Gesture and Speech driven Operating System

Multimodal Interaction Project 2021/2022

Matteo Orsini
1795119

Fabrizio Rossi
1815023

Abstract - *We realized a proof-of-concept of an operating system which replaces the traditional mouse and keyboard interaction with gestures and speech. In order to do this, we designed and implemented a gesture recognition system, a speech recognition system and a user interface. For the gesture recognition system, we acquired a new dataset, extracted hand keypoints thanks to the MediaPipe library and trained a new model to recognize the gestures we defined. Moreover, we used several pre-processing and data augmentation techniques. Instead, for the speech recognition system, we used the free Google API with a custom mechanism to start and stop the capture of the audio signal. Finally, we designed our user interface to provide all the necessary feedback to show the user the state of the system, and display hints about how it can be operated.*

1 Introduction

Nowadays, commercial operating systems are designed to work using the traditional mouse and keyboard as input devices, but there are more natural ways to interact with our computer. Specifically, Human-Computer Interaction (HCI) is the field of research which studies the interaction between a person and a machine. In the past, the design of the input signal was based on the requirements of the devices, not considering the needs of the users, since the aim was to ease the work the computer should do. Instead, newer and more powerful devices allowed us to develop solutions which put users at the center of the design process. In fact, in the latest years numerous devices have been proposed to enrich the user experience, both for entertainment purposes but also for productivity. For example, many mediums that are often used in human-to-human communication can be used also to interact with machines, like eyes, by tracking the gaze of a person, voice, by recognizing speech, or hand gestures, which can be used to recognize specific movements or signs.

Moreover, it is also possible to combine multiple modes of interactions in order to improve the user experience, creating a so-called Multimodal Interaction system. In particular, in our work we focused on speech and gesture recognition in order to interact with an operating system to carry out the daily actions that we perform with our machines. We chose these two modalities because they allow us to enjoy a very natural and potentially faster interaction. In our case, the user replaces mouse and keyboard with a webcam in order to capture a live video feed of his body, and a microphone, in order to capture his voice.

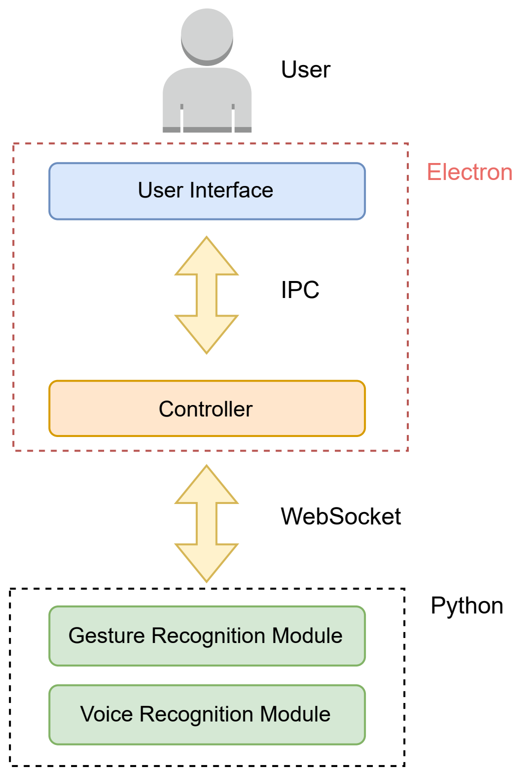

Fig. 1: Architecture of the system.

However, transforming these kind of input signals in commands which can be understood by the computer is not an easy task. In fact, input devices like mouse, keyboard or joystick, provide a digital signal which can be immediately interpreted by the system, while voice and gestures are inherently raw signals which must be first pre-processed in order to extract commands which can be directly executed. Moreover, recognition systems for these kinds of domains must address several challenges: for example, for gestures we can have occlusions or motion blur making the hand recognition hard or even impossible, and for voice, similar sounding words, which can easily be misinterpreted by the system. Due to the heavy processing needed by these complex signals, they are in general much more demanding in terms of computational resources for the system in which they run with respect to traditional input signals and this can be a problem even for not so old computers. In addition, since mouse and keyboard have been around for more than 50 years, users are familiar with some standard mappings, for example the right mouse button is usually assigned to show some options in the current context, but this is not the case for speech and gestures.

Finally, another problem which characterizes this kind of systems is the intra-class variance, i.e. the input signal may vary from person to person: for example different people may say the same word with a different accent or may perform the same gesture with a different hand pose.

