HAND SIGNALS WITH VISUAL DATA

FINAL REPORT

[CSC 59866 – DD1]
FALL 2019

BY
AYUSHYA AMITABH
CHRIS PANICAN
GERRY XU
OMAR ELNAGDY

Table of Contents

ABSTRACT

We decided to conduct this research because in this generation it is known to be wireless about everything, so why not be *KVM-free* as well. To be keyboard, video, and mouse free can really make all types of user's experience easy and smooth, with no complications. Imagine a user that's not in the tech field and has no experience with tech and just doesn't know all the commands. With our software whether you are in the tech field or in any field, we allow the common human gestures to be your controller without trying to learn new commands, which makes the human life much easier for them. This matters to us because we are in fact in the tech field and we can truly understand that even some of the basic commands are hard to remember since we have so much else on our minds. When conducting this research the process was difficult however the results came out as expected, which was to allow for simple gesture based controls on the end user's computer to perform an action based on the gesture taken from the webcam. This research is important to us and it should be important to everyone else because it's really only a matter of time till air gestures become the next touchpad, and getting ahead of the game with small gesture based controls now will allow a better understanding on how to create more simple air gestures for more complicated commands in the future.

INTRODUCTION

"Hand Signals with Visual Data" is the title of our proposed project that will work to create a software to allow for gesture based controls on the end user's computer with the aid of deep learning to help recognize the user's hand as well as the number of fingers being held up by the user.

MOTIVATION

The motivation behind "Hand Signals with Visual Data" is split between the idea of fantasy and the urge to push technology to new limits. Like self-driving cars, speech recognition, and many other dreams that Artificial Intelligence has helped us achieve - we believe that Deep Learning (sub-field of Artificial Intelligence) can help us achieve this fantasy too. The use of hand gestures has been a common place in many futuristic movies - a thing of fantasy that only computer generated imagery or CGI could achieve. This, however, is no longer true - with the recent developments in AI we can now work towards realizing the dream of controlling our computer with effortless hand gestures without having to lay hands on the computer itself.

A related work that inspired us to try our hands on our own version is by someone named Abhishek Singh who made a project called "CNNGestureRecognizer" which is using the OpenCV library to capture the user's hand gestures. His application uses a CNN model that is able to recognize up to 5 pretrained hand gestures such as, ok,

peace, stop, punch and nothing. The app is then able to guess the user's hand gesture against the pretrained gesture and the prediction data can be plotted in real time on a predication bar chart. His final product was being able to play a game that is built into the Chrome browser, the Dino Jump game, where he tests his trained gestures in this case the "punch" gesture, where he bind the "punch" gesture with the jump action of the Dino character.

This final report will cover the progress made by the team towards our final project. Through the course of this progress report we will also discuss the division of roles through the team, our achievements, ongoing progress, general project organization, the challenges we have faced, and our future goals.

Our project source code is being managed on GitHub and can be found at:

https://github.com/cpanican/capstone

TEAM CONTACTS

Ayushya Amitabh ..... aamitab000@citymail.cuny.edu

Chris Panican ..... cpanica000@citymail.cuny.edu

Gerry Xu ..... gxu000@citymail.cuny.edu

Omar Elnagdy ..... oelnagd000@citymail.cuny.edu

BACKGROUND STUDIES

Gesture recognition is an important topic in computer vision because of its wide range of applications, such as HCI, sign language interpretation, and visual surveillance. Myron W. Krueger was the first who proposed Gesture recognition as a new form of interaction between human and computer in the mid-seventies. The author designed an interactive environment called computer-controlled responsive environment, a space within which everything the user saw or heard was in response to what he/she did. Rather than sitting down and moving only the users fingers, he/she interacted with his/her body. Gesture recognition has been adapted for various other research applications from facial gestures to complete bodily human action. Thus, several applications have emerged and created a stronger need for this type of recognition system, which is why as a group we decided to take on study like this. A study we researched that we found very interesting would be Dong Guo, Yonghua Yan, and M. Xie's research which described an approach of vision-based gesture recognition for human–vehicle interaction. The models of hand gestures were built by considering gesture differentiation and human tendency, and human skin colors were used for hand segmentation. A hand tracking mechanism was suggested to locate the hand based on rotation and zooming models. The main research was focused on the analysis of interaction modes between human and vehicle under various scenarios such as: calling-up vehicle, stopping the vehicle, and directing the vehicle, etc.

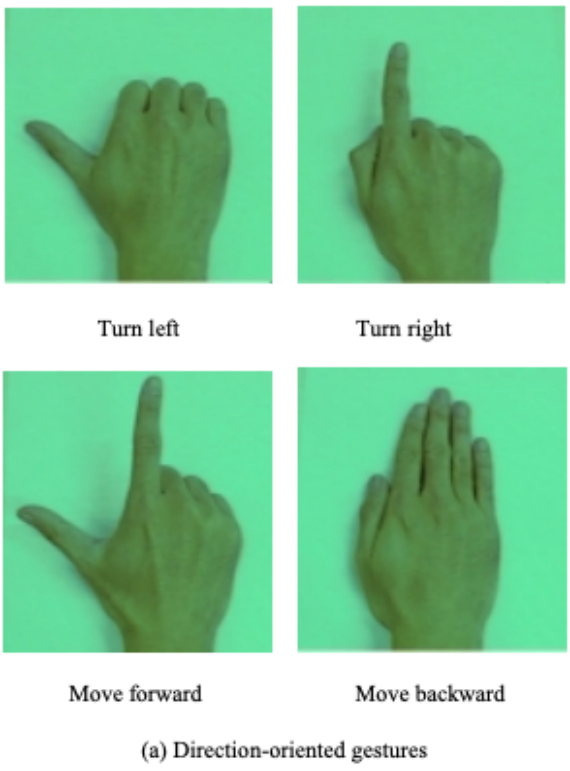

Figure 1: Various hand gestures and their corresponding actions

What Dong and his colleagues came to see and what we as a team came to avoid was the limitation of the use of skin colors method for hand segmentation which might dramatically affect the performance of the recognition system in the presence of skin-colored objects in the background. Hand gesture recognition studies started as early as 1992 when the first frame grabbers for colored video input became available, which enabled researchers to grab colored images in real time. This study signified the start of the development of gesture recognition because color information improves segmentation and real-time performance is a prerequisite for HCI. Hand gesture analysis can be divided into two main approaches, namely, glove-based analysis, vision-

based analysis. The one we will focus on is vision-based analysis. Vision-based analysis, is based on how humans perceive information about their surroundings. Through the research, several feature extraction techniques have been used to extract the features of the gesture images. A common feature extraction method was applied in another study for the problem of recognizing a subset of American Sign Language (ASL). In the classification phase, the author used a single-layer perceptron to recognize the gesture images. Using the same feature method, namely, orientation histogram, proposed a gesture recognition method using both static signatures and an original dynamic signature. The static signature uses the local orientation histograms in order to classify the hand gestures. Despite the limitations of orientation histogram, the system is fast due to the ease of the computing orientation histograms, which works in real time on a workstation and is also relatively robust to illumination changes. Most recognition phases are performed in real-time using a camera video. The recognition system can process 23 frames per second on a Quad Core Intel Processor. Through most of this research most authors concluded that the presence of skin-colored objects in the background may dramatically affect the performance of the system because the system relied on a skin-based segmentation method. Thus, one of the main weaknesses of gesture recognition from color images is the low reliability of the segmentation process, if the background has color properties similar to the skin. Furthermore, hand gesture recognition systems that use feature extraction methods suffer from working under different lighting conditions as well as scaling, translation, and rotation problems.

