Real Time Sign Language Detection Model

Nikunj Raghav
Department of Computer Science
University of Bath
Bath, BA2 7AY
nr601@bath.ac.uk

Dissertation 2022 - Research Project - Department of Computer Science, University of Bath, UK

Abstract

"In this dissertation we use neural networks and data extraction to research the problem of real time sign language detection. This dissertation reviews more than 30 papers, with the majority of them being published in the last 3 years in order to come up with the optimal solution for our problem. Based on the background research done, the author came up with the solution of using Mediapipe Holistic to extract keypoints from videos taken from webcam in combination with multi-layer LSTM neural network model to perform real time sign Language detection using OpenCV. The neural network model used in this paper comprises of 3 LSTM layers with 3 Dense layers and was compiled using ADAGRAD optimizer and ReLU activation function. The model was tested in real time for all signs in three different conditions, and model was able to accurately predict the sign 102 times, out of 105 times it was tested and each sign was detected within 15 seconds, giving us an accuracy of 97%."

Acknowledgment

I would like to thank the following people, without whom I would not have been able to complete this research. The Computer Science team at University of Bath, especially to my patient and supportive supervisor supervisor Dr Cherry Zhao, whose insight and knowledge into the subject matter steered me through this research. And special thanks to Marina De Vos, who helped me with all of the queries during my academic year. And my biggest thanks to my family for all the support you have shown me through this one year of my masters when I was 5000 miles away from home. Thanks for all your support, without which I would not have made it through my masters degree!

Contents

1 Introduction **5**

2 Literature Review **5**

2.1 Main Objectives of literature review 5

2.2 Techniques used for data collection? 6

2.2.1 Using Online Dataset 6

2.2.2 Using Sensor-Based System 6

2.2.3 Creating Own Dataset 7

2.3 Techniques implemented for enhancing the data? 7

2.3.1 Pre-Processing 7

2.3.2 Segmentation 7

2.4 Techniques used for feature extraction? 7

2.5 Algorithms implemented for sign language detection? 8

2.6 Conclusion from background research 9

3 Requirements **9**

4 Design **10**

4.1 Approach for solving this tasks 10

4.2 Creating Dataset 10

4.3 Pre-processing 12

4.4 Creating Neural Network Model 12

4.5 Sign language detection in real time 12

5 Implementation **12**

5.1 Libraries and packages used 13

5.2 Detecting landmarks 13

5.3 Setting up file path 15

5.4 Keypoints Extraction 15

5.5 Collecting keypoints 15

5.6 Pre Processing Data 16

5.7 Neural Network Model 17

5.7.1 Building Neural Network Model 17

5.7.2 Compiling the model 18

5.7.3 Training the Model 18

5.7.4 Visualizing TensorBoard 19

5.7.5 Visualizing Neural Network 20

5.8 Performing Evaluation 21

5.8.1 Multilabel Confusion Matrix 21

5.8.2 Accuracy Score 22
5.9 Performing detection in real-time 22

6 Testing **24**

7 Evaluation **25**

8 Conclusion and Future Works **25**

1 Introduction

Communication is defined as the act of sharing or exchanging information, ideas, or feelings[1]. To establish communication between two people, both must know and understand the same language. The modalities of communication for hearing impaired (deaf) and speechless (mute) people are different. According to WHO around 20% percent of the world population suffers from hearing loss which accounts for over 1.5 billion people, of which 430 million population is deaf[2]. Deaf and mute persons communicate with each other via hand gestures and sign language[3], which makes it difficult for them to communicate with other people as the majority of the population does not understand sign language, and because of this, they have to rely on human translators. And learning sign language is not an easy task as it requires a lot of study and training, and those without any hearing and speech problems find it hard to motivate themselves for learning sign language.

Sign language recognition is a challenge that has been addressed for many years but there still has not been a comprehensive solution. The study of sign language brings together a wide range of subjects and expertise and the majority of work designed to address this problem has relied on one of two approaches: **Contact-based systems**, such as data gloves, in which the signer wears a hardware glove which is a soft sensor-embedded glove that uses a deep learning algorithm to analyze the data from those sensors to allow real-time sign recognition[4]. Despite its high accuracy of over 90%, the glove-based approach appears to be a little unpleasant for practical application[5]. And the other is **Vision-based systems**, which rely only on cameras to analyze users' hand and body characteristics to successfully predict sign language[6]. Static and dynamic recognition are two types of vision-based methods, statics is concerned with the detection of static gestures (two-dimensional images), whereas dynamic is concerned with the capturing of gestures in real-time, this entails the employment of a camera to record movement. This method is significantly less expensive, and the rise of deep learning makes it even more effective. Vision-based hand gesture recognition is an area of active current research in computer vision and machine learning. Being a natural way of human interaction, it is an area that many researchers are working on, to make human-computer interaction (HCI) easier and natural, without the need for any extra devices[7].

This paper aims to use a vision-based system to tackle the problem of performing real-time sign language detection (on British Sign Language) with the help of deep neural network model. The author created the dataset using MediaPipe Holistic, which was then used to train a deep neural network model (comprising of 3 LSTM layers and 3 dense layer). The neural network was trained multiple times using different optimizer like ADAM, ADAGRAD, ADADELTA, SGD and different activation functions like ReLU, leaky ReLU, Sigmoid, Tanh and then their performance was evaluated against each other using a confusion matrix, finally ADAGRAD was used with ReLU as it gave the best performance for the model in comparison with any other combination that was tested. After successfully training the model every component was combined together using OpenCV in order to detect sign language in real time.

2 Literature Review

A search for published articles was undertaken on IEEE Explore and Google Scholar databases for this review. Papers were found using the keywords "sign language recognition" and "sign language detection". For this study, only papers published after 2016 were reviewed (except for one from 2014), and to ensure focus on recent breakthroughs, most of the reviewed papers were published within the last three years.
This review includes various academic articles, so to successfully analyze each paper in accordance with the target problem, it's better to break the problem into different research objectives.

2.1 Main Objectives of literature review

Continual developments are being done to bridge the communication gap between special needs individuals and normal people, this literature review will focus on visual-based sign language recognition systems and various recent breakthroughs that have been made in that field. The main objective of this study is to find the an effective real-time sign language recognition technique that is currently available using a standard video camera, meaning it must be precise, efficient, and easily

accessible. This literature review helped in the reaching to the most optimal design for solving the problem of performing sign language detection in real time.

1. Firstly the model will need data for training in order to perform sign language detection, so what are the techniques used for data collection?
2. After data is collected, what techniques were implemented for enhancing the data before performing feature extraction?
3. Techniques used for extracting features, so that features can be used by the classifier?
4. Algorithms incorporated for classification of obtained features in order to make a successful prediction?

These research objective will help in assembling and analyzing data from various publications to compare them against each other, these research objectives will ultimately help in finding the optimal technique for sign language recognition.

2.2 Techniques used for data collection?

Data-gathering step must be consistent, with all images having the same file type and dimensions. Obtaining a properly labeled image dataset is very important, as it determines how well the training phase of the model will go, and the training phase in turn determines the performance of the model. The models' chances of making accurate predictions will be increased drastically if it is trained with a good dataset[1].

Various methods of data collection were used in different papers according to the need of their projects, majority of them can be classified into three different categories, using an online dataset, using sensor-based systems, or creating their own dataset using cameras.

2.2.1 Using Online Dataset

Using data from an online dataset to train a model is a popular approach as the data is generally available in ready-to-use state and most of the time online dataset contains high quality and a vast amount of images. There are different types of datasets available online, for example, American Sign Language (ASL) dataset from Kaggle contains more than $12,500$ images[8], and a dataset containing both static $(54,000)$ and dynamic $(49,613)$ hand gesture images was used in another paper[9]. Some papers used video datasets like LSA64 containing over 500 videos[10] , and the Kinetics dataset which is a human action video dataset containing 400 human action classes, with over 400 clips in each class[11]. Using an online dataset can be beneficial as it provides a large number of high-quality images without the requirement for any additional hardware, it saves time and allows one to focus on other parts of the research. Sometimes selected dataset does not provide enough data (like online dataset from Massey University consists of only 900 images[12]), so to obtain additional data, data augmentation [2] might be used[14].

2.2.2 Using Sensor-Based System

Another method of collecting data is using sensor-based approach where some sort of wearable is used to classify sign language gestures[4] , this method is highly accurate in predicting the sign language but it is less practical and more expensive in comparison with other methods as it requires the use of additional hardware. Some papers also demonstrated the use of Kinect sensors provided by Microsoft[15][16] , "much like a webcam these sensors are peripheral devices that produces a depth map (which provides distance between sensor and pixel, for every pixel seen by sensors) in addition to an RGB image"[17] , thus offering additional specific information that can help in classification. Kinect sensors provide highly accurate data and make segmentation easier by differentiating foreground and background features[18] but they are expensive in comparison to normal 2D cameras and are sensitive to external infrared sources[19]. Other papers demonstrated the use of similar devices to Kinect like Leap Motion[20], which are in same price range but are more accurate.

[1] A good data set contains not only a large number of pictures but also a high density of target objects within those pictures.

[2] Data Augmentation is the process of adding to available data to develop new data, one of the common ways of achieving that is by simply rotating the angle of the image[13].

2.2.3 Creating Own Dataset

Creating own dataset was found to be the most popular approach due to the easy availability of the camera in smartphones and computers[21][22] , as it allows to tailor the dataset to meet requirements of the project[23][24] . But when creating dataset using camera, pre-processing is frequently required to ensure consistency with the classifier[3].

2.3 Techniques implemented for enhancing the data?

Depending on the source of data, or how data was collected there could be some sort of noise in images, and to remove that noise several methods are applied after data collection. These methods also enhance the data which helps in improving the performance of the classifier.

2.3.1 Pre-Processing

Pre-Processing is the process of altering data once it has been collected which generally involves the removal of potentially inaccurate data and noise. Gaussian Filter[22][25] which reduces the noise but also reduces the quality of the image, and Median Filter[12][26] which filters the noise but does no error propagation, are two popular noise reduction methods. Some papers also incorporated bootstrapping[11] to void the loss of information. Cropping, filtering, brightness and contrast adjustments for consistent input and image enhancement are also part of pre-processing[24][21][23] .

2.3.2 Segmentation

Segmentation can be defined as the technique for enhancing feature. Feature extraction is the phase in which keypoints are extracted for the classifier, but the classifier is only interested in hand gestures (that are executing signs), so it is important to enhance those features which act as the region of interest(ROI) for the classifier before sending the image for feature extraction[27], and segmentation is used for this very purpose[4].

Grayscaling[10][12] is part of segmentation, it converts a colored image to grayscale after which red, green, and blue parts have equal intensity in RGB space, so instead of 3 value for each pixel (which is the case for colored image), now it only needs to be specified with a single intensity value[25][28]. This will prevent the classifier from considering color in its calculations which makes it easier to process the grayscale image in comparison to the colored image. Grayscale also aids in recognizing the backdrops and focal points, which is crucial[29].

Thresholding[22][25][12] is another method that was commonly observed in segmentation and was applied after grayscaling, so the first step is to convert RGB images to grayscale and then convert them to binary images, after which the image must be cropped to remove any undesired parts, and the selected area is enhanced. Backdrop and focal points are selected based on the threshold that was set by the researcher.[30].

2.4 Techniques used for feature extraction?

Feature extraction limits the resources required for computation without losing significant or relevant information, this allows the creation of a sign recognition database[31]. **Canny edge detection** seems to be one of the popular methods,[32][25] as it is useful for determining the Region of Interest(ROI) by mapping out the hand's edges[33], then this ROI is sent to the classifier. Edge detection methods help in dealing with noisy environments but takes too much computation time[34].
Whereas another paper implemented the 7Hu Moments technique[32] to extract 7 moments from images, which are then used to create a database of gestures.

Whereas Some papers incorporated two descriptors, HOG (histogram of oriented gradients) and SIFT(Scale Invariant and Feature Transform) for feature extraction[22], these methods count instances of gradient orientation in specific areas of a picture. SURF (Speeded Up Robust Feature) which is very similar to SIFT but is faster in terms of calculation, was used in other papers[28], it is resistant to

[3]The need for pre-processing occurs due to poor image quality (depends on the camera used) or different image dimensions (use of different cameras) or both.

[4]The researcher must first determine which parts of the image are significant in order to successfully perform segmentation, which will then remove background and other things that would cause the classifier to fail.

scaling, rotation, and variance which makes it useful for picture classification since it allows classifier to manage varied sign rotations. In the case of video input, one paper used an Openpose library on summarized video sequences for extracting keypoints[35] which allowed them to extract a feature vector for all video sequences in their dataset.

From the reviewed papers, one of the most successful keypoint extraction approaches was the use of **MediaPipe Holistic**[23][36], because it accurately extracts keypoints from human pose and hands and also allows the use of custom build frameworks(Using TensorFlow-lie)[37], which makes data-flow easily manageable and configurable[38].

2.5 Algorithms implemented for sign language detection?

This study reviewed various techniques used for classification to perform sign language recognition, and many papers demonstrated the use of **Artificial Neural Network (ANN)**[33][35], as it provides a quick learning curve and fault-tolerant network. Papers using ANN showed great results in terms of accuracy, as can be seen in[33], which applied ANN after extracting features from video images to achieve a recognition rate of 95%. Whereas paper[35] used 9 neural networks (Different neurons in each layer) in different combinations to compare their performance. Although the model using ANN showed a slow convergence speed.

Out of all of the methods, the use of **Convolutional Neural Network (CNN)** was the most popular and unlike ANN, CNN has an additional convolutional layer that helps in identifying which node the data represents and uses this result to train the model. Using CNN appears to be better than ANN as it provides very accurate image classification even without segmentation or pre-processing as seen in some papers[12][39] and achieved an accuracy of over 98%. It could be further classified into different categories based on the type of CNN that was being used for SLR i.e. whether it was 2D[12][39][9] or 3D[40][15] CNN [5]. Better results can be achieved by increasing the number of layers in the neural network but that also increases the computational time, which makes getting the layers right very difficult, as seen in paper[9] which achieved an accuracy of over 99%. 3D CNN models appear to be quite effective when performing sign language recognition on a video dataset[40] and achieved an accuracy of close to 97%. Some papers also implemented **I3D** model[10][24], (which is a "3D Convolutional neural network model for video classification trained on the Kinetics dataset"[42][11]), and achieved an accuracy of up to 98%. One of the major issues with CNN methods is that they increase computational time and require strong hardware in order to train the model.

