

# Gesture Recognition Tutorial

## Introduction

Welcome to the Gesture Recognition Tutorial! In this tutorial, we'll guide you through using a Gesture Recognition library in C++. This library empowers users to effortlessly create models capable of recognizing gestures in images, live video streams, or recordings. This library can be your go-to for lots of cool applications, but just to name a few: you can use it to script American Sign Language (ASL) signs in videos or spice up your presentation game with hands-free controls.

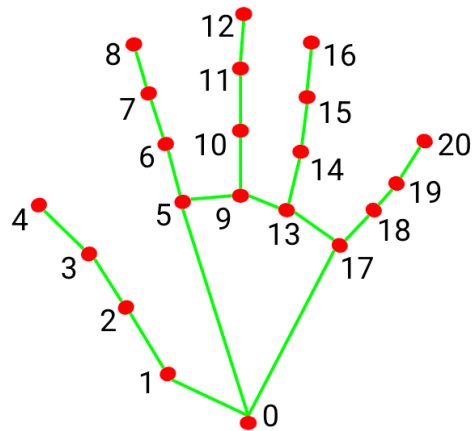
## Who Should Read

This tutorial is designed for developers and enthusiasts interested in creating gesture recognition models. While a basic understanding of C++ and Machine Learning is beneficial, we'll guide you through each step, assuming minimal prior knowledge. Whether you're a hobbyist looking to explore fun applications or a professional seeking hands-on experience with gesture recognition, this tutorial is for you. ***For more detailed information about each step, check out our manual.***

*Note: As of December 2023, this library is exclusively tested and compatible with MacOS. Our library is designed to run alongside Google's MediaPipe Libraries. We've set up a server that leverages Google's MediaPipe and has been configured to work for MacOS as a default. The server plays a vital role in obtaining hand landmarks, a crucial step in our program that simplifies the classification problem. Instead of dealing with numerous pixels for each image, our approach involves working with 21 landmarks for each image. It's worth noting that, for simplicity, our current model supports the classification of a single-hand gesture.*

## About Google's Mediapipe:

MediaPipe Solutions provides a suite of libraries and tools for quick integration of artificial intelligence (AI) and machine learning (ML) techniques into your applications. Notably, the MediaPipe Hand Landmarker task enables the detection of 21 hand landmarks in image coordinates (x, y, z) for each detected hand, offering reliable information about hand locations in images.



- |                       |                       |
|-----------------------|-----------------------|
| 0. WRIST              | 11. MIDDLE_FINGER_DIP |
| 1. THUMB_CMC          | 12. MIDDLE_FINGER_TIP |
| 2. THUMB_MCP          | 13. RING_FINGER_MCP   |
| 3. THUMB_IP           | 14. RING_FINGER_PIP   |
| 4. THUMB_TIP          | 15. RING_FINGER_DIP   |
| 5. INDEX_FINGER_MCP   | 16. RING_FINGER_TIP   |
| 6. INDEX_FINGER_PIP   | 17. PINKY_MCP         |
| 7. INDEX_FINGER_DIP   | 18. PINKY_PIP         |
| 8. INDEX_FINGER_TIP   | 19. PINKY_DIP         |
| 9. MIDDLE_FINGER_MCP  | 20. PINKY_TIP         |
| 10. MIDDLE_FINGER_PIP |                       |

Hand Landmarks definition (Source: [Mediapipe](#))

## Acknowledgment:

This project takes inspiration from a similar gesture recognition library in Python, the GRLib, whose algorithm has been nicely documented [here](#).

## About the Authors:

This library is a collaborative effort developed as the final group project for the COMS W4995: Design Using C++ course in Fall 2023, taught by Bjarne Stroustrup. The authors behind this library are Yana Botvinnik, Maitar Asher, Noam Zaid, Elvina Wibisono, and Elifia Muthia.

## Workflow:

Before diving into the detailed steps, the following workflow outlines the essential steps to effectively use the library.