OVERVIEW OF SYSTEM ARCHITECTURE

Hand Gestures with Visual Data consists of two main components called backend and frontend components. The diagram below presents a very basic form of our software architecture. Users access the frontend components (see right portion in figure 2) directly by interacting through the application. The frontend component can interact with the backend component to send inputs from frontend and apply the gestures as actions.

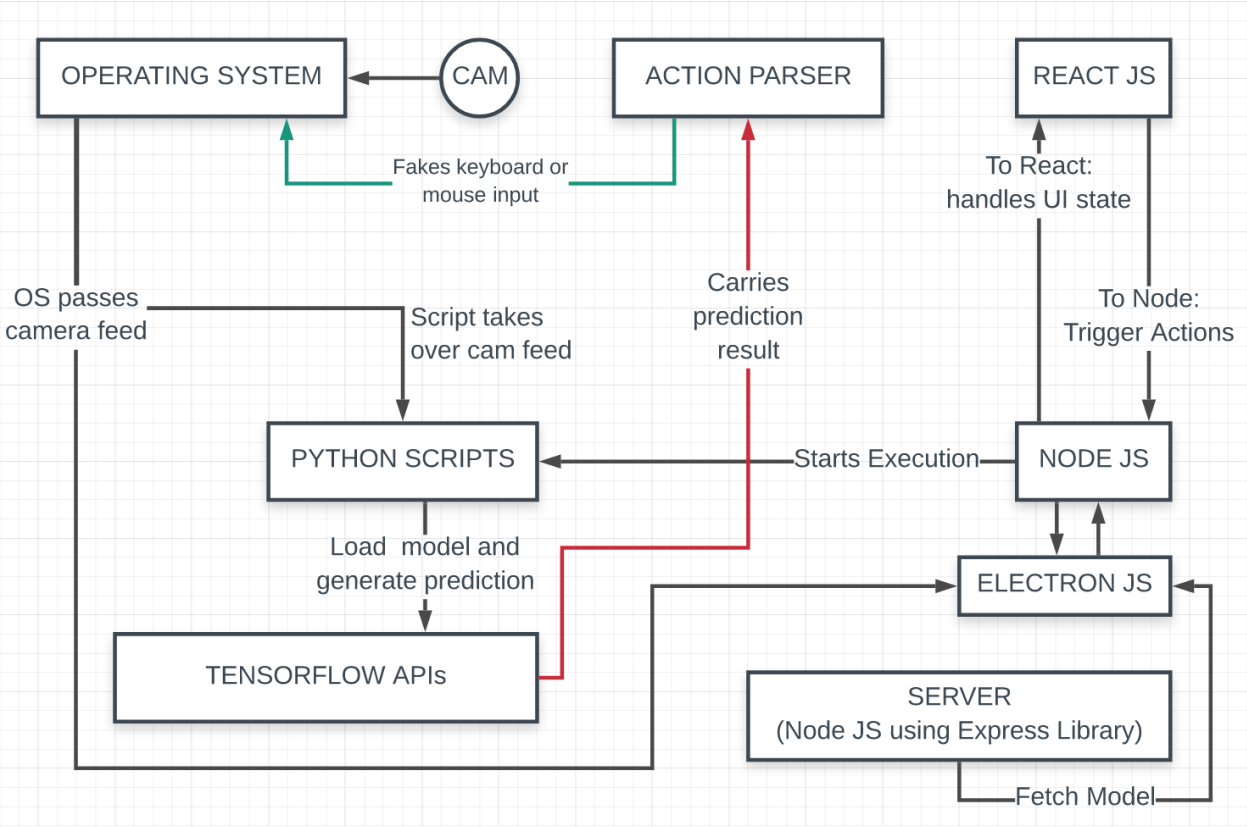

Figure 2: Hand Signal with Visual Data Software Architecture

TECHNOLOGIES USED

Using the latest stack is one of the requirements that our group wanted to implement. Being able to adapt to the latest technologies will favor us better outcomes. The technologies that will be used to create the software are listed and described below.

Electron Framework

The “Electron Framework” is an open-source project created and maintained by Github that simplifies making cross platform applications. Electron accomplishes this by utilizing Google’s open-source Chromium runtime and combining it with the Node.js runtime. Being built on a JavaScript runtime allows for the development and use of a single user interface across all platforms.

Node.js

Node.js is an asynchronous-event driven JavaScript runtime, designed to build scalable network applications. Our software will use the Node.js runtime to create an exclusively locally hosted server to allow for communication between our Deep Learning library, the system tool used to mock inputs, and the software’s user interface.

React

React is an open-source JavaScript library maintained by Facebook that allows for building component-based user interface development. It's used for handling view layer for web and applications, it combines the most common languages in web development, HTML, JavaScript, and CSS into a single component.

TensorFlow

TensorFlow is a free and open-source software library built by Google for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. We used TensorFlow's Python API's to accomplish the deep learning aspects of our project.

In the first couple of weeks of development our team has separately worked toward developing the client-side app and the machine learning model. To start off, let's take a closer look at the progress made towards our client-side app.

CLIENT-SIDE APP

As described in the architecture above, we are using a combination of Node.js along with the Electron Framework built on top of Node.js, and React library to manage our front-end DOM. In the first few weeks of development we have established code

standards, based off of Airbnb's Es-Lint configurations which can be found on their GitHub JavaScript Packages.

In accordance with Electron's developer guide, our app establishes two main processes – Main and Renderer. The Main process is in charge of handling the communication between the operating system and the Renderer process. The Main process is written using Node.js but will be capable of invoking other scripts if necessary. The Renderer process manages the GUI, although it does have its own utilities structure which allows for a modular approach to communication between the Renderer and Main processes.

The communication between the Main and Renderer process is done using instances of an Inter Process Communicator (IPC) which is packaged with the Electron framework. The IPC Main is invoked in the Main process while the IPC Renderer is selectively called in the Renderer process.

Below is an example of event listeners being registered in the Main process.

```
/*
File: app/public/electron.js
*/

  app.on('ready', () => {
  // EVENT LISTENERS
      ipcMain.on('get-window', getWindow);
      ipcMain.on('create-settings', createSettings);
      ipcMain.on('save-model', (event, arg) => {
  // …..
      createWindow();
  });
```

Here is an example of the event listener being triggered through the Renderer process.

```
/*
File: app/src/Utilities/Server.js
*/
  getDateModel = async (d) => new Promise((resolve, reject) => {
    ipc.on('save-model-done', (e, reply) => {
      resolve(true);
    });
    ipc.send('save-model', {
      url: this.__url + ENDPOINTS.DOWNLOAD_DATE.url(d),
      method: ENDPOINTS.DOWNLOAD_DATE.method,
    });
  })
```

The invocation of an IPC event allows for replies from the listener in the Main process, this enables a two-way channel between the IPC Main and IPC Renderer. These IPC tunnels are responsible for downloading models and will handle simulating key presses.

