Advanced Machine Learning Project

On

Hand Detection, Face Detection and Facial Mesh

Submitted by:

Aarush Bhardwaj

811126136

Introduction:

Machine learning is a branch of computer science that arose from the study of data pattern recognition as well as artificial intelligence's computational learning theory. It's a first-class ticket to today's most exciting data analytics jobs[1]. As the number of data sources grows, so does the computational power required to process them. Going straight to the data is one of the simplest ways to quickly acquire insights and make predictions.

Decision making, clustering, classification, forecasting, deep learning, inductive logic programming, support vector machines, reinforcement learning, similarity and metric learning, genetic algorithms, sparse dictionary learning, and other sub-problems are all studied in Machine Learning. The machine learning task of inferring a function from labeled data is known as supervised learning, or classification [2]. A training set and a test set are used in supervised learning. The purpose of the supervised learning algorithm is to infer a function that maps the input vector to the output vector with low error from the training and test set, which consists of a set of instances consisting of input and output vectors. In an ideal case, a model trained on a set of instances would correctly classify an unknown example, which would necessitate the model to generalize from the training set in a fair manner. In layman's words, supervised learning is the process of learning an idea, much like a brain which is exposed to a set of inputs and result vectors, and the brain learns the concept that connects the inputs to the outputs. The machine learning enthusiast can choose from a variety of supervised machine learning algorithms, such as Neural Networks, Decision Trees, Support Vector Machines, Random Forest, Nave Bayes Classifier, Bayes Net, Majority Classifier[4,7,8,9], each with its own set of advantages and disadvantages. According to the No Free Lunch theorem [3,] there is no single method that works in all instances.

In this project, we will use mediapipe library to detect hands and then try to control our device, just by using our hand gestures.

Problems and Issues in Machine Learning:

Before we begin, we must understand how to choose the best machine learning algorithm for the dataset. To choose an algorithm for a learning task intelligently, we must consider the following elements [4]:

1. Data Heterogeneity:

Many algorithms, such as neural networks and support vector machines, require homogenous numeric and normalized feature vectors. Because algorithms that use distance metrics are extremely sensitive to this, these methods should be used only as a last resort if the data is heterogeneous. Decision Trees are highly good at handling heterogeneous data.

2. Data Redundancy:

If the data contains duplicated information, such as highly correlated values, distance-based approaches are worthless due to numerical instability. In this circumstance, data can be subjected to some form of regularization to avoid this problem.

3. Dependent Characteristics:

If the feature vectors are interdependent, algorithms that monitor complicated interconnections, such as Neural Networks and Decision Trees, perform better than other algorithms.

4. Bias-Variance Tradeoff:

When trained on each of these data sets, a learning algorithm is biased for a particular input x if it predicts different output values when trained on different training sets, whereas a learning algorithm has high variance for a particular input x if it predicts different output values when trained on different training sets. The total of bias and variance of the learning process can be related to the prediction error of a learnt classifier, and neither can be high because the prediction error will be high. Machine learning algorithms can automatically adjust the balance between bias and variance, or manually adjust the balance using bias parameters, and adopting such methods will remedy this dilemma.

5. Curse of Dimensionality:

The machine learning algorithm can be confused by the big number of dimensions if the problem has a large number of dimensions and the problem only depends on a subspace of the input space with tiny dimensions. As a result, the algorithm's variance can be significant. In practice, the accuracy of the trained function is likely to improve if the data scientist can manually remove unnecessary features from the input data. In addition, many feature selection techniques, such as Principle Component Analysis for unsupervised learning, aim to discover the important features while excluding the unnecessary ones. This decreases the number of dimensions.

6. Overfitting:

The programmer should be aware that there is a chance that the output numbers contain inherent noise because of human or sensor failures. The algorithm must not attempt to infer a function that exactly matches all of the data in this scenario. Overfitting occurs when the data is fitted too carefully, resulting in the model answering flawlessly for all training examples but with a very high error for unknown samples. Stopping the learning process prematurely and applying filters to the data in the pre-learning phase to reduce sounds are two practical ways to avoid this.

We can only choose a supervised learning method that works for the dataset we're working on after evaluating all of these aspects. If we were dealing with a dataset that had heterogeneous data, for example, decision trees would outperform other methods. If the dataset we were working on contained 1000 dimensions, it would be wiser to do PCA on the data before using a supervised learning method.

Model Prepration: (Hand Detection)

For this project, I am going to use to MediaPipe, which is a google library that has pre-annotated 30K images of hands. MediaPipe Hands model makes use of a machine learning pipeline that consists of several models that work together: An orientated hand bounding box is returned by a palm detection model that acts on the entire image. A hand landmark model that returns high-fidelity 3D hand key points from the cropped image region determined by the palm detector. This approach is similar to the one used in our MediaPipe Face Mesh solution, which combines a face detector with a face landmark model.

Providing the hand landmark model with a correctly cropped hand image dramatically minimizes the requirement for data augmentation (e.g. rotations, translations, and scaling) and instead allows the network to focus on coordinate prediction accuracy. Furthermore, the pipeline crops can be created based on the hand landmarks recognized in the previous frame, and palm detection is only used to delocalize the hand when the landmark model can no longer detect its presence.

The pipeline is implemented as a MediaPipe *graph* that renders using a specific hand renderer subgraph and leverages a *hand landmark tracking subgraph* from the hand landmark module. A hand landmark subgraph from the same module and a palm detection subgraph from the palm detection module are used internally by the hand landmark tracking subgraph.