### 1. Set Up the Environment:

Ensure your development environment is set up with the necessary dependencies and configurations.

### 2. Set Up the MediaPipe Server:

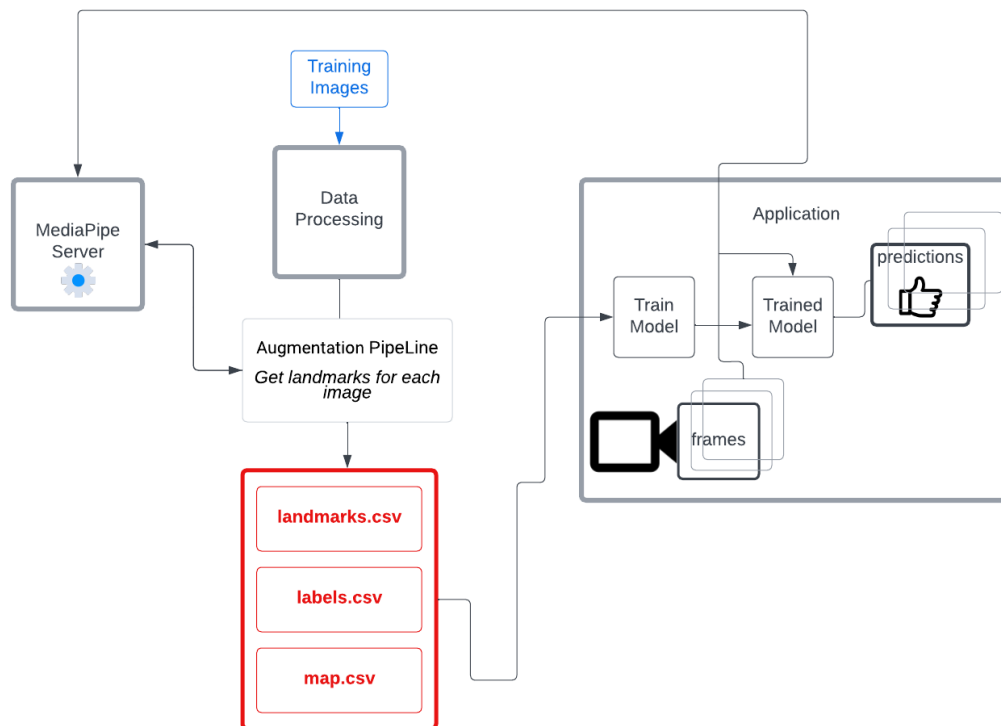
Configure and initiate the MediaPipe server, essential for obtaining hand landmarks crucial for model training.

### 3. Process Training Data:

Prepare your training data (images) by employing our processing application. This step entails extracting landmarks (from the server) and labels for each training image, creating a different structured representation of the data.

#### 4. Train the Model and Make Predictions:

Train your model using the preprocessed data. Once trained, you can leverage the model to make predictions on images, live videos, or recorded videos, showcasing the power of your gesture recognition model.



The workflow begins by configuring and initiating the **MediaPipe server**, a pivotal component responsible for obtaining hand landmarks. This step is essential for both model training and subsequent predictions. Moving to the **data processing** phase, the training data undergoes preparation using our specialized application. This involves extracting landmarks from the server and associating them with labels for each training image, resulting in a uniquely structured representation of the data. With the preprocessed data in hand (saved in csv files), **the model is trained** to recognize intricate gesture patterns. Once trained, the model becomes a potent tool for making predictions on various inputs, including images, live videos, or recorded videos. During the **prediction phase**, each image frame is sent to the server to obtain landmarks, and these landmarks are subsequently fed to the model for making predictions.