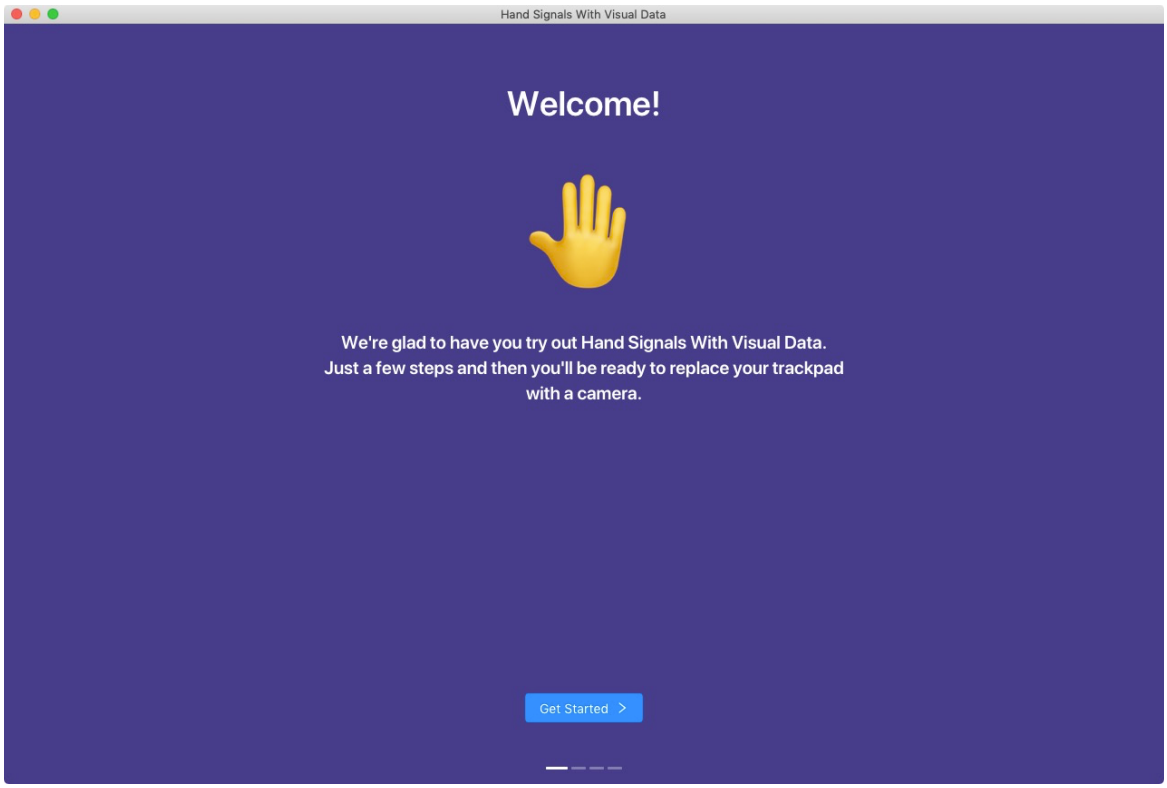

Figure 3: Welcome Screen of our application

This is the welcome page of our application, it tells the user that there are a few steps that need to be completed before the user can replace their trackpad with a camera. Clicking the “Get Started” button will navigate the page to the next component.

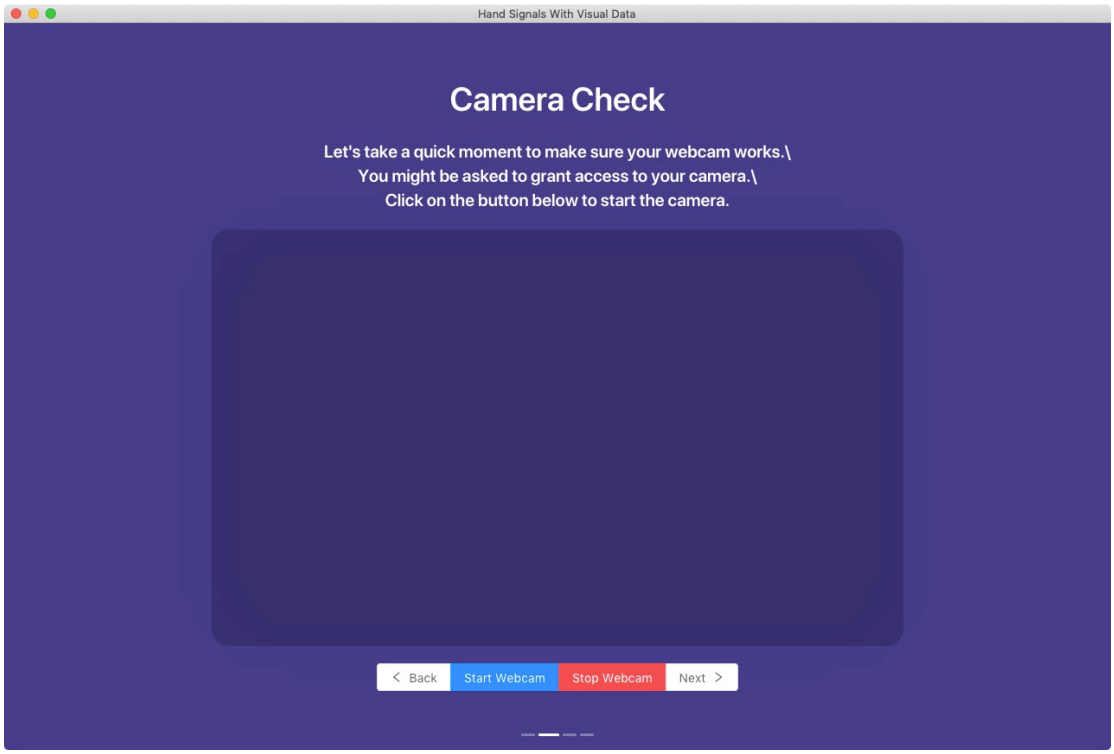

Figure 4: Camera Check Screen

In Figure 4, it asks the user to check if their webcam is working by clicking the “start webcam” button. If the user wants to stop the webcam, the “stop webcam” button does that. If the start webcam button doesn’t work, then the user is prompted that no cameras are detected.