***Graph code:***

# MediaPipe graph that performs multi-hand tracking with TensorFlow Lite on GPU.

# Used in the examples in

# mediapipe/examples/android/src/java/com/mediapipe/apps/handtrackinggpu.

# GPU image. (GpuBuffer)

input\_stream: "input\_video"

# Max number of hands to detect/process. (int)

input\_side\_packet: "num\_hands"

# Model complexity (0 or 1). (int)

input\_side\_packet: "model\_complexity"

# GPU image. (GpuBuffer)

output\_stream: "output\_video"

# Collection of detected/predicted hands, each represented as a list of

# landmarks. (std::vector<NormalizedLandmarkList>)

output\_stream: "hand\_landmarks"

# Throttles the images flowing downstream for flow control. It passes through

# the very first incoming image unaltered, and waits for downstream nodes

# (calculators and subgraphs) in the graph to finish their tasks before it

# passes through another image. All images that come in while waiting are

# dropped, limiting the number of in-flight images in most part of the graph to

# 1. This prevents the downstream nodes from queuing up incoming images and data

# excessively, which leads to increased latency and memory usage, unwanted in

# real-time mobile applications. It also eliminates unnecessarily computation,

# e.g., the output produced by a node may get dropped downstream if the

# subsequent nodes are still busy processing previous inputs.

node {

calculator: "FlowLimiterCalculator"

input\_stream: "input\_video"

input\_stream: "FINISHED:output\_video"

input\_stream\_info: {

tag\_index: "FINISHED"

back\_edge: true

}

output\_stream: "throttled\_input\_video"

}

# Detects/tracks hand landmarks.

node {

calculator: "HandLandmarkTrackingGpu"

input\_stream: "IMAGE:throttled\_input\_video"

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

input\_side\_packet: "NUM\_HANDS:num\_hands"

output\_stream: "LANDMARKS:hand\_landmarks"

output\_stream: "HANDEDNESS:handedness"

output\_stream: "PALM\_DETECTIONS:palm\_detections"

output\_stream: "HAND\_ROIS\_FROM\_LANDMARKS:hand\_rects\_from\_landmarks"

output\_stream: "HAND\_ROIS\_FROM\_PALM\_DETECTIONS:hand\_rects\_from\_palm\_detections"

}

# Subgraph that renders annotations and overlays them on top of the input

# images (see hand\_renderer\_gpu.pbtxt).

node {

calculator: "HandRendererSubgraph"

input\_stream: "IMAGE:throttled\_input\_video"

input\_stream: "DETECTIONS:palm\_detections"

input\_stream: "LANDMARKS:hand\_landmarks"

input\_stream: "HANDEDNESS:handedness"

input\_stream: "NORM\_RECTS:0:hand\_rects\_from\_palm\_detections"

input\_stream: "NORM\_RECTS:1:hand\_rects\_from\_landmarks"

output\_stream: "IMAGE:output\_video"

}

***Hand Landmark Tracking Subgraph:***

# MediaPipe graph to detect/predict hand landmarks on GPU.

#

# The procedure is done in two steps:

# - locate palms/hands

# - detect landmarks for each palm/hand.

# This graph tries to skip palm detection as much as possible by reusing

# previously detected/predicted landmarks for new images.

type: "HandLandmarkTrackingGpu"

# GPU image. (GpuBuffer)

input\_stream: "IMAGE:image"

# Max number of hands to detect/track. (int)

input\_side\_packet: "NUM\_HANDS:num\_hands"

# Complexity of hand landmark and palm detection models: 0 or 1. Accuracy as

# well as inference latency generally go up with the model complexity. If

# unspecified, functions as set to 1. (int)

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

# Whether landmarks on the previous image should be used to help localize

# landmarks on the current image. (bool)

input\_side\_packet: "USE\_PREV\_LANDMARKS:use\_prev\_landmarks"

# Collection of detected/predicted hands, each represented as a list of

# landmarks. (std::vector<NormalizedLandmarkList>)

# NOTE: there will not be an output packet in the LANDMARKS stream for this

# particular timestamp if none of hands detected. However, the MediaPipe

# framework will internally inform the downstream calculators of the absence of

# this packet so that they don't wait for it unnecessarily.

output\_stream: "LANDMARKS:multi\_hand\_landmarks"

# Collection of detected/predicted hand world landmarks.

# (std::vector<LandmarkList>)

#

# World landmarks are real-world 3D coordinates in meters with the origin in the

# center of the hand bounding box calculated from the landmarks.

#

# WORLD\_LANDMARKS shares the same landmark topology as LANDMARKS. However,

# LANDMARKS provides coordinates (in pixels) of a 3D object projected onto the

# 2D image surface, while WORLD\_LANDMARKS provides coordinates (in meters) of

# the 3D object itself.

output\_stream: "WORLD\_LANDMARKS:multi\_hand\_world\_landmarks"

# Collection of handedness of the detected hands (i.e. is hand left or right),

# each represented as a ClassificationList proto with a single Classification

# entry. (std::vector<ClassificationList>)

# Note that handedness is determined assuming the input image is mirrored,

# i.e., taken with a front-facing/selfie camera with images flipped

# horizontally.

output\_stream: "HANDEDNESS:multi\_handedness"

# Extra outputs (for debugging, for instance).

# Detected palms. (std::vector<Detection>)

output\_stream: "PALM\_DETECTIONS:palm\_detections"

# Regions of interest calculated based on landmarks.

# (std::vector<NormalizedRect>)

output\_stream: "HAND\_ROIS\_FROM\_LANDMARKS:hand\_rects"

# Regions of interest calculated based on palm detections.

# (std::vector<NormalizedRect>)

output\_stream: "HAND\_ROIS\_FROM\_PALM\_DETECTIONS:hand\_rects\_from\_palm\_detections"

# When the optional input side packet "use\_prev\_landmarks" is either absent or

# set to true, uses the landmarks on the previous image to help localize

# landmarks on the current image.

node {

calculator: "GateCalculator"

input\_side\_packet: "ALLOW:use\_prev\_landmarks"

input\_stream: "prev\_hand\_rects\_from\_landmarks"

output\_stream: "gated\_prev\_hand\_rects\_from\_landmarks"

options: {

[mediapipe.GateCalculatorOptions.ext] {

allow: true

}

}

}

# Determines if an input vector of NormalizedRect has a size greater than or

# equal to the provided num\_hands.

node {

calculator: "NormalizedRectVectorHasMinSizeCalculator"

input\_stream: "ITERABLE:gated\_prev\_hand\_rects\_from\_landmarks"

input\_side\_packet: "num\_hands"

output\_stream: "prev\_has\_enough\_hands"

}

# Drops the incoming image if enough hands have already been identified from the

# previous image. Otherwise, passes the incoming image through to trigger a new

# round of palm detection.

node {

calculator: "GateCalculator"

input\_stream: "image"

input\_stream: "DISALLOW:prev\_has\_enough\_hands"

output\_stream: "palm\_detection\_image"

options: {

[mediapipe.GateCalculatorOptions.ext] {

empty\_packets\_as\_allow: true

}

}

}

# Detects palms.

node {

calculator: "PalmDetectionGpu"