So, in the next sections we will describe all the challenges we faced and the solutions we propose in order to design a mockup of an operating system that can be controlled only using voice and hand gestures, focusing on usability aspects and providing a good user experience.

2 System Architecture

As a proof-of-concept we implemented just a subset of the numerous features of a complete operating system, writing an application which runs on top of Windows.

To accomplish this we used Electron, a framework to write desktop applications using web technologies.

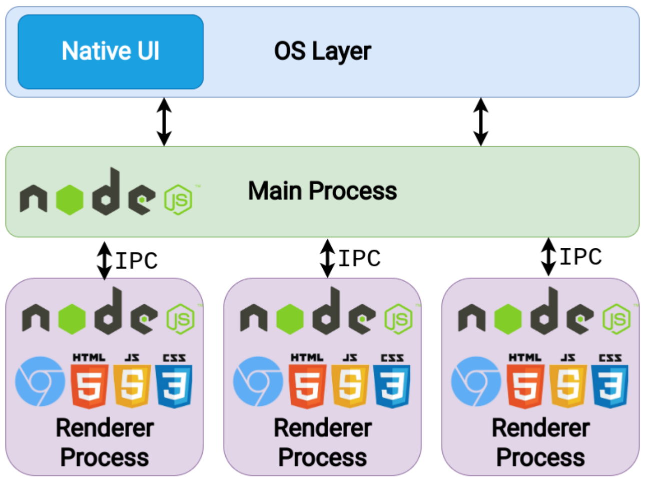

Fig. 2: Components of the Electron architecture.

In practice, Electron embeds a browser instance, i.e. Chromium, which will be launched when the user opens the application. In this context, the process responsible to manage the browser instance is called *renderer*.

Moreover, another process, called *main*, is launched by Electron inside a Node.js environment in order to communicate with the operating system APIs. A representation of the components of Electron is shown in Figure 2.

In our case, inside the browser we used TypeScript as programming language, which is a typed version of JavaScript, and the React framework, which is especially suited for interactive user interfaces.

From an high level point of view, our system architecture is composed by the following modules:

1. **User Interface**: set of visual components used in our application. The user interface will show the results of the commands issued by gesture or voice, and provide feedback to the user;
2. **Gesture Recognition Module**: captures, processes and classifies the raw video stream of both static and dynamic hand gestures;
3. **Speech Recognition Module**: captures and elaborates the audio signal from the microphone of the user when requested;
4. **Controller**: acts like an interface which manages the gesture and speech recognition modules and sends the prediction generated by the models to the user interface;
5. **Communication Bus**: allows the user interface to communicate with the gesture and speech recognition modules.

Regarding the features of the system, as we said, we implemented only a subset of all the features a complete operating system offers:

1. Launch and close an application
2. Change active application
3. Arrange the application windows in a certain layout
4. Use voice commands to start an app and shut down the system

Moreover, an operating system needs also some main applications that are provided out-of-the-box and therefore we implemented some examples of them:

1. **File explorer**: which let the user navigate through a hierarchy of folders going forward and backward or opening a file with the associated app;
2. **Photo viewer**: which let the user view a photo or navigate through a gallery of photos using gestures;
3. **Video player**: which let the user watch a video, play or pause it and toggle the full-screen mode;
4. **Text viewer**: which let the user inspect the content of a text file or simply show a text on the screen.

3 User Interface

The User Interface is one of the most important components of an operating system since it's the module that is used by users in order to understand the state of the system. Our operating system starts with a simple screen composed of some indications of the current date and time. Then, the user can open what we called *"Command mode"*, which allows the user to launch a new application, close or change the current app and arrange the layout in which the apps are displayed. This mode can be activated in any system state simply by holding the left palm in front of the camera, action that is reserved to the operating system and can't be used by other applications.

When at least one app is opened, a task bar will be shown at the bottom of the screen to display the apps that are running and the one which is currently selected. In addition, some other information is displayed in the task bar: the date and time, a hint of how to open the command mode and some feedback indicators for gesture and speech recognition.

In fact, we tried to provide several feedback for the user actions, since the user may not be aware if the system is actively detecting the user hands or voice. So, when a gesture is detected from the left or right hand, or when the system is listening for voice commands, the corresponding indicator will be highlighted to inform the user that the system is currently capturing a gesture or a voice command. In Figure 3 we can see how the command mode and task bar look like. Moreover, as a form of redundancy, we also added an auditory feedback when the system starts or stops listening to the microphone.