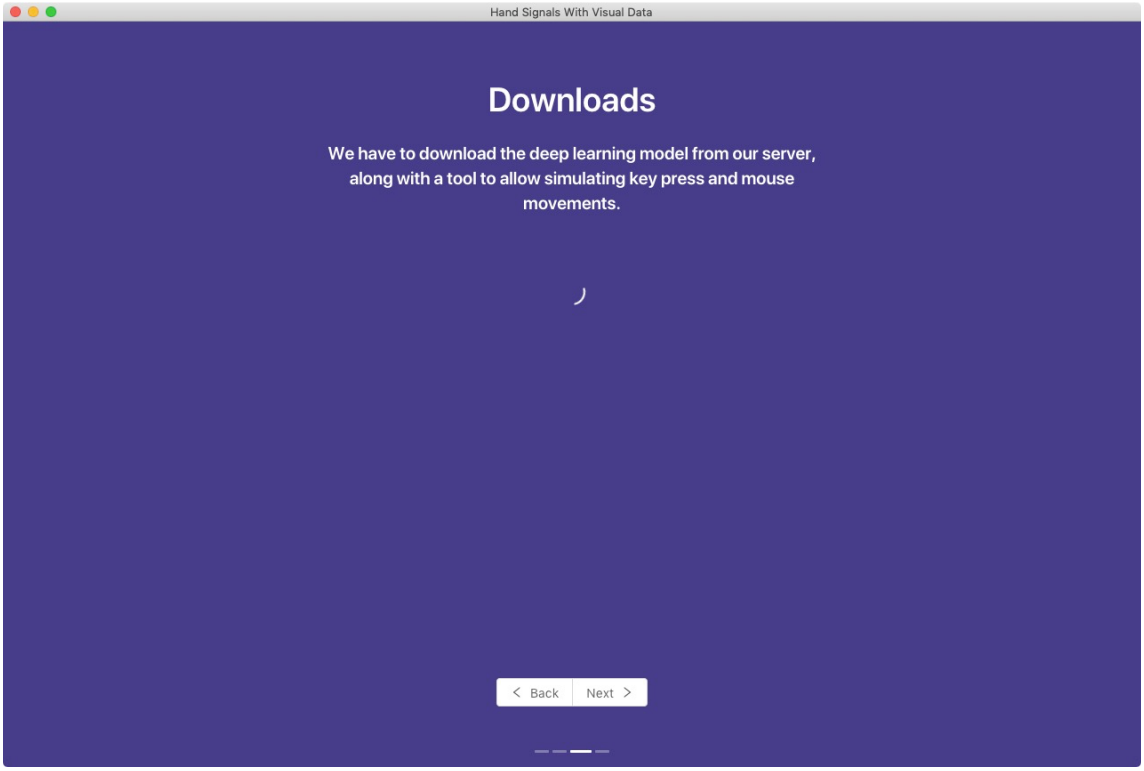

Figure 5: Download Model Screen

Here in figure 5, it tells the user that it needs to download the deep learning model from the server to allow simulating key press and mouse movement. If there are no models currently in our server, the app will be constantly searching in our server to fetch the uploaded model. The user can pick various models from our server. The next section will be showing that interface.

SERVER

Our server is also written in Node.js – the main purpose of our server is to provide downloadable models to the client app, this allows for version control of the models and

gives the user control over which version they want to use. Our server uses Express.js, a Node.js library that simplifies hosting a REST API using Node.js. The project structure for an active server is the following:

```
.server
+-- index.js                                    - server root
|   +-- uploads/                                - folder containing uploads
|   |   +-- status.json                         - maintains model version info
|   |   +-- {filename}-{datestring}.{extension} - format of saved models
```

The uploads directory stores the uploaded models and handles the version listing, in order to achieve this, our REST API establishes the following endpoints:

Method	URL	Purpose
GET	/	Checks if server is alive
GET	/upload	UI for uploading new model versions
PUT	/upload	Uploads file with unique name and updates server status
GET	/download/list	Gets list of available versions and lastest version
GET	/download/:date	Downloads specific version, latest if date is invalid
GET	/download	Downloads latest version

Figure 6: Table of Server APIs

Figure 7: Upload New Model UI on the Server

Figure 7 above is our UI for uploading new models in our server, we can simply drag any files that contains our model into it. This UI can only be accessed on the server so users can't change this data. After dragging in a file that contains our model, we can then click upload and upload the model to our server as seen in the figure 8 below:

Figure 8: Successfully Uploaded Model

After the model is successfully uploaded to the server, we can go back to the download section of the application to see that the model is retrieved and ready to be downloaded. In addition, we can also upload multiple models and users could pick a model that suits them the most. Figure 9 shows the interface users will see if a model is available to download.

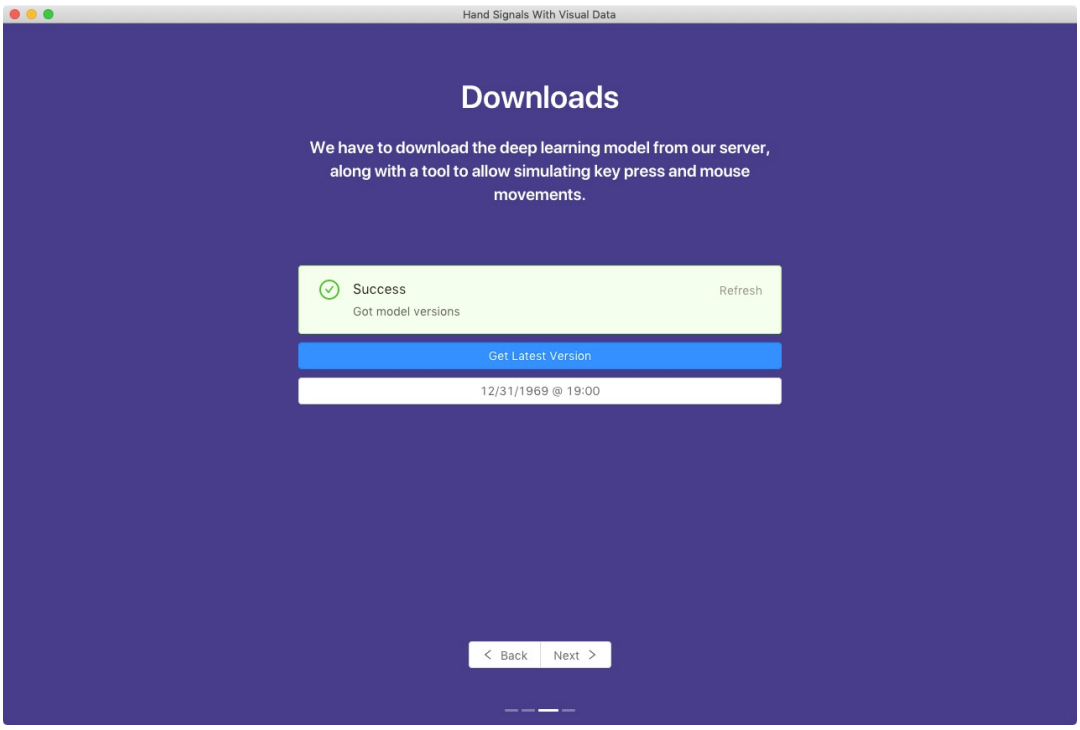

Figure 9: Download Section of Application

Simply clicking on the model that was just uploaded, the application will download the model and apply it for the current user. If the user wishes to change their model, then they can always rely on the navigation buttons to go back and change to a model that they wish to use.