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

input\_stream: "IMAGE:palm\_detection\_image"

output\_stream: "DETECTIONS:all\_palm\_detections"

}

# Makes sure there are no more detections than provided num\_hands.

node {

calculator: "ClipDetectionVectorSizeCalculator"

input\_stream: "all\_palm\_detections"

output\_stream: "palm\_detections"

input\_side\_packet: "num\_hands"

}

# Extracts image size.

node {

calculator: "ImagePropertiesCalculator"

input\_stream: "IMAGE\_GPU:palm\_detection\_image"

output\_stream: "SIZE:palm\_detection\_image\_size"

}

# Outputs each element of palm\_detections at a fake timestamp for the rest of

# the graph to process. Clones the image\_size packet for each palm\_detection at

# the fake timestamp. At the end of the loop, outputs the BATCH\_END timestamp

# for downstream calculators to inform them that all elements in the vector have

# been processed.

node {

calculator: "BeginLoopDetectionCalculator"

input\_stream: "ITERABLE:palm\_detections"

input\_stream: "CLONE:palm\_detection\_image\_size"

output\_stream: "ITEM:palm\_detection"

output\_stream: "CLONE:image\_size\_for\_palms"

output\_stream: "BATCH\_END:palm\_detections\_timestamp"

}

# Calculates region of interest (ROI) base on the specified palm.

node {

calculator: "PalmDetectionDetectionToRoi"

input\_stream: "DETECTION:palm\_detection"

input\_stream: "IMAGE\_SIZE:image\_size\_for\_palms"

output\_stream: "ROI:hand\_rect\_from\_palm\_detection"

}

# Collects a NormalizedRect for each hand into a vector. Upon receiving the

# BATCH\_END timestamp, outputs the vector of NormalizedRect at the BATCH\_END

# timestamp.

node {

name: "EndLoopForPalmDetections"

calculator: "EndLoopNormalizedRectCalculator"

input\_stream: "ITEM:hand\_rect\_from\_palm\_detection"

input\_stream: "BATCH\_END:palm\_detections\_timestamp"

output\_stream: "ITERABLE:hand\_rects\_from\_palm\_detections"

}

# Performs association between NormalizedRect vector elements from previous

# image and rects based on palm detections from the current image. This

# calculator ensures that the output hand\_rects vector doesn't contain

# overlapping regions based on the specified min\_similarity\_threshold.

node {

calculator: "AssociationNormRectCalculator"

input\_stream: "hand\_rects\_from\_palm\_detections"

input\_stream: "gated\_prev\_hand\_rects\_from\_landmarks"

output\_stream: "hand\_rects"

options: {

[mediapipe.AssociationCalculatorOptions.ext] {

min\_similarity\_threshold: 0.5

}

}

}

# Extracts image size.

node {

calculator: "ImagePropertiesCalculator"

input\_stream: "IMAGE\_GPU:image"

output\_stream: "SIZE:image\_size"

}

# Outputs each element of hand\_rects at a fake timestamp for the rest of the

# graph to process. Clones image and image size packets for each

# single\_hand\_rect at the fake timestamp. At the end of the loop, outputs the

# BATCH\_END timestamp for downstream calculators to inform them that all

# elements in the vector have been processed.

node {

calculator: "BeginLoopNormalizedRectCalculator"

input\_stream: "ITERABLE:hand\_rects"

input\_stream: "CLONE:0:image"

input\_stream: "CLONE:1:image\_size"

output\_stream: "ITEM:single\_hand\_rect"

output\_stream: "CLONE:0:image\_for\_landmarks"

output\_stream: "CLONE:1:image\_size\_for\_landmarks"

output\_stream: "BATCH\_END:hand\_rects\_timestamp"

}

# Detect hand landmarks for the specific hand rect.

node {

calculator: "HandLandmarkGpu"

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

input\_stream: "IMAGE:image\_for\_landmarks"

input\_stream: "ROI:single\_hand\_rect"

output\_stream: "LANDMARKS:single\_hand\_landmarks"

output\_stream: "WORLD\_LANDMARKS:single\_hand\_world\_landmarks"

output\_stream: "HANDEDNESS:single\_handedness"

}

# Collects the handedness for each single hand into a vector. Upon receiving the

# BATCH\_END timestamp, outputs a vector of ClassificationList at the BATCH\_END

# timestamp.

node {

calculator: "EndLoopClassificationListCalculator"

input\_stream: "ITEM:single\_handedness"

input\_stream: "BATCH\_END:hand\_rects\_timestamp"

output\_stream: "ITERABLE:multi\_handedness"

}

# Calculate region of interest (ROI) based on detected hand landmarks to reuse

# on the subsequent runs of the graph.

node {

calculator: "HandLandmarkLandmarksToRoi"

input\_stream: "IMAGE\_SIZE:image\_size\_for\_landmarks"

input\_stream: "LANDMARKS:single\_hand\_landmarks"

output\_stream: "ROI:single\_hand\_rect\_from\_landmarks"

}

# Collects a set of landmarks for each hand into a vector. Upon receiving the

# BATCH\_END timestamp, outputs the vector of landmarks at the BATCH\_END

# timestamp.

node {

calculator: "EndLoopNormalizedLandmarkListVectorCalculator"

input\_stream: "ITEM:single\_hand\_landmarks"

input\_stream: "BATCH\_END:hand\_rects\_timestamp"

output\_stream: "ITERABLE:multi\_hand\_landmarks"

}

# Collects a set of world landmarks for each hand into a vector. Upon receiving

# the BATCH\_END timestamp, outputs the vector of landmarks at the BATCH\_END

# timestamp.

node {

calculator: "EndLoopLandmarkListVectorCalculator"

input\_stream: "ITEM:single\_hand\_world\_landmarks"

input\_stream: "BATCH\_END:hand\_rects\_timestamp"

output\_stream: "ITERABLE:multi\_hand\_world\_landmarks"

