Gesture-Recognition Manual

Table of Contents

1. Getting Started

The first thing you want to do is to first clone the Gesture Recognition repository. Execute the following command to clone:

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

To use the Gesture Recognition library, there are a few dependencies that need to be installed.

a. Installing Homebrew

Most of the dependencies will be installed using the package manager Homebrew. In your Mac terminal, enter the following command to install Homebrew:

```
$ /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/ins
tall.sh)"
```

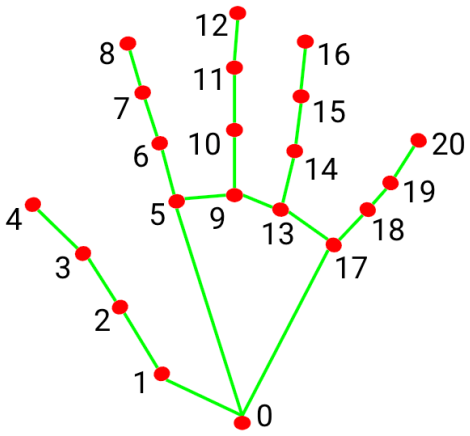
b. Installing OpenCV

Since we will be working with image and live video processing, we will use the open-source library OpenCV. To install the latest version of OpenCV, paste the following command in your terminal:

```
$ brew install opencv
```

2. Setting up Mediapipe Server

Mediapipe is a library provided by Google for customizable ML solutions in image and live video processing. Our Gesture-Recognition library depends on Mediapipe to obtain hand landmarks – a vector of 21 (x,y,z) coordinates that represents the different points of the hand in the image. Here, the x and y coordinates correspond to the (x,y) coordinates of the image on the screen and the z coordinate is the approximate depth or distance of the hand from the camera. Below is a visual representation of how the landmarks are represented and defined.



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

Figure 1: Hand Landmarks definition (Source: Mediapipe)

To use Mediapipe with our library, we have modified the original Mediapipe repository to function as a server that could be set up locally by the user. The following modified Mediapipe version has been configured to specifically run on MacOS.

a. Create a new directory within the src directory and navigate to the newly created directory

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

b. Cloning modified Mediapipe library

Locally clone the following Github Mediapipe repository. In your terminal, navigate to the folder you want to clone the Mediapipe repository to. Enter the following command:

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

Navigate into the Mediapipe repository:

```
$ cd mediapipe
```

You now have a local copy of the modified Mediapipe repository.

c. Installing Mediapipe-specific dependencies

To run Mediapipe, you will need to install a few dependencies. Google recommends using bazel to build and run Mediapipe. To install bazel in MacOS, paste the following command into your terminal:

```
$ brew install bazelisk
```

On top of OpenCV that has already been installed, another separate OpenCV@3 library will need to be installed as well. Paste the following command into your terminal:

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

Another open source library, ffmpeg, will need to be installed. Paste the following command into your terminal:

```
$ brew install ffmpeg
```

To install numpy, paste the following command into your terminal:

```
$ brew install numpy
```

Lastly, make sure you have **XCode** installed in your device. You can do this by going to the App store and downloading the app.

d. Turning on Mediapipe server

The mediapipe server code can be found in the following path:
mediapipe/mediapipe_samples/sample/mediapipe_sample.cc
The server runs on port 8080 as a default.

To run Mediapipe as a server, you will need to first build the file. You only need to build the file once after making any changes to the mediapipe_sample.cc file. Run the following command in the root directory of the Mediapipe repository. This might take around 5-10 minutes to build.

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

After building the file, execute the following command to run the server:

```
$ GLOG_logtostderr=1
bazel-bin/mediapipe/mediapipe_samples/mediapipe_sample/mediapipe
_sample
--calculator_graph_config_file=mediapipe/graphs/hand_tracking/ha
nd_tracking_desktop_live.pbtxt
```

e. Troubleshooting and Debugging

The current Mediapipe repository has been configured to default Mac settings. If you encounter any issues with dependencies, you might want to see the WORKSPACE file in the root directory.

