Sign Language Interpretation System: Software Design Document

William Hughes, Braden Bagby, Zachary Langford, Robert Stonner, David Gray

Contents

1.0 Introduction

This software project, known as the sign language interpretation system, seeks to create a system that interprets hand gestures through a camera as individual alphanumeric characters, which can then be collected as strings of command phrases or words and used as an output to systems such as a smart home device. To expand upon this system, we as a team will also separate the camera capturing components and smart home command output components from the main gesture and command

phrase processing components by putting the latter on a cloud environment that will be connected to by the former. This document will first outline this entire project in terms of its overall goals, scope, and architecture. Then, we will break the project into its major components, and provide a detailed implementation plan for those components, including design diagrams and architecture. Finally, this document will conclude with an overview of user interface components, a detailed analysis of the constraints that relate more specifically to this project, and a quality assurance and testing guide that identify critical parts of the software project.

1.1 Goals and objectives

The overall goal of our sign language interpretation system is to be able to recognize American Sign Language (ASL) symbols A through Z with an accuracy of ninety percent or greater. Another goal is for this system to be compatible with a smart home device like the way voice recognition is utilized.

The software in our project serves four main objectives: detecting a hand in the camera view, recognizing and translating hand gestures, predicting a valid command from a string of letters, and turning a command into action via smart home device API.

1.2 Statement of scope

This section contains the scope of this software project and the functionalities we feel are necessary to complete the project, as well as functionalities that are less important. This should generate a very broad idea of this project and what it does.

1.2.1 Essential Functionality

For this project, there are a few essential functionalities. This means that these functionalities are necessary to produce the main processes originally designated by the stakeholders for the project. This includes the following for the system as a whole:

- The system must receive input from a live camera feed. This will be known as the video feed input module.
- The system must be able to detect if a hand is within the frame of the live camera feed. This will be known as the hand tracking module.
- The system must be able to interpret hand gestures using a machine learning model for alphanumeric characters in American Sign Language that are performed by a hand when it is within the frame of the live camera feed into the corresponding characters in the English alphabet as string data within the application. This will be known as the gesture interpretation module.
- The system must be able to output sequences of characters from American Sign Language gestures made within a few seconds of each other. This will be known as the command output module.

- The system must have subprocess and error console logging for each component.
- The system must have a video debugging console containing a view of the live camera feed as it is recording.

1.2.2 Desirable Functionalities

The next set of functionalities are desirable for the system during development, but are not key to the basic, outlined functionality of the system:

- The system must collect the interpreted hand gesture characters from the English alphabet and can output a collection of them to another system, such as a smart home device that can perform command phrases. This would also be considered an addition to the command output module.
- The system must be able to correct for small amounts of misinterpreted hand gesture characters within a gesture interpretation module using a grammar module (for example, the command phrase "LJGHTON" could be reinterpreted by the grammar system, and correctly predict that the intended phrase was "LIGHTON"). This component will be known as the formal grammar module.
- The system must have a cloud environment that contains the gesture interpretation and grammar modules that is separate from the client-side video feed input and command phrase output modules, so that these separate sets of modules communicate over the internet (to increase the portability of the system). This will be known as the cloud environment.

1.2.3 Future Functionalities

These final listed functionalities represent future goals and aspirations of the software project that may be completed during development, but are unlikely to be completed and unnecessary to any core functionality:

- The system must be able to interpret hand gestures that correspond to entire words or phrases.
- The system must contain the ability for future developers to add new or custom hand gesture commands to the gesture interpretation module.
- The system must have direct integration into a smart home device via a custom application programming interface made for smart home devices. This would also be considered an addition to the command output module.

1.3 Software context

This software has many use case environments from a user perspective or a developer. For example, those who are otherwise unable to speak may be able to use this technology to interact with a smart home device, giving them the capabilities of any other user, such as turning lights on and off, controlling room temperature, and monitoring security devices such as cameras. The technology of this software project and its modularity allow it to be reused for many different applications where accurate capture of hand gestures in real time is important. The requirement of the software being able to have a minimal client-side application that sends data to and receives data from a server with the most computationally complex modules could allow for this software to be deployed as a mobile application that interacts with the gesture interpretation over the internet. The requirement of a robust application programming interface could allow the software to be integrated into a variety of other applications. Ultimately, this software is intended to fulfill user's and developer's desire for a robust, portable, and modular application that can detect and utilize hand gestures in real time.

1.4 Major constraints

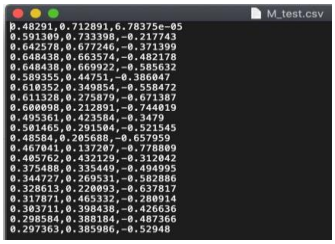
This software project has several major constraints. The largest of these constraints is the accuracy and timing of the gesture interpretation. The gesture interpretation process must be done in real time (real time in this case means an action performed between 50 and 500 milliseconds). It must also be able to interpret the correct gesture with 90% accuracy. This means that the software must be able to account for when the live camera feed has conditions of poor lighting or low video quality, and still give an accurate output and responsive output. It must also account for a detected hand being within the maximal distance of a room. The second largest of these constraints is the time for development, which is approximately 4 months. The accuracy of the gesture interpretation module may not be able to be refined as much as desired within this time. Therefore, we included the grammar module as a potential functionality integrated with the gesture interpretating module, to help correct for minor misinterpretations from the gesture interpreting module. Considering the entire set of American Sign Language gestures, we are interpreting with our system using a trained machine learning model, that set containing letters A through Z and 1 through 9, our system will also be limited by the number and quality of images we are able to find to train this model, as this will partially determine the accuracy of our model. The systems that we run our application on, both client side and server side, must have enough computing power to translate the live camera feed it receives into recognizable data for our machine learning model and run our machine learning model on the input data.

2.0 Data design

2.1 Internal Data structure for Formal Grammar Component

To interpret gestures from an image, we used machine learning to train a model based on a dataset made up of thousands of pictures of sign language signs. We transformed this dataset from a dataset of pictures to a dataset of CSV files. Each image became a corresponding CSV file containing MediaPipe's output of the x, y, and z coordinates of each 21 detected landmark on the hand in the image.

becomes



The formal grammar module will require a dictionary of predetermined commands that can be used in comparison against inputs from the gesture interpreter. This dictionary will be a list of phrases as strings that are commonly used with smart homes and similar devices.