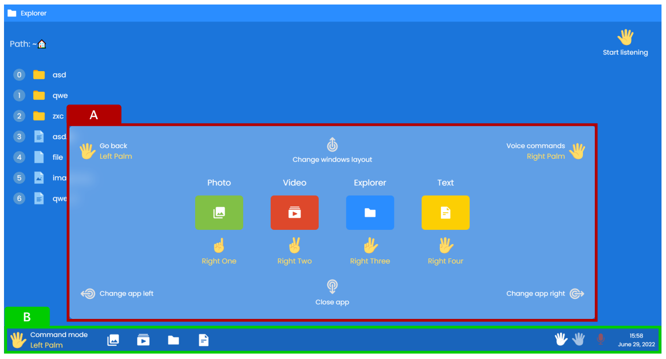

Fig. 3: Interface of the system which shows the command mode (A) and task bar (B).

In addition, we provided a feedback also during the arrangement of applications in a layout: we can divide the screen in maximum 4 sectors and assign an application to each one of them, as shown in Figure 4. Here, the feedback is provided through a preview of the layout as understood by the system, which can optionally be changed before confirming.

An instant feedback is provided also for voice commands: in fact we created a mechanism to show a preview of the text which is being recognized by the speech recognition module before the user has finished to say the command.

Another important aspect of the design of our interface are the *indicators*, showing which gestures, and associated actions, can be performed in the current context.

Finally, when we designed the used interface we tried to prevent bad things from happening asking confirmation for more critical actions, as a form of grounding. In fact, when a user tries to close an application using the swipe down gesture, before actually closing the app, we show a popup asking for confirmation.

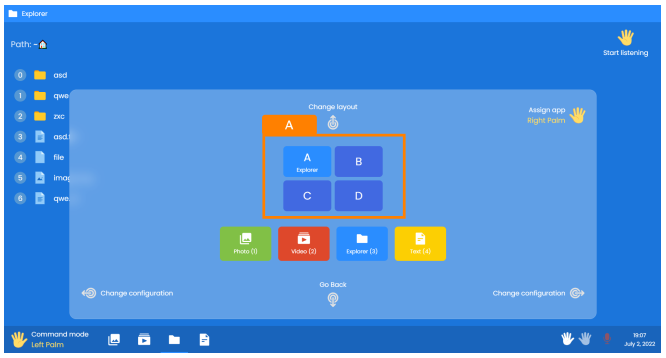

Fig. 4: Interface used to change the layout of the applications with a preview (A).

4 Gesture Recognition Module

The Gesture Recognition Module is the one in charge of capturing the video feed from the webcam and detecting when a hand gesture is performed, in order to classify the action intended by the user.

We designed our system to recognize 9 gestures: 5 static hand gestures and 4 dynamic. In particular, the static gestures are the numbers from 0 to 5 that need to be held as still as possible, and the dynamic gestures are hand movements in form of swipes into four directions: left, up, right and down.

In this case, since we designed peculiar gestures to detect and no method or dataset was already available for our needs, we created a dataset of videos specifically for our project and fit a Deep Learning model on the hand landmarks extracted from it.

4.1 Dataset

The dataset was acquired using two different models of commercial webcams, in two different scenarios and with a total of 5 subjects performing the requested gestures.

During the recording phase, we didn't need to care about recording videos in different light conditions or different environments since we used a very robust pre-trained model to extract landmarks, that could handle all these situations by itself.

Instead, we focused on recording videos with different wrist rotations and finger positions. Indeed, as we said previously, two different people may perform the same gesture with two different finger positions, e.g. making the *one* gesture with the thumb or with the index finger.

This concept is even more accentuated with dynamic gestures, like swipes, in fact there isn't any standard way to perform them. For this reason we defined a way, shown in Figure 5, hard to be confused with a static gesture, i.e. placing the hand vertically with the thumb towards the user for left and right swipes, and horizontally for top and bottom swipes.

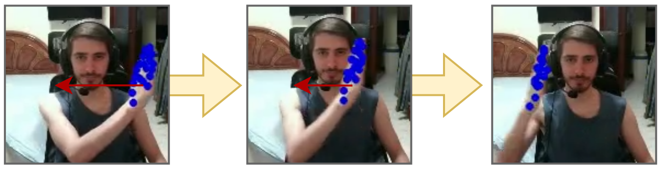

Fig. 5: Illustration of how we defined the swipe gesture.