Some papers also used **Recurrent Neural Network**[23][21], which remembers every information, and thus offers series prediction. One paper used different types of RNNs and compared their performance[23]. While another paper used RNN along with CNN[21], where CNN was used to recognize features and RNN was used to train on features.

Another popular classification technique that was found in this study was **K-Nearest Neighbor (KNN)**[20][22], which is a supervised learning algorithm that trains the model using a labeled dataset, it is used to predict which group a data point will belong to based on the group of the nearest data point. It is easy to implement and provides good accuracy but does not work well with a large dataset. When compared to Probabilistic Probabilistic Neural Network(PNN), PNN achieved better accuracy than KNN[32].

Some papers demonstarted use of Support Vector Machine(SVM)[28] which is a popular method for solving margin optimization problems. This method is memory efficient making it good for classification, but it does not perform well when image data have noise. While another paper[14] used the Hidden Markov Model (HMM) which is a very efficient learning algorithm but has a high computation time and needs a large amount of data for training.

One of the most effective classification methods found in this review was the **Long Short-Term Memory (LSTM)** model[16][24] which is a type of RNN that can maintain information for a longer period of time due to the presence of additional memory cell units, making it capable of learning long term dependencies [6], the LSTM uses feedback connection which allows it to classify the entire

[5] 2D convolution is generally used for images where the filter moves in two directions with 3-dimensional input and output, whereas 3D convolution is generally used for videos and has a filter that moves in three directions and having 4-dimensional input and output[41]

[6] LSTM is designed to avoid long dependency issues, and usually performs better than time recursive neural networks and hidden Markov models

sequence of data [7] . One paper combine LSTM with another Neural network[8], which uses the CNN for processing and the LSTM model for sign language recognition thus achieving a recognition rate of over 95%.

2.6 Conclusion from background research

From the detailed study (presented above) which is derived from a review of over 30 papers, it can be analyzed that SLR comprises 4 major steps which are Data Gathering, Pre-Processing, Feature Extraction, and Classification. We found that all methods used in different papers have their advantages and disadvantages, some techniques like edge detection help in dealing with noise but take too much computational time, and CNN appears to have the highest accuracy rate with the use of multiple network layers but it has strong hardware requirements to train the model without compromising computational time, and thus it produces better results in combination with LSTM layers. Based on the review it can be concluded that creating a high performance and efficient SLR system will require:

- Creation of dataset using camera (as it appears to be the most popular approach).
- Image pre-processing to remove noise and enhance ROI.
- Use of feature extraction method like MediaPipe Holistic(as it allows the use of custom build frameworks).
- Use of an effective classification technique like the CNN and LSTM Model.
- Putting it all together using OpenCV to successfully make predictions in real-time using a web camera.

3 Requirements

The inspiration behind this project is to find a solution that can help in reducing the communication barrier that exists between those who can speak and those who are deaf and mute, and for this reason we decided on developing a system that can help anyone in understanding sign language by detecting them in real-time. The target audience of this project would be anyone who doesn't know sign language but would like to communicate with those who can only speak using sign language. As potential users, we can say that the most important things we would want from a sign language detection model will be accuracy when performing detection, and fast computation time so that predictions can be made in real-time without having to wait too long for the results, plus a clean user interface so that we can easily understand which sign is getting detected by the model.

Based on the background research done we know that to make accurate predictions it is important to use some sort of deep neural network model rather than just using a machine learning model because because neural networks structure algorithms in such a manner that they can learn and make intelligent decisions on their own while machine learning algorithms bases judgments solely on what it has learnt[43]. The deep neural networks model also needs to be trained using feature-rich data, because even the most efficient algorithms can become useless in the absence of high-quality training data. Another major requirement of the project is to make predictions by analyzing the entire action rather than just one still image, in other words, to make an accurate prediction it is important that the neural network model uses multiple frames of data from the action being demonstrated in front of the webcam rather than just using a single frame at a particular point in time. To perform detection in real time a deep neural network model must make predictions in a matter of seconds otherwise the model will be too slow to have any real-time application. The user interface needs to be simple so that it can clearly indicate which sign language is being predicted by the model based on the actions that are being performed in front of the webcam. We will be doing everything from scratch which makes it impractical to make a high-quality dataset and then train the model for a large number of signs in the short amount of time that we have, that's why the aim will be to train the model for five to ten different types of signs.

[7] LSTM model overcomes the limitations of RNN and is capable of classifying a large amount of data efficiently.

So, in summary, we can conclude that for successful implementation of this project it is important to design a model which makes highly accurate predictions by training on a multi-frame dataset and then takes a very small amount of computation time to make predictions.

4 Design

This project was designed for British Sign language (BSL) and to test the model's effectiveness over different types of signs in BSL the model will be trained using seven different types of actions(signs). These actions were selected on random, to incorporate different types of hand and body movement.

Based on the Requirements the main goals for the project can be divided into four different steps:

1. Creating a high quality dataset on which model will be trained
2. Pre Processing the data so that it can be used for training
3. Successfully training the deep neural network model in order to make accurate predictions
4. Combining all of this together with openCV to make prediction in real time using a webcam

4.1 Approach for solving this tasks

The very first step in the process of sign language detection was to collect data using which the neural network model will be trained, and one of the major goal for this project was to create a dataset rather than using one that is available online, the background research that was done before undertaking this project showed that creating one's own dataset is a very popular approach as it gives the developer complete accountability and ownership for the project from start to end, it also helps in developing deep understanding of the problem and data, moreover it provides flexibility to the developer as they can tailor the dataset according to their requirements[44]. Mediapipe Holistic will be used to meet the requirement of creating dataset and collecting keypoints from the videos recorded using webcam, as it was one of the most popular approach for feature extraction (as shown in the Techniques used for feature extraction?). After collecting the keypoints, data will be pre-process which will involve creating arrays of keypoints, these arrays will be used for training the deep neural network model. Moreover once the model is trained, mediapipe holistic can be used to collect the features from the action being demonstrated in front of the camera, these collected feature will then used by the trained neural network to make accurate predictions.

Most state of the art neural network models (studied in background research) use bunch of CNN layers followed by LSTM layers and achieved high accuracy with such models, so for this model mediapipe holistic will be combined with the deep neural network model comprising of CNN layers followed by LSTM layers.

Finally all of these steps will be combined together using OpenCV so that the model can actually predict an action over number of frames in real time using a webcam. The user-interface is going to be very simple and minimal as the only purpose of the user interface is to display the detected sign language.

4.2 Creating Dataset

One of the major challenge with creating your own dataset is getting useful information out of that collected data that can be further used for training the neural network model. As mentioned earlier this challenge was overcome with the help of mediapipe holistic, which was used for data collection and feature extraction. Real-time perception of human pose, face landmarks, and hand tracking are all possible with MediaPipe. It provides quick and precise answers, but integrating them all in real-time into a semantically coherent end-to-end solution is a particularly challenging issue that requires simultaneous inference of numerous, dependent neural networks[45]. Mediapipe holistic allows to extract keypoints from your hand, body, and face. The pose, face, and hand landmark models are used by MediaPipe Holistic to create a total of 543 landmarks (33 pose landmarks, 468 face landmarks, and 21 hand landmarks per hand). this project stores these landmarks in numpy arrays which are later combined in one big numpy array in order to train the deep neural network. Pose landmarks consist of 4 coordinates whereas left hand, right hand and face landmarks each

of them consist of 3 coordinates each, which means every time in a single frame of a video 1662 keypoints will be extracted (33*4 + 468*3 + 21*3 + 21*3)[45].

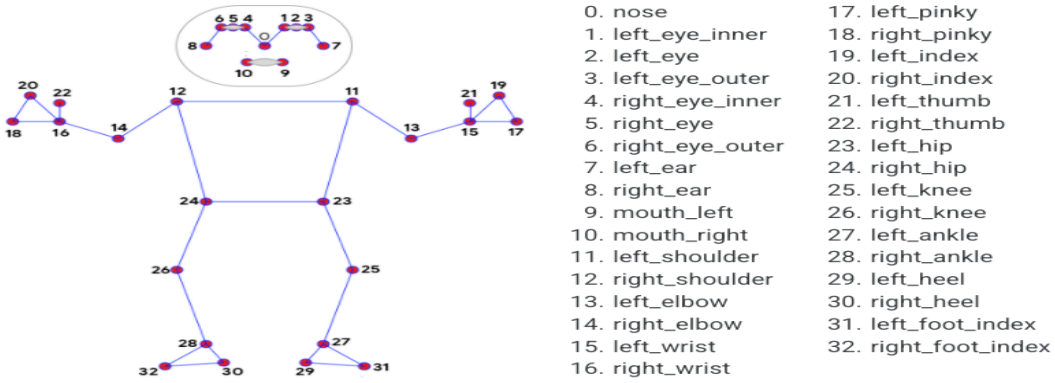

Figure 1: Pose Landmarks [Mediapipe 2020]
[46]

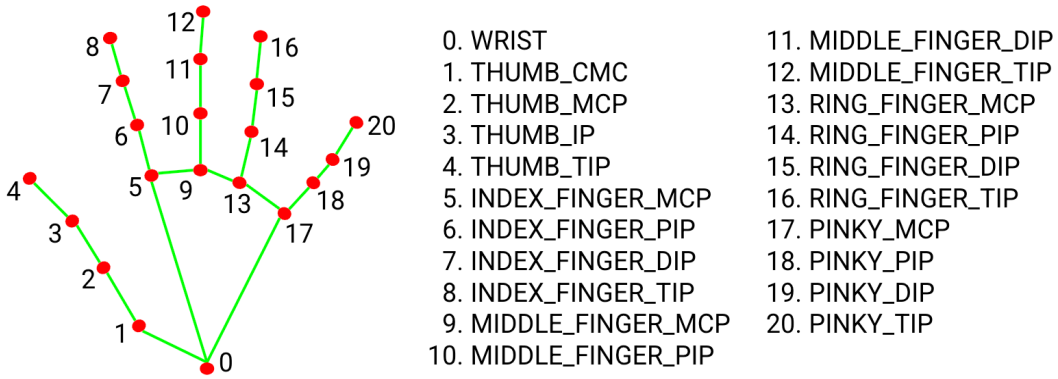

Figure 2: Hand Landmarks [Mediapipe 2020]
[47]

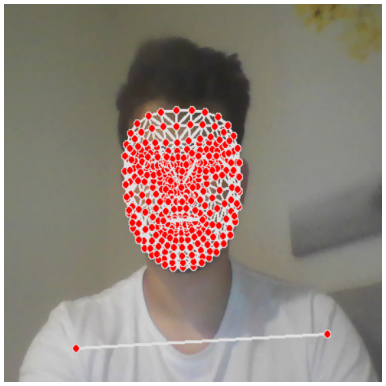

Figure 3: Face Mesh with 468 Landmarks

This project collected data for seven different types of signs (hello ,tea ,good ,thanks, like, sweet, I love you), over 30 frames which means 30 different sets of preceding keypoints were recorded to be able to classify that sign. To provide high quality and quantity of training data for the model, 80 sequences (videos) were recorded for every type of sign. Although it would have been much better to train the model for more signs (like for 10 different signs) but since we created our own dataset, even

creating 560 (7 different types of action * 80 sequence for each action) high quality videos took a large amount of time.

4.3 Pre-processing

The input data used in this project is a series of 30 arrays as there are 30 frames for each sequence, and each one of those arrays and contains 1662 values which represents keypoints for a single frame, in other words each arrays represent landmarks value for a single frame. Every action was recorded for 80 sequences, so every single action contains 80*30 arrays. So in total there were 7*80*30*1662 keypoints collected (7 different types of actions, 80 sequence for each action, 30 frame over each sequence, 1662 keypoints in a single frame). Once the data was collected it was time to pre-process it, so that it can be used for training the deep neural network. The collected data was then organised into one big array, and the shape for that array was [560, 30, 1662] which represents 560 sequences in total .

4.4 Creating Neural Network Model

TensorFlow is a very popular machine learning library and was used to create the deep neural network for this project. TensorFlow is a flexible open-source library that allows to develop faster and more reproducible deep learning model in a very simple way. TensorFlow was originally written by Google brain in 2015[48]. As mentioned earlier most state of the art models for sign language detection tends to use neural network with bunch of CNN layers followed by LSTM layers, or just CNN layers. The first deep neural network model that was developed for this project consist of CNN layers followed by LSTM layers but the results achieved were no where near where it can be actually useful for performing action detection in real time, we tried changing the number of layers, using different type of activation functions and optimizers for the neural network, and even tried increasing the number of sequences in dataset for limited signs but the results were still very poor to be of any use for detecting sign language in real-time, and even worse results were observed when only CNN layers were used. After that we tried combining mediapipe holistic with a neural network model that only consist of LSTM layers, because from background research it was concluded that LSTM models were one of the most effective classification methods as they provide high accuracy results even when trained over short amount of dataset. The results that were achieved with only LSTM layer model were really impressive. Apart from being hyper accurate the LSTM model was a much denser neural network, the combination of CNN and LSTM model had over 30 million parameters in neural network whereas LSTM model with same number of layers had only close to 600,000 parameters thus making LSTM model much easier and faster to train. Moreover since the LSTM model was less dense and much more simple it made detection really fast in comparison to other models (CNN with LSTM layer model and only CNN layer Model) that were trained with same dataset, hence making it possible to use this model to perform real-time action detection.