2.2 Global data structure

NOT APPLICABLE

2.3 Temporary data structure

2.3.1 Temporary Data Structures for Network Component

- Input Data Structure - Numpy ndarray (two dimensional)
- UDP packet - UTF-8 Encoded Byte string
- Component output - JPEG image
- Frame Buffer - Binary Heap

UDP packet data structure table for custom protocol

Packet Section	Frame Sequence Number	Frame Size (INT)	Slice Sequence Number (INT)	Slice Size	Data
Data Type (before string conversion)	INT	INT	INT	INT	Numpy ndarray
Size	10 Bytes	10 Bytes	10 Bytes	10 Bytes	Slice Size Bytes

Value Range	0…*	Constant 86,400	0…19	Constant FrameSize//20	<7,200 bytes

2.3.2 Temporary Data Structures for Modified MediaPipe Component

MediaPipe outputs 21 landmarks from the hand relative to a bounding box around the detected hand. This output is sent to a file or over a TCP socket connection. It is structured like a CSV file, each landmarks x, y, and z coordinates are on one line separated by commas.

2.3.3 Temporary Data Structures for Formal Grammar Component

For the formal grammar module, the main data that will be sent to it as an input will be a string containing a sequence of characters from the gesture interpretation module. This output will be sent to the formal grammar module as a part of a larger collective Python application that runs the formal grammar module and the gesture interpretation module. Once the formal grammar module receives this input, the string will run as it is received through the formal grammar interpreter, which will work similarly to a search autocomplete for suggestion function. The autocomplete function can have two possible data outputs. If the function can determine a valid command from the list of possible commands, we provide based on the input string, it will output the command phrase to the command output module, and subsequently this string will be sent to a smart home device via a TCP connection over a network. If the autocomplete function is unable to determine a valid command from the input string, it will log an error to the console as an output.

2.4 Database description

NOT APPLICABLE

3.0 Architectural and component-level design

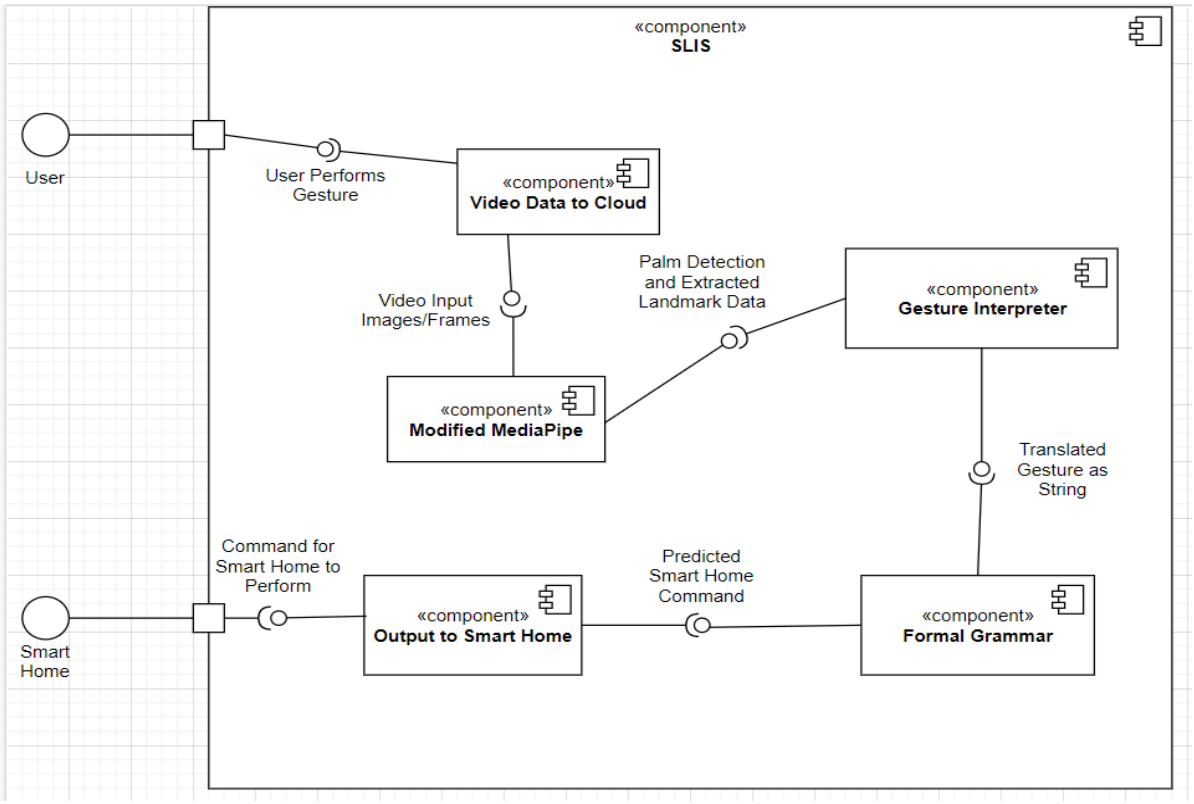
Considering the linear nature of our software, the architecture that we have based our development on is the layered architecture. The method of input for the software is not unique every time, other than the fact that the sign language itself may be different. The input must travel through different systems/modules to produce the output needed. Each layer is isolated from itself in terms of development, making changes and modifications to one layer will not impact the system, granted those modifications work properly. The layers will constantly communicate with each other, starting from the beginning when the user makes a gesture in front of a camera, which the camera will send captured video to our server receiver layer. Each image/frame will then be written to a file and sent to be further analyzed by the gesture interpreter layer. From here, the gesture interpreter will use the landmark

data from each frame to interpret the gesture being made, sending the result as a string to the formal grammar layer. The formal grammar layer will take that result and predict what command is being gestured referencing a dataset of common smart home commands, which will then send that command to our smart home environment. The layers in the system are closed which means that the process of one layer must be completed before moving to the next. Each layer will process the input differently using its own logic unique from other layers; however, the input and output of the layers will always be the same to properly communicate with the separate layers.

3.1 System Structure

The system structure used for the SLIS is a modular structure. From the design goals we must achieve, the modular approach is the best applicable method to use. The modules that we have include our video capture module, MediaPipe and custom modules to interface with it, gesture interpretation module, and our formal grammar module which will interface with the smart home environment. We have utilized Python for a large chunk of our software to create modules such as our gesture interpretation module and formal grammar module. Each module can be broken down into more specific and essential functionalities that need to interact together to create our finished product. Our video capture and MediaPipe modules work very closely together, communicating via TCP to send frames to MediaPipe where from there, MediaPipe analyzes the image to detect the palm, and extract the landmarks to begin interpretation. The gesture interpretation module then uses a TensorFlow machine learning model to interpret the gestures received by MediaPipe, outputting a string of the interpretation. That output is sent to out formal grammar module which will then use a library of common smart home commands to predict what command the interpretation best fits. This command is then sent to our smart home environment and executed properly per command. The interchangeability and customizable aspects of these modules is what encapsulates the modular structure. Each module is essential in executing one key function to then communicate down the chain of modules to accomplish our main goal.