Moreover, each gesture was recorded first with one hand and then with the other, and the dynamic gestures were also performed at different speeds.

Then, we cut the videos making the start and end of each sample coincide with the start and end of the gesture present in it, removing any other hand movement, like raising the hand to prepare to perform the gesture.

Finally, in order to let the model discriminate between a gesture and a non-relevant movement of the hand, we recorded several samples with hand movements not associated to any gesture we came up with and put them in a single class called "*none*".

In Table 1 we reported the number of samples captured for each class and even if the number of samples of the dataset may seems low, we performed several data augmentation techniques to increase the number of variations, finding out that the number is sufficient to train a robust model.

Class	**Samples**
none	73
zero	14
one	22
two	23
three	22
four	22
palm	22
swipe up	22
swipe down	21
swipe left	26
swipe right	22
Total	289

Table 1: Number of samples for each gesture.

4.2 Pre-processing

In this section we will describe the pre-processing stages applied to our dataset and in order to follow better the steps we will reference Figure 6.

First of all, we gather the video frames from the webcam using the OpenCV library for Python (i.e. step A in the figure). Then, we extract the landmarks from the hand present in the frames (i.e. step B). In order to do this, we used the MediaPipe Hands library provided by Google for Python, which can detect hands and identify 21 landmarks for each one of them. This library has some parameters that can be tuned and in particular we set the maximum hands to detect to 1, since in the video samples of our dataset there is only one hand at a time, the minimum confidence threshold for a hand to be recognized at 0.5 and we set the static images mode to false. The latter means that the MediaPipe framework will treat the input data as a video, and once the hand is detected, it is only going to track it in the following frames, making the computation much faster than detecting it for every frame.

The extracted landmarks are returned as a tuple (x, y, z) for each of the 21 landmarks and the coordinates are normalized by MediaPipe in the range $[0, 1]$ with respect to the frame width and height, which was resized to 320×180.

After the extraction of the landmarks we proceed with a specific stage only for samples of none gestures (i.e. step C). This is motivated by an assumption we made: our model in general should output the *none* gesture much more frequently with respect to the other classes, because not all the hand movements, especially for us Italians, are directed to issue commands to the system but are actually correlated to other human actions, like scratching your head or gesticulation. Thus, we designed the system in order to prefer that the user repeats the gesture if the system is not sure on the result of the recognition, i.e. the output has a low confidence. This prevents the system from performing a non desirable action which could impact the interaction of the user. So, we augmented the none samples by cutting some random pieces of the none videos, using a specified minimum length, and adding them back in addition to the training set increasing the number of samples of the none class.

Now, all the frames pass through a sampling procedure (i.e. step D): we sample a fixed number of them and we batch them to make the training process faster and more stable. In fact, batching the samples would not be possible using all the frames since each video has a variable duration. In practice, we sample them by taking evenly spaced frames from the whole video.

Then, we perform another specific transformation but this time only for static gestures (i.e. step E). This is motivated by the fact that a slight rotation of the hand should not impact the final prediction. So, we augmented the dataset by adding some duplicated versions of the static gestures, applying a small random rotation around the roll axis, but being careful not to touch dynamic gestures since in that case a rotation would actually change the direction of the swipe.

Finally, a normalization step (i.e. step F) was performed for all the samples in the dataset, in order to transform the landmarks in a representation that can be easily interpreted by the model: this is the part that makes efficient use of our small dataset, making it much more valuable. In this step we normalize the size of the hand in the range $[0, 1]$, by comput-

