Simplifying Sign Language Detection for Smart Home Devices using Google MediaPipe

Braden Bagby, David Gray, Riley Hughes, Zachary Langford, and Robert Stonner

***Abstract*—Smart home technology has become more and more ubiquitous in homes. In recent years, the demand for different smart home and digital personal assistant technology has increased dramatically. The technology for performing a variety of different tasks has become quite advanced, from simple tasks such as web searches, to more complex tasks such as recording notes or turning on lights. Over time, research in various fields of computer science have enhanced the capabilities of smart homes with refined convolutional machine learning models that constantly analyze sound input for activation phrases and context dependent correction of detected words and phrases in commands. However, one type of smart device input technology that is often neglected is input via hand gesture, specifically sign language, and using that to spell out command phrases. The lack of support for hand gestures as an input for smart devices excludes some communities of people from using these devices.**

In this paper, we are proposing a unique approach to the various problems that are faced when capturing hand gestures in American Sign Language (ASL) for use as an input to a smart device system. Our project takes advantage of Google's rapidly developing MediaPipe hand-tracking technology to simplify the machine learning models that are detecting the ASL hand gestures and translating them into English alphabetic characters. Our prototype application also allows for a client-server separation of the more resource intensive parts of the application and the camera input and command output technology. We will take a look at some current research done into hand gesture detection technology, the Python framework we have built surrounding MediaPipe and MediaPipe itself, how we trained the machine learning model for hand gesture detection and its accuracy based on a few metrics, and finally some thoughts on future work that could be done based on the results of this project.

Keywords—machine learning, sign language, MediaPipe, smart home, smart device, hand gesture, IoT

I. Introduction

In recent years, computers have become more and more essential to our everyday lives and the functions that we perform daily. Our society is increasingly reliant on computers, and it is important to have the most basics functions of those computers as readily available as possible. For example, turning lights on and off, locking doors, calling a friend, and searching the internet are all essential functions which are now commonly done with the help of computerized systems and the internet of things (IoT). This network of internet connect computers is easily access from a common, centralized technology hub known as a smart home, smart device, or digital assistant.

This smart device generally takes an input that comes more naturally to a human. This is commonly a voice command such as "turn on the lights" or "lock the front door". These are commands that make sense to a human. The smart device typically takes this phase's sound in via a microphone and runs the detected audio data through a machine learning model that has been trained on a large amount of audio data for many different spoken words and phrases. These models can be something as simple as a multilayer perceptron, but generally a convolutional neural network is used, which is able better at finding patterns in complex data. The machine learning model, however it is implemented, attempts to find the words that have been spoken from the audio data that has been detected, output its best guess at the spoken phrase, and then connect that phrase to a specific everyday process (e.g. turning on the lights) that has already been linked to the smart device.

This exercise in machine learning has been near perfected when it comes to voice detection technology. There are a wide variety of readily available smart devices and digital assistants on many platforms that will easily detect voice commands and perform their associated actions. However, there is a gap in this technology when it comes to using any other natural human communication medium to input commands to a smart device. Specifically, this project was started because of a desire to research and build new and better ways to take an input of hand-based sign language gestures from a video feed and transform them into a command string that can be performed by devices within the internet of things.

This technology is important so that smart devices can be inclusive towards those with certain disabilities, and may only be able to use sign language to communicate, or for any users who may prefer to use sign language to communicate over spoken language. Communication and interaction technologies for the deaf or those who use sign language are often quite expensive and computationally expensive. They are also often difficult to install and maintain. With our approach, we hope to create a system that uses existing technologies that are well maintained to that our system can be more easily installed and updated. We also hope to introduce the capability of the system being distributed in a client-server architecture, which would allow for the more computationally expensive server portion of the system that interprets the hand gesture images to be separated from the more accessible and easily integrated client module.

There have been a few attempts to make readily available hand gesture smart home technology. Our initial goal as stated was to create a robust system that captures hand gestures in American Sign Language, capturing basic English alphanumeric signs A through Z and 1 through 9, with a 90% accuracy at detecting any individual sign in real time. In this case, real time is when an action occurs between 50 and 500 milliseconds after the gesture action has been initiated.

