landmarks on the frame using mp\_drawing.draw\_landmarks().
   7. **Feature extraction:** For each detected hand, the code extracts the x and y coordinates of the landmarks. It computes the minimum values of x and y coordinates (min(x\_) and min(y\_)) to normalize the landmark data. The normalized coordinates are appended to the data\_aux list.
   8. **Drawing a bounding box and predicting gesture:** The code calculates the coordinates for drawing a bounding box around the detected hand by multiplying the normalized minimum and maximum x and y values with the frame dimensions. It checks if the length of the data\_aux list is greater than 42 (indicating a valid hand gesture with 21 landmarks) to avoid erroneous predictions. Then, it uses the trained model to predict the gesture label for the current hand gesture by passing the extracted features (np.asarray(data\_aux)) to the model's predict() function.
   9. **Displaying the predicted gesture:** The code visualizes the predicted gesture by drawing a rectangle around the hand using cv2.rectangle() and displaying the predicted character using cv2.putText().
   10. **Displaying the frame:** The code shows the annotated frame with the bounding box and predicted gesture using cv2.imshow(). It waits for a key press using cv2.waitKey().
   11. **Exiting the loop and releasing resources:** The code continues the loop until the 'Esc' key is pressed. After exiting the loop, it releases the video capture using cap.release() and closes all the windows using cv2.destroyAllWindows().

# EXPERIMENTAL ANALYSIS WITH CODE:

### CAPTURE IMAGES:

import os

import cv2

DATA\_DIR = './data'

if not os.path.exists(DATA\_DIR):

    os.makedirs(DATA\_DIR)

number\_of\_classes = 26

dataset\_size = 500

cap = cv2.VideoCapture(0)

for j in range(number\_of\_classes):

    if not os.path.exists(os.path.join(DATA\_DIR, str(j))):

        os.makedirs(os.path.join(DATA\_DIR, str(j)))

    print('Collecting data for class {}'.format(j))

    done = False

    while True:

        ret, frame = cap.read()

        cv2.putText(frame, 'Press "s" to capture images', (100, 50), cv2.FONT\_HERSHEY\_DUPLEX, 0.8, (0, 255, 0), 2,

                    cv2.LINE\_AA)

        cv2.imshow('frame', frame)

        if cv2.waitKey(25) == ord('s'):

            break

    counter = 0

    while counter < dataset\_size:

        ret, frame = cap.read()

        cv2.imshow('frame', frame)

        cv2.waitKey(25)

        cv2.imwrite(os.path.join(DATA\_DIR, str(j), '{}.jpg'.format(counter)), frame)

        counter += 1

cap.release()

cv2.destroyAllWindows()

### CREATE DATASET :

import os

import pickle

import mediapipe as mp

import cv2

import matplotlib.pyplot as plt

mp\_hands = mp.solutions.hands

mp\_drawing = mp.solutions.drawing\_utils

mp\_drawing\_styles = mp.solutions.drawing\_styles

hands = mp\_hands.Hands(static\_image\_mode=True, min\_detection\_confidence=0.3)

DATA\_DIR = './data'

data = []

labels = []

for dir\_ in os.listdir(DATA\_DIR):

    for img\_path in os.listdir(os.path.join(DATA\_DIR, dir\_)):

        data\_aux = []

        x\_ = []

        y\_ = []

        img = cv2.imread(os.path.join(DATA\_DIR, dir\_, img\_path))

        img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

        results = hands.process(img\_rgb)

        if results.multi\_hand\_landmarks:

            for hand\_landmarks in results.multi\_hand\_landmarks:

                for i in range(len(hand\_landmarks.landmark)):

                    x = hand\_landmarks.landmark[i].x

                    y = hand\_landmarks.landmark[i].y

                    x\_.append(x)

                    y\_.append(y)

                for i in range(len(hand\_landmarks.landmark)):

                    x = hand\_landmarks.landmark[i].x

                    y = hand\_landmarks.landmark[i].y

                    data\_aux.append(x - min(x\_))

                    data\_aux.append(y - min(y\_))

            data.append(data\_aux)

            labels.append(dir\_)

f = open('data.pickle', 'wb')

pickle.dump({'data': data, 'labels': labels}, f)

f.close()