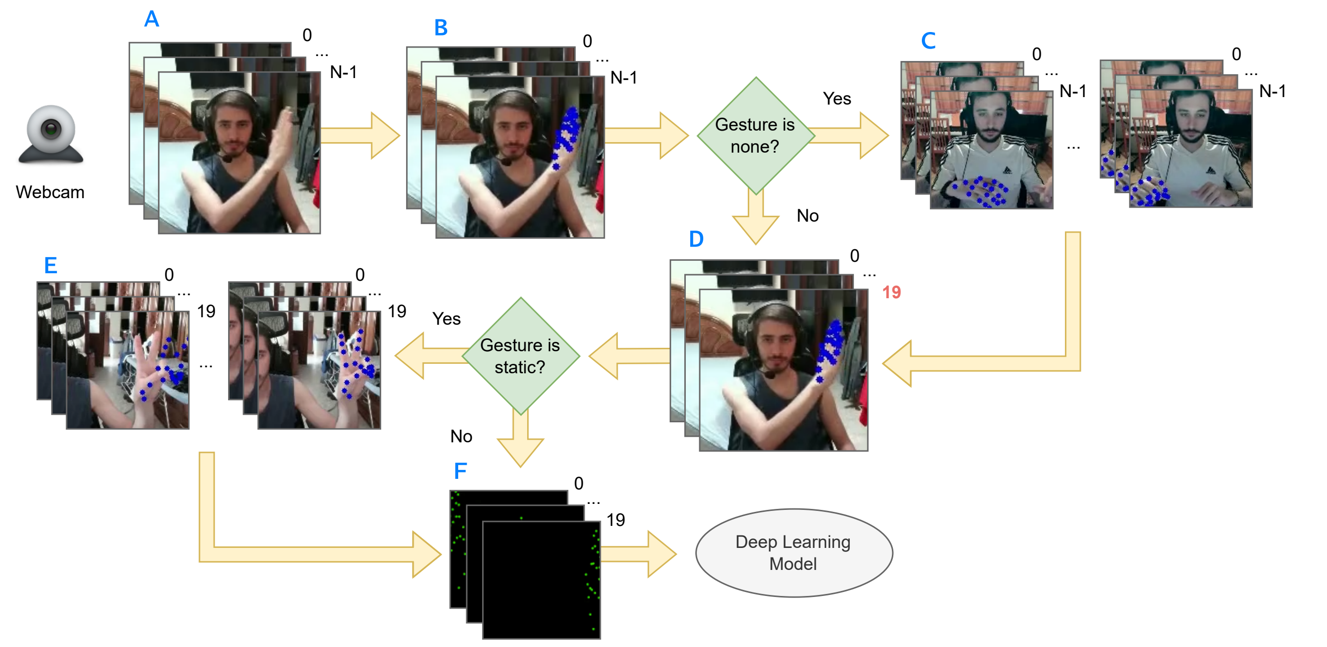

Fig. 6: Pipeline of our pre-processing stages: acquire frames from the webcam (A), extract landmarks (B), none augmentation (C), sampling (D), rotation augmentation (E) and normalization (F).

ing the minimum and maximum coordinates that are reached by any of the keypoints detected for the entire duration of each single video clip. In this way we can reach translation and zoom invariance, eliminating the need to record videos where we change the hand position in the frame or perform the same gesture by getting closer or further from the webcam.

4.3 Architecture

The Deep Learning model, whose architecture is shown in Figure 7, was implemented with the TensorFlow library for Python and it is composed by 3 Long Short-Term Memory (LSTM) layers with respectively 16, 32 and 16 units in output with the ReLU activation function, followed by 2 Dense layers both with 16 output neurons and the ReLU activation function. Finally, we have another Dense layer with a number of output neurons equal to the number of gestures we want to identify, i.e. 10 in our case, and the softmax activation function to obtain a probability distribution of the classes in output.

For our model we used LSTM layers because they are more suitable for sequences of data thanks to their feedback connections. These connections allow the model to output the next prediction based on information gathered during the previous iterations of the model. LSTMs can also be less affected by the vanishing gradient problem which characterize the training of traditional Recurrent Neural Networks (RNNs).

The model has only about 14.000 trainable parameters thanks to the fact that we only use the 21 landmarks offered by the MediaPipe library, ensuring that the model can be run on less powerful devices.

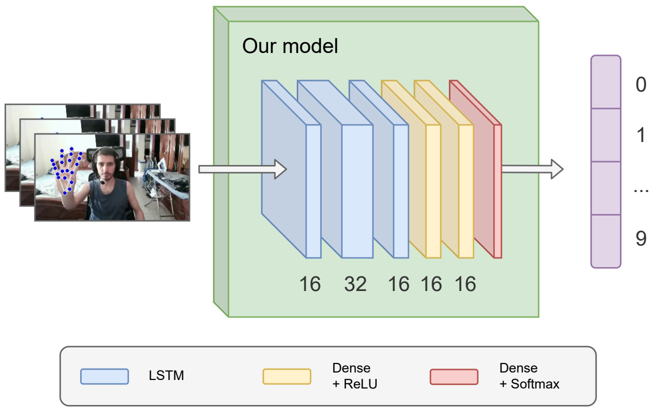

Fig. 7: Architecture of the model.