Our project specifically focused on an approach that involved Google's new, rapidly growing, and open-source project, MediaPipe. MediaPipe has several pre-trained, high accuracy image recognition machine learning models that run on a live video feed. There are models for various parts of the human body. Our project used the single hand recognition

module and used this module to create a simpler, faster hand recognition machine learning model on top of this framework.

II. Literature Review

Hand gesture recognition is a relatively difficult problem to solve in the field of machine learning. Most of these initial attempts at creating an extremely accurate machine learning model that detects hand gestures from image frames use conventional convolutional neural networks. In *Sign Language Gesture Recognition* [1], the project contributors took an approach of training a machine learning model using a custom library of 80000 individual numeric signs with more than 500 pictures per sign. Their system is exemplary in showing a refined approach to a convolutional neural network for hand gestures. The system is split into a few major parts, those being a hand detection system that uses a training database of pre-processed images, and then a gesture recognition system.

The image pre-processing uses what is known as feature extraction in order to standardize the input information before we either use it to train a machine learning model or use the pre-trained machine learning model to predict something. The image in the case of [1] is made to be grayscale, a standardized resolution, and is also put through a process that gives object contours within the image. Once the image is pre-processed, the feature extraction techniques flatten the image into a smaller amount of one-dimensional components from the set of two-dimensional pixels. This technique helps highlight certain features about the pixel data from images in a way that is easier for the convolutional neural network that is being trained on this data to understand.

In addition [1] also uses a well-developed and use machine learning process known as a convolutional neural network. With a normal neural network, we have dense networks of connected but individual nodes of information. A convolutional neural network allows the nodes of a network to have large amounts of multidimensional information in each node. Each of these nodes can also be made to look at certain features that are extracted during feature extraction. A convolutional network is also a type of feed-forward network, which means that it takes into account things that have been assessed during the previous layer's analysis. This can help when assessing some hand gestures that have movement elements to them in addition to static finger positions. In their system, [1] used a Python Open CV library for image processing and input and a TensorFlow convolutional neural network, and was able to achieve a hand gesture recognition system that has an accuracy of around 95% to 98%, citing the possibility that complex or noisy backgrounds or poor lighting conditions may affect this accuracy.

In the paper *RGBD Video Based Human Hand Trajectory Tracking and Gesture Recognition System* [2] uses a few additional techniques in hand tracking in two and three-dimensional space. First, they use a couple of methods to help extract hand features from image frames, namely skin saliency, and motion and depth-based filters. Skin saliency takes the fact that skin tones are generally within specific ranges in order to increase the accuracy of feature extraction in image pre-processing. The motion and depth filters reduce noise within the image in the background and where uninteresting features and movements arise. All of these processing methods were also used in turn with a customized convolutional neural network and a large dataset of hand gesture videos in a variety of environments. With added gesture recognition techniques that track hand gesture sequences in context of the gesture being performed, this project resulted in an extremely accurate hand tracking and gesture recognition at a classification accuracy of around 98%.

The project *Dynamic Sign Language Recognition for Smart Home Interactive Application Using Stochastic Linear Formal Grammar* [3] applies a novel technique in sign language recognition, but also introduces smart home integration as a consideration for their output. Their system looked at a smaller amount of larger, dynamic hand gestures using the entire arm rather than the finger spelling American Sign Language like our project. It uses a machine learning approach that uses what is called a bag-of-features method to identify hand gestures from raw video input. This method separates various parts of the body involved in gestures, such as the palm, fingers, wrist, upper arm, and so on, allows certain features to be added or separated. This eliminates a cumbersome need to constantly track different parts of the body for gesture recognition, because we can approximate the position of various parts based on other features.

This approach uses a stochastic linear formal grammar module. This module effectively increases the accuracy of the machine learning model that classifies incoming hand gesture sequences from images by looking at the gesture sequences in turn, as a sort of feed-forward grammar checking. The system has a default, predetermined grammar based on the possible sequences of gestures that can be made. Although this may be more difficult to scale to hand gesture sequences such as finger spelling, where there are many more possible sequences, there is still promise that some sort of post processing of predicted gestures within a sequence produced from a machine learning system may improve accuracy.