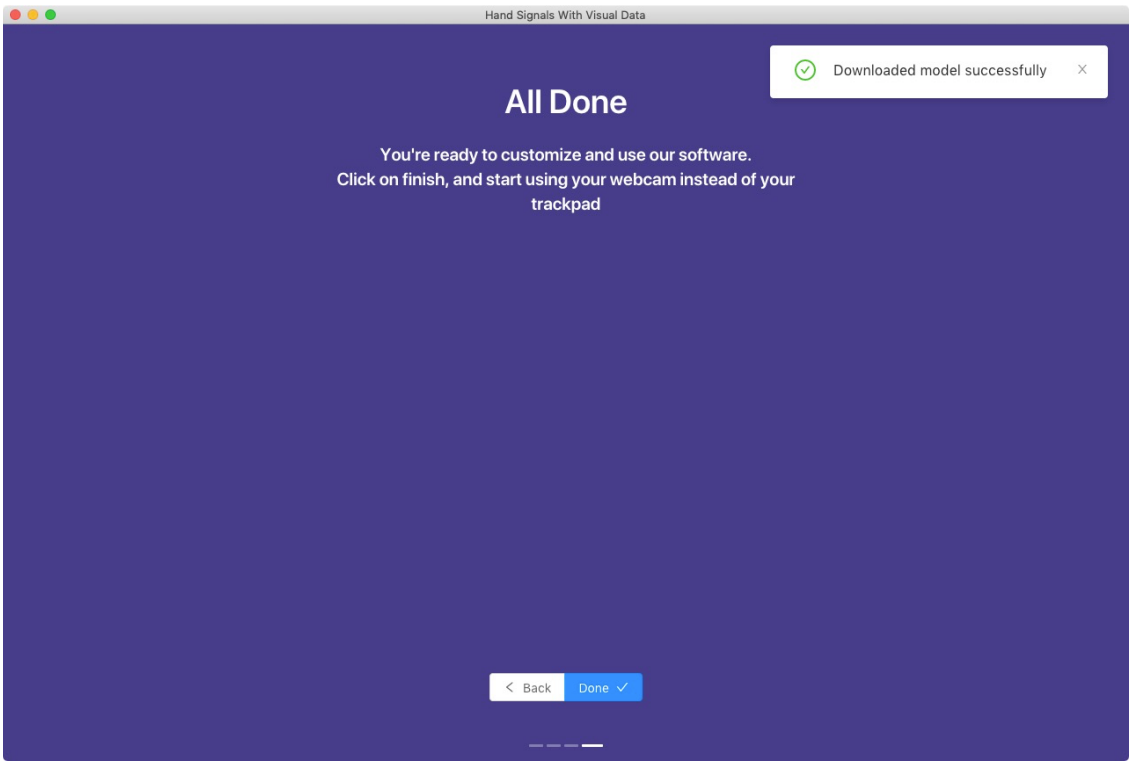

Figure 10: Completion Screen

Figure 10 shows the completion screen after setting up the program. After clicking done, the program will run in the background to save CPU/RAM resources. The user can repeat this process by running the program again.

DEEP LEARNING

In our project, we will utilize deep learning by gathering visual data from a webcam and use that data to trigger commands in the user's machine. Our group picked CNN because of its ability to process images. Other neural networks, such as RNN, cannot excel in this task.

CNN contains fully connected layers in which each neuron in one layer is connected to all neurons in the next layers. There are 4 building blocks in making a CNN:

1. Convolution – receiving input signals from another layer and using filters/kernels to convolve across the input. This is where the network "learns" important filters as it runs.
2. Subsampling – also known as pooling layer. It reduces the dimensions of data with a filter size and a stride and also controls overfitting. It is usually inserted after a convolution.
3. Activation – controls how information flows from one layer to another.
4. Fully connected – connects all neurons in a layer from its previous layer to its sub-layers.

In the first few weeks of development, our group has made enough progress to show a functioning product. One of the accomplishments our team has made so far is that we were able to create a Convolutional Neural Network to detect different hand gestures. The program is able to use the user's webcam as input, then it will output a predicted gesture. Shown below are screenshots of our model's prediction input and output.

Figure 11: First few weeks of progress

For our current model we are using Digit/Number classification similar to what we used in the MNIST database classification.

To generate our dataset, we are using a script that continuously extracts frames from a video stream, similar to what the end usage would be using a computers built-in webcam. These frames are then fed into our model trainer which is defined as:

```
model = Sequential()
model.add(Conv2D(nb_filters, (nb_conv, nb_conv),
                        padding='valid',
                        input_shape=(img_channels, img_rows, img_cols)))
convout1 = Activation('relu')
model.add(convout1)
model.add(Conv2D(nb_filters, (nb_conv, nb_conv)))
convout2 = Activation('relu')
```

```
model.add(convout2)

model.add(MaxPooling2D(pool_size=(nb_pool, nb_pool)))

model.add(Dropout(0.5))



model.add(Flatten())

model.add(Dense(128))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(nb_classes))

model.add(Activation('softmax'))
```

Our model has following twelve layers:

```
_________________________________________________________________

Layer (type)                 Output Shape              Param #

=================================================================

conv2d_1 (Conv2D)            (None, 32, 198, 198)      320

_________________________________________________________________

activation_1 (Activation)    (None, 32, 198, 198)      0

_________________________________________________________________

conv2d_2 (Conv2D)            (None, 32, 196, 196)      9248

_________________________________________________________________

activation_2 (Activation)    (None, 32, 196, 196)      0
```

```
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 98, 98)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 32, 98, 98)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 307328)            0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               39338112
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
_________________________________________________________________
activation_4 (Activation)    (None, 5)                 0
=================================================================
```

EXPERIMENTS AND RESULTS

To train our model, we started by setting the steps to 100, which in our case is 1 epoch, so we can see what the accuracy is. As seen in figure 12, the graphs plotted using TensorBoard shows the training accuracy vs. number of steps. As the steps increases

we can see the accuracy increases and almost hitting 1 at certain steps. This shows that the model we designed are being trained correctly. Figure 13 shows the graph for validation accuracy vs. number of steps plotted using TensorBoard, validation accuracy is the data used to validate the generalisation ability of the model during the training process, which in this case is a linear line.