4.5 Sign language detection in real time

Once the model was successfully trained using the sign language dataset, we put all of it together using openCV to perform sign language detection in real time using webcam. OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library that was developed to offer a standard infrastructure for computer vision applications and to speed up the incorporation of machine perception in the production of commercial goods[49], the openCV library is cross-platform and free to use under open source-BSD license[50]. As mentioned earlier **user interface** was very simple which just displays the detected sign language, but sometimes model was detecting more than one sign language for a particular action so probability visualization was incorporated to show the sign getting identified more clearly, probability visualization allows to see how probabilities are being calculated in real time for different actions and thus giving a better idea of which sign is being detected by the model.

5 Implementation

We have already talked about the design of the project and how all of it was structured in order to achieve the final goal. For better understanding of the project it is important to talk about how

various components were implemented, and how they were connected together in order to perform object detection. This entire project is implemented using python programming language, python was chosen for this project because it is high level object oriented programming language which offers many benefits for deep learning projects. Python is simple and consistent which makes writing deep learning models much easier, python also has large database of libraries and frameworks (like Keras, TensorFlow, NumPy, and more) which are very useful in building deep learning models, being platform independent it also offers the advantage of running the software solution developed in python over any operating system (like Linux, Windows, Mac, and more), moreover python has a big community which helps in getting help for almost every problem one can encounter while development[51].

For the ease of explanation and making the overall code structure much more clear the whole project implementation will be divided into different steps, and in this section we will be going through each of these steps in details.

5.1 Libraries and packages used

For this project we have installed nine different libraries (see Figure 17). First one is OpenCV which is a computer vision library that allows users to work with their webcam[50]. Second one is matplotlib which allows to visualize images in a much better and easier manner[52]. Third one is mediapipe holistic which allows to extract keypoints from face, hands and body, we will be using openCV together with mediapipe holistic to collect keypoint from our body and then saving those as frames which will represents sequence of events for a particular action(sign). We also used scikit-learn for leveraging training and testing split and also for creating the evaluation matrix[53]. TensorFlow was installed for building and training the deep neural network model[48]. Last three libraries shown in figure 17 were installed to plot the neural network, which helps in understanding how the model is working behind the scenes[54]. After installing all these libraries the necessary packages were imported into the project (see Figure 18).

5.2 Detecting landmarks

The very first part of the project was to access the webcam using OpenCV (see Figure 19).

After successfully accessing the webcam with OpenCV additional layer of mediapipe holistic was implemented with webcam in order to detect landmarks. Two variables were created, first variable to download the mediapipe holistic model, which will be used to make detection and second variable is a mediapipe drawing utility which makes it easier to draw landmarks (see Figure 20)[45].

Once the mediapipe holistic is setup, it was time to create a function to start making detection. There are steps that needs to be followed in order to make detection using mediapipe holistic (see Figure 21) because OpenCV read the image in the BGR channel format but detection done by mediapipe needs to be in RGB format[55].

Pseudo-code for detection_with_mediapipe function:
Convert the grabbed image from BGR to RGB
Set the image to unwritable to save memory
Perform detection and store them in 'results' variable
Set image back to writable
Convert the image from RGB to BGR
Return image and result

Now the result from this detection was applied to the OpenCV loop (that was created earlier to access webcam), although this will not do the rendering but it will store the results in a variable called *results*[45] (see Figure 22) .

The next step was to create a function that will accept the image and model in order to visualise the landmarks in frame. This function will take the detection from mediapipe model and render those on the image with the help of mediapipe drawing utility[45][56] (see Figure 23).

Figure 4: Image feed running with mediapipe holistic

After combining these landmarks with OpenCV feed, user can see the landmarks within the feed. (see Figure 24). But by default mediapipe shows same color for all type of landmarks (face, right hand, left hand and pose) which makes it difficult to differentiate between different types of landmarks.

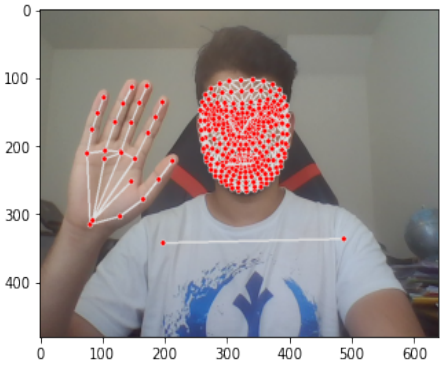

Figure 5: Image with default landmarks

In order to make different landmarks visible within the frame to the user, another function was created which was similar to visualize_landmarks functions, but this time custom formatting (different color and shapes) was used instead of the default formatting (see Figure 25). These new customized landmarks were then combined with the OpenCV image feed (see Figure 26), and now user can finally see different landmarks (with custom formatting) in the frame[45][56] .

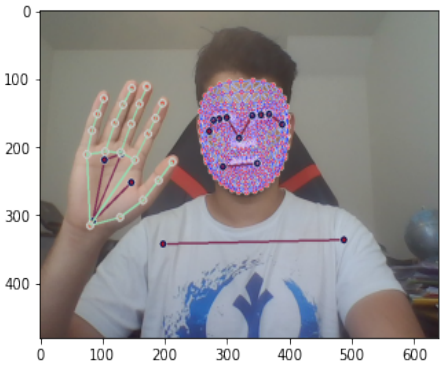

Figure 6: Image with customised landmarks

5.3 Setting up file path

Before extracting keypoints we need to create folders for storing keypoints and define file paths where extracted keypoints from different actions(signs) will be saved. The keypoints were extracted for seven different actions (hello ,tea ,good ,thanks, like, sweet, and I love you) eighty sequences(video) for each sign, and every sequence is going to have thirty frames worth of data. So a file path was set up accordingly, one main folder called 'keypoints_collected' consisting of seven sub folder (which represents actions) and each of those seven folders consisting of eighty sub folders (which represent sequences), and every sub folder (sequence) will contain thirty NumPy arrays(which represent frames) (see Figure 27).

5.4 Keypoints Extraction

After the mediapipe holistic was successfully combined with the webcam using OpenCV, it was time to extract keypoints value in a format which can be later used by the model. As it was earlier discussed in the design section Creating Dataset the MediaPipe Holistic creates 33 pose landmarks, 468 face landmarks, and 21 hand landmarks per hand. And each pose landmark consist of 4 coordinates whereas every other landmark consist of 3 coordinates, and same can be verified from the data stored in variable *results*(see Figure 28). So to store these landmarks in useful manner they were concatenated into a numpy array (see Figure 29). After this the array was flatten so that all the landmarks can be available inside one big array, without flattening their will just be multiple sets of landmarks in the array which will not be feasible for training the deep neural network model (see Figure 30).

And if at a particular point of time a given frame does not have value for any of the four landmarks then a numpy zero array of same size and dimensions will be saved for that landmark (see Figure 31). The shape and size for different types of landmark will vary based on number of landmarks created by Mediapipe Holistic, and how many coordinates that particular landmark had. So in total four loops were run (because there are 4 different types of landmarks), one through each type of landmark in order to extract their values in numpy array and then all four numpy arrays were concatenated together (see Figure 32).

Pseudo-code for extracting keypoints:
If values exist for a particular landmark type
  Loop through every landmark
  Extract the value for each coordinate of landmarks in a numpy array
  Flatten the NumPy Arrays
Else
  Create NumPy zeroes array of same size and shape
Return a concatenated NumPy array (with pose keypoints, face keypoints, left hand keypoints and right hand keypoints)

Now this function will return a numpy array containing all of the keypoints detected by Mediapipe Holistic, and numpy zeroes array in place of those landmarks that were not detected by Mediapipe Holistic, either way the final numpy array will always have the same number of keypoints (i.e. face + pose+ left hand + right hand keypoints). So the final numpy array will have 1662 keypoints (468*3 + 33*4 + 21*3 + 21*3) (see Figure 33).

5.5 Collecting keypoints

It was finally time to create dataset by collecting keypoints for all 7 signs. For collecting data using the webcam the same loop was reused that was created for Detecting landmarks, but instead of looping consistently through webcam this time the loop was run for eighty sequences(video) in each sign and there were 30 frames within each sequence.

So basically the loop will run seven times (for each sign), then eighty times within each sign (80 sequence), then 30 times within each sequence (30 frames) (see Figure 34). When this loops was run the keypoint collection was happening so fast which made it very hard to switch positions between different collections, therefore 3 seconds break was added between each new sequence collection in order to allow the user to reposition themselves. Having breaks allows to reset and reposition yourself

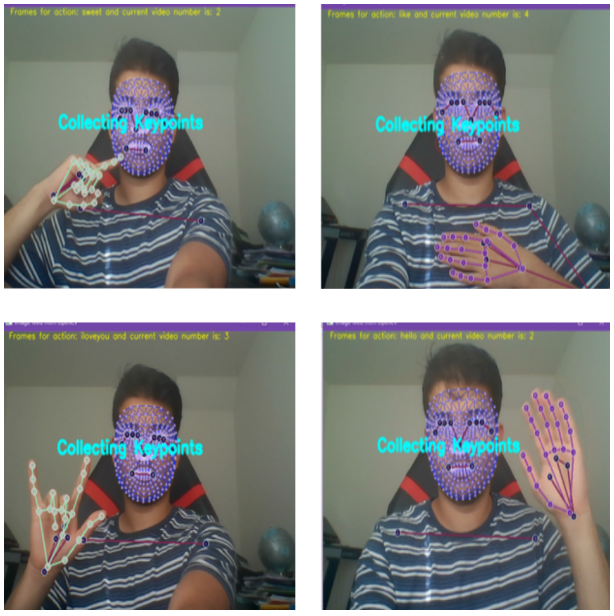

Figure 7: Collecting keypoints for different signs

to collect frames from start to finish, it is important to reposition between each action in order to collect wide variety of data for the model. Moreover between each collection the information was displayed about which action class is being collected right now in order to make it easier for the user to perform the right action, plus a message was also displayed right before collecting keypoints in order to alert the user (see Figure 35). After collecting data for all the signs, Keypoints Extraction was performed and all the data was stored in their respective folders(see Figure 36).

Pseudo-code for Collecting Keypoints:

```
for every action(7 Actions)
    for every sequence in that action(80 Sequences)
        for every frame in that sequence(30 Frames)
            Read image feed
            Make Detection
            Draw landmarks
            If at frame zero
                Display starting collecting keypoints
                Display which action class is being collected
                Take a 3sec break
            Else
                Display which action class is being collected
            Extract Keypoints
            Collect Keypoints in respective folders
            Break through loop if 'escape' is pressed
Close image feed
```

5.6 Pre Processing Data

After all the keypoints are collected it was time to pre process the data so that it can be used for training the deep neural network model.

First a dictionary was created to represent each sign (see Figure 37) called 'dictionary_of_signs'. All of the keypoints were already collected and using the dictionary_of_signs they will be structured into one big array, structuring the data is important in order to use it for training purposes. To structure all the data in one big array, an array of size (560,30,1662) was required, i.e. an array containing 560

array(because eighty sequences for seven actions) with 30 frames in each one of those arrays and 1662 keypoints in every frame (see Figure 38).

Pseudo-code for Pre-Processing Data:
Initializing two blank arrays, 'features_data' and 'labels_data'
'features_data' will represent X_data
'labels_data' will represent y_data
for every sign(seven signs in total)
  for every sequence in that sign (80 Sequences)
    Initialize a blank array 'frames_in_sequence' ('frames_in_sequence' will represent all of the frames for a particular sequence)
    for every frame in the sequence (30 Frames)
      Loading every frame using NumPy (by passing full path to different NumPy arrays)
      Append all the arrays to 'frames_in_sequence' array
    Append 'frames_in_sequence' array to 'features_data' array
    Append 'dictionary_of_signs' to 'labels_data' array

After the data is structured together it needs to pre-processed before it can be used for training the deep neural network model. The X_data was generated by converting 'feature_data' array into a NumPy array. And for y_data "to_categorical" function from Keras utilities[57] was used to hot-encode label_data into a binary flag array which represents value for different signs (see Figure 39).

After this the data was split into training and testing partitions. "train_test_split" function from scikit-learn was used to make training and testing partition so that training can be done on one partition and then test it on the other partition of data[58]. To do the split the result of "train_test_split" function were unpacked into training and testing variables (see Figure 40). The test_size=0.05 inside the function indicates the percentage of the data that should be held over for testing/validation, in this case value is set to 0.5 meaning test partition is going to be 5% of our data. This split function randomly divides the dataset rows so that we get disjoint training and testing sub-datasets[58]. 'X' represents independent features(excluding target variable) whereas 'y' represents dependent variables, called (target). To train the model we will use X_train as the features and y_train as the ground truth. Similarly, when testing the model we will use X_test as the features and y_test to validate the predicted labels[59][60]. For dataset used in this project the shape of X_train, X_test, y_train, y_test are show in figure 40.

5.7 Neural Network Model

Before reaching at this step we have already pre processed our data and divided it into training and testing partitions. As mentioned earlier in design phase (Creating Neural Network Model), TensorFlow was used to create the Deep Neural Network as it allows to build simple and fast deep learning models.

All the dependencies we will need for building neural network were imported at the starting of code (see Figure 41). The first dependencies was Sequential, we will be using multiple layers in the neural network and TensorFlow allows to do so by initializing the model as Sequential. Then LSTM and Dense layers were imported for the neural network model (because we decided on using LSTM layer model as explained in Creating Neural Network Model). In order to track the training process and have a better understanding of the neural network, TensorBoard was imported. And finally plot_model was imported which allows to visualize neural network by plotting it's structure.

TensorBoard is web application which is part of the TensorFlow Package, it allows to track and monitor the training process by logging the training process inside a v2 file. A separate Logfile folder was created to store the TensorBoard callback file (see Figure 42), this file will allow the visualization of TensorBoard during and after training[61].

5.7.1 Building Neural Network Model

Like mentioned before, in order to use multiple layers in neural network it needs to initialized as Sequential. The number of nodes (or hidden neurons) or layers one should select in their neural model is not fixed nor is their any hard-and-fast rule, so the best way is to do a trial-and-error method which will yield the best results for your particular scenario[62]. So we created a Sequential model

and kept on adding additional layers one at a time until we got good results from our neural network architecture, the results obtained with just one LSTM layer were very subpar; with two layers, the results were marginally better; nevertheless, the best accuracy was obtained with the addition of three LSTM layers. LSTM layers in TensorFlow takes 3 keywords argument and one positional arguments[63] .