}

# Collects a NormalizedRect for each hand into a vector. Upon receiving the

# BATCH\_END timestamp, outputs the vector of NormalizedRect at the BATCH\_END

# timestamp.

node {

calculator: "EndLoopNormalizedRectCalculator"

input\_stream: "ITEM:single\_hand\_rect\_from\_landmarks"

input\_stream: "BATCH\_END:hand\_rects\_timestamp"

output\_stream: "ITERABLE:hand\_rects\_from\_landmarks"

}

# Caches hand rects calculated from landmarks, and upon the arrival of the next

# input image, sends out the cached rects with timestamps replaced by that of

# the input image, essentially generating a packet that carries the previous

# hand rects. Note that upon the arrival of the very first input image, a

# timestamp bound update occurs to jump start the feedback loop.

node {

calculator: "PreviousLoopbackCalculator"

input\_stream: "MAIN:image"

input\_stream: "LOOP:hand\_rects\_from\_landmarks"

input\_stream\_info: {

tag\_index: "LOOP"

back\_edge: true

}

output\_stream: "PREV\_LOOP:prev\_hand\_rects\_from\_landmarks"

}

***Hand Render Subgraph:***

# MediaPipe graph to render hand landmarks and some related debug information.

type: "HandRendererSubgraph"

# GPU buffer. (GpuBuffer)

input\_stream: "IMAGE:input\_image"

# Collection of detected/predicted hands, each represented as a list of

# landmarks. (std::vector<NormalizedLandmarkList>)

input\_stream: "LANDMARKS:multi\_hand\_landmarks"

# Handedness of the detected hand (i.e. is hand left or right).

# (std::vector<ClassificationList>)

input\_stream: "HANDEDNESS:multi\_handedness"

# Regions of interest calculated based on palm detections.

# (std::vector<NormalizedRect>)

input\_stream: "NORM\_RECTS:0:multi\_palm\_rects"

# Regions of interest calculated based on landmarks.

# (std::vector<NormalizedRect>)

input\_stream: "NORM\_RECTS:1:multi\_hand\_rects"

# Detected palms. (std::vector<Detection>)

input\_stream: "DETECTIONS:palm\_detections"

# Updated GPU buffer. (GpuBuffer)

output\_stream: "IMAGE:output\_image"

# Converts detections to drawing primitives for annotation overlay.

node {

calculator: "DetectionsToRenderDataCalculator"

input\_stream: "DETECTIONS:palm\_detections"

output\_stream: "RENDER\_DATA:detection\_render\_data"

node\_options: {

[type.googleapis.com/mediapipe.DetectionsToRenderDataCalculatorOptions] {

thickness: 4.0

color { r: 0 g: 255 b: 0 }

}

}

}

# Converts normalized rects to drawing primitives for annotation overlay.

node {

calculator: "RectToRenderDataCalculator"

input\_stream: "NORM\_RECTS:multi\_hand\_rects"

output\_stream: "RENDER\_DATA:multi\_hand\_rects\_render\_data"

node\_options: {

[type.googleapis.com/mediapipe.RectToRenderDataCalculatorOptions] {

filled: false

color { r: 255 g: 0 b: 0 }

thickness: 4.0

}

}

}

# Converts normalized rects to drawing primitives for annotation overlay.

node {

calculator: "RectToRenderDataCalculator"

input\_stream: "NORM\_RECTS:multi\_palm\_rects"

output\_stream: "RENDER\_DATA:multi\_palm\_rects\_render\_data"

node\_options: {

[type.googleapis.com/mediapipe.RectToRenderDataCalculatorOptions] {

filled: false

color { r: 125 g: 0 b: 122 }

thickness: 4.0

}

}

}

# Outputs each element of multi\_palm\_landmarks at a fake timestamp for the rest

# of the graph to process. At the end of the loop, outputs the BATCH\_END

# timestamp for downstream calculators to inform them that all elements in the

# vector have been processed.

node {

calculator: "BeginLoopNormalizedLandmarkListVectorCalculator"

input\_stream: "ITERABLE:multi\_hand\_landmarks"

output\_stream: "ITEM:single\_hand\_landmarks"

output\_stream: "BATCH\_END:landmark\_timestamp"

}

# Converts landmarks to drawing primitives for annotation overlay.

node {

calculator: "LandmarksToRenderDataCalculator"

input\_stream: "NORM\_LANDMARKS:single\_hand\_landmarks"

output\_stream: "RENDER\_DATA:single\_hand\_landmark\_render\_data"

node\_options: {

[type.googleapis.com/mediapipe.LandmarksToRenderDataCalculatorOptions] {

landmark\_connections: 0

landmark\_connections: 1

landmark\_connections: 1

landmark\_connections: 2

landmark\_connections: 2

landmark\_connections: 3

landmark\_connections: 3

landmark\_connections: 4

landmark\_connections: 0

landmark\_connections: 5

landmark\_connections: 5

landmark\_connections: 6

landmark\_connections: 6

landmark\_connections: 7

landmark\_connections: 7

landmark\_connections: 8

landmark\_connections: 5

landmark\_connections: 9

landmark\_connections: 9

landmark\_connections: 10

landmark\_connections: 10

landmark\_connections: 11

landmark\_connections: 11

landmark\_connections: 12

landmark\_connections: 9

landmark\_connections: 13

landmark\_connections: 13

landmark\_connections: 14

landmark\_connections: 14

landmark\_connections: 15

landmark\_connections: 15

landmark\_connections: 16

landmark\_connections: 13

landmark\_connections: 17

landmark\_connections: 0

landmark\_connections: 17

landmark\_connections: 17

landmark\_connections: 18

landmark\_connections: 18

landmark\_connections: 19

landmark\_connections: 19

landmark\_connections: 20

landmark\_color { r: 255 g: 0 b: 0 }

connection\_color { r: 0 g: 255 b: 0 }

thickness: 4.0

}

}

}

# Collects a RenderData object for each hand into a vector. Upon receiving the

# BATCH\_END timestamp, outputs the vector of RenderData at the BATCH\_END

# timestamp.

node {

calculator: "EndLoopRenderDataCalculator"

input\_stream: "ITEM:single\_hand\_landmark\_render\_data"

input\_stream: "BATCH\_END:landmark\_timestamp"

output\_stream: "ITERABLE:multi\_hand\_landmarks\_render\_data"