# 1 - Set Up the Environment

*Copy and paste the subsequent commands into your Mac terminal. Commands c-f are essential for configuring the MediaPipe server.*

**a. Clone the Gesture-Recognition repository**

```
$ git clone  
https://github.com/maitarasher/Gesture-Recognition.git
```

**b. Install Homebrew (a package manager to install library dependencies)**

**c. Install OpenCV**

```
$ brew install opencv
```

**d. Install bazelisk**

```
$ brew install bazelisk
```

**e. Install opencv@3**

```
$ brew install opencv@3
```

**f. Install ffmpeg**

```
$ brew install ffmpeg
```

**g. Install Numpy**

```
$ brew install numpy
```

# 2 - Set Up the MediaPipe Server

As a reminder: the MediaPipe Server is in place to extract hand landmarks from images, serving the dual purpose of processing the training data for the model and processing real-time image frames to be able to make predictions.

To run the Mediapipe server locally, follow these steps:

**a. Create a new folder within the src directory , and move to the newly created directory.**

```
$ mkdir src/dependencies  
  
$ cd src/dependencies
```

**b. Clone the modified Mediapipe repository from this GitHub link. In your terminal, navigate to the desired folder for the clone and execute the command:**

```
$ git clone https://github.com/elifia-muthia/mediapipe.git
```

**c. Move into the newly cloned Mediapipe repository**

```
$ cd mediapipe
```

*Now, you have a local copy of the customized Mediapipe repository.*

*Locate the server code in the following path:*

*mediapipe/mediapipe\_samples/sample/mediapipe\_sample.cc.*

*The server operates on port 8080 by default.*

**d. Build the file**

*You only need to build the file once.*

Execute the following command (in the root directory of the Mediapipe repository, where you are currently located). This might take about 5-10 minutes to complete.

```
$ bazel build -c opt --define MEDIAPIPE_DISABLE_GPU=1  
mediapipe/mediapipe_samples/mediapipe_sample:mediapipe_samp  
le
```

**e. Execute the following command to run the server**

```
$ GLOG_logtostderr=1  
bazel-bin/mediapipe/mediapipe_samples/mediapipe_sample/medi  
apipe_sample  
--calculator_graph_config_file=mediapipe/graphs/hand_tracki  
ng/hand_tracking_desktop_live.pbtxt
```

*You should be seeing something like this*

```
(base) maitarasher@Maitars-MacBook-Pro mediapipe % GLOG_logtostderr=1 bazel-bin/mediapipe/mediapipe_samples/  
mediapipe_sample/mediapipe_sample --calculator_graph_config_file=mediapipe/graphs/hand_tracking/hand_trackin  
g_desktop_live.pbtxt  
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR  
I0000 00:00:1701694936.063294      1 mediapipe_sample.cc:123] Initialize the calculator graph.  
_Port: 8080
```

## 3 - Process Training Data

**a. Clone our Gesture-Recognition repository from this [GitHub link](#). In your terminal, navigate to the desired folder for the clone and execute the command:**

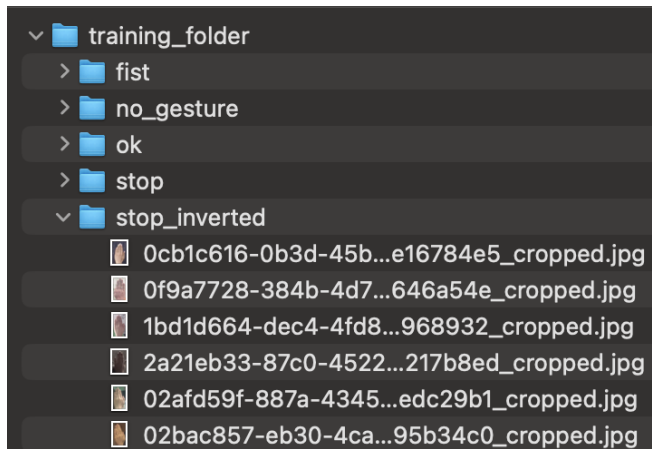
```
$ git clone  
git@github.com:maitarasher/Gesture-Recognition.git
```

**b. Move into the newly cloned Gesture-Recognition repository**

```
$ cd Gesture-Recognition
```

**c. Have your dataset of different gestures images ready**

Ensure that all the training images are organized like the following directory format (each class of gestures is in a separate subfolder)



**d. Customize your augmentation pipeline as needed**

The processing code (processing.cpp) which includes the augmentation pipeline is available at the following path: Gesture-Recognition/processing\_data.