All of these previous methods prove that hand gesture recognition, or hand recognition in general, is achievable with a high degree of accuracy with complex mathematical processing and large datasets. For our project, our initial research found an interesting and quickly developing open-source framework from Google known as MediaPipe [4]. MediaPipe is a large collection of high accuracy human body part detection and tracking models. These models are trained on some of the best, largest, and most diverse datasets from Google, and they track key points on certain parts of the body as skeletons of nodes and edges, with the nodes, or landmarks, being three-dimensional normalized coordinate points. MediaPipe is a framework that allows developers to use the custom-built models from Google, or models built by developers using TensorFlow lite, in a way where the pipelines of information flow are easily adaptable and modifiable via graph files. A pipeline in MediaPipe is composed of nodes on a graph, specified in a pbtxt file. These nodes are connected to C++ files that expand upon the base calculator class in MediaPipe. This class gets contracts of media streams from other nodes in the graph, like a video stream for example, ensures that it is connected, and then can output its own stream of processed data once the rest of the pipeline nodes are connected. Each stream of information to each calculator is sent via packet objects, which encapsulate many different types of information. Side packets can also be introduced into the graph, where auxiliary data like constants or static properties can be introduced to a calculator node. This simplicity of structure in the data pipeline allows for additions and modifications to be done more simply and with more precise control over how data

is being passed, how often, and how we are manipulating it between processes.

For MediaPipe hands module specifically [5], we have a machine learning model that specifically looks at singular hands within an image feed. MediaPipe hands consists of two different machine learning models to detect the hand. First, the image is pre-processed to set the image quality to a standard 256 x 256 size JPEG, grayscale, and then the image is given a contour filter. The first MediaPipe model then attempts to detect the palm of the hand, if there is one within the image, and crops the image to another standard sized bounding box (see the size of the image in Figure 1). Once this bounding box is determined for the first time, MediaPipe will not have to spend nearly as much computing power attempting to find the hand within the image. If it knows where the palm was in the previous frame, it can more easily find it in the next frame, assuming the palm does not move to a great degree. Once the palm detection has been run, MediaPipe runs its hand landmark detection model. As stated previously, this model takes a standardized image of a hand and produces a set of 21 hand landmarks. These landmarks are normalized to the standard image size, a value between 1 and -1, and contain x and y coordinates, as well as a z coordinate that represents relative depth for each point (see the red dots on the image in Figure 1).

Figure 1: A set of connected hand landmarks superimposed onto the original hand image from MediaPipe Hands with live video feed.

III. Solutions Details

A. Video Feed Networking & Command Output

The first part of our solution involves a client-side module that performs two tasks, and theoretically allows our more intensive processing via machine learning to be done on a separate server module, containing the MediaPipe instance and the gesture interpreter model. The client module takes input frames from a connected video camera device. The video is captured from the integrated video camera as individual frames by Open CV and each frame is stored during runtime as a two-dimensional numpy ndArray of size 86,400 bytes (480 x 60 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 video frames 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.

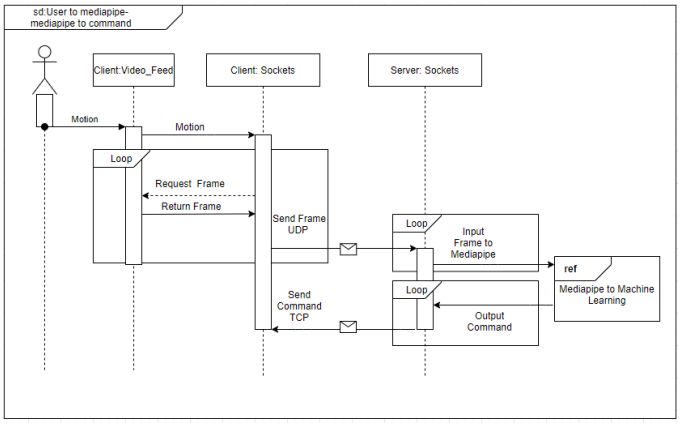

Figure 2: A state machine diagram representing how captured video frames are transferred from the client to the server.