Making sure the input layer has the appropriate number of input features is the first thing to get right[64], so in first LSTM layer(input layer) 'input_shape' was set to (30,1662) because the X_train array used for training the model has a shape of (532,30,1662) (see Figure 40)) meaning it has 532 array with 30 frames in each one of those array and 1662 keypoints in each one of those frames.

With LSTM layer in TensorFlow it is important to return sequences because the next layer is going to use those values[62]. Different types of activation functions were tried with model like relu, sigmoid, tanh, Leaky relu[65] and the best results were obtained with relu.

The LSTM layers were followed by Dense layers, these layers are fully connected layers meaning each neuron in the dense layer receives input from all neurons of its previous layer[66]. Similar to LSTM layer there is no hard-and-fast rule for how many dense layer should you use, so we kept on adding additional layers one at a time until we got good results, and best results were achieved with three dense layers. In dense layer first argument allows you to choose the number of neurons or nodes in the layer, and the second argument allows you to provide the activation function[67].

So overall architecture of the neural network model is as followed (see Figure 43):

- First LSTM layer with 64 neurons and 'return_sequences=True' with 'relu' activation function and as mentioned before 'input_shape' was set to (30,1662).
- Second LSTM layer with 128 neurons and 'return_sequences=True' with 'relu' activation function.
- Third LSTM layer with 64 neurons and 'return_sequences=False'(we don't have to return sequences after third LSTM layer because next layer is a Dense layer) with 'relu' activation function.
- Dense layer with 64 neurons and 'relu' activation function.
- Dense layer with 32 neurons and 'relu' activation function.
- Final Dense layer with with 'signs.shape[0]' and 'softmax' activation function

The final layer with 'softmax' activation function will return an array with values having probabilities between 0 and 1 (sum of all the values will be 1), and the highest value within that array will represent the action that is being detected. The size of this array will be seven because of 'signs.shape[0]' that is being passed in this layer (which represents seven signs).

5.7.2 Compiling the model

To compile our model we tested different optimizers like 'ADAGRAD', 'ADDELTA', 'ADAM', 'SGD'[68] and received best performance with 'ADAGRAD' (see Testing). Model was compiled using 'model.compile'[69] with 'ADAGRAD' activation function, and 'categorical_crossentropy' loss function (this is the loss function that needs to be used with multi-class classification model[70]). We also specified metrics as 'categorical_accuracy'[71] to track accuracy while training (see Figure 44).

5.7.3 Training the Model

Model was trained with 'model.fit' using 'X_train' and 'y_train' dataset that were created during Pre Processing Data step, use 'X_train' was used as the features and 'y_train' as the ground truth. And callback was set to 'tensorboard_callback' to store the training log file (see Figure 45). The model was trained for 400 epochs and it achieved 1.0000 (100%) 'categorical_accuracy' while training (see Figure 46), although sometimes model saturated at 'categorical_accuracy' of 1.0000 after training only for 200 epochs, so in those scenarios training was interrupted before reaching 400 epochs.

5.7.4 Visualizing TensorBoard

TensorBoard is a platform for providing the measurements and visuals needed during the machine learning workflow. It makes it possible to visualise the model graph, follow experiment metrics like loss and accuracy, and much more.

Log file for TensorBoard was already saved to monitor the training, and it can be opened in the browser through command line. Open command prompt and navigate to the project folder, then go to Logfile folder, then go inside train folder, here you will find the v2 file and in order to open this file in your browser just run "tensorboard –logdir=.", this will give the url for TensorBoard web-app which can now be accessed through browser (see Figure 47).

The TensorBoard web app can be used to visualize the neural network architecture (see Figure **??**). The web app can also be used to track the epoch categorical accuracy and epoch loss during training as shown in figures 8 9 :

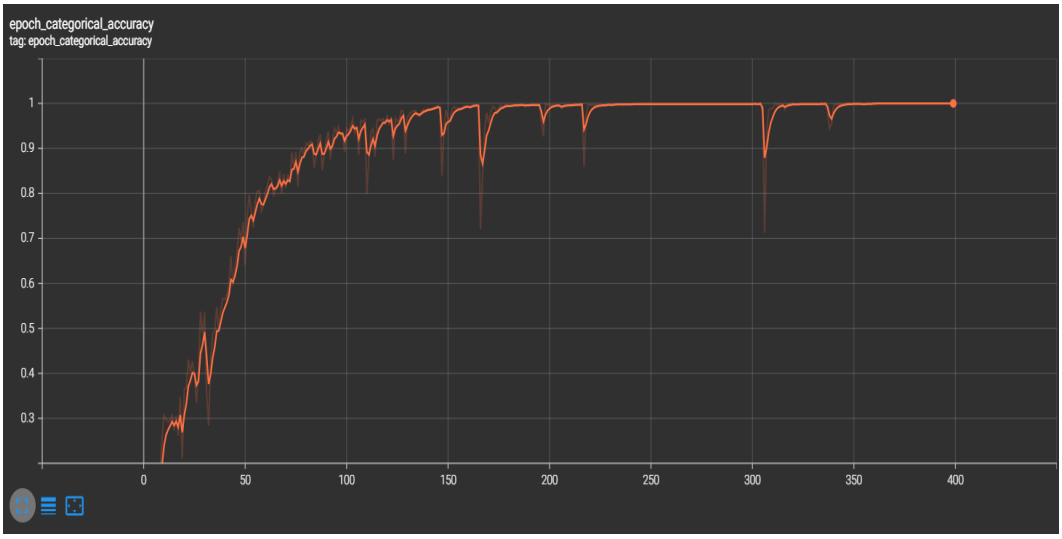

Figure 8: Graph for Epoch Categorical Accuracy (X-Axis showing accuracy from 0 to 1 and Y-Axis showing Number of epochs from 0 to 400)

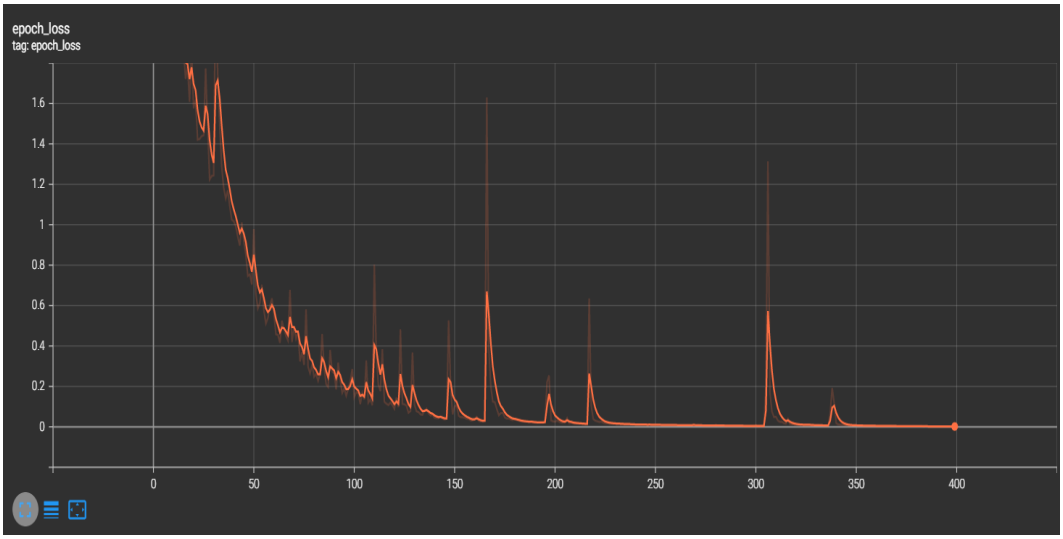

Figure 9: Graph for Epoch Loss (X-Axis showing loss and Y-Axis showing Number of epochs from 0 to 400)

5.7.5 Visualizing Neural Network

Neural network models can be visualised and understood more clearly with the help of the Keras Python deep learning package[72]. Keras provides a way to summarize a model by using 'model.summary ()' , summary is textual and provides the number of parameters (weights) in each layer, the order in which the layers appear in the model, the output shape of each layer, and the total number of parameters (weights) in the model. From figure 10 we can see shape, output and number of parameters for each layer, along with total number of parameters in the model.

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

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

 lstm_2 (LSTM)                (None, 64)                49408

 dense (Dense)                (None, 64)                4160

 dense_1 (Dense)              (None, 32)                2080

 dense_2 (Dense)              (None, 7)                 231

=================================================================
Total params: 596,807
Trainable params: 596,807
Non-trainable params: 0
_________________________________________________________________
```

Figure 10: Model Summary for Neural Network obtained using 'model.summary()' function

Additionally, Keras offers a function to visualise the neural network plot, which is helpful in understanding overall neural network model architecture. The plot_model() function in Keras creates a plot for neural network model[73] as shown in figure 11.

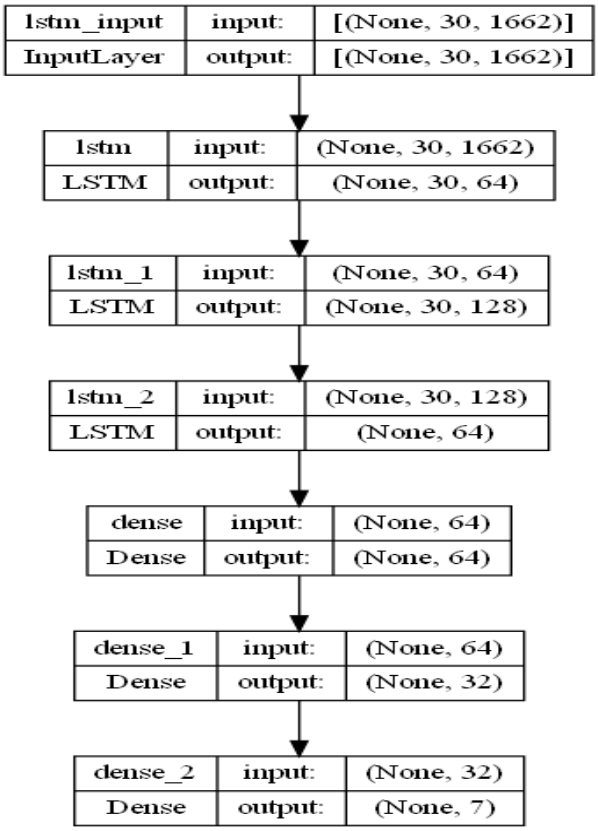

Figure 11: Neural Network Model Plot created using 'plot_model()'

5.8 Performing Evaluation

After training the model it is generally a good idea to perform some evaluation in order to determine how the model is performing, for carrying out this evaluation the 'multilabel_confusion_matrix' along with the 'ccuracy_score' packages were imported from scikit-learn library. The 'multilabel_confusion_matrix' provides the confusion matrix for each one of the labels(in this case signs) which in turns allows to evaluate the results of detection[74] i.e., which detection are detected as true positive, true negative, false positive, or false negative [8]. Before starting evaluation we need to make some predictions, these predictions were stored in 'y_hat' [9], later these 'y_hat' values were converted to a list along with 'y_train' values (the y_train values were generated during Pre Processing Data).

5.8.1 Multilabel Confusion Matrix

The 'y_hat' and 'y_train_list' were passed into 'multilabel_confusion_matrix' function which returns a confusion matrix in the shape of 2x2, containing the values in order of labels (in this case seven labels/signs hence seven matrices). Confusion matrix is used for evaluating the performance of a classification model where output can be two or more classes, it is extremely useful for measuring Recall, Precision, Specificity and Accuracy[76].

The confusion matrix is a table with 4 different combinations of predicted and actual values organised in the order as shown in the figure 12

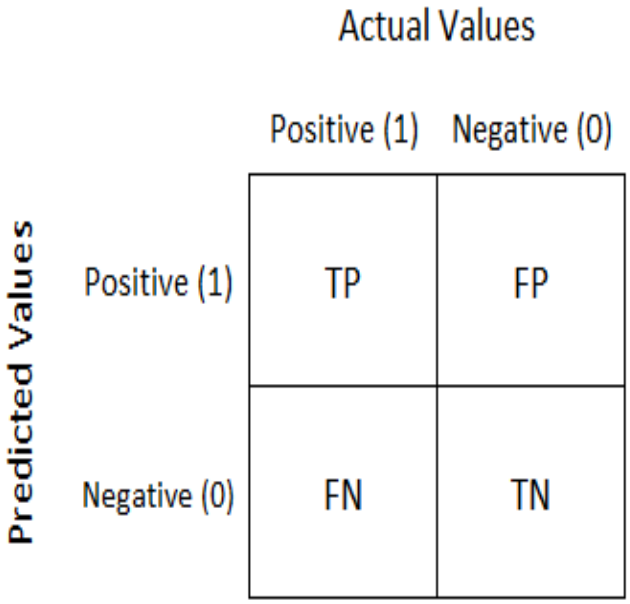

Figure 12: Confusion Matrix
[77]

So the best result from evaluation should have most of the values either in the top left corner which represents true positive or in bottom right corner which represents true negative, and this is the case with confusion matrix obtained for the trained model as shown in figure 13.

[8] An outcome where the model properly predicted the positive class is referred to as a true positive. An outcome where the model properly predicted the negative class is referred to as a true negative. Similar to a false negative, a false positive is a result when the model forecasts the positive class inaccurately. A false negative is a result where the model forecasts the negative class inaccurately[75]

[9] The y-hat values are the estimated or anticipated values in a regression or other predictive model

```
In [313]: #getting results from model.predict
          prediction_result = model.predict(X_test)

          1/1 [==============================] - 0s 365ms/step

In [314]: #calculating y_hat with model.predicta
          y_hat = model.predict(X_train)