Modify this part of the code if you wish to customize the augmentation pipeline:

C/C++

```
// (2) Create Pipeline structure

// Create a pipeline with 1 hand
Pipeline my_pipeline(1);

// Add stages to the pipeline (brightness, rotation)
my_pipeline.add_stage(40.0, 0.0);
my_pipeline.add_stage(40.0, -45.0);
my_pipeline.add_stage(40.0, 45.0);
my_pipeline.add_stage(40.0, 90.0);
```

Feel free to incorporate as many stages as you desire into your pipeline. Each stage represents an additional augmentation of your data and requires two parameters: the degree of brightness and rotation to apply. The model will each each augmented image to the server to obtain landmarks **until a valid landmark is identified**.

**Why consider tweaking the pipeline?** Certain images in your training data might

be excessively dark, and brightening them can enhance the MediaPipe server's accuracy in obtaining landmarks. Additionally, the positioning of some hands in images may pose challenges for the MediaPipe server's initial detection; in such cases, rotating the images can be beneficial. Making these adjustments has the potential to significantly **improve the accuracy rates of your model**.

**e. Build the processing application:**

A CMakeLists.txt has been provided to build *processing.cpp*. It can be found in *Gesture-Recognition/processing\_data* folder

```
$ cd processing_data
$ mkdir build
$ cd build
$ cmake ..
$ make
```

**f. Compile the processing application** to generate landmarks representation of your data

```
$ ./processing <training_images_dir_path> <output_folder>
```

For Example:

```
$ ./processing /Desktop/training_folder ../data
```

Ensure you provide the following command line arguments:

- `<training_images_dir_path>`: Path to your training images from step c
- `<output_folder>`: Path to save the representation of your data (folder should exist).

Upon execution, three files will be created in `<output_folder_path>`:

**labels.csv**: Class labels for each hand landmark identified from the image. Each line corresponds to the entries in Landmarks.csv.

**landmarks.csv**: Landmarks for each hand found in the image. Each line contains 21 (x, y, z) coordinates separated by semicolons (;). Each line corresponds to the entries in labels.csv.

**map.csv:** The key listing the string label corresponding to each numeric class label found in labels.csv. This map is used to convert predicted numeric classes by the classifier back to their string labels.

## 4 - Train the Model and Make Predictions

In this tutorial, we will walk through the `asl_application.cpp` example located at `Gesture-Recognition/gesture_asl`. By doing so, we'll equip you with a template that you can employ for your own purposes.

### a. Connect to MediaPipe Server

- Include necessary libraries and establish a connection with the MediaPipe server.
- Relative code can be found at `Gesture-Recognition/src/mediapipe_client/mediapipe_client.cpp`

```
C/C++
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <opencv2/opencv.hpp>
#include <arpa/inet.h>
#include <regex>
#include "../src/feature_extraction/pipeline.hpp"
#include "../src/feature_extraction/stage.hpp"
#include "../src/load_data/folder_loader.hpp"
#include "../src/load_data/save_and_read.cpp"
#include "../src/data_classes.hpp"
#include "../src/mediapipe_client/mediapipe_client.hpp"
#include "../src/classifier/knn.hpp"

const int PORT = 8080;
const char* SERVER_IP = "127.0.0.1";

int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    int clientSocket = connectToServer(PORT, SERVER_IP);
    if (clientSocket == -1) {
        return -1;
    }
}
```



```

// rest of code
// ....
// ....

close(clientSocket);
return 0;
}

```

## b. Load Saved Data

- Load the data saved in Step 3 from CSV files.
- Relative code can be found at  
[Gesture-Recognition/src/load\\_data/save\\_and\\_read.cpp](#)