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 make it more likely that the server will miss frames, resulting in a less stable video feed with missing or misplaced slices. 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.

The server UDP receiver 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 the UDP receiver 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 MediaPipe. This entire process is shown in Figure 2.

B. MediaPipe Integration

For our MediaPipe integration, we built a Python framework on top of MediaPipe, and also integrated hand landmark output as a modification to the MediaPipe Hand detection code itself. MediaPipe has defined inputs and outputs and performs work each frame or at specified intervals. MediaPipe comes with a hand tracking project that detects a hand and tracks its movement using 21 defined landmarks on the detected hand. Each of these landmarks represents a three dimensional coordinate, normalized to a standardized range. We modified this hand tracking example application for our use in two different cases. In the 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. Figure 3 shows a sample hand image used in training and the set of hand landmark coordinates that have been output by MediaPipe. In the second case, we ran MediaPipe as a module in our main application and

determine the landmarks of a video feed in real time. This will be referred to as MediaPipe Run.

Figure 3: A sample hand gesture image frame from the training set, and the output of normalized hand landmarks to a csv file (one row per landmark).

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.

C. Machine Learning Model

Our machine learning model was designed to take an input of hand landmark coordinate sets from our library of landmarks sets extracted from our training images of hands performing American Sign Language gestures for letters A through Z. Our original image dataset was about 15000 images distributed evenly among all 26 characters. When we initially ran this set through our MediaPipe train module, some of the landmark coordinate sets for certain images were not accurate to the gesture being performed. This is most likely because, when we fed all the images through MediaPipe, we were only sending a single image frame. Because of the two machine learning models MediaPipe Hands runs the images through, one where it finds the palm and the other where it tracks the hand, MediaPipe is more accurate on a live video feed where it can detect for the palm in a few frames before detecting the fingers for the first time. We then went through all the coordinate sets for each image, and determined which sets correctly matches or closely matched the hand gesture, and used those to train our convolutional model. Each letter had about 300 sets of landmarks to train on per gesture. We shaped out data of 21 landmarks, those being 63 floating point numbers, and placed each set of landmarks corresponding to a finger on the hand into its own separate tensor. Our machine learning model consists of three convolutional 2D layers in TensorFlow with a rectified linear unit activation function on each, which are then flattened and then put through a dense layer with the same activation function.

C. Grammar and Command Output

The grammar module is a smaller component that exists as a part of the same Python application process that runs the gesture interpreter component. The 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 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 grammar component.

This module performs 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.

The command output module 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 and the grammar module. The commands will then be mapped to smart home API functions. For our project, our system merely connects to a simple smart device API called IFTTT, which currently only has the functionality to output the command to text, but has the capability of doing a wide variety of commands, similar to the ones stated at the introduction to this project.