3.1.1 Architecture diagram



3.2 Description of Client to MediaPipe-Server Component

The client to MediaPipe component's function is to deliver valid data to the MediaPipe program. The SLIS hand tracking MediaPipe instance is modified to receive still JPEG images.

3.2.1 Processing Narrative for Client to MediaPipe-Server Component

This component should facilitate the creation of appropriate sockets for the various inter-process communications. It will implement a protocol for sending video frames to a server and receiving them on the server side. The component will also be responsible for preparing and presenting video frames to MediaPipe.

3.2.2 Interface Description for Client to MediaPipe-Server Component

The local client uses OpenCv to obtain video frames from an integrated video camera. OpenCV stores video frames as two-dimension numpy arrays known as ndArrays. These ndArrays are the root input data format for the entire SLIS system. The Network to MediaPipe server component sends these arrays to the server and encodes them into JPEG images/ These JPEG images are this component's final data structures that it outputs on a TCP socket.

3.2.3 Processing Detail for Client to MediaPipe-Server Component

The component is a client-server architecture. The client's function is to receive input from the user. It does this via a live integrated video camera. The client then utilizes User Datagram Protocol to send the video feed to a remote server. The server receives the data packets from the client and extracts individual frames from the video feed that are ultimately sent to a local port where they can be picked up by media pipe. The client-server architecture sends and receives the data packets according to a custom application layer protocol that is described in section 3.2.3.3.

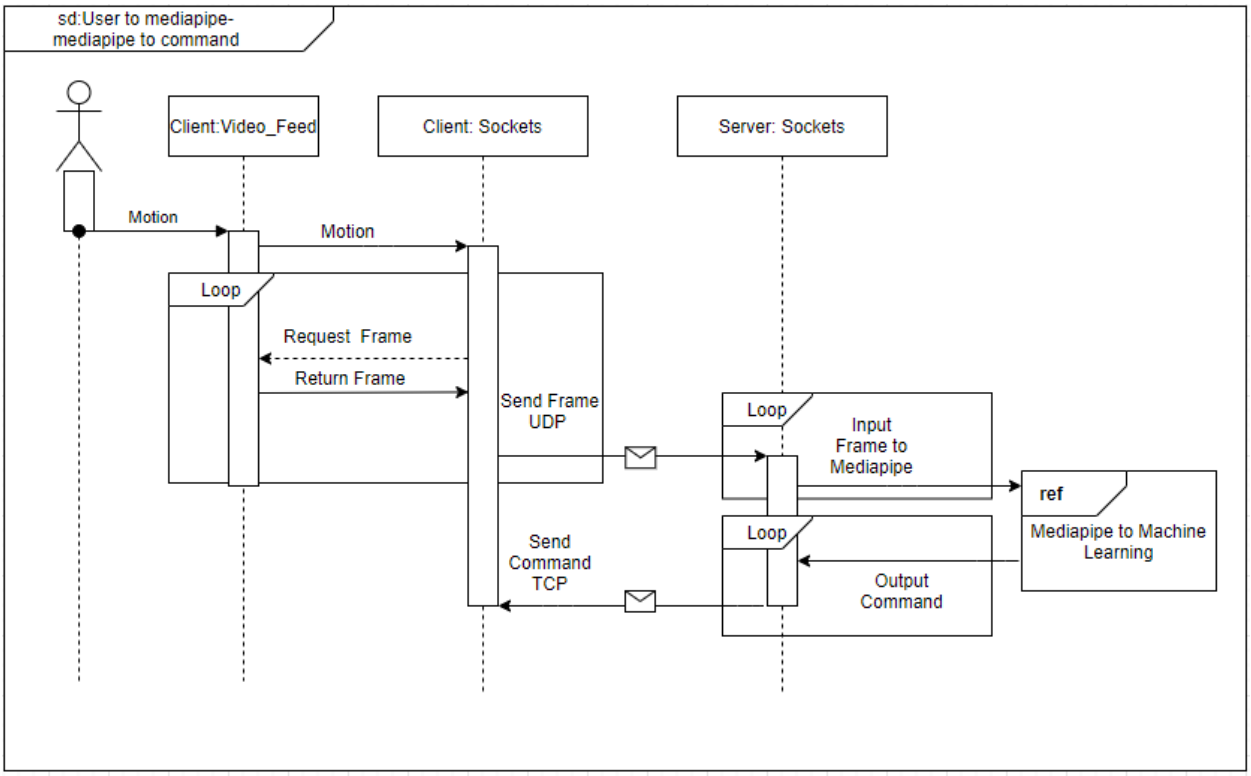
3.2.3.1 Restrictions/Limitations for Client to MediaPipe-Server Component

- The video frames must be of size/resolution 480x60x3. This is OpenCV's default resolution.
- Each frame should be sliced a minimum of 20 times before being sent individually over the UDP stream

3.2.3.2 Performance issues for Client to MediaPipe-Server Component

The protocol handles dropped packets with a timeout on each frame, if the frame is not completely reconstructed after n milliseconds, the receiver moves on to the next frame. The timeout time n is a constant variable that can be changed to account for lag. However, the shorter the timeout variable the more likely the frame is to be skipped before receiving a slightly delayed packet. The shorter the timeout variable the more likely the server is to miss frames resulting in a more glitchy video feed. There is an inverse relationship between lag and glitch. The timeout variable should be adjusted for optimization during setup to find optimal lag to glitch ratio on the system hosting the component.

3.2.3.3 Algorithmic model for each operation



The video is captured from the integrated video camera as individual frames by Python-OpenCV and each frame is stored during runtime as a two-dimensional numpy ndArray of size 86,400 bytes (480x60 pixels x3 RGB values). Each sub-array represents a sequential pixel in the frame and consists of exactly three floating point numbers that represent the red, green, and blue values of that pixel.

A custom application layer protocol was written to transmit videoframes to a server. Each frame is split into exactly twenty slices to be sent over User Datagram Protocol to the listening server. Each slice is converted into a character string and prepended with a 40-byte header with four separate ten-byte sections that represent the current frame sequence, the frame size, the current slice sequence, and the slice size. The string is then encoded into a UTF-8 byte string that can be sent through a UDP socket. See Table in section 2.3.1 for a table describing the data packet.