### TRAIN MODELS :

***#Random Forest:***

import pickle

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import numpy as np

data\_dict = pickle.load(open('./data.pickle', 'rb'))

data = np.asarray(data\_dict['data'])

labels = np.asarray(data\_dict['labels'])

x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.3, shuffle=True, stratify=labels)

model = RandomForestClassifier()

model.fit(x\_train, y\_train)

y\_predict = model.predict(x\_test)

score = accuracy\_score(y\_predict, y\_test)

print('{}% of samples were classified correctly!'.format(score \* 100))

f = open('model.p', 'wb')

pickle.dump({'model': model}, f)

f.close()

***#SVM:***

import pickle

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import numpy as np

data\_dict = pickle.load(open('./data.pickle', 'rb'))

data = np.asarray(data\_dict['data'])

labels = np.asarray(data\_dict['labels'])

x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.3, shuffle=True, stratify=labels)

model = SVC(probability=True)  *# Use SVC for Support Vector Machines*

model.fit(x\_train, y\_train)

y\_predict = model.predict(x\_test)

score = accuracy\_score(y\_predict, y\_test)

print('{}% of samples were classified correctly!'.format(score \* 100))

f = open('model.p', 'wb')

pickle.dump({'model': model}, f)

f.close()

***#Multiayer Perceptron:***

import pickle

from sklearn.neural\_network import MLPClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import numpy as np

data\_dict = pickle.load(open('./data.pickle', 'rb'))

data = np.asarray(data\_dict['data'])

labels = np.asarray(data\_dict['labels'])

x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.3, shuffle=True, stratify=labels)

model = MLPClassifier(hidden\_layer\_sizes=(100, 100), max\_iter=1000)

model.fit(x\_train, y\_train)

y\_predict = model.predict(x\_test)

score = accuracy\_score(y\_predict, y\_test)

print('{}% of samples were classified correctly!'.format(score \* 100))

f = open('model.p', 'wb')

pickle.dump({'model': model}, f)

f.close()

### INFERENCE MODEL :

import pickle

import cv2

import mediapipe as mp

import numpy as np

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

model\_dict = pickle.load(open('./modelMLP.p', 'rb')) *#change model to “modelRF.p” or “modelSVM.p”*

model = model\_dict['model']

cap = cv2.VideoCapture(0)

mp\_hands = mp.solutions.hands

mp\_drawing = mp.solutions.drawing\_utils

mp\_drawing\_styles = mp.solutions.drawing\_styles

hands = mp\_hands.Hands(static\_image\_mode=True, min\_detection\_confidence=0.3)

labels\_dict = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W', 23: 'X', 24: 'Y', 25: 'Z'}

while True:

    data\_aux = []

    x\_ = []

    y\_ = []

    ret, frame = cap.read()

    H, W, \_ = frame.shape

    frame\_rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

    results = hands.process(frame\_rgb)

    if results.multi\_hand\_landmarks:

        for hand\_landmarks in results.multi\_hand\_landmarks:

            mp\_drawing.draw\_landmarks(

                frame,

                hand\_landmarks,

                mp\_hands.HAND\_CONNECTIONS,  *# hand connections*

                mp\_drawing\_styles.get\_default\_hand\_landmarks\_style(),

                mp\_drawing\_styles.get\_default\_hand\_connections\_style())

        for hand\_landmarks in results.multi\_hand\_landmarks:

            for i in range(len(hand\_landmarks.landmark)):

                x = hand\_landmarks.landmark[i].x

                y = hand\_landmarks.landmark[i].y

                x\_.append(x)

                y\_.append(y)

            for i in range(len(hand\_landmarks.landmark)):

                x = hand\_landmarks.landmark[i].x

                y = hand\_landmarks.landmark[i].y

                data\_aux.append(x - min(x\_))

                data\_aux.append(y - min(y\_))

        x1 = int(min(x\_) \* W) - 10

        y1 = int(min(y\_) \* H) - 10

        x2 = int(max(x\_) \* W) - 10

        y2 = int(max(y\_) \* H) - 10

        if len(data\_aux) > 42:

            continue

        prediction = model.predict([np.asarray(data\_aux)])

        predicted\_proba = model.predict\_proba([np.asarray(data\_aux)])[0]

        match\_percentage = round(max(predicted\_proba) \* 100, 2)

        if match\_percentage < 95.0:

            predicted\_character = "Unknown"

            display\_text = predicted\_character

        else:

            predicted\_character = labels\_dict[int(prediction[0])]

            display\_text = **f**"{predicted\_character} ({match\_percentage}%)"

        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)

        cv2.putText(frame, display\_text, (x1, y1 - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 1.3,