C/C++

```

int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data
    std::vector<Hand_Landmarks> all_images_landmarks_from_csv =
readFromCSV("../gesture_asl/data/asl_landmarks.csv");
    std::vector<float> all_labels_from_csv =
readLabelsFromCSV("../gesture_asl/data/asl_labels.csv");
    std::unordered_map<float, std::string> stringLabelMap =
readMapFromCSV("../gesture_asl/data/asl_label_map.csv");

    // rest of code
    // ....
    // ....

    close(clientSocket);
    return 0;
}

```

## c. Train the model

- Choose the appropriate model for your application. Each model returns the model and the accuracy of the model. Check the accuracy before

proceeding to predictions.

KNN: Call KNN\_build, specify K\_number (should be the number of classes)

```
C/C++
int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data
    // ....

    // (3) Train the KNN model
    auto [knn,knn_accuracy] = KNN_build(all_images_landmarks_from_csv,
all_labels_from_csv, K_number);

    // rest of code
    // ....
    // ....

    close(clientSocket);
    return 0;
}
```

SVM: Calls SVM\_build with svm.autoTrain(), if parameters are already known they can be changed inside the svm.hpp

```
C/C++
int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data
    // ....

    // (3) Train the SVM model
    auto [svm,svm_accuracy] = SVM_build(all_images_landmarks_from_csv,
all_labels_from_csv,num_classes);

    // rest of code
    // ....
    // ....
}
```

```

close(clientSocket);
return 0;
}

```

DTree: Calls DTree\_build with max\_depth parameter.

```

C/C++
int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data
    // ....

    // (3) Train the Decision Tree model
    auto [dtree, dtree_accuracy] = DTree_build(all_images_landmarks_from_csv,
all_labels_from_csv, depth_tree);
    // rest of code
    // ....
    // ....

    close(clientSocket);
    return 0;
}

```

#### d. Make Predictions and Return Script

For KNN use findNearest()

```

C/C++
int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data

```

```

// ....

// (3) Train the model
// ....

// (4) Make predictions
cv::VideoCapture capture;
capture.open(0);
bool grab_frames = true;

int frameWidth = static_cast<int>(capture.get(cv::CAP_PROP_FRAME_WIDTH));
int frameHeight = static_cast<int>(capture.get(cv::CAP_PROP_FRAME_HEIGHT));

cv::namedWindow("ASL Application", cv::WINDOW_NORMAL);
cv::resizeWindow("ASL Application", frameWidth, frameHeight);

while (grab_frames) {
    cv::Mat inputImage;
    capture >> inputImage;

    std::vector<Hand_Landmarks> landmarks;
    bool success = getLandmarksFromServer(clientSocket, inputImage,
landmarks);
    if (success == false) {
        return -1;
    }

    std::string text;
    for (auto& lm : landmarks) {
        cv::Mat result;
        cv::Mat input_cvMat(1, 63, CV_32F);
        input_cvMat = lm.toMatRow();
        knn->findNearest(input_cvMat, 3, result);
        float predict = result.at<float>(0, 0);
        // convert class prediction number into string class label
        text = stringLabelMap[predict];
    }
    // rest of code
    // ....
    // ....

    close(clientSocket);
    return 0;
}

```

For SVM and DTree use model.predict()

C/C++

```
int main(int argc, char* argv[]){

    // (1) create a client socket and connect to the MediaPipe Server
    // ....

    // (2) Load the data
    // ....

    // (3) Train the model
    // ....

    // (4) Make predictions

    // ....
    // ....

    std::string text;
    for (auto& lm : landmarks) {
        cv::Mat result;
        cv::Mat input_cvMat(1, 63, CV_32F);
        input_cvMat = lm.toMatRow();
        svm->predict(input_cvMat, result);
        float predict = result.at<float>(0, 0);
        // convert class prediction number into string class label
        text = stringLabelMap[predict];
    }

    // rest of code

    // ....

    // ....

    close(clientSocket);
```

```
return 0;  
}
```

## 5 - Running the ASL Application Example

You can find our example implementation of our library for ASL Alphabet Recognition in the `gesture_asl` folder of the repository! **Make sure that you have the Mediapipe server running before you execute the following commands.** Here, we will walk you through on how to run the ASL application. We have already run our images through the processing script and obtained the corresponding **labels.csv**, **landmarks.csv**, and **map.csv** that are now ready for training. You can find these files in the `data/asl` directory.