4.4 Results

Before starting the training process, we divided 70% of the dataset into training set, to perform the fitting of the model, and the remaining 30% of the dataset into the test set, to evaluate the performance of the model. Then, we performed the augmentation of the none samples, as described in Section 4.2, 5 times for each sample in the training set, with a minimum length of 40 frames, making the number of none samples 5 times bigger. Moreover, the rotation of the static gestures was performed with a degree of rotation randomly generated between -0.5 and 0.5 radians, 50 times for each sample of the training set. Finally, after the steps described, we obtained a dataset whose distribution is shown in Table 2, which has been pre-processed as stated in Section 4.2, sampling 20 frames for each exemplar.

Thus, we trained our model for 35 epochs with a batch

Class	Samples
none	15249
zero	510
one	765
two	816
three	816
four	765
palm	765
swipe up	816
swipe down	765
swipe left	918
swipe right	765
Total	22950

Table 2: Number of samples for each class after the dataset augmentation.

size of 32, using the Adam optimizer with a learning rate of 0.001 and the categorical cross entropy as loss function, on a GTX 1060 3GB. The results obtained shown in the confusion matrices reported in Figure 8, denote a very robust model where almost all the samples are correctly classified both in the training and test set.

In Figure 9, we can see the training and test accuracy of the model during the training phase. From the plot we can notice that the training accuracy is always increasing, which indicates that the model learned the task, since we have a value which is almost 1. Moreover, inspecting the test accuracy we can see how the model is performing with data that has never saw during the training phase and also in this case the accuracy is about 0.96, meaning that the model is classifying correctly the 96% of the test samples.

4.5 Inference

Once trained the model, it's time to use it with a real-time webcam feed. This is a challenging task since the model only saw videos of gestures perfectly cut, but the webcam feed is a continuous stream of frames and therefore, there is the need to tokenize the gestures, i.e. to identify when the gesture starts and when it ends. We addressed this challenge using a sliding window implemented using two landmarks accumulators, one for each hand.

So, when a new frame arrives from the webcam, we compute the landmarks for both hands thanks to the MediaPipe library, which indicates also the handedness of the landmarks detected, i.e. if the hand is left or right. Then, we put the landmarks detected in the accumulator based on the hand to which they corresponds to. So, when the size of the accumulator for one hand is equal to the predefined window size, which we set to 20, we use the model to output a classification for the landmarks present in that hand accumulator. Before doing so, however we need to pre-process the landmarks in a similar way to the one applied during training: we sample the landmarks corresponding to 20 frames from the window and then normalize them. Finally, when the window has accumulated enough frames to slide, we shift it by a stride value which we set to 5.

This means that at inference time we have two parameters, the window size and the stride of the window, which directly influence the performance and recognition speed of the system. In fact, if we have a webcam that generates F frames per second, a window size of length W and a stride S, we have that the first prediction will be outputted after $\frac{W}{F}$ seconds, instead in the general case we will have a new prediction each $\frac{S}{F}$ seconds, since we keep $W-S$ frames from the accumulator of the previous prediction. In practice, we also need to account the processing time of the entire pipeline, which is about 1.3 seconds, that needs to be added to the time previously formulated.

Finally, during the implementation and test of the window mechanism, we noticed that we could implement another technique to improve the robustness of our model: we implemented a cooldown mechanism in which after a prediction different from *none*, we wait 2 seconds before using the model again, but we still accumulate the landmarks during this period. This strategy is used to let the user change the hand position without triggering any new action which wasn't intended by him.

5 Speech Recognition Module

The Speech Recognition Module is used to turn the speech captured from a microphone into text which will be then used to execute actions inside the system. In order to implement this module we used the Speech Recognition Python library. The latter is able to perform speaker independent speech recognition with few lines of code thanks to the several engines and APIs, both online and offline, integrated in it. Moreover, thanks to this library we are able to easily change the engine or API used to interpret the speech, which makes it very convenient in our modular system.

In particular, we focused on two free APIs that don't require any particular processing before using them: Google and Sphinx. We tested both of them and the Google API seems to perform better in most scenarios, recognizing text more accurately than Sphinx, but in contrast the latter works in offline mode, so it doesn't require any Internet connection in order to perform the recognition. Taking into account these considerations, we opted to use the Google API because we preferred to have a more accurate prediction even if an Internet connection is needed to output it.