          17/17 [==============================] - 1s 39ms/step

In [315]: #converting y_train and y_hat values to list
          y_train_list = np.argmax(y_train, axis=1).tolist()
          y_hat = np.argmax(y_hat, axis=1).tolist()

In [316]: #using 'multilabel_confusion_matrix' function from sikit-learn to get the confusion matrix
          #confusion matrix give us analysi of prediction i.e. wether the prediction was true poisitve, true negative,
          #false positive, false negative
          multilabel_confusion_matrix(y_train_list, y_hat)

Out[316]: array([[[455,   0],
                  [  0,  77]],

                 [[454,   0],
                  [  0,  78]],

                 [[456,   0],
                  [  0,  76]],

                 [[458,   0],
                  [  0,  74]],

                 [[454,   0],
                  [  0,  78]],

                 [[455,   0],
                  [  0,  77]],

                 [[460,   0],
                  [  0,  72]]], dtype=int64)

In [317]: #Calculating accuracy_score
          accuracy_score(y_train_list, y_hat)

Out[317]: 1.0
```

Figure 13: Screenshot of code snippet showing data passed to multilabel_confusion_matrix function and confusion matrix obtained from it

5.8.2 Accuracy Score

Then 'accuracy_score' function was also used with 'y_hat' and 'y_true_list' values, which calculates the accuracy for a set of predicted labels against the true labels . This function returns values between 0 and 1, where 1 represents 100% accuracy[78], and as can be seen from figure 13 the trained model achieved 100% accuracy with this function.

5.9 Performing detection in real-time

After training the model and evaluating it's performance it was time to test the model using webcam to perform detection. We reuse the OpenCV loop that was created earlier for accessing webcam and initialized two additional empty lists, first one called 'stack_frames' which will collect keypoints from latest 30 frames in order to make predictions and second one called 'predicted_values' to concatenate history of predicted signs in order to display latest prediction. A 'threshold_value' was also defined so that only results above this threshold value gets rendered (see Figure 48).

To start making prediction we first need to extract keypoints from the sign that is being demonstrated in front of the camera and for that we use previously created 'keypoints_extraction' function, and keypoints from last 30 frames recorded in front of camera are stored inside 'stack_frames' list (see Figure 49). Then 'model.predict' was applied to 'stack_frames' and the results were stored in a 'predicted_values' (see Figure 50). After which the results were displayed at the bottom of the feed using OpenCV (see Figure 52).

Since model is making prediction based on the complete action, it takes the model some time to analyze the entire action and it makes multiple prediction before getting to the final prediction because of which model was displaying multiple signs before making the right prediction. To resolve this Probability visualization was setup which shows probabilities being calculated in real time, and after model successfully predicts an action the probability visualization becomes stable and stays on that sign(see Figure 51) .

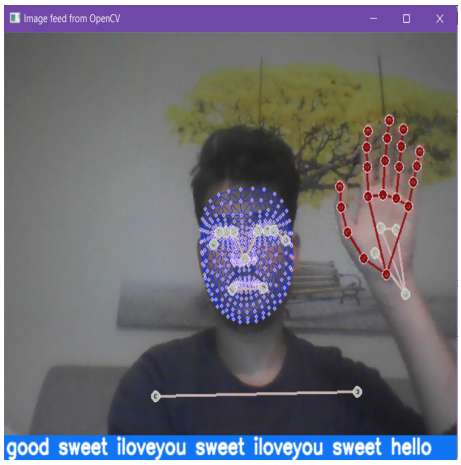

Figure 14: Model predicted the right sign 'hello' but made bunch of other predictions before getting to the right one

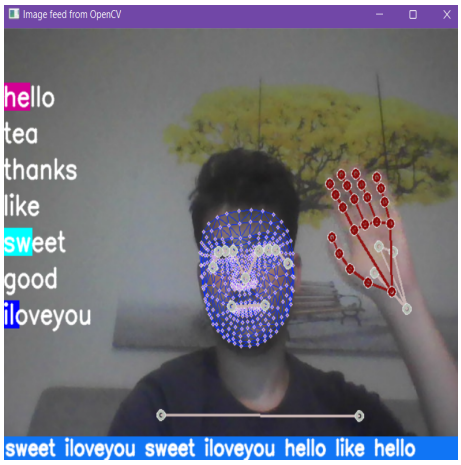

Figure 15: Probability visualization showing probabilities being calculated in real time for sign 'hello'

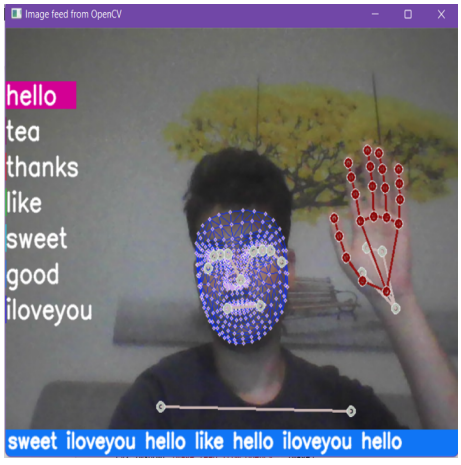

Figure 16: Probability visualization staying on sign 'hello' after successfully predicting the action within couple of seconds

Pseudo-code for displaying results:
Initialize 'stack_frames' and 'predicted_values' list
Setup threshold_value to 0.85
Perform detection with Mediapipe on frames collected from webcam
Extract keypoints from frames and store them in 'stack_frames' list
If length of 'stack_frames' is equal to 30
  Pass the keypoints from 'stack_frames' to model.predict function to make prediction
  If the results obtained from model exceed threshold_value
    If length of 'predicted_values' is greater than zero
      If the current sign does not equal to last sign in 'predicted_values' list(Done to prevent displaying same sign again)
        Append the current sign to the 'predicted_values' list
    Else (meaning if 'predicted_values' list is empty, then we can just append to 'predicted_values')
      Append the predicted sign to the 'predicted_values' list
  If length of 'predicted_values' list exceeds 7
    Get the last 7 values from 'predicted_values' list (at a time we will only display 7 signs so we need to make sure we are displaying 7 latest signs)
  Call probability visualization function
Render a rectangle using OpenCV at the bottom of the feed
Display the text from 'predicted_values' list over the rectangle using OpenCV

6 Testing

Finally it's time to talk about the results achieved. The deep neural network model was trained with different optimizer like ADAGRAD ADADELTA, SGD, ADAM (see Figures: 54,55,56,57) multiple times for same epochs and their performance was evaluated using Accuracy Score, and their best results are listed below in the table 1:

Optimizer	Accuracy Score (0-1)
ADAGRDA	1
ADADELTA	0.911
SGD	0.916
ADAM	0.146

Table 1: Table showing best accuracy scores for different optimizers

From these results we can conclude that this particular neural network model when used in combination with Mediapipe Holistic performs best with ADAGRAD optimizer (out of all of the optimizers tested here). This neural network model achieved an accuracy_score of 100% all 5 times when tested against training dataset.

Since it's a sign language detection model, the real test comes when the model is tested in real time using a webcam. The model was tested in real time for all seven sign in three different conditions (for five times in each condition), so in total the model was tested for 105 actions (7*3*5), for testing purposes the maximum time allowed to predict the sign was set to 15 seconds and within this time limit model made 102 accurate predictions giving us an accuracy of 97%.

Moreover, the time taken to predict the action was recorded each time the model was tested (five times in each condition) and the average time taken for each action is shown in the table 2, and the results were recorded based on how much time it took on average for the model to accurately detect the sign. Given the fact that model was trained with a dataset that was created only by a single user (which generally leads to homogeneous data) the model performed really well. The model showed good results in all different backgrounds for limited signs it was trained on, showing that it's performance does not depend on the surrounding conditions.

Time Taken on average in each condition to correctly predict the action (in seconds)			
Signs Performed	First Indoor Environment	Second Indoor Environment	Outdoor environment
Hello	5	5	7
Tea	7	7	12
Thanks	5	5	8
Like	5	6	8
Sweet	6	7	10
Good	6	5	8
I Love You	5	5	7

Table 2: Table showing time taken by the model on average (average of 5 times) to correctly predict different signs in different environment conditions

7 Evaluation

The main Requirements of the project were, first to create a feature rich data set using mediapipe holistic, then train a deep neural network using that dataset, finally combine all of it together with OpenCV to perform real time sign language detection using webcam and achieve highly accurate detection results, and all of these requirements were achieved by the end of the project. Creating the dataset was very challenging as the author had to perform same actions repeatedly in front of webcam for hours but in the end a high quality dataset was created using mediapipe holistic. All of the designs that were discussed were successfully implemented, however, the initial target was to train the model for ten signs but in order to get good results we had to use 560 sequences for seven signs i.e eighty videos worth of data per sign, and when the same number of videos were used to train the model for more than seven signs the results achieved were comparatively very poor, indicating that the model requires large amount of heterogeneous data in order to train for ten signs. So in the end the model was only trained for seven signs as their was not enough time to collect more high quality data. At the end we believe that the designs decisions were correct as the model achieved most of it's targets, with one major exception of being trained for only seven different type of signs, but that was a shortcoming on the part of dataset due to lack of time on hand. The project also meet the requirements from a user's perspective as it was able successfully detect sign languages in real time using a webcam.

8 Conclusion and Future Works

In conclusion we achieved what we set out to do, which was to create a model to perform sign language detection in real-time. Considering the model was only trained with data obtained form one user it performed really well, it successfully predicted sign languages based on complete action rather than using a still image. The model achieved an accuracy of over 97% irrespective of the environment in which it is being tested, this showed that it can perform well in different environment (even in outdoor conditions) .

For future works we would like to incorporate more sign languages, currently the model was only trained for seven signs because of limited time on hand for creating dataset. For some cases model took more than 10 seconds to make accurate prediction which is too long that's why a better performing model is needed. Right now the model was trained using dataset created by one person, to improve performance it will be better to have data from different users as it will help in creating a more heterogeneous dataset. Moreover the data was recorded using standard laptop webcam, for creating even better dataset we can use a high quality camera because having a feature rich dataset will improve the overall performance of model.

References

[1] Joshua Taylor and Ian Parberry. *What is communication?* en. Nov. 2014. URL: https://entrepreneurhandbook.co.uk/what-is-communication/.

[2] World Health Organization. *Deafness and Hearing Loss*. en. 2022. URL: https://www.who.int/health-topics/hearing-loss#tab=tab_2.

[3] sense.org.uk. *Sign Language*. en. URL: https://www.sense.org.uk/get-support/information-and-advice/communication/sign-language/.

[4] Minhyuk Lee and Joonbum Bae. "Deep learning based real-time recognition of dynamic finger gestures using a data glove". In: *IEEE Access* 8 (2020), pp. 219923–219933. URL: https://ieeexplore.ieee.org/document/9264164.

[5] vaishells. *Sign Language Recognition for Computer Vision Enthusiasts*. en. June 2021. URL: https://www.analyticsvidhya.com/blog/2021/06/sign-language-recognition-for-computer-vision-enthusiasts/.

[6] Mohammed Safeel et al. "Sign language recognition techniques-a review". In: *2020 IEEE International Conference for Innovation in Technology (INOCON)*. IEEE. 2020, pp. 1–9. URL: https://ieeexplore.ieee.org/document/9298376.

[7] José Herazo. *Sign language recognition using deep learning*. en. Aug. 2020. URL: https://towardsdatascience.com/sign-language-recognition-using-deep-learning-6549268c60bd.

[8] Wanbo Li, Hang Pu, and Ruijuan Wang. "Sign Language Recognition Based on Computer Vision". In: *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. IEEE. 2021, pp. 919–922. URL: https://ieeexplore.ieee.org/document/9498024.

[9] Rajarshi Bhadra and Subhajit Kar. "Sign Language Detection from Hand Gesture Images using Deep Multi-layered Convolution Neural Network". In: *2021 IEEE Second International Conference on Control, Measurement and Instrumentation (CMI)*. IEEE. 2021, pp. 196–200. URL: https://ieeexplore.ieee.org/document/9362897.

[10] Herman Gunawan, Narada Thiracitta, Ariadi Nugroho, et al. "Sign language recognition using modified convolutional neural network model". In: *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*. IEEE. 2018, pp. 1–5. URL: https://ieeexplore.ieee.org/document/8627014.

[11] Joao Carreira and Andrew Zisserman. "Quo vadis, action recognition? a new model and the kinetics dataset". In: *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 6299–6308. URL: https://ieeexplore.ieee.org/document/8099985.

[12] Murat Taskiran, Mehmet Killioglu, and Nihan Kahraman. "A real-time system for recognition of American sign language by using deep learning". In: *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*. IEEE. 2018, pp. 1–5. URL: https://ieeexplore.ieee.org/document/8441304.

[13] Jacob Solawetz. *Why and How to Implement Random Rotate Data Augmentation*. en. June 2020. URL: https://blog.roboflow.com/why-and-how-to-implement-random-rotate-data-augmentation/.

[14] Dan Guo et al. "Sign language recognition based on adaptive hmms with data augmentation". In: *2016 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2016, pp. 2876–2880. URL: https://ieeexplore.ieee.org/document/7532885.

[15] Nantinee Soodtoetong and Eakbodin Gedkhaw. "The efficiency of sign language recognition using 3D convolutional neural networks". In: *2018 15th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE. 2018, pp. 70–73. URL: https://ieeexplore.ieee.org/document/8619984.

[16] Tao Liu, Wengang Zhou, and Houqiang Li. "Sign language recognition with long short-term memory". In: *2016 IEEE international conference on image processing (ICIP)*. IEEE. 2016, pp. 2871–2875. URL: https://ieeexplore.ieee.org/document/7532884.

[17] *Getting Started with Kinect and Processing*. en. URL: https://shiffman.net/p5/kinect/.

[18] MReza et al. *Investigating the impact of a motion capture system on Microsoft Kinect v2 recordings: A caution for using the technologies together*. en. Sept. 2018. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6157830/.

[19] Mostafa Karbasi et al. "Analysis and enhancement of the denoising depth data using Kinect through iterative technique". In: *Jurnal Teknologi* 78.9 (2016).

[20] Yaofeng Xue et al. "A Chinese sign language recognition system using leap motion". In: *2017 International Conference on Virtual Reality and Visualization (ICVRV)*. IEEE. 2017, pp. 180–185. URL: https://ieeexplore.ieee.org/document/8719155.

[21] Kshitij Bantupalli and Ying Xie. "American sign language recognition using deep learning and computer vision". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 4896–4899. URL: https://ieeexplore.ieee.org/document/8622141.

[22] Bhumika Gupta, Pushkar Shukla, and Ankush Mittal. "K-nearest correlated neighbor classification for Indian sign language gesture recognition using feature fusion". In: *2016 International conference on computer communication and informatics (ICCCI)*. IEEE. 2016, pp. 1–5. URL: https://ieeexplore.ieee.org/document/7479951.

[23] Anusorn Chaikaew, Kritsana Somkuan, and Thidalak Yuyen. "Thai sign language recognition: an application of deep neural network". In: *2021 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunication Engineering*. IEEE. 2021, pp. 128–131. URL: https://ieeexplore.ieee.org/document/9425711.

[24] Wuyang Qin et al. "Sign Language Recognition and Translation Method based on VTN". In: *2021 International Conference on Digital Society and Intelligent Systems (DSInS)*. IEEE. 2021, pp. 111–115. URL: https://ieeexplore.ieee.org/document/9670588.

[25] A Sharmila Konwar, B Sagarika Borah, and CT Tuithung. "An American Sign Language detection system using HSV color model and edge detection". In: *2014 International Conference on Communication and Signal Processing*. IEEE. 2014, pp. 743–747. URL: https://ieeexplore.ieee.org/abstract/document/6949942.

[26] C Jose L Flores, AE Gladys Cutipa, and R Lauro Enciso. "Application of convolutional neural networks for static hand gestures recognition under different invariant features". In: *2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*. IEEE. 2017, pp. 1–4.

[27] PulkitS. *Computer Vision Tutorial: A Step-by-Step Introduction to Image Segmentation Techniques (Part 1)*. en. Apr. 2019. URL: https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/.

[28] Cheok Ming Jin, Zaid Omar, and Mohamed Hisham Jaward. "A mobile application of American sign language translation via image processing algorithms". In: *2016 IEEE Region 10 Symposium (TENSYMP)*. IEEE. 2016, pp. 104–109. URL: https://ieeexplore.ieee.org/document/7519386.

[29] Sourabh$_{Sinha}$. *Python | Grayscaling of Images using OpenCV*. en. July 2021. URL: https://www.geeksforgeeks.org/python-grayscaling-of-images-using-opencv/.

[30] Data Carpentry. *Thresholding*. en. URL: https://datacarpentry.org/image-processing/07-thresholding/.

[31] OpenCV Org. *What is Feature Extraction?* en. URL: https://deepai.org/machine-learning-glossary-and-terms/feature-extraction.

[32] Umang Patel and Aarti G Ambekar. "Moment based sign language recognition for indian languages". In: *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*. IEEE. 2017, pp. 1–6. URL: https://ieeexplore.ieee.org/document/8463901.

[33] Ariya Thongtawee, Onamon Pinsanoh, and Yuttana Kitjaidure. "A novel feature extraction for american sign language recognition using webcam". In: *2018 11th Biomedical Engineering International Conference (BMEiCON)*. IEEE. 2018, pp. 1–5. URL: https://ieeexplore.ieee.org/document/8609933.

[34] OpenCV Org. *Canny Edge Detection*. en. URL: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html.

[35] André Neyra-Gutiérrez and Pedro Shiguihara-Juárez. "Feature extraction with video summarization of dynamic gestures for peruvian sign language recognition". In: *2020 IEEE XXVII International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*. IEEE. 2020, pp. 1–4. URL: https://ieeexplore.ieee.org/document/9220243.

[36] Chang Jo Kim and Han-Mu Park. "Per-frame Sign Language Gloss Recognition". In: *2021 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2021, pp. 1125–1127. URL: https://ieeexplore.ieee.org/document/9621167.

[37] Sholikhatul Amaliya et al. "Study on Hand Keypoint Framework for Sign Language Recognition". In: *2021 7th International Conference on Electrical, Electronics and Information Engineering (ICEEIE)*. IEEE. 2021, pp. 446–451. URL: https://ieeexplore.ieee.org/document/9616851.

[38] OpenCV Org. *MediaPipe Holistic — Simultaneous Face, Hand and Pose Prediction, on Device*. en. Dec. 2020. URL: https://ai.googleblog.com/2020/12/mediapipe-holistic-simultaneous-face.html.

[39] Yangho Ji, Sunmok Kim, and Ki-Baek Lee. "Sign language learning system with image sampling and convolutional neural network". In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2017, pp. 371–375. URL: https://ieeexplore.ieee.org/document/7926567.

[40] Nobuhiko Mukai, Shoya Yagi, and Youngha Chang. "Japanese Sign Language Recognition based on a Video accompanied by the Finger Images". In: *2021 Nicograph International (NicoInt)*. IEEE. 2021, pp. 23–26. URL: https://ieeexplore.ieee.org/document/9515951.

[41] Shiva Verma. *Understanding 1D and 3D Convolution Neural Network | Keras*. en. Sept. 2019. URL: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610.

[42] Deepmind. *I3D model*. en. Feb. 2018. URL: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610.

[43] Priya Pedamkar. *Machine Learning vs Neural Network*. en. 2022. URL: https://www.educba.com/machine-learning-vs-neural-network/.

[44] Andrew D. *Building Your Own Dataset: Benefits, Approach, and Tools*. en. 2022. URL: https://towardsdatascience.com/building-your-own-dataset-benefits-approach-and-tools-6ab096e37f2.

[45] GOOGLE LLC. *MediaPipe Holistic*. en. 2020. URL: https://google.github.io/mediapipe/solutions/holistic.html.

[46] GOOGLE LLC. *Pose Landmarks from Mediapipe*. en. 2020. URL: https://google.github.io/mediapipe/images/mobile/pose_tracking_full_body_landmarks.png.

[47] GOOGLE LLC. *Hand Landmarks from Mediapipe*. en. 2020. URL: https://google.github.io/mediapipe/images/mobile/hand_landmarks.png.

[48] GOOGLE LLC. *Tensorflow*. en. 2015. URL: https://www.tensorflow.org/.

[49] OpenCV Team. *openCV*. en. 2022. URL: https://opencv.org/.

[50] OpenCV Team. *About openCV*. en. 2022. URL: https://opencv.org/about/.

[51] Nazar Kvartalnyi. *WHY USE PYTHON FOR MACHINE LEARNING?* en. 2022. URL: https://inoxoft.com/blog/why-use-python-for-machine-learning/.

[52] The Matplotlib development team. *Matplotlib: Visualization with Python*. en. 2022. URL: https://matplotlib.org/.

[53] scikit-learn team. *scikit-learn*. en. 2022. URL: https://scikit-learn.org/stable/.

[54] Jason Brownlee. *How to Visualize a Deep Learning Neural Network Model in Keras*. en. 2017. URL: https://machinelearningmastery.com/visualize-deep-learning-neural-network-model-keras/.

[55] Sharon Kinyan. *MediaPipe HandPose Detection using Python*. en. 2021. URL: https://www.section.io/engineering-education/handpose-detection-using-mediapipe-and-python/.

[56] Thenjiwe kubheka. *Artificial Intelligence Project: Pose Detection*. en. 2022. URL: https://levelup.gitconnected.com/artificial-intelligence-project-pose-detection-564dead6f10c.

[57] TensorFlow. *tf.keras.utils.to$_c$ategorical*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical.

[58] scikit-learn developers. *sklearn.model$_s$election.train$_t$est$_s$plit*. en. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[59] Stack Exchange. *What do x and y mean when working with data in the machine learning domain?* en. 2022. URL: https://datascience.stackexchange.com/questions/105104/what-do-x-and-y-mean-when-working-with-data-in-the-machine-learning-domain.

[60] Stack Exchange. *X, Y names in data-science*. en. 2020. URL: https://datascience.stackexchange.com/questions/78214/x-y-names-in-data-science.

[61] TensorFlow. *tf.keras.callbacks.TensorBoard*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/TensorBoard.

[62] Karsten Eckhardt. *Choosing the right Hyperparameters for a simple LSTM using Keras*. en. 2018. URL: https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046.

[63] TensorFlow. *tf.keras.layers.LSTM*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM.

[64] Jason Brownlee. *Your First Deep Learning Project in Python with Keras Step-by-Step*. en. 2022. URL: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/.

[65] Sagar Sharma. *Activation Functions in Neural Networks*. en. 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[66] Palash Sharma. *Keras Dense Layer Explained for Beginners*. en. 2020. URL: https://machinelearningknowledge.ai/keras-dense-layer-explained-for-beginners/.

[67] TensorFlow. *tf.keras.layers.Dense*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense.

[68] Sanket Doshi. *Various Optimization Algorithms For Training Neural Network*. en. 2019. URL: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.

[69] ProjectPro. *How to compile a keras model?* en. 2022. URL: https://www.projectpro.io/recipes/compile-keras-model.

[70] TensorFlow. *tf.keras.metrics.categorical$_c$rossentropy*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/categorical_crossentropy.

[71] TensorFlow. *tf.keras.metrics.categorical$_a$ccuracy*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/categorical_accuracy.

[72] Jason Brownlee. *How to Visualize a Deep Learning Neural Network Model in Keras*. en. 2017. URL: https://machinelearningmastery.com/visualize-deep-learning-neural-network-model-keras/.

[73] TensorFlow. *tf.keras.utils.plot$_m$odel*. en. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/utils/plot_model.

[74] scikit-learn developers. *sklearn.metrics.multilabel$_c$onfusion$_m$atrix*. en. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.multilabel_confusion_matrix.html.

[75] Google Developers. *Classification: True vs. False and Positive vs. Negative*. en. 2022. URL: https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative.

[76] Sarang Narkhede. *Understanding Confusion Matrix*. en. 2018. URL: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[77] Sarang Narkhede. *Comparison between Machine Learning Deep Learning*. en. 2018. URL: https://miro.medium.com/max/445/1*Z54JgbS4DUwWSknhDCvNTQ.png.

[78] scikit-learn developers. *sklearn.metrics.accuracy$_s$core*. en. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html.

APPENDIX