IV. Solution Evaluation

A. Efficacy of Machine Learning Model

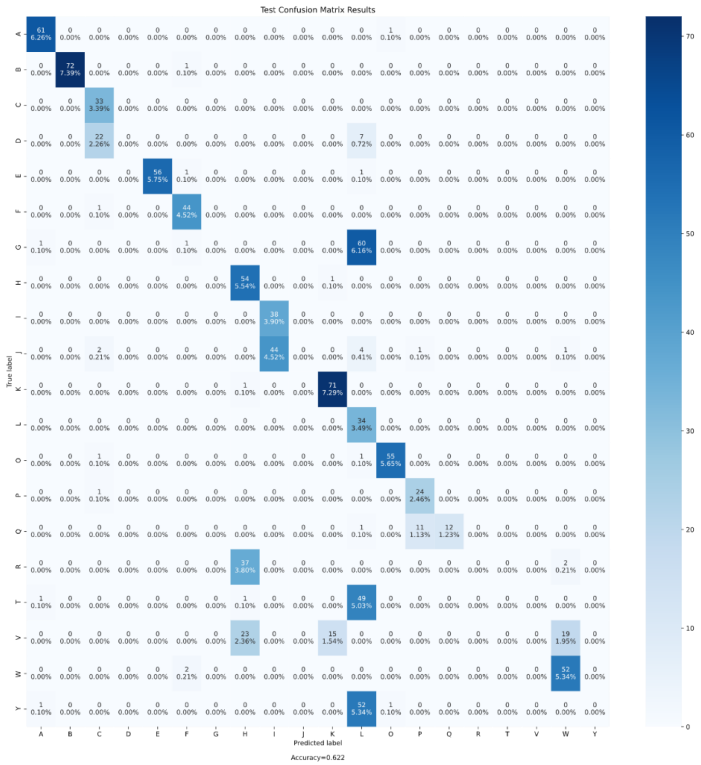

Figure 4: Confusion Matrix of CNN with 20 Classes.

To evaluate our machine learning models we used a series of different measures including accuracy, precision, recall, and f1-score. Accuracy is the ratio of correct predictions to the total predictions made. Since we have a multi-class target the accuracy can give misleading results by hiding information using a broad overview of the results. For this reason, we must use precision, recall, and f1-score. Precision calculates the ability of a classifier to not label a true negative observation as positive. Recall calculates the ability of a classifier to find positive observations in the data. F1-score helps to measure recall and precision at the same time finding a harmonic mean by punishing extreme values.

To calculate these measures, we first pass the 20% of our data we reserved for testing purposes through the models and record the results in a confusion matrix. The confusion matrices in Figure 4 and 5 compare the class predicted by the model to the actual class recorded in the testing data. The y-axis refers to the true label and x-axis refers to the predicted label. The opacity of each square represents the number of samples that were classified accordingly. The optimal results would be a diagonal line of dark colored squares starting from the topmost, left most square and ending at the bottom-most, right most square. That pattern would explain that every sample has been correctly classified.

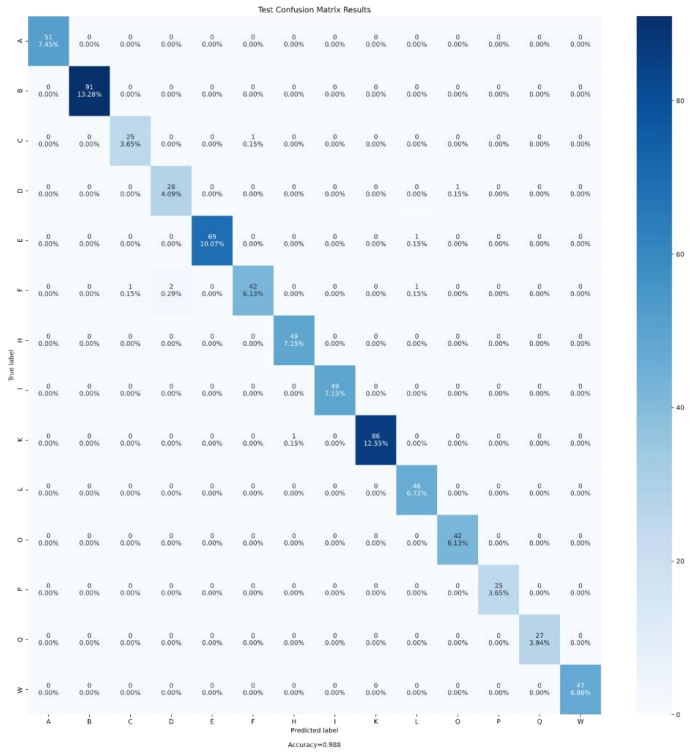

Figure 5: Confusion Matrix of CNN with 14 Classes.