}

# Don't render handedness if there are more than one handedness reported.

node {

calculator: "ClassificationListVectorHasMinSizeCalculator"

input\_stream: "ITERABLE:multi\_handedness"

output\_stream: "disallow\_handedness\_rendering"

node\_options: {

[type.googleapis.com/mediapipe.CollectionHasMinSizeCalculatorOptions] {

min\_size: 2

}

}

}

node {

calculator: "GateCalculator"

input\_stream: "multi\_handedness"

input\_stream: "DISALLOW:disallow\_handedness\_rendering"

output\_stream: "allowed\_multi\_handedness"

node\_options: {

[type.googleapis.com/mediapipe.GateCalculatorOptions] {

empty\_packets\_as\_allow: false

}

}

}

node {

calculator: "SplitClassificationListVectorCalculator"

input\_stream: "allowed\_multi\_handedness"

output\_stream: "handedness"

node\_options: {

[type.googleapis.com/mediapipe.SplitVectorCalculatorOptions] {

ranges: { begin: 0 end: 1 }

element\_only: true

}

}

}

# Converts classification to drawing primitives for annotation overlay.

node {

calculator: "LabelsToRenderDataCalculator"

input\_stream: "CLASSIFICATIONS:handedness"

output\_stream: "RENDER\_DATA:handedness\_render\_data"

node\_options: {

[type.googleapis.com/mediapipe.LabelsToRenderDataCalculatorOptions]: {

color { r: 255 g: 0 b: 0 }

thickness: 10.0

font\_height\_px: 50

horizontal\_offset\_px: 30

vertical\_offset\_px: 50

max\_num\_labels: 1

location: TOP\_LEFT

}

}

}

# Draws annotations and overlays them on top of the input images. Consumes

# a vector of RenderData objects and draws each of them on the input frame.

node {

calculator: "AnnotationOverlayCalculator"

input\_stream: "IMAGE\_GPU:input\_image"

input\_stream: "detection\_render\_data"

input\_stream: "multi\_hand\_rects\_render\_data"

input\_stream: "multi\_palm\_rects\_render\_data"

input\_stream: "handedness\_render\_data"

input\_stream: "VECTOR:0:multi\_hand\_landmarks\_render\_data"

output\_stream: "IMAGE\_GPU:output\_image"

}

***Hand Landmark Subgraph:***

# MediaPipe graph to detect/predict hand landmarks on CPU.

type: "HandLandmarkGpu"

# GPU image. (GpuBuffer)

input\_stream: "IMAGE:image"

# ROI (region of interest) within the given image where a palm/hand is located.

# (NormalizedRect)

input\_stream: "ROI:hand\_rect"

# Complexity of the hand landmark model: 0 or 1. Landmark accuracy as well as

# inference latency generally go up with the model complexity. If unspecified,

# functions as set to 1. (int)

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

# 21 hand landmarks within the given ROI. (NormalizedLandmarkList)

# NOTE: if a hand is not present within the given ROI, for this particular

# timestamp there will not be an output packet in the LANDMARKS stream. However,

# the MediaPipe framework will internally inform the downstream calculators of

# the absence of this packet so that they don't wait for it unnecessarily.

output\_stream: "LANDMARKS:hand\_landmarks"

# Hand world landmarks within the given ROI. (LandmarkList)

# World landmarks are real-world 3D coordinates in meters with the origin in the

# center of the given ROI.

#

# WORLD\_LANDMARKS shares the same landmark topology as LANDMARKS. However,

# LANDMARKS provides coordinates (in pixels) of a 3D object projected onto the

# 2D image surface, while WORLD\_LANDMARKS provides coordinates (in meters) of

# the 3D object itself.

output\_stream: "WORLD\_LANDMARKS:hand\_world\_landmarks"

# Handedness of the detected hand (i.e. is hand left or right).

# (ClassificationList)

output\_stream: "HANDEDNESS:handedness"

# Transforms a region of image into a 224x224 tensor while keeping the aspect

# ratio, and therefore may result in potential letterboxing.

node {

calculator: "ImageToTensorCalculator"

input\_stream: "IMAGE\_GPU:image"

input\_stream: "NORM\_RECT:hand\_rect"

output\_stream: "TENSORS:input\_tensor"

output\_stream: "LETTERBOX\_PADDING:letterbox\_padding"

options: {

[mediapipe.ImageToTensorCalculatorOptions.ext] {

output\_tensor\_width: 224

output\_tensor\_height: 224

keep\_aspect\_ratio: true

output\_tensor\_float\_range {

min: 0.0

max: 1.0

}

gpu\_origin: TOP\_LEFT

}

}

}

# Loads the hand landmark TF Lite model.

node {

calculator: "HandLandmarkModelLoader"

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

output\_side\_packet: "MODEL:model"

}

# Runs a TensorFlow Lite model on GPU that takes an image tensor and outputs a

# vector of tensors representing, for instance, detection boxes/keypoints and

# scores.

node {

calculator: "InferenceCalculator"

input\_side\_packet: "MODEL:model"

input\_stream: "TENSORS:input\_tensor"

output\_stream: "TENSORS:output\_tensors"

}

# Splits a vector of tensors to multiple vectors according to the ranges

# specified in option.

node {

calculator: "SplitTensorVectorCalculator"

input\_stream: "output\_tensors"

output\_stream: "landmark\_tensors"

output\_stream: "hand\_flag\_tensor"

output\_stream: "handedness\_tensor"

output\_stream: "world\_landmark\_tensor"

options: {

[mediapipe.SplitVectorCalculatorOptions.ext] {

ranges: { begin: 0 end: 1 }

ranges: { begin: 1 end: 2 }

ranges: { begin: 2 end: 3 }

ranges: { begin: 3 end: 4 }

}

}

}

# Converts the hand-flag tensor into a float that represents the confidence

# score of hand presence.

node {

calculator: "TensorsToFloatsCalculator"

input\_stream: "TENSORS:hand\_flag\_tensor"

output\_stream: "FLOAT:hand\_presence\_score"