The server program *udp_recieve.py* retrieves the data packets from the specified ports and reassembles the slices into their original sequences before decoding them back into an ndArray and pushing them onto a heap that is used as a buffer for the incoming video frames.

A separate function in *udp_recieve.py* then grabs frames from the reconstructed video feed, encodes them into JPEG images and sends them over a TCP socket that serves as an inter-process communication with media pipe.

3.3 Description for Modified MediaPipe Component

MediaPipe is a framework for building machine learning pipelines. A MediaPipe application is made up of graphs and calculators. Calculators are the processing parts of the application. They have defined inputs and outputs and perform work each frame or at specified intervals. Graphs define how calculators are connected to one another. This creates an organized structure for your machine learning pipeline. MediaPipe comes with a hand tracking project that detects a hand and tracks its movement using 21 defined landmarks on the detected hand. This is very useful for hand gesture recognition, and we modified this hand tracking example application for our use in two different cases. First case, to create a custom dataset of CSV files containing hand landmarks for different hand gestures. This MediaPipe build will be referred to as MediaPipe Trainer. The second case, to run as a module in our main application and determine the landmarks of a video feed in real time, will be referred to as MediaPipe Run.

We customized MediaPipe's hand tracking program to perform two different methods of input. The hand tracking program normally captures frames from your device's webcam in order to perform the landmark detection. For our use case, we need to provide two custom types of input: a single image that can be defined by a command line argument and a JPEG image stream sent over TCP. To provide a single image, we modified MediaPipe to take a command line argument for the image file path and use OpenCV to read the image. After reading the image and performing landmark detection one time, the program outputs the detected landmarks to a CSV file and terminates. This first case is done for the MediaPipe Trainer build. Providing a constant JPEG image stream over TCP was slightly more complicated. A separate thread listens for a TCP connection to receive data. This thread saves that data to a buffer. This buffer is then accessed by MediaPipe's main process loop (which is normally called on every frame from the webcam feed). This allows MediaPipe to read the continuous stream of JPEG images to the TCP port as if it was reading straight from a web cam. This continuous input is done for the MediaPipe Run build.

We also customized media pipe to perform two different methods of output. We need to output the coordinates of the 21 points detected from the hand in the image. This was done by modifying the graph of the program and inserting a custom calculator. This calculator receives the 21 detected landmarks as input and outputs them to either a CSV file or TCP connection. The MediaPipe Trainer build outputs to a CSV file, while the MediaPipe Run build outputs over TCP.

We now have two custom MediaPipe builds. The MediaPipe Trainer build is used to convert any dataset of images to a dataset of CSV files containing the 21 detected landmarks from each image. After conversion, a TensorFlow model can be trained on the new dataset. The MediaPipe Run build receives a constant TCP stream of JPEG images and constantly outputs the 21 detected points over TCP. This is done so another process can receive these 21 points and run it through the trained TensorFlow model to classify a gesture.

3.3.1 Processing narrative for Modified MediaPipe Component

The Modified MediaPipe component has two uses. The MediaPipe Trainer build is used to convert any dataset of images of hand gestures to a dataset of 21 landmarks. This is useful for training a model for classifying gestures in real time. The MediaPipe Run build is used in the final application for continuously converting a stream of images to their detected landmarks and outputting these 21 landmarks to the Gesture Interpreter Component.

3.3.2 Modified MediaPipe Component interface description

There are two interfaces or ways to interact with the custom MediaPipe builds. The MediaPipe Trainer build can run from the command line and passed command line arguments to specify input image file path and output CSV file path. This way has no visible UI. The MediaPipe Run build when ran shows a screen with the current frame being processed and 21 landmarks on the detected hand. This may be disabled after debugging is done if not needed.

3.3.3 Modified MediaPipe Component processing detail

3.3.3.1 Design Class hierarchy for Modified MediaPipe Component

NOT APPLICABLE

3.3.3.2 Restrictions/limitations for Modified MediaPipe Component

MediaPipe has support for many different platforms but some platforms full support is not guaranteed. We tried running MediaPipe on Windows, Mac, and Linux. We successfully built MediaPipe on Linux but had trouble with the other operating system. This leads a limitation of only being able to run this component on a Linux machine.

Detecting hands and landmarks is not a perfect process. In our testing, lighting conditions can affect whether MediaPipe will successfully detect the landmarks. Though it does work well, MediaPipe will need decent lighting to detect landmarks.

3.3.3.3 Performance issues for Modified MediaPipe Component

MediaPipe is very processing intensive. The component will run slow if the machine it is ran on cannot handle the power it requires. MediaPipe also requires a GPU to be able to run with the speed it needs. With this in mind, we slowed down the frame rate input to MediaPipe.

3.3.3.4 Design constraints for Modified MediaPipe Component

MediaPipe component must be run on Linux Device.

MediaPipe component requires GPU to run and must process at a framerate that allows the machine to continue to run smoothly.

3.3.3.5 Processing detail for each operation of Modified MediaPipe Component

3.3.3.5.1 Processing narrative for each operation

Input:

The MediaPipe Run build receives a JPEG image stream as input to perform landmark detection. This requires a TCP thread that is always listening for a new frame. This thread opens a TCP Socket and listens. First set of data it receives, it saves the number of bytes but does not save any data. This allows it to detect the size of the images it will be receiving. The second time it starts receiving data, it prepares a buffer of that length and fills it with the incoming data. It also uses a mutex to block the buffer while writing. The MediaPipe process loop needs to read from this thread every time it needs a new frame. It blocks the mutex, reads the buffer, and unblocks. It then uses OpenCV to decode the image and continues to landmark detection as if it read the frame from the webcam.

The MediaPipe Training build is given a command line argument to specify an image file path. It reads this image once and continues to landmark detection.

Hand Tracking or Landmark Detection:

Landmark detection is done by MediaPipe itself, we only modified it to output the data in a format that is useful to us. Landmark detection is done by detecting a hand in the frame, focusing on a zone around the hand called the “bounding box”, and detecting coordinates on 21 defined points on the hand relative to the bounding box. It is not necessary to understand how MediaPipe performs landmark detection to build this component.

Output:

After every landmark detection, the MediaPipe Run build outputs the 21 detected points over a TCP socket connection to be picked up buy another process for classification. The MediaPipe Training build outputs the detected landmarks to a file path that can be specified in the command line arguments.

3.3.3.5.2 Algorithmic model for each operation

MediaPipe Training Build:

Input:

MediaPipe Training build’s input is very simple. It takes command line arguments to specify and image path and an output location.

Output:

MediaPipe Training build is simply modified to loop through the landmarks after detection and write each x, y, and z coordinates to a CSV file.