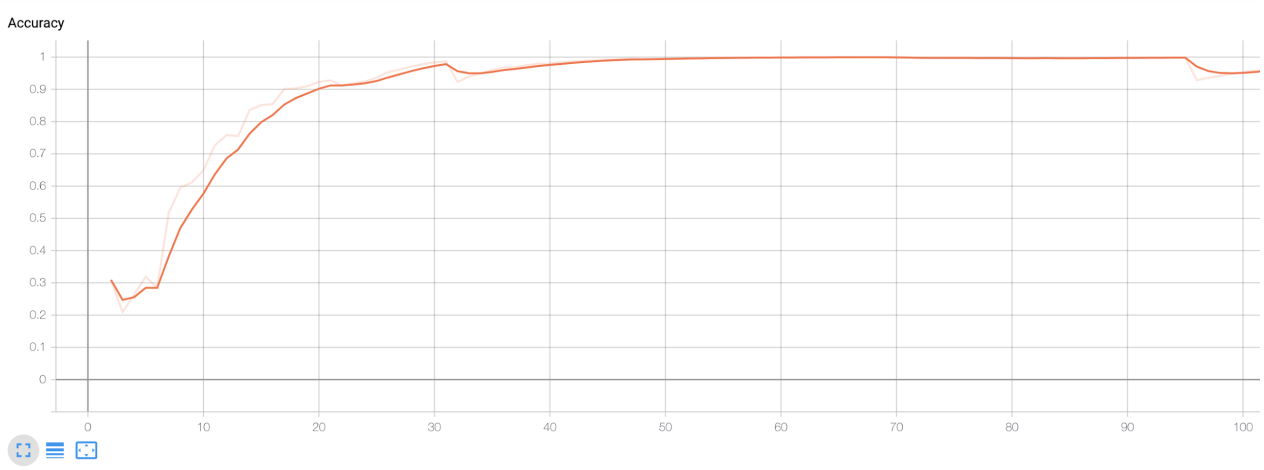

Figure 12: Training Accuracy vs Num of Steps (1 epoch) in TensorBoard

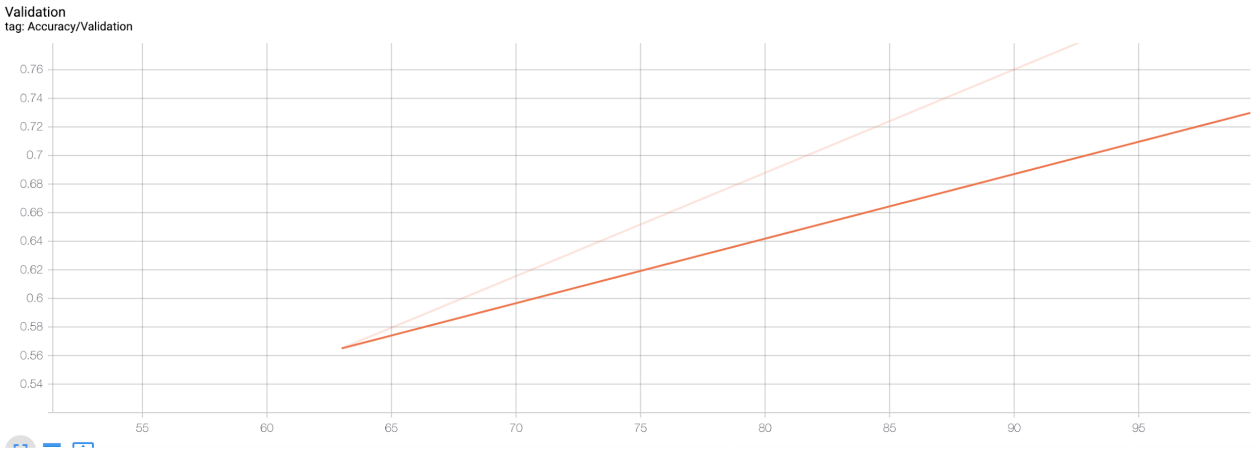

Figure 13: Validation Accuracy vs Num of Steps (1 epoch) in TensorBoard

On the other hand, after 100 steps, the training loss values vs. number of steps graph as shown in figure 14 is also plotted using TensorBoard. We see a decline in the loss value as the steps increases, which should be the case since the lower the loss, the better the model. The main objective in a learning model is to reduce or minimize the loss function's value. Figure 15 below shows a graph of validation loss vs. number of steps using TensorBoard, validation loss is decreasing as steps increases and in figure 13, validation accuracy is increasing as steps increases, which is what a model should be showing when it's learning correctly.

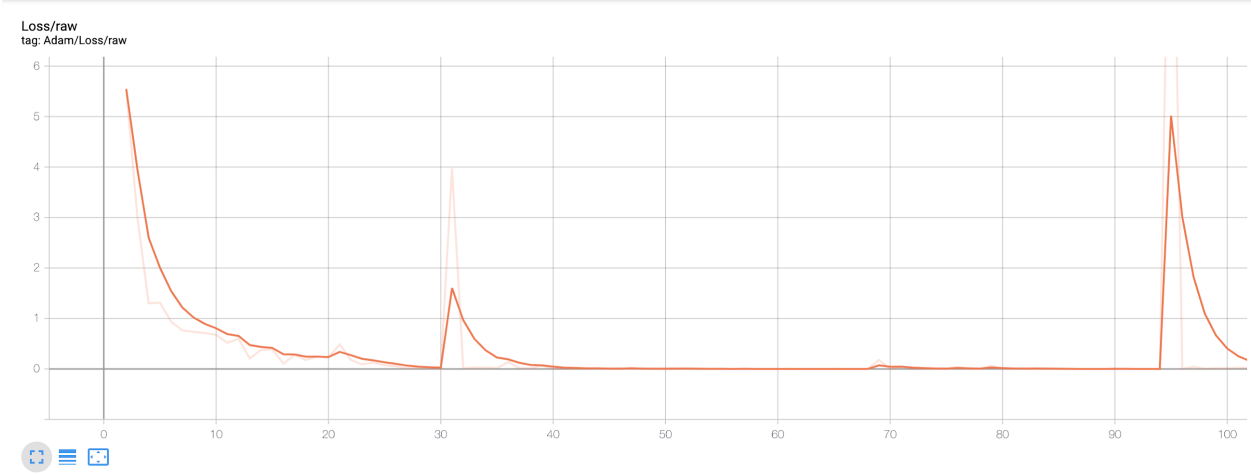

Figure 14: Training Loss vs Num of Steps (1 epoch) in TensorBoard

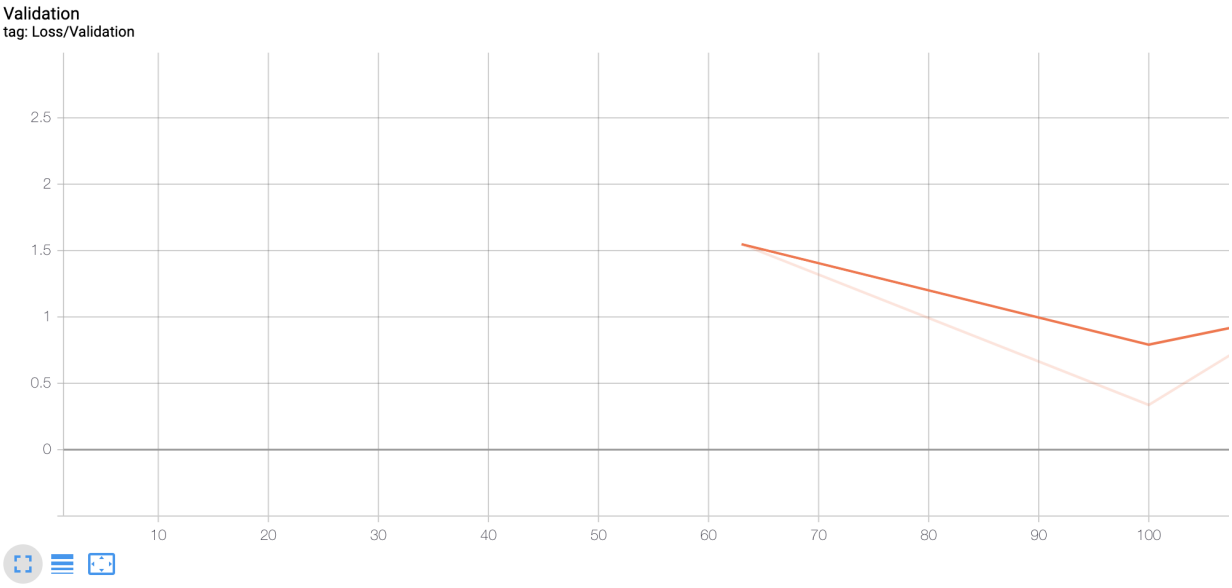

Figure 15: Validation Loss vs Num of Steps (1 epoch) in TensorBoard

Next we decided to train our model for fifteen epochs or 1500 steps, we were able to achieve the following accuracy and loss in figure 16 and 17. We have observed that this is the most optimal way to train our data in our local machines. After trial and error, our team settled to at least 15 epochs which could give us a very similar and desirable accuracy.