Inside WORKSPACE, you can find references to external dependencies required to run the output. So, it is important to have the correct version and correct path to be able to run the program.

OpenCV and FFmpeg require your path to the installed library, so having the correct path is really important.

If you have an error in running the program, the program could not find OpenCV. Please refer to the WORKSPACE file and under the `new_local_repository` function , and depending on what Operating System you are using please find the function name corresponding to your Operating System. Below is an example when running in MacOS you will want to refer to this part of the code :

```
new_local_repository(
    name = "macos_opencv",
    build_file = "@//third_party:opencv_macos.BUILD",
    # prefix in "opencv_macos.BUILD".
    path = "/usr/local",  # e.g. /usr/local/Cellar for HomeBrew
)
```

In this code it is pointing that your OpenCV@3 is in the “/usr/local” path. Double check if OpenCV@3 exists within that path, if not, you might want to change the path. For example:

```
new_local_repository(
    name = "macos_opencv",
    build_file = "@//third_party:opencv_macos.BUILD",
    path = "/opt/homebrew/",
)
```

Here we changed the path to “/opt/homebrew/”, this path correctly points to OpenCV@3. If you change the OpenCV path in WORKSPACE, you should also change the corresponding FFmpeg path.

Find a new_local_repository under the name of your OS and FFmpeg (ex: “macos_ffmpeg”) and change the path the same as the modified path you established for “macos_opencv” and it pointing to FFmpeg directory. For example:

…..

```
path = "/opt/homebrew/ffmpeg",

}
```

Here you are able to fix the <opencv2/opencv2.h> not exist error, and you can try rerunning the program again. If you encounter the error again, please double check if the path is pointing to the specified library correctly, and please double check if you have OpenCV and OpenCV@3 installed too and both are in the same path.

It is also important to note that the external dependencies are linked to each other in workspace, so it is very important to reference these external dependencies with the correct versions and the correct version of their dependencies, to be able to successfully run the program.

3. Data Preprocessing: Preparing your data for training

a. Finding the right dataset

The algorithm implemented in this library's performance relies heavily on the dataset that it is trained on and finding the right dataset is crucial. Below are a few things to keep in mind when choosing a dataset for this project:

- Images should only contain one hand at a time. If there are more than one hand in the image, make sure to crop the image so it only contains the desired gesture.
    - If you are using a COCO dataset that contains multiple hands in an image, see part 3c of this manual.
- The more images the better. Ensure that the number of images per class is balanced.
- Kaggle is a good resource for looking for datasets.

b. Working with the library's processing script

We have provided a processing script that will take in your images for training and extract hand landmarks from the images using our Mediapipe server. This script can be found in processing_data/processing.cpp from the root directory of the Gesture-Recognition library. This script will output 3 files that's contents all correspond to each other:

1. **labels.csv** : Class labels for each hand landmark identified from the image. Each line in this file corresponds to each line in Landmarks.csv.
2. **landmarks.csv** : Landmarks for each hand found in the image. Each line contains 21 (x,y,z) coordinates separated by semicolons (;). Each line in this file corresponds to each line in Labels.csv.
3. **map.csv** : The key that lists the string label that corresponds to each numeric class label found in Labels.csv. Used to convert predicted numeric classes by the classifier back to its string label.

To use the library's processing script, ensure that all the training images are organized like the following directory format:

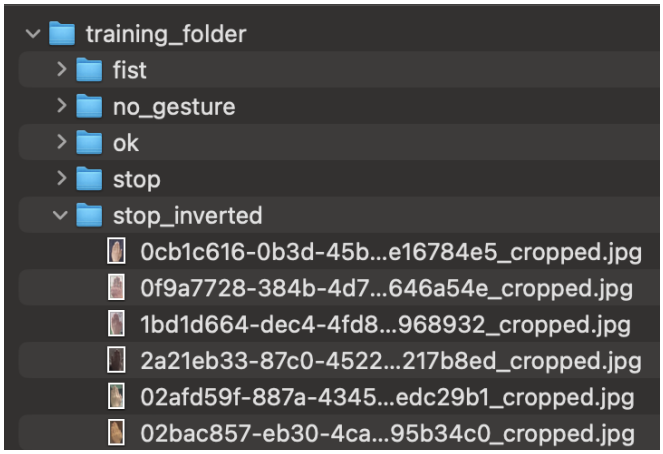


Figure 2: Example training images directory

All training images should be in a single directory (for example *training_folder* in figure 2) that's path will be passed into the processing script as a command line argument. Each image should be organized into a directory that's title will be the string class label. For example, all images of a fist should be in a directory titled '*fist*'.

A CMakeLists.txt has been provided to build the processing script that can be found in the same folder. To build, navigate into the processing_data directory and execute the following series of commands:

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

To run the processing script, you will need to provide 2 command line arguments:

1. the directory to your folder containing the images in the format specified above
2. the directory where the 3 output files (Labels.csv, Landmarks.csv, Map.csv) will be stored.

For the second command line argument, we have created a designated *data* folder in the root directory of the Gesture-Recognition library to save the 3 output files of your project. Figure 3 shows an example usage where information of an asl dataset has been saved.

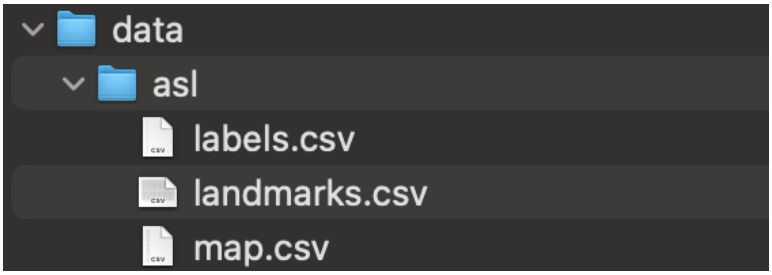


Figure 3: Example usage for the data output directory

Execute the following command in the processing_data/build directory to run the processing script:

```
$ ./processing [training_images_dir_path] [output_folder]
```

Example command:

```
$ ./processing /Desktop/asl_dataset ../../data/asl
```

c. Modifying processing script for data augmentation

Our library offers the ability to perform data augmentation. Each image is run through a pipeline that can augment the images by modifying the brightness and rotation. Modify the following block of code in the processing script to augment the images as needed.

```
// 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, -15.0);
my_pipeline.add_stage(40.0, 15.0);
my_pipeline.add_stage(40.0, -30.0);
my_pipeline.add_stage(40.0, 30.0);
my_pipeline.add_stage(40.0, -45.0);
my_pipeline.add_stage(40.0, 45.0);
```

Figure 4: Code block for pipeline that performs data augmentation

d. Processing COCO dataset

When the chosen dataset contains two hands in an image, it might be a problem when training the data. Since mediapipe will try to get both of the landmarks and labeling it to the wrong hand. To avoid this problem, we would want to have a dataset where there's one hand at a time. So we created a script that cropped the object based on the given bounding boxes from its respective JSON file. It is also required to have a JSON dataset corresponding to an image that informs the respective image's label and bounding box. It is recommended that the dataset is in COCO format. We created a script under processing_data/processing_cocodataset/process_coco.cpp, in this script the images are automatically cropped based on the bounding boxes and saved these cropped images under a folder named based on its label. This script will also create a new folder within the directory called output_folder containing the cropped images.