MediaPipe Run Build:

Input:

MediaPipe has two main threads we need to focus on for customization. Input Image thread and the Process Update Loop thread.

Input Image Thread:

The input image thread is started immediately. It opens a TCP port for receiving JPEG encoded image data that it will eventually save to a buffer. To create this char buffer, we first need to know the size of the image we are receiving. This is automatically calculated the first time any data is sent. The first image frame that is sent to the socket is used to calculate the size of the buffer and no data is stored. It continually loops until no more data is supplied keeping track of the total amount of bytes received. After it is done receiving data, it knows how large the images will be, prepares a buffer of this size, and sets a flag so it knows it is ready to begin saving data to the buffer.

After the first image is scrapped and used to calculate the imageSize, it begins saving the data on the succeeding connections. It creates a temporary buffer to store the image data in and continuously loops until it receives no more data. Once it is done receiving data, it locks an imageBuffer mutex, copies the memory from the temporary buffer to the global imageBuffer, and unlocks the mutex. This allows the mutex to stay unlocked while it receives data, and only lock down for a short span of time once all data has been received.

Process Update Loop Thread:

The process update loop is run every time MediaPipe expects a new image frame to perform detection on. Before customization, MediaPipe pulled an image frame from the device's webcam. Instead, the custom MediaPipe gets its image frame from the imageBuffer that was filled with the TCP Input Image Thread. It first locks the imageBuffer mutex, copies the imageBuffer over to a local buffer, and then unlocks the mutex. Next, it uses OpenCV to decode the JPEG image data. MediaPipe then continues its normal processes.

Output:

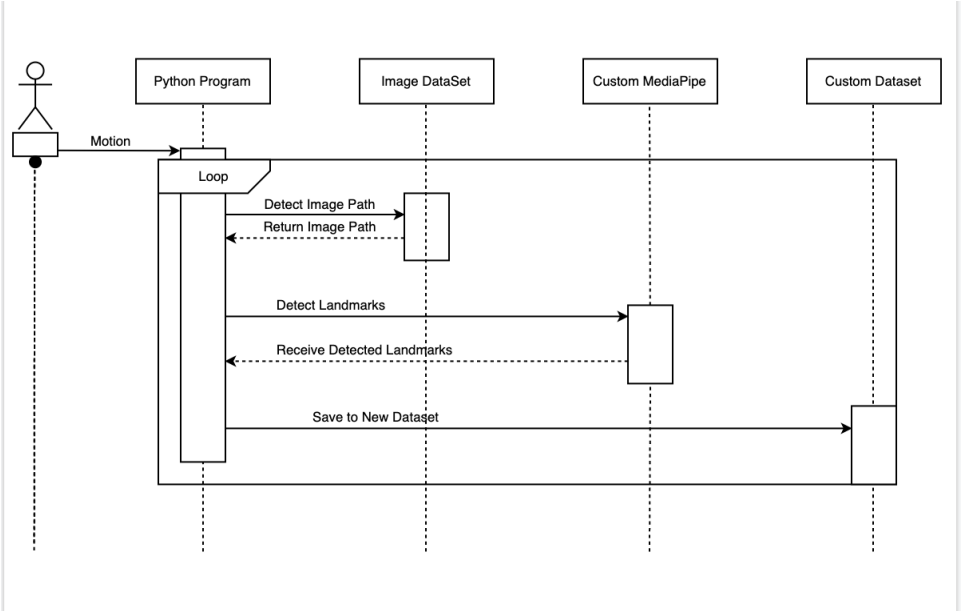
Output for MediaPipe Run build is very similar to the training build. After the landmarks are detected, MediaPipe is modified to loop through the detected landmarks and create a String in the format of a CSV file. Each line is an x, y, and z coordinate of a detected landmark. Finally, this string is output over a TCP socket.

3.3.4 Dynamic Behavior for Modified MediaPipe Component

NOT APPLICABLE

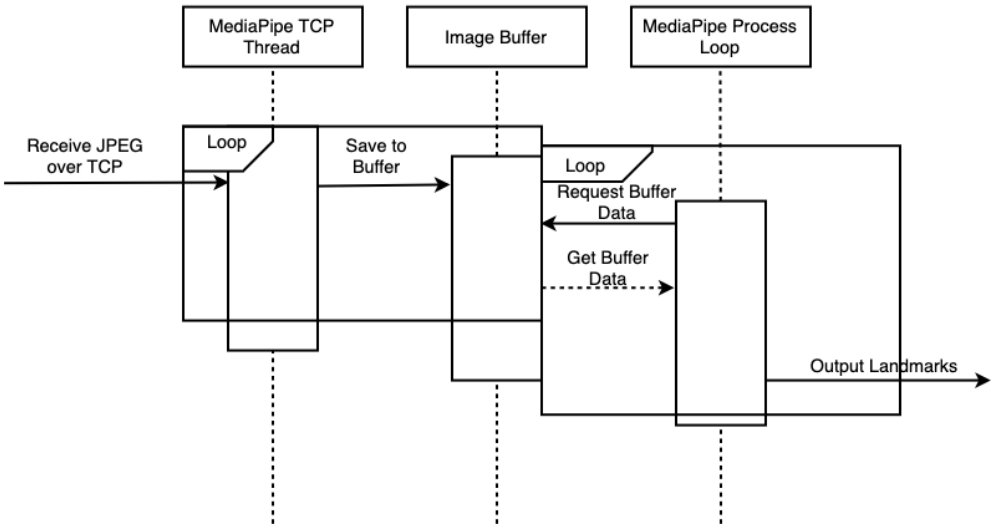
3.3.5 Interaction Diagrams

3.3.5.1 MediaPipe Training Build Interaction Diagram



This sequence diagram shows the MediaPipe Training Build that takes an input of an image path, is ran once, and outputs 21 points to the file path it also received as an input command line argument. This Custom MediaPipe is used to convert a dataset of images to a corresponding dataset of CSV files containing the detected 21 landmark points. This is done by writing a simple python program that runs the Custom MediaPipe binary for every image in the image dataset.

3.3.5.1 MediaPipe Run Build Interaction Diagram



This sequence diagram shows the MediaPipe Run build. It receives a JPEG encoded image over TCP and saves it to a buffer in one thread. Another thread reads that buffer and outputs the landmarks.

3.4 Description for Gesture Interpreter Component

The gesture interpreter is a component in our system that determines what character is being signed based on the landmarks gathered from the modified MediaPipe component.

3.4.1 Processing narrative for Gesture Interpreter Component

The gesture interpreter is responsible for making use of landmark data. This component bridges the gap between a vector of floating pointing numbers (landmarks) and English letters that a person can comprehend. The process behind the translation is a trained neural network which access the landmarks as a vector of floating-point numbers and outputs the confidence of the predicted letter.