Now, from the root directory, navigate into the `gesture_asl` folder where the ASL application is hosted.

```
$ cd gesture_asl
```

You will need to first build the application. Execute the following commands:

```
$ mkdir build
```

```
$ cd build
```

```
$ cmake ..
```

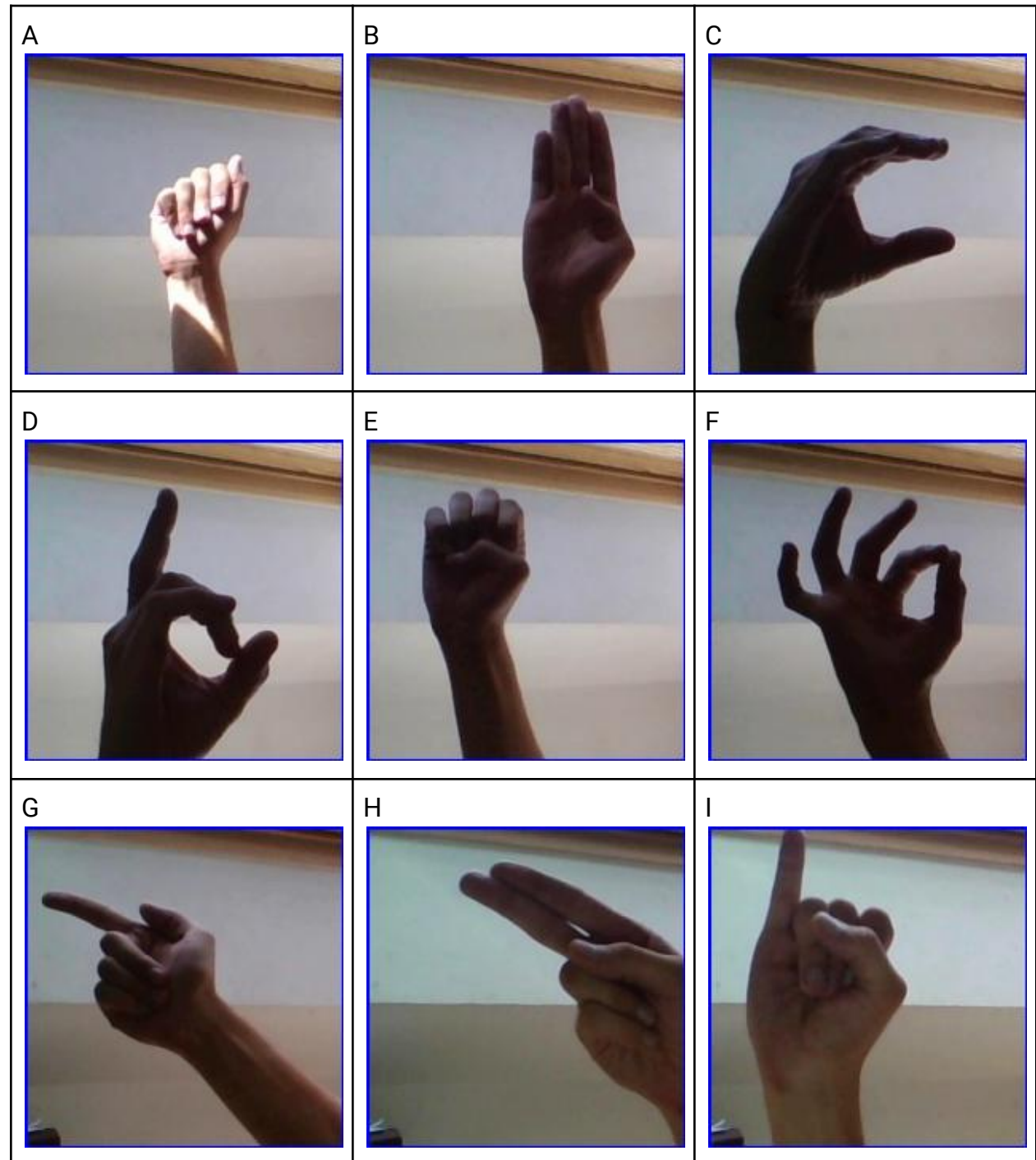
```
$ make
```

To run the ASL application, you will need to provide a path to the directory where the **labels.csv**, **landmarks.csv**, and **map.csv** are. You can use the following relative path to run the application.

```
$ ./asl_application ../../data/asl
```

The application is set to use the SVM classifier as a default. The application will need to train the SVM first at the beginning which may take a few minutes. Once the training has been completed, a camera window will appear on your screen. Now, you can sign the different ASL alphabets with your hand to the camera and the predicted alphabet will be printed in the bottom right-hand corner of the screen.

Here is a reference for the ASL alphabets to sign!



J



K



L



M



N



O



P



Q



R





S



T



U



V



W



X



Y



Z



## 6 - Running the Powerpoint Controller Example

You can find our example implementation of our library for Powerpoint Gesture Recognition tool in the `gesture_pptx` folder of the repository! **Make sure that you have the Mediapipe server running before you execute the following commands.** Here, we will walk you through on how to run the Powerpoint Gesture Recognition tool. We have already run our images through the processing script and obtained the corresponding **labels.csv**, **landmarks.csv**, and **map.csv** that are now ready for training. You can find these files in the `data/asl` directory.

### Dataset

In this particular application, we used the [HaGrid](#) dataset. This dataset contains images and its respective JSON file containing the object's information (such as labels, bounding boxes, landmarks, etc). When you have two hands in one image, it is recommended that you only have one hand in an image. We made a script where it crops the image based on the bounding boxes and labeled and saved the cropped image automatically. You can find this script in [processing\\_data/processing\\_cocodataset/process\\_coco.cpp](#).