To use this script we have to install some 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>
```

4. Choosing and working with classifiers

Our library provides three possible options for the classifier model : KNN, SVM, and Decision Tree. Your choice of the right model can depend on the following things:

	Size of the data set	Complexity and interpretation of the algorithm	Noise in your data set	Number of features to take into account	Other points to consider
KNN	Works well with small datasets. Inefficient with large datasets as it calculates the distances between each data point.	Very easy to understand and to adjust K parameters.	Sensitive to noise and missing data	Does not work well in high dimensions	● Can adjust itself to new data easily. ● No training period required. ● Memory intensive.
SVM	Works well in small datasets. Not suitable for large data sets.	SVM has many parameters (gamma, kernel, etc.) that affect the performance.	Sensitive to noise.	Very effective in high dimensional spaces	● Can handle nonlinear separation. ● Memory efficient ● Global optimality.
DTree	Efficient for large Datasets. Can be unstable for small datasets.	Easy to interpret and understand. Does not require choosing any parameters.	Small changes in data can create a completely different Tree.	Can identify the important features.	● Not well suited for XOR relationships ● Handles Nonlinear data ● Prone to overfitting.

You can try to run your application with all three classifiers and decide which one is more relevant to your dataset based on the accuracy it returns. To see how to build and use the classifiers please refer to our **tutorial section 4**.

5. Working with ASL and Powerpoint Applications

To see how to run the ASL Alphabet Recognition application, see Part 5 of the tutorial.

To see how to run the Powerpoint Gesture Control application, see Part 6 of the tutorial.

a. Dealing with Customized Gestures for Powerpoint Gesture Control

If you are training on different sets of gestures, ensure that you replace the curr_gesture options (the part in the if statement where *curr_gesture == <class_label>*) in the following if statements with the correct class labels as specified in your **map.csv.** Here is the code snippet from gesture_pptx/gesture_pptx.cpp

```
std::cout << prev_gesture << "\t" << curr_gesture << std::endl;
if (prev_gesture == curr_gesture && curr_gesture == "ok") {
    std::cout << "START SLIDE\n";
    handleGesture("StartSlide");
}
else if (prev_gesture == curr_gesture && curr_gesture == "fist") {
    std::cout << "END SLIDE\n";
    handleGesture("EndSlide");
}
else if (prev_gesture == curr_gesture && curr_gesture == "like") {
    std::cout << "NEXT SLIDE\n";
    handleGesture("NextSlide");
}
else if (prev_gesture == curr_gesture && curr_gesture == "dislike") {
    std::cout << "PREV SLIDE\n";
    handleGesture("PrevSlide");
}
```

b. Troubleshooting and Debugging Tips

- Ensure that you have the most recent version or commit of the Mediapipe server.
- Ensure that the Mediapipe server has been built and is currently running
- If relative paths are not working, try with an absolute path instead
- It is recommended that you test the applications with one hand at a time and not two hands
- Training images should only contain the hand gesture of the class that it is training on and only one hand at a time
- When running the ASL application, training the classifiers especially with SVM at the beginning of the application might take about 2-5 minutes since the ASL application has to train on a large dataset.
- Make sure that there is a balanced amount of training images for each class
- If you are having trouble turning on your laptop camera, it may be the case that your laptop is attempting to connect to a camera from another device, especially if you are using an iPhone or an iPad. To disable this feature on your iPhone/iPad, go to:
    - Settings > General > AirPlay & Handoff > Turn off Continuity Camera

Acknowledgements

We would like to express our gratitude to the following individuals and organizations for their contributions to this project:

- The creators and maintainers of Mediapipe for providing a critical component in our system.
- The creators of GRLib for the inspiration of this project.
- Kaggle community for the HaGRID and ASL Alphabet dataset used in our project.
- Professor Bjarne Stroustrup from Columbia University for his valuable insights on the project design in C++.

We also acknowledge the support of the COMS 4995: Design Using C++ teaching assistants, Ulas Alakent and Srishti Srivastava, for their feedback and engagement.

Team Members

The design and development of Gesture Recognition Library in C++ were made possible through the collaborative efforts of the following team members:

- Elifia Muthia
    - Barnard College
    - em3308@barnard.edu
- Elvina Wibisono
    - Barnard College
    - ew2709@barnard.edu
- Yanna Botvinnik
    - Barnard College
    - yb2462@barnard.edu
- Maitar Asher
    - Columbia University
    - ma4265@columbia.edu
- Noam Zaid
    - Columbia University
    - nmz2117@columbia.edu