3.4.2 Gesture Interpreter Component interface description.

The gesture interpreter takes an input of twenty-one landmarks on the hand and outputs the predicted character as text. A dense feedforward neural network is used to make this operation possible. The neural network accepts a flattened vector of sixty-three floating point numbers derived from the twenty-one landmarks - each landmark containing x, y, and z coordinates - as an input. The network then outputs from twenty-nine nodes - representing the twenty-six letters of the alphabet, space, delete, or nothing - where each node returns a floating-point number that reflects the networks confidence in that character being the target. The node with the most confidence will decide the classification.

3.4.3 Gesture Interpreter Component processing detail

3.4.3.1 Design Class hierarchy for Gesture Interpreter Component

NOT APPLICABLE

3.4.3.2 Restrictions/limitations for Gesture Interpreter Component

The gesture interpreter is straightforward and has few restrictions. One restriction is the input must be in the correct format. For our project the input is a vector of sixty-three floating point numbers where the first twenty-one numbers are the x-coordinates, the next twenty-one numbers are the y-coordinates, and the last twenty-one numbers are the z-coordinates of the landmarks from the hand. Another restriction is the length of time the neural network takes to be trained. Based on our experience it takes approximately an hour to train this neural network.

3.4.3.3 Performance issues for Gesture Interpreter Component

The gesture interpreter does not always accurately predict the correct letter. To solve this problem the formal grammar component will deal with incorrect predictions.

3.4.3.4 Design constraints for Gesture Interpreter Component

The gesture interpreter must be a tensor flow model.
The gesture interpreter must have sixty-three input nodes.
The gesture interpreter must have twenty-nine output nodes.

3.4.3.5 Processing detail for each operation of Gesture Interpreter Component

3.4.3.5.1 Processing narrative for each operation

The gesture interpreter is a feedforward dense neural network containing one input layer, two hidden layers, and one output layer. The two hidden layers each contain twenty-nine nodes. With each node of the hidden layers there are a vector of weights, which are fine tuned in the training process, that effectively separate the training data into different classes using hyperplanes. Once separated a rectified linear unit (ReLU) activation function is applied to the data making the output one of two numbers representing a true or false expression. After taking all of this into account the network outputs twenty-nine nodes floating-point numbers that reflect the networks confidence in that character being the target. The node with the most confidence will decide the classification.

The gesture interpreter neural network must be trained. Our model was trained with more than 50,000 samples of landmarks from high-resolution images.

The training process takes around an hour and will likely take more than one try to get the correct fit.

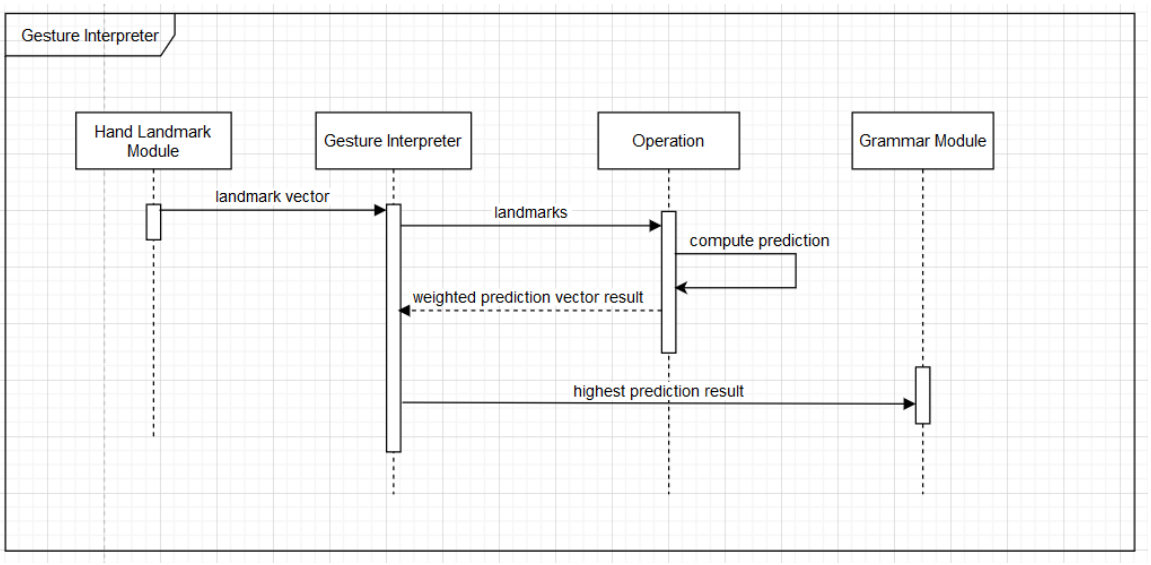
3.4.3.5.2 Algorithmic Model for each operation

The gesture interpreter neural network classifies samples using a set of weights, biases, and activation functions. During the training process the randomly initialized weights and biases will be tuned by the results from each pass of the training samples through the network. To create an accurate gesture interpreter a neural network must be trained with a lot of data. Our model was trained with more than 50,000 samples. The samples came from an online library of high-resolution images of American Sign Language letters. First, the landmarks must be extracted from each image by passing the images through MediaPipe and writing the coordinates to a comma separated value (csv) file. Once in a csv file the landmarks can then be used as training samples. It is important to have a large amount of training and testing samples so the neural network can be properly trained and tested. TensorFlow has tools that make it easy to train a neural network, but a hyperparameter that must be specified is the number of epochs to train the network on. For our model we used two hundred as our number of epochs. This number can vary widely, and it is recommended to try different numbers via trial and error as to not overfit the model.

3.4.4 Dynamic Behavior for Gesture Interpreter Component

First, the gesture interpreter communicates with MediaPipe to get the input that is needed to perform its operations. The input that it receives is a vector of sixty-three floating point numbers that depict the landmark data of the hand. This input is then processed through our neural network to accurately interpret the gesture into its English meaning. Next, the interpreter will communicate with the formal grammar module in the sense that it will send the interpreted gesture as a string variable, where it will be further processed into a command. The gesture interpreter will only interact with the two layers stated above.

3.4.5 Interaction Diagrams for Gesture Interpreter component



3.5 Description for Formal Grammar Component

The formal grammar module is a component in our system that receives an input string of English characters that are interpreted by the gesture interpreter from American Sign Language hand gestures, and outputs a valid command phrase based on this input string, if possible.