}

# Applies a threshold to the confidence score to determine whether a hand is

# present.

node {

calculator: "ThresholdingCalculator"

input\_stream: "FLOAT:hand\_presence\_score"

output\_stream: "FLAG:hand\_presence"

options: {

[mediapipe.ThresholdingCalculatorOptions.ext] {

threshold: 0.5

}

}

}

# Drops handedness tensor if hand is not present.

node {

calculator: "GateCalculator"

input\_stream: "handedness\_tensor"

input\_stream: "ALLOW:hand\_presence"

output\_stream: "ensured\_handedness\_tensor"

}

# Converts the handedness tensor into a float that represents the classification

# score of handedness.

node {

calculator: "TensorsToClassificationCalculator"

input\_stream: "TENSORS:ensured\_handedness\_tensor"

output\_stream: "CLASSIFICATIONS:handedness"

options: {

[mediapipe.TensorsToClassificationCalculatorOptions.ext] {

top\_k: 1

label\_map\_path: "mediapipe/modules/hand\_landmark/handedness.txt"

binary\_classification: true

}

}

}

# Drops landmarks tensors if hand is not present.

node {

calculator: "GateCalculator"

input\_stream: "landmark\_tensors"

input\_stream: "ALLOW:hand\_presence"

output\_stream: "ensured\_landmark\_tensors"

}

# Decodes the landmark tensors into a list of landmarks, where the landmark

# coordinates are normalized by the size of the input image to the model.

node {

calculator: "TensorsToLandmarksCalculator"

input\_stream: "TENSORS:ensured\_landmark\_tensors"

output\_stream: "NORM\_LANDMARKS:landmarks"

options: {

[mediapipe.TensorsToLandmarksCalculatorOptions.ext] {

num\_landmarks: 21

input\_image\_width: 224

input\_image\_height: 224

# The additional scaling factor is used to account for the Z coordinate

# distribution in the training data.

normalize\_z: 0.4

}

}

}

# Adjusts landmarks (already normalized to [0.f, 1.f]) on the letterboxed hand

# image (after image transformation with the FIT scale mode) to the

# corresponding locations on the same image with the letterbox removed (hand

# image before image transformation).

node {

calculator: "LandmarkLetterboxRemovalCalculator"

input\_stream: "LANDMARKS:landmarks"

input\_stream: "LETTERBOX\_PADDING:letterbox\_padding"

output\_stream: "LANDMARKS:scaled\_landmarks"

}

# Projects the landmarks from the cropped hand image to the corresponding

# locations on the full image before cropping (input to the graph).

node {

calculator: "LandmarkProjectionCalculator"

input\_stream: "NORM\_LANDMARKS:scaled\_landmarks"

input\_stream: "NORM\_RECT:hand\_rect"

output\_stream: "NORM\_LANDMARKS:hand\_landmarks"

}

# Drops world landmarks tensors if hand is not present.

node {

calculator: "GateCalculator"

input\_stream: "world\_landmark\_tensor"

input\_stream: "ALLOW:hand\_presence"

output\_stream: "ensured\_world\_landmark\_tensor"

}

# Decodes the landmark tensors into a list of landmarks, where the landmark

# coordinates are normalized by the size of the input image to the model.

node {

calculator: "TensorsToLandmarksCalculator"

input\_stream: "TENSORS:ensured\_world\_landmark\_tensor"

output\_stream: "LANDMARKS:unprojected\_world\_landmarks"

options: {

[mediapipe.TensorsToLandmarksCalculatorOptions.ext] {

num\_landmarks: 21

}

}

}

# Projects the world landmarks from the cropped hand image to the corresponding

# locations on the full image before cropping (input to the graph).

node {

calculator: "WorldLandmarkProjectionCalculator"

input\_stream: "LANDMARKS:unprojected\_world\_landmarks"

input\_stream: "NORM\_RECT:hand\_rect"

output\_stream: "LANDMARKS:hand\_world\_landmarks"

}

***Palm Detection Subgraph:***

# MediaPipe graph to detect palms with TensorFlow Lite on GPU.

type: "PalmDetectionGpu"

# GPU image. (GpuBuffer)

input\_stream: "IMAGE:image"

# Complexity of the palm detection model: 0 or 1. Accuracy as well as inference

# latency generally go up with the model complexity. If unspecified, functions

# as set to 1. (int)

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

# Detected palms. (std::vector<Detection>)

# NOTE: there will not be an output packet in the DETECTIONS stream for this

# particular timestamp if none of palms detected. However, the MediaPipe

# framework will internally inform the downstream calculators of the absence of

# this packet so that they don't wait for it unnecessarily.

output\_stream: "DETECTIONS:detections"

# Transforms an image into a 256x256 tensor while keeping the aspect ratio, and

# therefore may result in potential letterboxing.

node {

calculator: "ImageToTensorCalculator"

input\_stream: "IMAGE\_GPU:image"

output\_stream: "TENSORS:input\_tensor"

output\_stream: "LETTERBOX\_PADDING:letterbox\_padding"

options: {

[mediapipe.ImageToTensorCalculatorOptions.ext] {

output\_tensor\_width: 192

output\_tensor\_height: 192

keep\_aspect\_ratio: true

output\_tensor\_float\_range {

min: 0.0

max: 1.0

}

border\_mode: BORDER\_ZERO

gpu\_origin: TOP\_LEFT

}

}

}

# Generates a single side packet containing a TensorFlow Lite op resolver that

# supports custom ops needed by the model used in this graph.

node {

calculator: "TfLiteCustomOpResolverCalculator"

output\_side\_packet: "opresolver"

options: {

[mediapipe.TfLiteCustomOpResolverCalculatorOptions.ext] {

use\_gpu: true

}

}

}

# Loads the palm detection TF Lite model.

node {

calculator: "PalmDetectionModelLoader"

input\_side\_packet: "MODEL\_COMPLEXITY:model\_complexity"

output\_side\_packet: "MODEL:model"