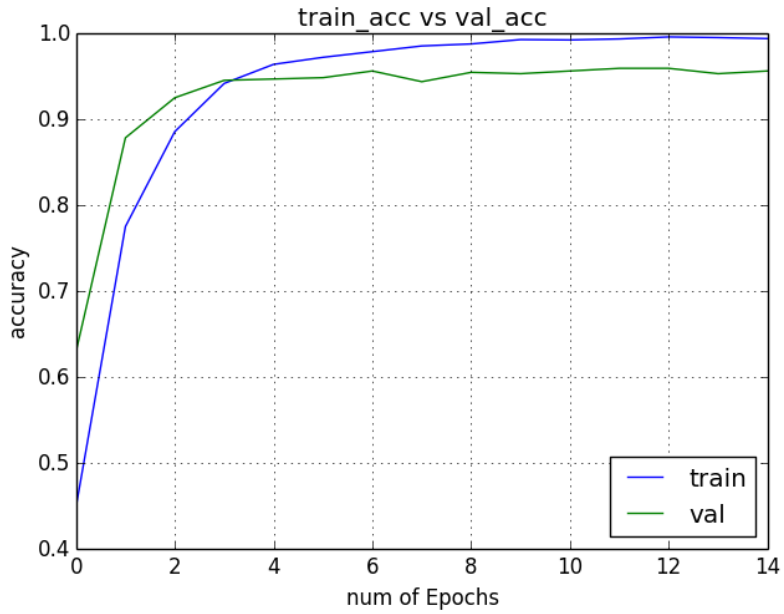

Figure 16: Summary of Accuracy vs Num of Epochs (14 epochs)

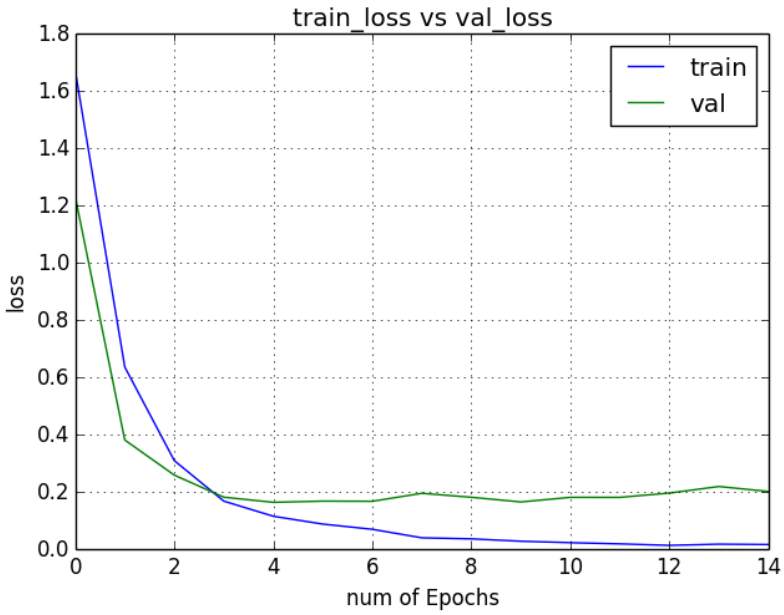

Figure 17: Summary of Loss vs Num of Epochs (14 epochs)

CNN is good in detecting edges and that's why it's useful for image classification kind of problems. In order to understand how the neural net is understanding the different gesture input it's possible to visualize the layer feature map contents.

Using Keras we can visualize the layers:

```
layer = model.layers[layerIndex]



get_activations = K.function([model.layers[0].input, K.learning_phase()],
[layer.output,])

activations = get_activations([input_image, 0])[0]

output_image = activations
```

A bug we first ran into, but has since been fixed was when we were generating and training a new gesture, the accuracy/confidence of that gesture was over 100%. The two images below shows what it was before the bug fix and after the bug fix:

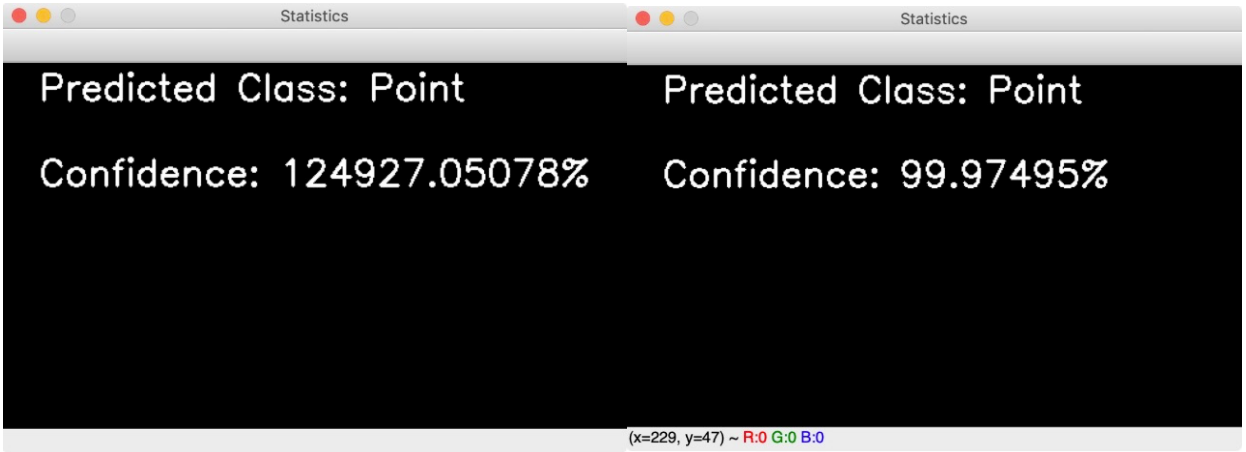

Figure 18: Before and After Confidence Fix

The image on the left shows that the gesture “point” had a confidence of over 100%, which is incorrect, the image on the right shows the fixed version and now it shows the correct confidence value for the gesture.

Here are the visualizations generated for the palm gesture:

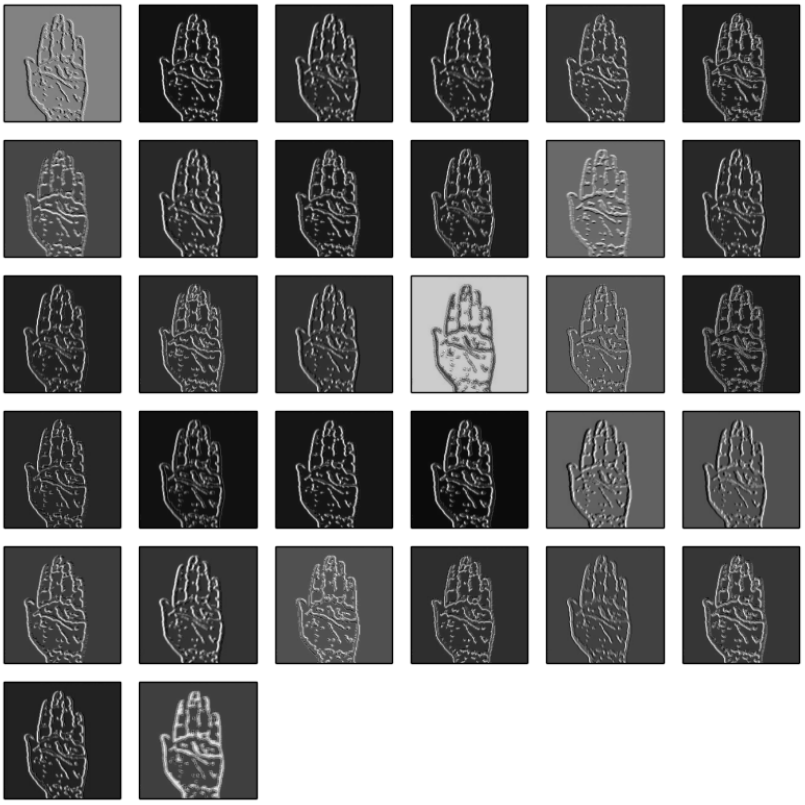

Figure 19: Visualizations of Palm Gesture

Another accomplishment our team has made so far is that we are able to map our hand gestures to the keyboard key-related actions using cliclick.

```
/*
File: ml/ContinuousGesturePredictor.py
*/
  commandDictonary = {
  // EVENT LISTENERS
      “Palm”: “play-pause”,
      “Fist”: “”,
      “Point”: “volume-up”,
      “Swing”: “mute”
  }
  parser = actionParser(commandDictonary)
  parser.doCommand(className[predictedClass])
```

In the code above, you can see that gesture “Palm” is mapped to the “play-pause” action on the keyboard, “Fist” is not mapped to anything at the moment, “Point” is mapped to the “volume-up” action and “Swing” is mapped to the “mute” action.