Before running the script, you should run the commands below for the dependencies:

```
$ brew install jsoncpp
$ brew install pkg-config
```

This script uses cmake to run the script, a `CMakeLists.txt` has been provided to build the script that can be found in the same folder. To build, navigate into the current directory of [processing\\_cocodataset](#) and execute the following series of commands:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

When it is done building, we will run the executable line and provide two arguments, which are the absolute path of the dataset and the absolute path of the output folder.

```
$ ./coco_dataset_export <coco_folder_path> <output_folder>
```

After having your dataset ready, you can continue running the application!

Now, from the root directory, navigate into the `gesture_pptx` folder where the Powerpoint Gesture Recognition tool application is hosted.

```
$ cd gesture_pptx
```

You will need to first build the application. Execute the following commands:

```
$ mkdir build
```

```
$ cd build
```

```
$ cmake ..
```

```
$ make
```


To run the Powerpoint application, you will need to provide a path to the directory where the **labels.csv**, **landmarks.csv**, and **map.csv** are. You will also need to give a path to where your Powerpoint presentation is. You can use the following relative path to run the application.

```
$ ./gesture_pptx ../../data/pptx <path_to_pptx>
```

For example:

```
$ ./gesture_pptx ../../data/pptx /Desktop/test.pptx
```

The application is set to use the SVM classifier as a default. The application will need to train the SVM first at the beginning which may take a few minutes. Once the training has been completed, you will see the PowerPoint presentation opened on your screen if it hasn't already been opened. Now, you can control the Powerpoint presentation by doing the following gestures to the camera:

|   |                      |
|---|----------------------|
|  | Start Slideshow mode |
|---|----------------------|



Next slide



Previous slide



End Slideshow mode