```
In [1]: !pip install opencv-python
        !pip install matplotlib
        !pip install mediapipe
        !pip install sklearn
        !pip install tensorflow
        !pip install tensorflow-gpu
        !pip install pydot
        !pip install pydotplus
        !conda install graphviz
```

Figure 17: Libraries installed

```
In [2]: import cv2
        import numpy as np
        import os
        from matplotlib import pyplot as plt
        import time
        import mediapipe as mp
        from sklearn.model_selection import train_test_split
        from tensorflow.keras.utils import to_categorical
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import LSTM, Dense
        from tensorflow.keras.callbacks import TensorBoard
        from keras.utils.vis_utils import plot_model
        from sklearn.metrics import multilabel_confusion_matrix, accuracy_score
```

Figure 18: Packages imported

APPENDIX

Accessing Webcam

In [3]:
```
capture = cv2.VideoCapture(0) #grab video capture device 0
#for my machine device 0 was webcam
#the device number can varry depending on which machine the code is running on

while capture.isOpened(): #running the loop till video camera is open

    # Reading the feed from webcam
    return_value, frame = capture.read()

    # Displaying feed to the user
    cv2.imshow('Image feed from OpenCV', frame)

    # Breaking from the loop
    # Wait for the key to be pressed inside the frame
    # Break out of loop if escape key is pressed
    if cv2.waitKey(10) & 0xFF == ord('\x1b'):
        break

capture.release() #release the webcam
cv2.destroyAllWindows() #close down all frames
```

Figure 19: Accessing Webcam of device Using OpenCV

Setting up MediaPipe

Sharon Kinyan (2021), Detecting handposes using images captured by our webcam (Title of Program), https://www.section.io/engineering-education/handpose-detection-using-mediapipe-and-python/

In [2]:
```
#drawing utility helps in drawing landmarks
mp_drawing = mp.solutions.drawing_utils

# Actual Holistic Model
mp_holistic = mp.solutions.holistic
```

Figure 20: Variables for importing MediaPipe

In [17]:
```
#Function for performing detection with mediapipe
#OpenCV read the image in the BGR channel format but detection done by mediapipe needs to be in format of RGB

def detection_with_mediapipe(image, model):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # Convert image from BGR to RGB
    image.flags.writeable = False                  # Set image to unwritable
    results = model.process(image)                 # Perfrom Detection with MediaPipe
    image.flags.writeable = True                   # Set image back to writable
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR) # Convert image from RGB to BGR
    return image, results
```

Figure 21: Function for performing detection with Mediapipe

APPENDIX

Combining Mediapipe With OpenCV

In [6]:
```
capture = cv2.VideoCapture(0)
# Access mediapipe model with OpenCV

# min_detection_confidence is initial detection confidence
# min_tracking_confidence is the preceeding tracking confidence
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic_model:
    while capture.isOpened():  #running the loop till video camera is open

        # Reading the feed from webcam
        return_value, frame = capture.read()

        #Detecting with MediaPipe
        image, results = detection_with_mediapipe(frame, holistic_model)
        print(results)

        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', frame)