However, since we chose the limited free version, which is not on the same level as their paid Cloud Speech solution, we incurred in some problems. For example, we initially planned to use the word *"launch"* in order to open an application, but we noticed that it was easily mistaken for other words, so we decided to use the word *"start"* which is better

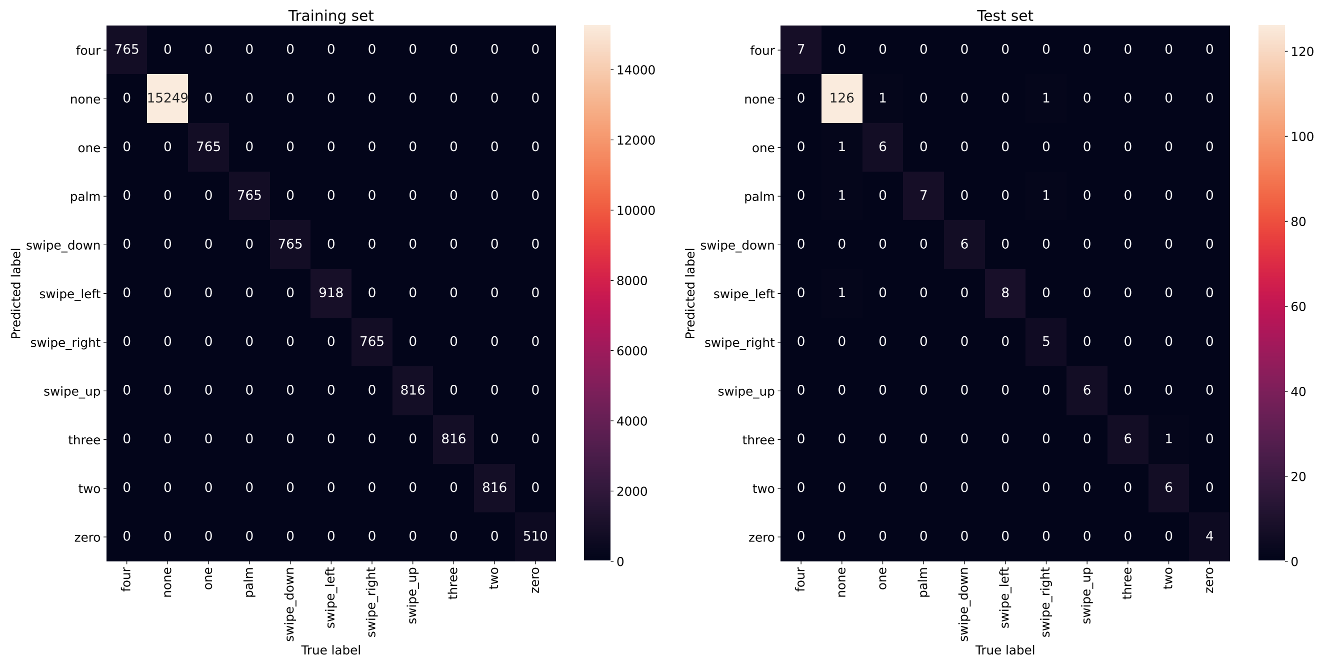

Fig. 8: Confusion matrix of the gesture recognition module.

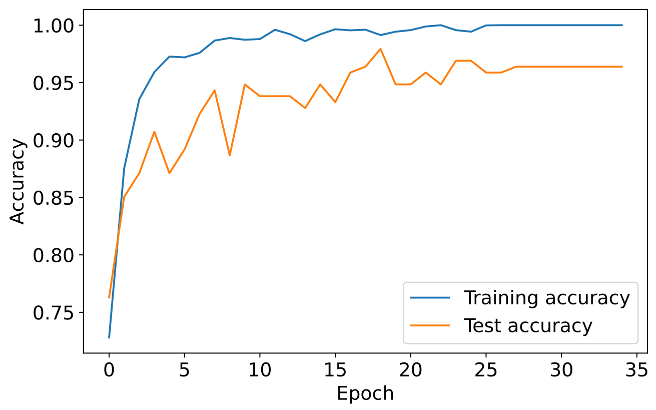

Fig. 9: Training and test accuracy of the model during training.

recognized by the Google API.

In addition, out of the box the speech recognition library is going to output the predicted text when it detects that the user is not talking anymore. In our case, however, we don't want to output a prediction when the user remains in silence, but when the user performs a particular action, i.e. lowering the palm. So, we implemented a mechanism which will start registering audio samples from the microphone when the user request to start the speech recognition process, and stop the recording when the user requests it to stop, as in a user initiative dialogue strategy. This mechanism allows us also to maintain a high level of privacy since we are not always listening to the user microphone, practice not always accepted by users. In more details, every time the recognition process is requested, a new thread is spawned, which will be in charge of recording the audio samples from the microphone, and when the user requests to stop the process, the thread will provide all the audio samples captured in this time interval to the Google API which will output the text.