                    (0, 0, 0), 3, cv2.LINE\_AA)

    cv2.imshow('frame', frame)

    if cv2.waitKey(1) & **0x**FF == ord('q'):

        break

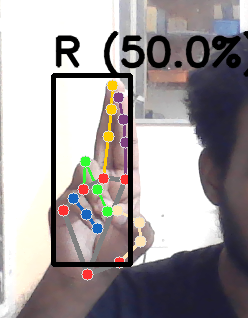
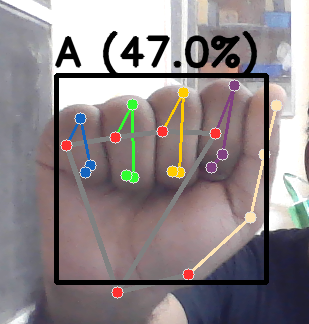
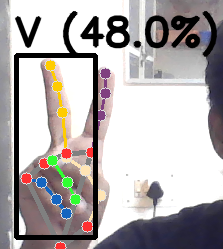
cap.release()

cv2.destroyAllWindows()

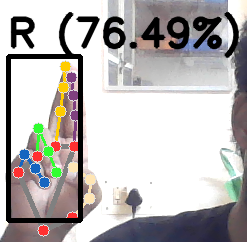
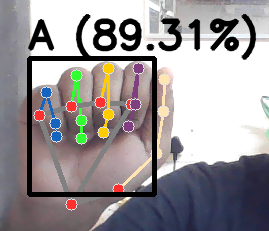
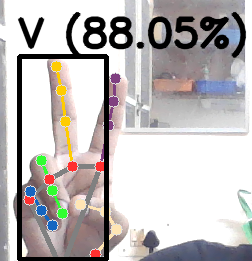
# COMPARATIVE ANALYSIS :

****

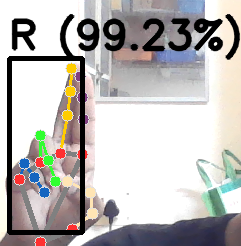
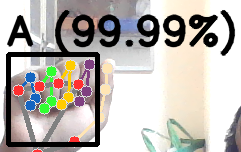
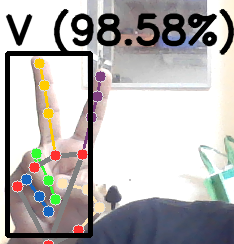
*Figure 3. Comparison of models by training accuracies*

*Figure 4. Testing accuracy of Random Forest*

*Figure 5. Testing accuracy of SVM*

*  *

*Figure 6. Testing accuracy of MLP*

# CONCLUSION WITH FUTURE ENHANCEMENT :

In conclusion, we have successfully implemented a real-time sign language recognition system using OpenCV, Mediapipe, and machine learning algorithms such as Random Forest, SVM, and Multilayer Perceptron. By leveraging the power of computer vision and machine learning techniques, we were able to detect and track hand signs in real-time video input from a camera.

In addition to the successful implementation of the real-time sign language recognition system, it is noteworthy to mention the training and testing accuracies achieved by the machine learning algorithms. During the training phase, the Random Forest algorithm exhibited the highest accuracy with a perfect score of 100%. The SVM algorithm followed closely with an accuracy of 99.84%, while the MLP algorithm achieved an accuracy of 99.89%.

However, when evaluating the performance of these algorithms on unseen data during the testing phase, interesting trends emerged. The MLP algorithm consistently outperformed the others, demonstrating the highest testing accuracy ranging from 96% to an impressive 99.99%. This indicates the MLP algorithm's ability to generalize well to new and unseen sign language gestures, showcasing its robustness and effectiveness.

On the other hand, the SVM algorithm exhibited a moderate testing accuracy, ranging from 65% to 80%. While still providing reasonable accuracy, it suggests some limitations in handling variations and complexities present in real-world sign language gestures.

Random Forest algorithm showcased the lowest testing accuracy, hovering around 50%. This discrepancy between its high training accuracy and relatively lower testing accuracy implies potential overfitting, where the model may have memorized the training data but struggled to generalize effectively to new samples.

These insights shed light on the comparative performance of the machine learning algorithms in real-time sign language recognition. The MLP algorithm demonstrated superior generalization capabilities, making it a favourable choice for real-world applications. Future work could focus on further improving the performance of the SVM and Random Forest algorithms to enhance their testing accuracy and bridge the gap between training and testing performance and also implementing a possible way to formulate entire sentences based on gestures.