        # Breaking from the loop
        # Wait for the key to be pressed inside the frame
        # Break out of loop if escape key is pressed
        if cv2.waitKey(10) & 0xFF == ord('\x1b'):
            break
    capture.release() #release the webcam
    cv2.destroyAllWindows() #close down all frames
```

Figure 22: Combining Mediapipe detection with the webcam

Rendering Landmarks on the feed

Thenjiwe kubheka (2022), Artificial Intelligence Project: Pose Detection (Title of Program),https://levelup.gitconnected.com/artificial-intelligence-project-pose-detection-564dead6f10c

In [148]:
```
def visualize_landmarks(image, results):
    #using mediapipe drawing utilities to draw landmarks on images
    mp_drawing.draw_landmarks(image, results.face_landmarks, mp_holistic.FACEMESH_TESSELATION) # Draw face connections
    mp_drawing.draw_landmarks(image, results.pose_landmarks, mp_holistic.POSE_CONNECTIONS) # Draw pose connections
    mp_drawing.draw_landmarks(image, results.left_hand_landmarks, mp_holistic.HAND_CONNECTIONS) # Draw left hand connections
    mp_drawing.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.HAND_CONNECTIONS) # Draw right hand connections
```

Figure 23: Drawing landmarks on the image with MediaPipe draw_landmakrs Function

APPENDIX

In [10]:
```
capture = cv2.VideoCapture(0)
# Access mediapipe model with OpenCV

# min_detection_confidence is initial detection confidence
# min_tracking_confidence is the preceeding tracking confidence
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic_model:
    while capture.isOpened():  #running the loop till video camera is open

        # Reading the feed from webcam
        return_value, frame = capture.read()

        #Detecting with MediaPipe
        image, results = detection_with_mediapipe(frame, holistic_model)
        print(results)

        # Render landmarks on feed
        visualize_landmarks(image, results)

        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', image)

        # Breaking from the loop
        # Wait for the key to be pressed inside the frame
        # Break out of loop if escape key is pressed
        if cv2.waitKey(10) & 0xFF == ord('\x1b'):
            break
    capture.release() #release the webcam
    cv2.destroyAllWindows() #close down all frames
```

Figure 24: Rendering landmarks on image feed

In [373]:
```
def visualize_custom_landmarks(image, results):
    # Draw face connections
    mp_drawing.draw_landmarks(image, results.face_landmarks, mp_holistic.FACEMESH_TESSELATION,
                              mp_drawing.DrawingSpec(color=(255,105,180), thickness=1, circle_radius=1),
                              mp_drawing.DrawingSpec(color=(255,45,10), thickness=1, circle_radius=1)
                              )
    # Draw pose connections
    mp_drawing.draw_landmarks(image, results.pose_landmarks, mp_holistic.POSE_CONNECTIONS,
                              mp_drawing.DrawingSpec(color=(178, 190, 181), thickness=2, circle_radius=4),
                              mp_drawing.DrawingSpec(color=(178,180,200), thickness=2, circle_radius=2)
                              )
    # Draw left hand connections
    mp_drawing.draw_landmarks(image, results.left_hand_landmarks, mp_holistic.HAND_CONNECTIONS,
                              mp_drawing.DrawingSpec(color=(8, 8, 136), thickness=2, circle_radius=4),
                              mp_drawing.DrawingSpec(color=(20, 20, 146), thickness=2, circle_radius=2)
                              )
    # Draw right hand connections
    mp_drawing.draw_landmarks(image, results.right_hand_landmarks, mp_holistic.HAND_CONNECTIONS,
                              mp_drawing.DrawingSpec(color=(159, 43, 104), thickness=2, circle_radius=4),
                              mp_drawing.DrawingSpec(color=(159, 63, 144), thickness=2, circle_radius=2)
                              )
```

Figure 25: Drawing landmarks with custom formatting on the image

APPENDIX

In [16]:
```
capture = cv2.VideoCapture(0)
# Access mediapipe model with OpenCV

# min_detection_confidence is initial detection confidence
# min_tracking_confidence is the preceeding tracking confidence
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic_model:
    while capture.isOpened():  #running the loop till video camera is open

        # Reading the feed from webcam
        return_value, frame = capture.read()

        #Detecting with MediaPipe
        image, results = detection_with_mediapipe(frame, holistic_model)
        print(results)

        # Render custom landmarks on feed
        visualize_custom_landmarks(image, results)

        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', image)

        # Breaking from the loop
        # Wait for the key to be pressed inside the frame
        # Break out of loop if escape key is pressed
        if cv2.waitKey(10) & 0xFF == ord('\x1b'):
            break
    capture.release() #release the webcam
    cv2.destroyAllWindows() #close down all frames
```

Figure 26: Rendering custom formatted landmarks on feed

Setting up folders and file paths

In [210]:
```
#Setup folders for storing keypoints for every type of sign

# Different types of sign
signs = np.array(['hello','tea','thanks','like','sweet','good','iloveyou'])

# Eighty videos worth of data
total_sequences = 80

# Each video is going to be 30 frames in length
number_of_frames = 30

# Folder for storing Sequences
keypoints_collected = os.path.join('keypoints_collected')

#creating file path for every 80 sequence
for sign in signs:
    for sequence in range(total_sequences):
        try:
            os.makedirs(os.path.join(keypoints_collected, sign, str(sequence)))
        except:
            pass
```

Figure 27: Setting up folders for Keypoints Extraction

APPENDIX

```
In [36]: len(results.face_landmarks.landmark)
Out[36]: 468

In [44]: len(results.pose_landmarks.landmark)
Out[44]: 33

In [46]: len(results.left_hand_landmarks.landmark)
Out[46]: 21

In [50]: len(results.left_hand_landmarks.landmark)
Out[50]: 21
```

Figure 28: Length of different landmarks array

Keypoints Extraction

```
In [242]: len(results.face_landmarks.landmark)
Out[242]: 468

In [246]: #saving all face Landmarks
          face_landmark = []
          #for every value inside face landmarks
          for val in results.face_landmarks.landmark:
              #extract value for each coordinate
              test = np.array([val.x, val.y, val.z])
              face_landmark.append(test)

In [248]: len(face_landmark)
Out[248]: 468
```

Figure 29: Storing values for face landmarks in NumPy Array

APPENDIX

```
In [249]: #without flattening the array
          face_landmark = np.array([[val.x, val.y, val.z] for val in results.face_landmarks.landmark])

In [251]: len(face_landmark)

Out[251]: 468

In [252]: #array after flattening
          face_landmark = np.array([[val.x, val.y, val.z] for val in results.face_landmarks.landmark]).flatten()

In [253]: len(face_landmark)

Out[253]: 1404

In [254]: face_landmark.shape

Out[254]: (1404,)
```

Figure 30: Flattening the NumPy array to change the shape of array

```
#Extracting face landmarks
if results.face_landmarks:
    face_array = np.array([[val.x, val.y, val.z] for val in results.face_landmarks.landmark]).flatten()
else:
    face_array = np.zeros(1404) #468 Landmarks * 3 face coordinates
```

Figure 31: Creating NumPy zeroes array for face landmarks of same shape and size

```
In [231]: def keypoints_extraction(results):
              #Storing all keypoints as numpy arrays
              #If for any image there is no keypoints, then nympy zeroes array of same size will be stored

              #Extracting face landmarks
              if results.face_landmarks:
                  face_array = np.array([[val.x, val.y, val.z] for val in results.face_landmarks.landmark]).flatten()
              else:
                  face_array = np.zeros(1404) #468 Landmarks * 3 face coordinates

              #Extracting face landmarks
              if results.pose_landmarks:
                  pose_array = np.array([[val.x, val.y, val.z,val.visibility] for val in results.pose_landmarks.landmark]).flatten()
              else:
                  pose_array = np.zeros(132) #33 Pose Landmarks * 4 Pose coordinates

              #Extracting right hand Landmarks
              if results.right_hand_landmarks:
                  right_hand_array = np.array([[val.x, val.y, val.z] for val in results.right_hand_landmarks.landmark]).flatten()
              else:
                  right_hand_array = np.zeros(63) #21 Right hand landmakrs * 3 Right hand coordinates

              #Extracting left hand landmarks
              if results.left_hand_landmarks:
                  left_hand_array = np.array([[val.x, val.y, val.z] for val in results.left_hand_landmarks.landmark]).flatten()
              else:
                  left_hand_array = np.zeros(63) #21 left hand landmakrs * 3 left hand coordinates

              return np.concatenate([ pose_array, face_array, left_hand_array ,right_hand_array ])
```

Figure 32: Extracting Keypoints for every landmark

APPENDIX

```
In [79]: #final array with all keypoints
         keypoints_extraction(results)

Out[79]: array([ 0.62328637,  0.78348631, -0.02788057, ...,  0.86940926,
                 0.52089745, -0.0532684 ])

In [80]: #shape of final array with all keypoints
         keypoints_extraction(results).shape

Out[80]: (1662,)

In [81]: #total number of keypoints
         468*3+33*4+21*3+21*3

Out[81]: 1662
```

Figure 33: Size of final numpy array containing all keypoints

```
In [25]: #collecting Keypoints through webcam
capture = cv2.VideoCapture(0)
# Access mediapipe model with OpenCV

# min_detection_confidence is initial detection confidence
# min_tracking_confidence is the preceeding tracking confidence
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic_model:

    # Loop through each action
    for sign in signs:
        # Loop through sequences (videos)
        for sequence in range(total_sequences):
            # Loop through sequence length (video length)
            for frames in range(number_of_frames):

                # Reading the feed from webcam
                return_value, frame = capture.read()

                #Making detection with MediaPipe
                image, results = detection_with_mediapipe(frame, holistic_model)

                # Rendering custom landmarks on feed
                visualize_custom_landmarks(image, results)

                # Collecting Keypoints
                collect_keypoint(frames,sign,sequence)

                # Save keypoints in respective folders
                save_keypoints(results,sign,sequence,frames)

                # Breaking from the loop
                # Wait for the key to be pressed inside the frame
                # Break out of loop if escape key is pressed
                if cv2.waitKey(10) & 0xFF == ord('\x1b'):
                    break

    capture.release() #release the webcam
    cv2.destroyAllWindows() #close down all frames
```

Figure 34: Running the loop to collect keypoints

APPENDIX

In [23]:
```
#function for collecting keypoints from feed
def collect_keypoint(frames, sign, sequence):
    #if at frame zero
    #display "Collecting Keypoints"
    #display action and sequence number
    #wait for 2 seconds
    if frames == 0:
        cv2.putText(image, 'Collecting Keypoints', (120,200),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255, 0), 4, cv2.LINE_AA)
        cv2.putText(image, 'Frames for action: {} and current video number is: {}'.format(sign, sequence), (15,12),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 255), 1, cv2.LINE_AA)
        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', image)
        cv2.waitKey(2000)
    #if not at frame zero
    #display action and sequence number
    else:
        cv2.putText(image, 'Frames for action: {} and current video number is: {}'.format(sign, sequence), (15,12),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 255), 1, cv2.LINE_AA)
        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', image)
```

Figure 35: Displaying information to user between collection

In [170]:
```
### saving keypoints in previously setup file path
#7 signs
#Each sign with 80 sequences
#Each sequence with 30 frames
def save_keypoints (results,sign,sequence,frames):
    extracted_keypoints = keypoints_extraction(results)
    npy_path = os.path.join(keypoints_collected, sign, str(sequence), str(frames))
    np.save(npy_path, extracted_keypoints)
```

Figure 36: Storing collected keypoints in their respective folders

Pre Processing Data

In [175]:
```
#creating a dictionary of all signs for creating labels
#this is done for mapping all the labels
dictionary_of_signs = {label:num for num, label in enumerate(signs)}
```

In [176]:
```
dictionary_of_signs
```

Out[176]:
```
{'hello': 0,
 'tea': 1,
 'thanks': 2,
 'like': 3,
 'sweet': 4,
 'good': 5,
 'iloveyou': 6}
```

Figure 37: Creating Label dictionary for each action

APPENDIX

In [211]:
```
#Initialize two empty list for storing features and labels
#features_data will be X-data
#labels_data will be y-data
feature_data, labels_data = [], []

#loop through each sign
for sign in signs:
    #loop through each sequence
    for sequence in range(total_sequences):
        #this list will represent all of the frames in a particular sequence
        frames_in_sequence = []
        #loop through each frame
        for frames in range(number_of_frames):
            #loading each frame
            data = np.load(os.path.join(keypoints_collected, sign, str(sequence), "{}.npy".format(frames)))
            frames_in_sequence.append(data)
        #append frames from each sequence to feature_data
        feature_data.append(frames_in_sequence)
        #creating labels from all the signs
        labels_data.append(dictionary_of_signs[sign])
```

In [212]:
```
np.array(feature_data).shape
```

Out[212]: (560, 30, 1662)

In [213]:
```
np.array(labels_data).shape
```

Out[213]: (560,)

Figure 38: Structuring all the data in one big array

Creating X and y data

In [257]:
```
#creating X_data
#storing sequences into numpy array
X_data = np.array(feature_data)
```

In [258]:
```
X_data.shape
```

Out[258]: (560, 30, 1662)

In [259]:
```
#creating y_data
# "to_categorical" function from Keras utilities was used to hot-encode label_data into a binary flag
y_data = to_categorical(labels_data).astype(int)
```

In [260]:
```
y_data
```

Out[260]:
```
array([[1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 1],
       [0, 0, 0, ..., 0, 0, 1],
       [0, 0, 0, ..., 0, 0, 1]])
```

In [261]:
```
y_data.shape
```

Out[261]: (560, 7)

Figure 39: Creating X and y data

APPENDIX

Seperate Training and Testing partititons

```
In [219]: #Seperating training and testing partitions
          #"train\_test\_split" function from scikit-learn was used to make training and testing partition

          X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, test_size=0.05)

In [220]: X_train.shape
Out[220]: (532, 30, 1662)

In [221]: X_test.shape
Out[221]: (28, 30, 1662)

In [222]: y_train.shape
Out[222]: (532, 7)

In [223]: y_test.shape
Out[223]: (28, 7)
```

Figure 40: Separating pre processed data into training and testing partition

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.callbacks import TensorBoard
from keras.utils.vis_utils import plot_model
```

Figure 41: Dependencies used for building neural network model

Building Neural Network Model

Jason Brownlee (2022), Your First Deep Learning Project in Python with Keras Step-by-Step (Title of Program), https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

Karsten Eckhardt (2018), Choosing the right Hyperparameters for a simple LSTM using Keras (Title of Program),https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046

```
In [232]: #creating log directory for storing log file
          log_dir = os.path.join('Logfile')
          #setting up tensorflow callback
          tensorboard_callback = TensorBoard(log_dir=log_dir)
```