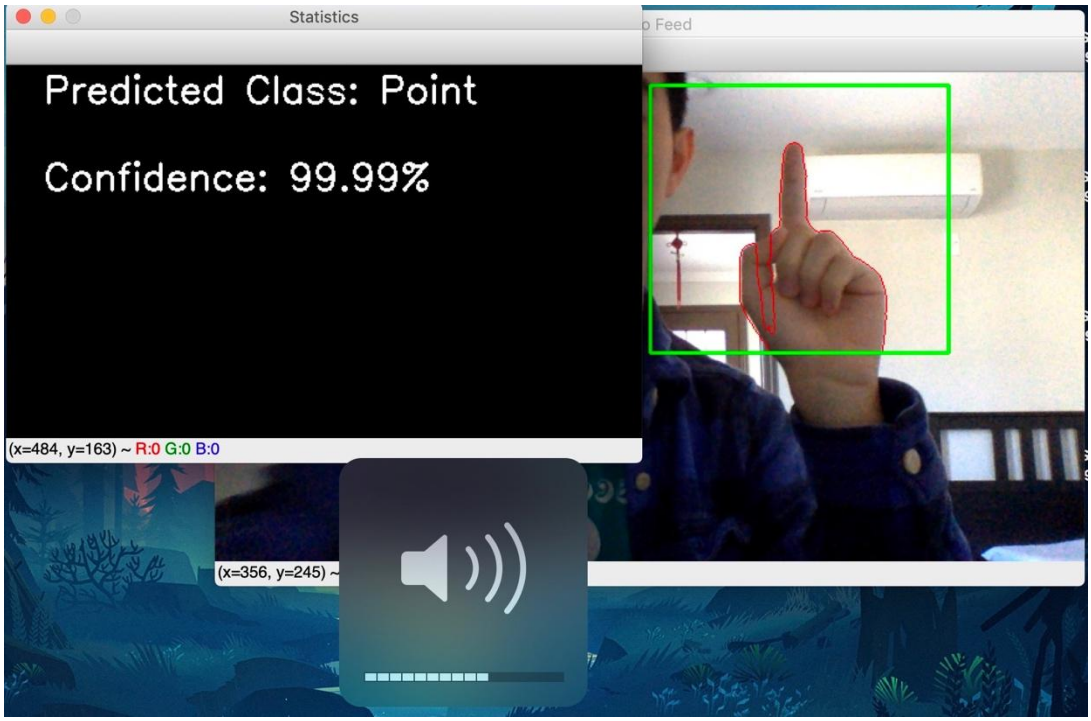

Figure 20: Point Gesture Emulate Volume Up Key Press

The image above shows a demonstration of an example of the “Point” gesture that will increase the volume on your computer every time the “Point” gesture is captured by your webcam.

CONCLUSION

Challenges

Our team is aware of the fair amount of hurdles we are currently facing in our road ahead. One major issue that we are trying to resolve is having a more optimized model in our program. There will be some overlaps between different classes because we have to account to multiple variables when collecting the data. Users might have different kinds of hardware or permissions that may block us from gathering images. Other notable challenge is moving gestures. Right now, our program can only detect static images but our main goal is to detect movements as well. Another major issue we were having trouble to resolve would be the background issue. More or less, the background should be static since its just comparing it to an initial capture and every time we run the program it takes a photo and then it just compares any changes to the live image. The question is what background will we remove? This turned out to be an important question, since the more specific a model is in terms of objects, angle, etc. the higher quality the separation will be. When starting our work, we thought big: a general background remover that will automatically identify the foreground and background in every type of image. But after training our first model, we understood that it would be better to focus our efforts in a specific set of images. Therefore, we decided to focus on our hands only.

The first step in most vision-based gesture recognition systems is the hand region detection and segmentation. This segmentation can be a particularly challenging task when it comes to complex backgrounds and varying illumination. In such environments, most hand detection techniques fail to obtain the exact region of the hand shape, especially in cases of dynamic gestures. Meeting these requirements becomes even more difficult, due to real-time operation demand. To overcome these problems, we attempted a new method for real-time hand detection in a complex background.

Future Goals

Our team plans to tackle the following to-do's after accomplishing our current tasks. We believe that after finishing this list, then we will have minimum viable product:

1. Better gesture detection with camera
    a. Motion detection
    b. Background elimination
        - We will try to employ a combination of existing techniques, based on motion detection and introduce a novel skin color classifier to improve segmentation accuracy. Motion detection is based on image differencing and background subtraction. Skin color detection is attempted via a color classification technique that

employs online color training, so that the system can dynamically adapt to the variety of lighting conditions and the user's skin color as well as possible. Finally, the derived motion, color and arrangement of information are combined to detect the hand region. Experimental results might show significant improvement in hand region detection.

2. Easy to use and intuitive interface

- Allow our application to be a plugin, where a user can turn it off or on.
- Allow for custom gestures based off the user's preference on a specific command

3. Documentation

- Video Tutorial / Youtube Tutorial on how to run our application from scratch and learn how to implement a new gesture to the application as well

Summary

To summarize, we see technology as being an extension of the human body and throughout this project we came to learn how vital a software like this could be. Thus, we hope in the future we grow our application to be part of the everyday human life.

REFERENCES

Deng, L.; Yu, D. (2014).
"Deep Learning: Methods and Applications"
Foundations and Trends in Signal Processing.
https://www.microsoft....DeepLearning-NowPublishing-Vol7-SIG-039.pdf

Dong, G., Yan, Y., Xie, M.: Vision-based hand gesture recognition for human–vehicle interaction.(1998)
https://www.researchgate.net/profile/Ming_Xie4/publication/2386425_Vision-Based_Hand_Gesture_Recognition_for_Human-Vehicle_Interaction/links/0c960525886596b6b9000000/Vision-Based-Hand-Gesture-Recognition-for-Human-Vehicle-Interaction.pdf
Paper Presented at the Proceedings of the International Conference on Control, Automation and Computer Vision

OpenPose
https://github.com/CMU-Perceptual-Computing-Lab/openpose

Handtrack.js
https://github.com/victordibia/handtrack.js/
Library for prototyping real-time hand detection

Gesture Recognition
https://github.com/asingh33/CNNGestureRecognizer
https://github.com/avidLearnerInProgress/hand-gesture-recognition \
Projects with examples for gestures recognition

TensorFlow Examples
https://github.com/tensorflow/examples
General examples from TensorFlow on gesture classification and object recognition

Kaggle Dataset
https://www.kaggle.com/.../hand-gesture-recognition-database-with-cnn/
Database is composed by 10 different hand-gestures

Project Repository
https://github.com/cpanican/capstone
GitHub repository containing the source code for our project