}

# Runs a TensorFlow Lite model on GPU that takes an image tensor and outputs a

# vector of tensors representing, for instance, detection boxes/keypoints and

# scores.

node {

calculator: "InferenceCalculator"

input\_stream: "TENSORS:input\_tensor"

output\_stream: "TENSORS:detection\_tensors"

input\_side\_packet: "CUSTOM\_OP\_RESOLVER:opresolver"

input\_side\_packet: "MODEL:model"

options: {

[mediapipe.InferenceCalculatorOptions.ext] {

delegate { gpu {} }

}

}

}

# Generates a single side packet containing a vector of SSD anchors based on

# the specification in the options.

node {

calculator: "SsdAnchorsCalculator"

output\_side\_packet: "anchors"

options: {

[mediapipe.SsdAnchorsCalculatorOptions.ext] {

num\_layers: 4

min\_scale: 0.1484375

max\_scale: 0.75

input\_size\_width: 192

input\_size\_height: 192

anchor\_offset\_x: 0.5

anchor\_offset\_y: 0.5

strides: 8

strides: 16

strides: 16

strides: 16

aspect\_ratios: 1.0

fixed\_anchor\_size: true

}

}

}

# Decodes the detection tensors generated by the TensorFlow Lite model, based on

# the SSD anchors and the specification in the options, into a vector of

# detections. Each detection describes a detected object.

node {

calculator: "TensorsToDetectionsCalculator"

input\_stream: "TENSORS:detection\_tensors"

input\_side\_packet: "ANCHORS:anchors"

output\_stream: "DETECTIONS:unfiltered\_detections"

options: {

[mediapipe.TensorsToDetectionsCalculatorOptions.ext] {

num\_classes: 1

num\_boxes: 2016

num\_coords: 18

box\_coord\_offset: 0

keypoint\_coord\_offset: 4

num\_keypoints: 7

num\_values\_per\_keypoint: 2

sigmoid\_score: true

score\_clipping\_thresh: 100.0

reverse\_output\_order: true

x\_scale: 192.0

y\_scale: 192.0

w\_scale: 192.0

h\_scale: 192.0

min\_score\_thresh: 0.5

}

}

}

# Performs non-max suppression to remove excessive detections.

node {

calculator: "NonMaxSuppressionCalculator"

input\_stream: "unfiltered\_detections"

output\_stream: "filtered\_detections"

options: {

[mediapipe.NonMaxSuppressionCalculatorOptions.ext] {

min\_suppression\_threshold: 0.3

overlap\_type: INTERSECTION\_OVER\_UNION

algorithm: WEIGHTED

}

}

}

# Adjusts detection locations (already normalized to [0.f, 1.f]) on the

# letterboxed image (after image transformation with the FIT scale mode) to the

# corresponding locations on the same image with the letterbox removed (the

# input image to the graph before image transformation).

node {

calculator: "DetectionLetterboxRemovalCalculator"

input\_stream: "DETECTIONS:filtered\_detections"

input\_stream: "LETTERBOX\_PADDING:letterbox\_padding"

output\_stream: "DETECTIONS:detections"

}

**Palm Detection Model**

We created a single-shot detector model (which is implemented in GoogleNet) tailored for mobile real-time purposes, comparable to the face detection model in MediaPipe Face Mesh, to detect initial hand placements. Hand detection is a difficult task: both our lite and complete models must be able to detect occluded and self-occluded hands and work across a wide range of hand sizes with a big scale spread (>20x) relative to the image frame. Whereas faces contain strong contrast patterns, such as around the eyes and mouth, hands lack similar traits, making it more difficult to accurately distinguish them based on their visual features alone. Providing additional context, such as arm, body, or human traits, instead helps with precise hand localization.

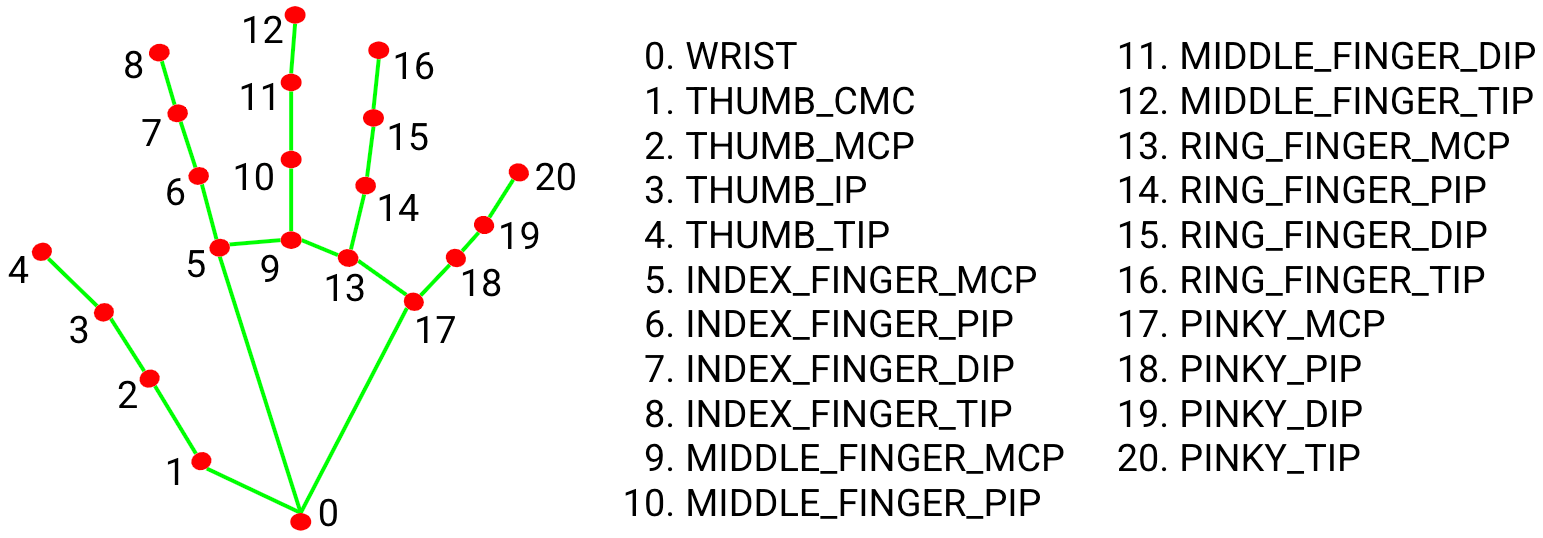
Our strategy employs a variety of strategies to address the aforementioned issues. First, instead of training a hand detector, we train a palm detector because estimating bounding boxes of inflexible objects like palms and fists is much easier than recognizing hands with articulated fingers. Furthermore, because palms are smaller objects, the non-maximum suppression method performs effectively even in two-hand self-occlusion situations such as handshakes. Furthermore, palms can be simulated using square bounding boxes (anchors in ML language) that ignore other aspect ratios, resulting in a reduction of 3-5 anchors. Second, even for little objects, an encoder-decoder feature extractor is used for larger picture context awareness (similar to the RetinaNet approach). Finally, because to the significant scale variance, we limit focus loss during training to support a large number of anchors.