3.5.1 Processing Narrative for Formal Grammar Component

The formal grammar component is a smaller component that exists as a part of the same python application process that runs the gesture interpreter component. The formal grammar module exists to add additional accuracy to our gesture interpreter, which may have some difficulties interpreting hand gestures, thus resulting in an incorrect guess. The formal grammar module will first receive an input string from the gesture interpreter after the gesture interpreter has completed running the set of hand landmark points through the model for an individual sequence of hand gestures within a specific time frame. The gesture interpreter will gather the set of interpreted gestures as English characters and give them to the formal grammar component. This component will perform an autocomplete and autosuggest function, like a simple edit distance calculation, on the input string, and attempt to find the most similar command phrase that this input string is trying to express with possible errors, based on a pre-gathered library of possible commands based on common smart home input commands. If the autocomplete and autosuggest functions are unable to find any command phrases that are within a certain distance from the input string after analysis, then it will let the rest of the entire system know by outputting an error to the console. Otherwise, the system will send the command

the input string most likely represents to the command output component client over a TCP network connection, which will then output the command to a smart home or similar device.

3.5.2 Formal Grammar Component Interface Description.

The formal grammar component will have three interfaces. First, it will interface with the gesture interpreter component as a part of the same python application. The formal grammar component will receive an input of a string of English alphanumeric characters composed of letters A through Z and numbers 1 through 9. This will be passed via a function call to the formal grammar component from within the application. The formal grammar component will output in two ways. First, it will interface with the command output client, sending valid commands that have been interpreted from the input string from the gesture interpreter. This output will also be a string of English alphanumeric characters composed of letters A through Z and numbers 1 through 9, as well as spaces between words. The formal grammar component can also output an error to the application console if it is unable to determine a command phrase that is close enough to the input string from the gesture interpreter.

3.5.3 Formal Grammar Component Processing Detail

3.5.3.1 Design Class hierarchy for Formal Grammar Component

NOT APPLICABLE

3.5.3.2 Restrictions/limitations for Formal Grammar Component

The grammar module will be receiving an input of strings, therefore the strings that it receives should, in a perfect world, be spelt correctly. However, in the case the strings are not, the module will be predicting what command it will need to send. There could be some limitations as to how accurate it would be with a very poorly spelt sequence, or how long it would take to predict a command with say more than 20 words. Numbers will be assumed to be used in associating with a specific noun, such as ‘Light 1, Light 2’, etc. The positioning of numbers could in fact throw off the prediction capabilities of the module. For characters/strings that are unrecognizable to the module, an ‘invalid command’ event will be triggered, skipping over to the next input sequence of strings.

3.5.3.3 Performance issues for Formal Grammar Component

The formal grammar component should be able to perform its calculation within a few milliseconds of receiving the input string from the gesture interpreter component. This will still add time to the entirety of the application’s processing time

to output commands, but it should not largely affect how close the program is able to function near real time.

3.5.3.4 Design constraints for Formal Grammar Component

The biggest constraint with this component is being able to run at a speed that complies with the global constraint of the application needing to run its process in real time. This component must also be able to interface with the gesture interpreter component as well as the MediaPipe to smart home component that connects to the smart home or similar device.

3.5.3.5 Processing detail for each operation of Formal Grammar Component

3.5.3.5.1 Processing narrative for each operation

The formal grammar module will, in its entirety, be a function call to a separate part of a running python application. The gesture interpreter component will, after gathering all the interpreted gestures within a specific amount of time, will output a string of alphanumeric characters to this formal grammar component function. Within this function, the formal grammar component will take the input string and compare it against a predetermined library of valid command phrases based on common functionalities used in smart homes or similar devices. The function will find the edit distance between the input string and each command phrase. The command output will take the top option if it is close within a certain value to the input string. If there is no command from the predetermined list that is close enough to the input string, then the application will output an error message to the application console, and continue functioning, waiting for more input strings from the gesture interpreter.

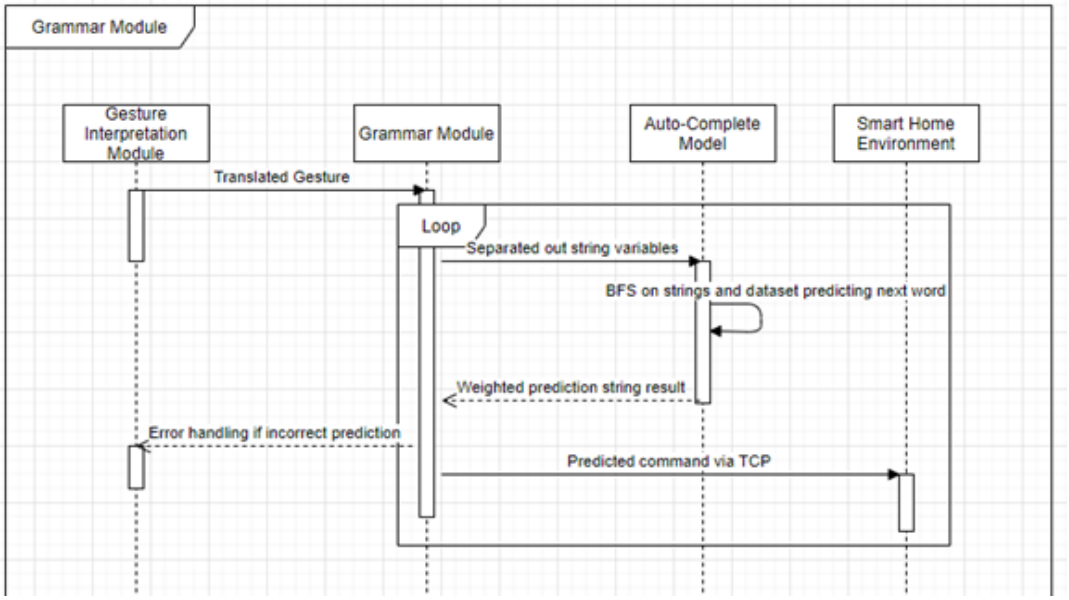
3.5.3.5.2 Algorithmic model for each operation

Commands will be received from the gesture interpretation model via a string variable. The grammar module will then take the string and parse through it, separating the full sequence of strings into separate strings for each word. Before we can go further, we will need a dataset of many different commands, some of the same commands but written in different ways. Then, we will need to parse through each command, giving every word a specific weight, words that are more common in the dataset will be given a higher weight and vice versa. The idea is to then take the command from the gesture interpretation module and perform a BFS over the words, using the weighted results we got from the dataset to predict what word comes next in the command. If the command has been predicted correctly, the module will prepare to send the command over to a smart home device via a TCP connection. However, if the module is unable to predict the command with an existing command in the dataset, then an error will occur stating that the command is invalid. The module will then move on to the next command given from the gesture interpretation module. With a successful working grammar module, we will be able to communicate correct commands to the smart home device to be performed.