Figure 42: File path for TensorBoard log file

APPENDIX

```
In [233]: #Initializing the model as sequential
model = Sequential()
#Adding three LSTM layers
#setting the right amount of input_shape in first layer
model.add(LSTM(64, return_sequences=True, activation='relu', input_shape=(30,1662)))
model.add(LSTM(128, return_sequences=True, activation='relu'))
#setting the return_sequences=False in last LSTM layer
model.add(LSTM(64, return_sequences=False, activation='relu'))
#three dense layers
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
#final layer which will return probabilities of signs
model.add(Dense(signs.shape[0], activation='softmax'))
```

Figure 43: Building Neural Network Model

```
In [234]: #Compiling the Neural Network Model with 'ADAGRAD' optimizer and 'categorical_crossentropy' loss function
#Metrics was set to 'categorical_accuracy' to track the accuracy while training
model.compile(optimizer='ADAGRAD', loss='categorical_crossentropy', metrics=['categorical_accuracy'])
```

Figure 44: Compile Neural Network Model

```
In [227]: #training the model for 400 epochs
#setting the callback to 'tensorboard_callback'
model.fit(X_train, y_train, epochs=400, callbacks=[tb_callback])
```

Figure 45: Training Neural Network Model

APPENDIX

```
In [227]: #training the model for 400 epochs
          #setting the callback to 'tensorboard_callback'
          model.fit(X_train, y_train, epochs=400, callbacks=[tb_callback])
          Epoch 392/400
          17/17 [==============================] - 2s 112ms/step - loss: 0.0024 - categorical_accuracy: 1.0000
          Epoch 393/400
          17/17 [==============================] - 2s 111ms/step - loss: 0.0023 - categorical_accuracy: 1.0000
          Epoch 394/400
          17/17 [==============================] - 2s 100ms/step - loss: 0.0023 - categorical_accuracy: 1.0000
          Epoch 395/400
          17/17 [==============================] - 2s 102ms/step - loss: 0.0022 - categorical_accuracy: 1.0000
          Epoch 396/400
          17/17 [==============================] - 2s 105ms/step - loss: 0.0021 - categorical_accuracy: 1.0000
          Epoch 397/400
          17/17 [==============================] - 2s 103ms/step - loss: 0.0022 - categorical_accuracy: 1.0000
          Epoch 398/400
          17/17 [==============================] - 2s 102ms/step - loss: 0.0020 - categorical_accuracy: 1.0000
          Epoch 399/400
          17/17 [==============================] - 2s 119ms/step - loss: 0.0020 - categorical_accuracy: 1.0000
          Epoch 400/400
          17/17 [==============================] - 2s 106ms/step - loss: 0.0019 - categorical_accuracy: 1.0000
Out[227]: <keras.callbacks.History at 0x2168d528760>
```

Figure 46: Training model for 1000 epochs

```
Command Prompt - tensorboard  --logdir=.
C:\Users\Nikunj\Desktop\7 ADAGRAD2\Logfile\train>tensorboard --logdir=.
2022-09-08 08:09:45.846559: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
ry 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2022-09-08 08:09:45.846730: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do n
ot have a GPU set up on your machine.
2022-09-08 08:09:50.664353: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
ry 'nvcuda.dll'; dlerror: nvcuda.dll not found
2022-09-08 08:09:50.664476: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (
303)
2022-09-08 08:09:50.669428: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic inform
ation for host: Snicker-Doodle
2022-09-08 08:09:50.669593: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: Snicker-Doodle
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.9.1 at http://localhost:6006/ (Press CTRL+C to quit)
```

Figure 47: Opening TensorBoard web-app through command prompt

```
In [393]: #Initialize threshold value
          threshold_value = 0.85
          #Intialize an empty list to collect keypoints from actions performed infront of webcam
          stack_frames = []
          #Intialize an empty list to store the predicted values based on the actions
          predicted_values = []
          #Color array for probability visualization
          vibgyor = [(148,0,211), (75,0,130), (0, 0, 255),(0, 255, 0),(255, 255, 0), (255, 127, 0),(255, 0 , 0)]
```

Figure 48: Initialize variables for making prediction in real time

APPENDIX

```
In [389]: #Function of storing frames inside 'stack_frames' list
          def frames_from_feed(results,stack_frames):
              #Extract keypoints
              extracted_keypoints = keypoints_extraction(results)
              #store extracted keypoints in 'stack_frames' list
              stack_frames.append(extracted_keypoints)
              #get the keypoints for latest 30 frames
              #because our dataset is made up of keypoints from 30 frames
              stack_frames = stack_frames[-30:]
              return stack_frames
```

Figure 49: Function for storing latest 30 frames

```
In [390]: #Function to provide data to display after detection
          def visulization_function(stack_frames,predicted_values, signs, image, colors):
              #if there are 30 frames in the 'stack_frames' list
              if len(stack_frames) == 30:
                  #make predcition using model.predict
                  result = model.predict(np.expand_dims(stack_frames, axis=0))[0]

                  #if the value got from prediction is more than the set threshold_value
                  if result[np.argmax(result)] > threshold_value:
                      #then check if their is already some value in the 'predicted_values' list
                      if len(predicted_values) > 0:
                          #if there is some value in 'predicted_values' list then check the last value
                          #because we don't want to display the same sign again
                          if signs[np.argmax(result)] != predicted_values[-1]:
                              #if the value is not same then store the predicted sign in 'predicted_values' list
                              predicted_values.append(signs[np.argmax(result)])

                      #if there is no value in the 'predicted_values' list
                      else:
                          #straight away add the predicted sign in 'predicted_values' list
                          predicted_values.append(signs[np.argmax(result)])

                  #To make sure we don't display more than 7 signs at a time
                  #grab the last 7 values from predicted_values list if length of list exceeds 7
                  if len(predicted_values) > 7:
                      predicted_values = predicted_values[-7:]

                  # calling probability_visualization
                  image = probability_visualization(result, signs, image, vibgyor)

              return predicted_values, image
```

Figure 50: Function for making prediction and storing the predicted signs

APPENDIX

In [396]:
```
#defining probability visualization to see how probabilities are being calculated in real-time
def probability_visualization(result, signs, input_image, vibgyor):
    output_image = input_image.copy()
    for num, prob in enumerate(result):
        #specifying the positions of rectangles that will fill the colors based on probaility predicted
        cv2.rectangle(output_image, (0,60+num*40), (int(prob*100), 90+num*40), vibgyor[num], -1)
        #putting the text with color in those rectangles
        cv2.putText(output_image, signs[num], (0, 85+num*40), cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 2, cv2.LINE_AA)
    return output_image
```

Figure 51: Probability Visualization to show how probabilities are being calculated for predicted signs in real-time

In [397]:
```
capture = cv2.VideoCapture(0)
# Set mediapipe model
with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as holistic_model:
    while capture.isOpened():


        # Reading the feed from webcam
        return_value, frame = capture.read()

        #Detecting with MediaPipe
        image, results = detection_with_mediapipe(frame, holistic_model)

        # Render custom landmarks on feed
        visualize_custom_landmarks(image, results)

        #Collect the frames from the feed and store the latest 30 frames in stack_frames list
        stack_frames = frames_from_feed(results,stack_frames)

        #make prediction based on the values in 'stack_frames' list
        predicted_values, image = visulization_function(stack_frames, predicted_values, signs, image, colors)

        #Display the predicted values at the bottom of screen
        #add a rectangle at the bottom of the feed
        cv2.rectangle(image, (0,450), (640, 480), (245, 117, 16), -1)
        #Display predicted signs over the rectangle
        cv2.putText(image, ' '.join(predicted_values), (3,470),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2, cv2.LINE_AA)

        # Displaying feed to the user
        cv2.imshow('Image feed from OpenCV', image)

        # Breaking from the loop
        # Wait for the key to be pressed inside the frame
        # Break out of loop if 'escape' key is pressed
        if cv2.waitKey(10) & 0xFF == ord('\x1b'):
            break
    capture.release()
    cv2.destroyAllWindows()
```

Figure 52: Displaying Predicted sign using OpenCV

```
In [29]: model.compile(optimizer='ADAM', loss='categorical_crossentropy', metrics=['categorical_accuracy'])

In [23]: #model.fit(X_train, y_train, epochs=2000, callbacks=[tb_callback])

In [19]: res = model.predict(X_test)

         1/1 [==============================] - 0s 363ms/step

In [21]: y_hat = model.predict(X_train)

         17/17 [==============================] - 1s 35ms/step

In [22]: y_true = np.argmax(y_train, axis=1).tolist()
         y_hat = np.argmax(y_hat, axis=1).tolist()

In [24]: multilabel_confusion_matrix(y_true, y_hat)

Out[24]: array([[[456,   0],
                 [ 76,   0]],

                [[  0, 454],
                 [  0,  78]],

                [[459,   0],
                 [ 73,   0]],

                [[454,   0],
                 [ 78,   0]],

                [[454,   0],
                 [ 78,   0]],

                [[458,   0],
                 [ 74,   0]],

                [[457,   0],
                 [ 75,   0]]], dtype=int64)

In [25]: accuracy_score(y_true, y_hat)

Out[25]: 0.14661654135338345
```

Figure 53: Evaluating results from ADAM optimizer

APPENDIX

```
In [19]: model.compile(optimizer='ADAGRAD', loss='categorical_crossentropy', metrics=['categorical_accuracy'])

In [30]: # model.fit(X_train, y_train, epochs=2000, callbacks=[tb_callback])

In [21]:  model.save('action.h5')

In [22]: res = model.predict(X_test)

         1/1 [==============================] - 1s 679ms/step

In [23]: y_hat = model.predict(X_train)

         17/17 [==============================] - 2s 98ms/step

In [27]: y_true = np.argmax(y_train, axis=1).tolist()
         y_hat = np.argmax(y_hat, axis=1).tolist()

In [28]: multilabel_confusion_matrix(y_true, y_hat)

Out[28]: array([[[455,   0],
                 [  0,  77]],

                [[456,   0],
                 [  0,  76]],

                [[455,   0],
                 [  0,  77]],

                [[459,   0],
                 [  0,  73]],

                [[458,   0],
                 [  0,  74]],

                [[454,   0],
                 [  0,  78]],

                [[455,   0],
                 [  0,  77]]], dtype=int64)

In [29]: accuracy_score(y_true, y_hat)

Out[29]: 1.0
```

Figure 54: Evaluating results obtained from training the model with ADAGRAD optimizer

```
In [32]: model.compile(optimizer='ADADELTA', loss='categorical_crossentropy', metrics=['categorical_accuracy'])

In [39]: # model.fit(X_train, y_train, epochs=2000, callbacks=[tb_callback])

In [34]: res = model.predict(X_test)

         1/1 [==============================] - 1s 594ms/step

In [35]: y_hat = model.predict(X_train)

         17/17 [==============================] - 1s 56ms/step

In [36]: y_true = np.argmax(y_train, axis=1).tolist()
         y_hat = np.argmax(y_hat, axis=1).tolist()

In [37]: multilabel_confusion_matrix(y_true, y_hat)

Out[37]: array([[[447,  11],
                 [  8,  66]],

                [[447,   6],
                 [  4,  75]],

                [[460,   0],
                 [  9,  63]],

                [[453,   4],
                 [  0,  75]],

                [[443,  12],
                 [  8,  69]],

                [[450,   5],
                 [ 13,  64]],

                [[445,   9],
                 [  5,  73]]], dtype=int64)

In [38]: accuracy_score(y_true, y_hat)

Out[38]: 0.9116541353383458
```

Figure 55: Evaluating results obtained from training the model with ADADELTA optimizer

APPENDIX

```
In [43]: model.compile(optimizer='SGD', loss='categorical_crossentropy', )

In [50]: #model.fit(X_train, y_train, epochs=2000, callbacks=[tb_callback])

In [45]: res = model.predict(X_test)

         1/1 [==============================] - 1s 555ms/step

In [46]: y_hat = model.predict(X_train)

         17/17 [==============================] - 1s 74ms/step

In [47]: y_true = np.argmax(y_train, axis=1).tolist()
         y_hat = np.argmax(y_hat, axis=1).tolist()

In [48]: multilabel_confusion_matrix(y_true, y_hat)

Out[48]: array([[[453,   5],
                 [  5,  69]],

                [[434,  19],
                 [  7,  72]],

                [[460,   0],
                 [  0,  72]],

                [[454,   3],
                 [  1,  74]],

                [[444,  11],
                 [  5,  72]],

                [[455,   0],
                 [ 13,  64]],

                [[446,   8],
                 [ 15,  63]]], dtype=int64)

In [49]: accuracy_score(y_true, y_hat)

Out[49]: 0.9135338345864662
```

Figure 56: Evaluating results obtained from training the model with SGD optimizer

```
In [29]: model.compile(optimizer='ADAM', loss='categorical_crossentropy', metrics=['categorical_accuracy'])

In [23]: #model.fit(X_train, y_train, epochs=2000, callbacks=[tb_callback])

In [19]: res = model.predict(X_test)

         1/1 [==============================] - 0s 363ms/step

In [21]: y_hat = model.predict(X_train)

         17/17 [==============================] - 1s 35ms/step

In [22]: y_true = np.argmax(y_train, axis=1).tolist()
         y_hat = np.argmax(y_hat, axis=1).tolist()

In [24]: multilabel_confusion_matrix(y_true, y_hat)

Out[24]: array([[[456,   0],
                 [ 76,   0]],

                [[  0, 454],
                 [  0,  78]],

                [[459,   0],
                 [ 73,   0]],

                [[454,   0],
                 [ 78,   0]],

                [[454,   0],
                 [ 78,   0]],

                [[458,   0],
                 [ 74,   0]],

                [[457,   0],
                 [ 75,   0]]], dtype=int64)

In [25]: accuracy_score(y_true, y_hat)

Out[25]: 0.14661654135338345
```

Figure 57: Evaluating results obtained from training the model with ADAM optimizer