Moreover, as a feedback, we also created a mechanism to show a preview of the text said by the user until that moment. In the implementation, when a certain number of seconds has passed, i.e. 2 seconds in our case, we created a new thread, which will simply process the text in parallel to avoid to stop the capturing of the audio samples, and when the process is completed, it will invoke a callback to pass the text to the controller first, and then, to the user interface.

6 Controller and Communication Bus

The Controller is the component in charge of managing the gesture and speech recognition modules and sending the prediction coming from the two Python modules to the UI. Instead, the communication bus is the software channel which will convey information from the Python modules to the UI and viceversa.

In our system, the code which implements the controller is splitted between the main process of Electron and some components of the React client.

Inside the main process we can find the listeners for the events from Python and from the UI, which are delivered through the communication bus. Moreover, the main process is able to interface with the system APIs which allow us to start or stop the Python process and read the content of a file when a user requests it.

Instead the React client, not only is in charge of display the UI contents, but it also handles the mapping between gestures and actions, registering a callback for each of them using a priority system. Moreover, it will parse the text received from the speech recognition module, checking if the text matches the structure of one of the predefined voice commands, and executing the corresponding action if that's the case. In addition, we also implemented a system to simulate gestures using the keyboard, in order to test the system without using them in the early stages of development.

Regarding the communication bus, we need two different bilateral communications: one between the UI and main process, and the other between the main process and the Python modules. The first one was implemented using the Inter-Process Communication (IPC) methods provided by Electron, which uses a publisher-subscriber mechanism in order to exchange messages in the form of events. The same pattern is used for the second bilateral communication between the main process and the Python modules, but this time we needed something to let two different processes written in two different programming languages communicate. So, we used WebSockets, which allow us to establish a low latency communication thanks to web technologies, in a client-server modality, where the main process is the server and Python is the client. In order to use this technology we used the Socket.IO library which abstracts the concept of WebSockets and make the development with such technology easier for both languages.

7 Conclusions

In summary, in our project we realized a proof-of-concept of an operating system which allows a user to interact with it using gestures and speech. In order to accomplish this we created three modules: one for gesture recognition, another for speech recognition and the last one for the user interface. All of them were managed by a controller which used two different communication bus to exchange messages among them. For the gesture recognition module, a phase of data acquisition was necessary, in which we acquired several video samples for the gestures we defined to use in our system. Then, we pre-processed and augmented the landmarks extracted from the acquired video samples and obtained a bigger and diversified dataset, which allowed us to perform a training phase with good results. Finally, we had to address the challenges related to the use of the model in a real scenario. For this reason, we devised a sliding window mechanism in order to process the raw video stream and output a prediction with a cooldown mechanism. Instead, for the speech recognition module we were able to use the free Google APIs for speech thanks to an external Python library. However, we needed to implement a recording mechanism with different threads to process the incoming audio signals when requested by the user. Finally, in the user interface module we tried to address the issues related to the gesture and speech modes focusing on usability. For example, we provided several feedback to the user and tried to add as many hints as needed for the users of our system.

In conclusion, we would like to address the advantages and disadvantages of using a system of this kind with respect to a traditional mouse and keyboard based system. The advantages are mostly related to the fact that users doesn't need to be in any physical contact with the computer in order to operate it: in fact gesture and speech signals are available also at a certain distance with commercial webcam and microphone, while the same is not true for mouse and keyboard, which are input devices intended to be used close to the computer and in general with a desk. Instead, our system can be used in any scenarios in which users don't have access to mouse and keyboard, for example during a presentation or while watching a movie on the sofa. Moreover, voice is even a more natural and faster way to interact with the computer since it is the main interaction modality used by humans. However, we need to consider that voice may not be ideal in a loud environment.

In addition, we can also consider this type of systems for applications that require more interactivity with their users: an example can be videogames offering a new experience for the players. However, in our opinion, this cannot be adapted easily for productivity applications that have a huge amount of features as they would require too many gestures to activate them all. In fact, semaphoric gestures don't satisfy well the use cases of productivity apps which would benefit more from gestures for the direct manipulation of virtual objects, since they offer a more natural interaction than mouse and keyboard in those contexts.