3.5.4 Dynamic Behavior for Formal Grammar Component

The formal grammar module will be interacting primarily with the gesture interpretation module and the smart home environment. Like previously stated, the grammar module works solely off the string output from the gesture interpretation module. The string variable will need to be sent directly to the grammar module for it to be processed by the auto-complete functionality of the module. There will be a component of the module that will log any errors or invalid commands to be utilized with the gesture interpretation system. This communication is vital in the event of a misinterpreted command, alerting the user that there was an error in trying to process their gestures and to try again. The grammar module will also need to have a stable connection to the smart home environment in order to successfully send over the string output via TCP, for the smart home to interpret much like any other input for a command.

3.5.5 Interaction Diagrams for Formal Grammar component



3.6 Description of MediaPipe to Smart Home

The MediaPipe to smart home component's function is to deliver an interpreted command back to the originating client so that the client can call a smart home API.

3.6.1 Processing narrative

This component should facilitate the creation of TCP sockets on the server and client ends so that commands can be sent over the network after they have been

interpreted by the gesture interpretation system. The commands will then be mapped to smart home API functions.

3.6.2 MediaPipe to Smart Home interface description.

The input for the component will be a string that is normalized to match a list of accepted command strings. The output will be a correlated smart home action such as a light switching on or off or a song being played.

3.6.3 MediaPipe to Smart Home Component Processing Detail

3.6.3.1 Design Class hierarchy for MediaPipe to Smart Home Component

NOT APPLICABLE

3.6.3.2 Restrictions/limitations for MediaPipe to Smart Home Component

The commands capable of being executed in a smart home environment will be constrained to the smart home API used and the functions it provides. Only commands able to be interpreted by the gesture interpretation system will be able to be sent to the smart home API

3.6.3.3 Performance issues for MediaPipe to Smart Home Component

The API call rate will be limited to the speed of the interpretation system and the API itself. The local API calls will only be able to execute one at a time.

3.6.3.4 Algorithmic model for each operation

The formal grammar module will present an interpreted string command to a TCP socket program on the server. The socket program will convert the string to a UTF-8 byte string and sent through TCP socket to the originating client machine. The Client machine will decode the byte string. The resulting string will be mapped to an API function that triggers the smart home action.

4.0 User interface design

The user interface modules for this system include the following: a video stream output console that shows the video stream that is being used as an input to our MediaPipe instance with the hand landmark tracking overlay, a video stream output console for the video streaming client, and a text console log that shows errors and subprocesses for the video streaming component, the gesture interpreting component, the formal grammar component, and the command output component. The set of user interfaces that exist for this system are meant to be minimal and more

intended for developer use for debugging and application diagnostics rather than for an end user.

4.1 Description of the user interface

4.1.1 Screen images

The screen images for this system include a video stream console from the video streaming client and a video stream output console that shows the video stream that is being used as an input to our MediaPipe instance with the hand landmark tracking overlay. These will both display in a separate window for the application, and they will display the frames of the video stream as they send and receive them respectively.

4.1.2 Objects and actions

The screen objects that exist for this system are the windows that display the text and video console logs for each of the components as described previously. There are no action objects that exist for this system for any users.

4.2 Interface design rules

The SLIS has a very minimal user interface. The primary input device is an integrated video camera. The video display may be turned on or off in a configurations file and is a provided option for assistance in verifying the system is working during setup.

The camera feed process should always be active on the client host machine so that it may be triggered via designated gesture to start receiving command input from the user. This will act as a visual alternative to the standard activation of voice command smart home devices.

4.3 Components available

- Primary user interface: video camera gesture input
- Configuration Interface: Any text editor may be used to edit configuration files
- Setup and verification interface: Video output is provided to verify that the camera is working
- Error logging and Command Line input/output: This can be used during setup and trouble shooting. Error messages will be sent to a text-based interface and various testing, building, and runtime configuration options may be set from the command line.

5.0 Restrictions, limitations, and constraints

One of the biggest limitations that we had was getting a working environment to build, run, and test MediaPipe on. MediaPipe performs best on a Linux environment, it is still experimental with Windows environments, however there are work arounds with using Docker to build a MediaPipe image. That also proved to be troublesome, we found our best success was to get an Ubuntu environment running via a hard drive partition and to build MediaPipe through that. The system must also have a functioning camera that can capture high resolution video feed. The video frames need to be of size/resolution of 480x60x3 since that is the default resolution for OpenCV to function properly. To send the frames over via UDP stream, the frames must be sliced to a minimum of 20 times prior to being sent.

For translating the sign language, we utilized a Tensor Flow model. This is the best approach to achieve translation in a real time format. However, in order to train the model, we needed to use a separate Python file apart from the entire system in order to keep training the model out of the initial function of the interpreter. The training time is approximately one hour and took multiple tries to fit the model correctly. In order to achieve optimal results from the neural network a large amount of training data must be used. In our case we used over 50,000 samples of landmark data. When trained correctly this neural network will perform at adequate speeds to reach the fast response time we are looking for. The speed this system must work at is real time (500 milliseconds).

6.0 Testing Issues

Once we have a functional product, we will need to conduct precise testing to ensure that our software meets the design requirements. More specifically, these tests need to not only satisfy our requirements, but to root out any bugs that could potentially harm the performance of our software. First, testing on the general function of being able to successfully translate gestures that the user creates into commands used for a smart home. However, we can take this to a more extensive level, using multiple different types of gestures to test the accuracy of our interpretations. We will test gestures of varying length, to ensure that our software can perform the translation and interpretation in real time. Not only will tests like this show if our software is functioning properly, but it will also give us an idea of how we can further improve it. We will also need to test building our project on a fresh machine to ensure that our software is portable. The main aspects that we will focus on during testing will be portability, accuracy, and efficiency. These aspects are the focus points of our project, and what our design goals are centered around.

6.1 Performance bounds

Some components require large amounts of processing power. A computer with at least 16 GB of ram is recommended. The MediaPipe component must be run on a system with a GPU.

6.2 Identification of critical components

Gesture Interpreter Component

The Gesture Interpreter Component needs a machine learned model that is trained to the highest degree of accuracy. This model needs to be trained and tested until it reaches the highest accuracy and precision.

Formal Grammar Component

Formal grammar component needs to be tested and optimized extensively to assure a high accuracy. Character strings that are close to a command but not exactly a command need to be interpreted as the command.