Looking at Figure 4, we observe that the convolutional neural network with twenty classes does not follow the diagonal line of optimal results. The model has incorrectly classified forty-four samples of J's as I's, fifty-two samples of Y's as L's, sixty samples of G's as L's, and twenty-two samples of D's as C's. These instances of misclassifications lead to lower scores in accuracy, precision, recall, and f1, shown by Figure 6. Since this is not a binary classifier, we must calculate the weighted average of all the scores. The results in Figure 6 show accuracy as 62%, weighted average precision as 50%, weighted average recall as 62%, and weighted average f1-score as 54%. To achieve higher scores, we removed the classes with bad results, which included labels G, J, R, S, T, and V leaving us with a total of fourteen classes.

The convolutional neural network with fourteen classes, shown in Figure 5, displays almost perfect results with only a few outliers in classes C, D, E, and F totally around 1% of the testing data. These results lead to very high accuracy, precision, recall, and f1-score. As seen in Figure 7, the accuracy, weighted average precision, weighted average recall, and weighted average f1-score are all around 99%.

	precision	recall	f1-score	support
0.0	0.95	0.98	0.97	62
1.0	1.00	0.99	0.99	73
2.0	0.55	1.00	0.71	33
3.0	0.00	0.00	0.00	29
4.0	1.00	0.97	0.98	58
5.0	0.90	0.98	0.94	45
6.0	0.00	0.00	0.00	62
7.0	0.47	0.98	0.63	55
8.0	0.46	1.00	0.63	38
9.0	0.00	0.00	0.00	52
10.0	0.82	0.99	0.89	72
11.0	0.16	1.00	0.28	34
12.0	0.96	0.96	0.96	57
13.0	0.67	0.96	0.79	25
14.0	1.00	0.50	0.67	24
15.0	0.00	0.00	0.00	39
16.0	0.00	0.00	0.00	51
17.0	0.00	0.00	0.00	57
18.0	0.70	0.96	0.81	54
19.0	0.00	0.00	0.00	54
accuracy			0.62	974
macro avg	0.48	0.61	0.51	974
weighted avg	0.50	0.62	0.54	974

Figure 6: Classification Report of CNN with 20 Classes

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	51
1.0	1.00	1.00	1.00	91
2.0	0.96	0.96	0.96	26
3.0	0.93	0.97	0.95	29
4.0	1.00	0.99	0.99	70
5.0	0.98	0.91	0.94	46
6.0	0.98	1.00	0.99	49
7.0	1.00	1.00	1.00	49
8.0	1.00	0.99	0.99	87
9.0	0.96	1.00	0.98	46
10.0	0.98	1.00	0.99	42
11.0	1.00	1.00	1.00	25
12.0	1.00	1.00	1.00	27
13.0	1.00	1.00	1.00	47
accuracy			0.99	685
macro avg	0.98	0.99	0.99	685
weighted avg	0.99	0.99	0.99	685

Figure 7: Classification Report of CNN with 14 Classes

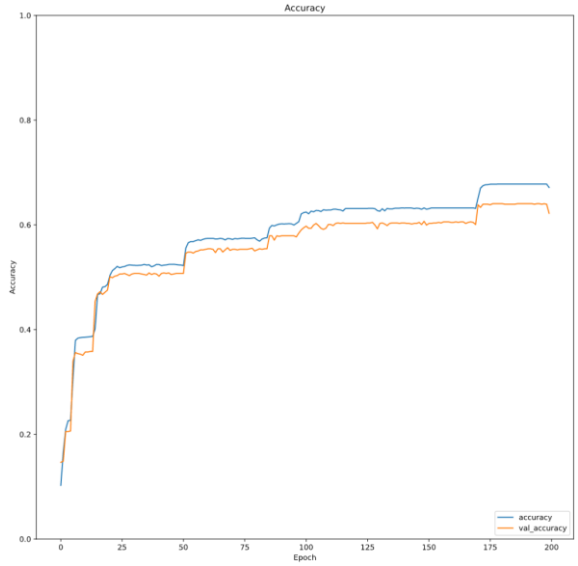

Figure 8: Accuracy of CNN with 20 Classes.

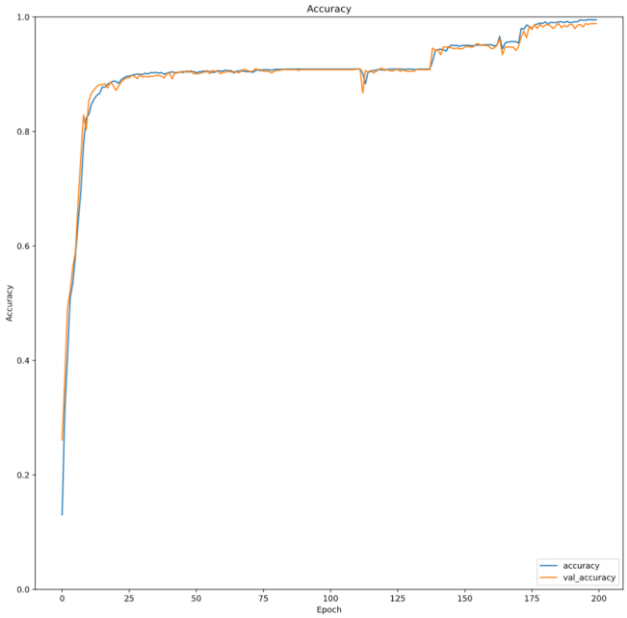

Figure 9: Accuracy of CNN with 20 Classes.

B. Timing Approximation

We ran tests on several live video feeds to record the average time it took to run an image frame through the MediaPipe integration module and the gesture interpreter. We decided to not include the time for the video feed networking component, or anything being outputted by or received by the client module, because those components are merely limited by internet transfer rate. We also did not include the time the gesture sequence, that is, the outputted command, takes to be processed by the grammar module, because this is also a trivial computation, and only take a millisecond or two at the most.

Doing tests with every single gesture in our library after training the model, it took an average of 71 milliseconds to run an image frame through MediaPipe via the MediaPipe integration module. The gesture interpreter module then took an average of 156 milliseconds to process the set of landmarks that have been given as an output from MediaPipe. This means that our core sign language interpretation takes, on average, 227 milliseconds to process a given gesture in sign language. This is important, because although our accuracy may be a bit off our target to an extent, and lower than some other researchers performing similar operations, our real-time calculation goal is well under the desired threshold of between 50 milliseconds to 500 milliseconds.

V. Conclusion

Our tests adequately show that MediaPipe can be easily used as a tool to accurately determine hand gestures. Although our model fell short of some of our goals in terms of the variety of different hand gestures that can be detected, ultimately, we were able successfully include enough of a variety of letter that, with a larger, more accurate, and more diverse training set of hand landmarks, could easily be expanded upon and strengthened in accuracy. Avenues for further research could include further manipulation of hand landmark data for more complex and dynamic gestures, such as tracking individual velocity of landmarks for signs such as J or Z, or any other moving hand

gestures. One could also improve upon the machine learning model used by testing different activation function or layers to see if accuracy with the 20 classes can be improved. Ultimately, further research into hand gesture recognition systems specifically for smart devices and including MediaPipe's state-of-the-art technology is very likely, as MediaPipe allows for an ease of breaking down and analyzing complex hand tracking information, without having to construct a brand new convolutional neural network for image recognition from scratch.

REFERENCES

[1] R. Sharma, R. Khapra, N. Dahiya, "Sign Language Gesture Recognition.," in *Sign*, June 2020, pp.14-19

[2] W. Liu, Y. Fan, Z. Li, Z. Zhang, "Rgbd video based human hand trajectory tracking and gesture recognition system," in *Mathematical Problems in Engineering*, Jan. 2015

[3] MR. Abid, EM. Petriu, E. Amjadian, "Dynamic sign language recognition for smart home interactive application using stochastic linear formal grammar," in *IEEE Transactions on Instrumentation and Measurement*, Sep. 2015, pp. 596-605

[4] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, Cl. Chang, MG. Yong, J. Lee, WT. Chang, "Mediapipe: A framework for building perception pipelines,", June 2019, doi:1906.08172

[5] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, CL. Chang, M. Grundmann. "MediaPipe hands: on-device real-time hand tracking", June 2020