With the above techniques, we achieve an average precision of 95.7% in palm detection. Using a regular cross entropy loss and no decoder gives a baseline of just 86.22%.

**Hand Landmark Model**

After the palm detection over the whole image our subsequent hand landmark model performs precise keypoint localization of 21 3D hand-knuckle coordinates inside the detected hand regions via regression, that is direct coordinate prediction. The model learns a consistent internal hand pose representation and is robust even to partially visible hands and self-occlusions.

We manually tagged 30K real-world photos with 21 3D coordinates to obtain ground truth data, as seen below (we take Z-value from image depth map, if it exists per corresponding coordinate). We also render a high-quality synthetic hand model over various backgrounds and map it to the associated 3D coordinates to better cover the available hand poses and provide additional supervision on the nature of hand geometry.



**Output:**

1. *MULTI-HAND-LANDMARKS:*

A collection of detected/tracked hands, each of which is represented by a list of 21 hand landmarks, each of which is made up of the letters *x, y,* and *z*. The image width and height are used to normalize *x* and *y* to *[0.0, 1.0]*. The landmark depth is represented by *z*, with the origin being the depth at the wrist, and the smaller the value, the closer the landmark is to the camera. The magnitude of *z* is measured using a scale that is similar to that of *x*.

1. *MULTI\_HAND\_WORLD\_LANDMARKS:*

In this collection of detected/tracked hands, each hand is represented by a list of *21 hand landmarks* in world coordinates. Each landmark consists of three numbers: *x, y, and z*, which represent real-world 3D coordinates in meters, with the origin at the hand's approximate geometric center.

1. *MULTI\_HANDEDNESS:*

Collection of the detected/tracked hands' handedness (i.e. is it a left or right hand). Each hand is made up of two parts: a *label* and a *score*. *label* is a string with the value *"Left"* or *"Right"* in it. *score* is the projected handedness's estimated probability, which is always greater than or equal to *0.5*. (and the opposite handedness has an estimated probability of *1 - score*).

**Gesture Volume Control**

Now we used this Hand Detection Module to actually control the volume of my computer. We accomplished the above task by creating 2 additional libraries:

1. Hand\_Tracking\_Module.py:

This file uses the mediapipe library to use all the functionality of hands.py file.

***Code:***

import cv2  
import mediapipe as mp  
import time  
  
class handDetector():  
 def \_\_init\_\_(self, mode=False, max\_hands=4,  
 model\_complexity=1, detection\_confidence=0.5, tracking\_confidence=0.5):  
 self.mode = mode  
 self.max\_hands = max\_hands  
 self.model\_complexity = model\_complexity  
 self.detection\_confidence = detection\_confidence  
 self.tracking\_confidence = tracking\_confidence  
  
 self.mp\_Hands = mp.solutions.hands  
 self.hands = self.mp\_Hands.Hands(self.mode, self.max\_hands, self.model\_complexity,  
 self.detection\_confidence, self.tracking\_confidence)  
 self.mpDraw = mp.solutions.drawing\_utils  
  
 def find\_hands(self, image, draw = True):  
  
 image\_RGB = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)  
 self.results = self.hands.process(image\_RGB)  
  
 #print(results.multi\_hand\_landmarks)  
  
 if self.results.multi\_hand\_landmarks:  
 for hand\_landmarks in self.results.multi\_hand\_landmarks:  
  
 if draw:  
 # To draw the points and connections on hands  
 self.mpDraw.draw\_landmarks(image, hand\_landmarks, self.mp\_Hands.HAND\_CONNECTIONS)  
  
 return image  
  
 def find\_position(self, image, hand\_number=0, draw=True):  
  
 landmark\_list = []  
  
 if self.results.multi\_hand\_landmarks:  
 myhand = self.results.multi\_hand\_landmarks[hand\_number]  
  
 for id, landmark in enumerate(myhand.landmark):  
 h, w, c = image.shape  
  
 cx, cy = int(landmark. x \* w), int(landmark. y \* h)  
 #print(id, cx, cy)  
 landmark\_list.append([id, cx, cy])  
  
 # Making a circle around a landmark  
 if draw:  
 cv2.circle(image, (cx, cy), 7, (255, 0, 0), cv2.FILLED)  
  
 return landmark\_list  
  
  
  
def main():  
 previous\_time = 0  
 current\_time = 0  
 cap = cv2.VideoCapture(0)  
 detector = handDetector()  
  
 while True:  
 success, image = cap.read()  
 image = detector.find\_hands(image)  
 landmark\_list = detector.find\_position(image)  
 if len(landmark\_list) != 0:  
 print(landmark\_list[4])  
  
 current\_time = time.time()  
 fps = 1 / (current\_time - previous\_time)  
 previous\_time = current\_time  
  
 cv2.putText(image, str(int(fps)), (10, 70), cv2.FONT\_HERSHEY\_PLAIN, 3, (255, 0, 255), 3)  
 cv2.imshow("Image", image)  
 cv2.waitKey(1)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

1. Volume\_Gesture\_Control.py:

This file works in tandem with the Hand\_Tracking\_Module to ultimately give us the functionality to control the volume of my pc by just using the fingers.

***Code:***

import cv2  
import time  
import numpy as np  
import Hand\_Tracking\_Module as htm  
import math  
from ctypes import cast, POINTER  
from comtypes import CLSCTX\_ALL  
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume  
  
  
devices = AudioUtilities.GetSpeakers()  
interface = devices.Activate(  
 IAudioEndpointVolume.\_iid\_, CLSCTX\_ALL, None)  
volume = cast(interface, POINTER(IAudioEndpointVolume))  
  
#volume.GetMute()  
#volume.GetMasterVolumeLevel()  
volume\_range = volume.GetVolumeRange()  
min\_vol = volume\_range[0]  
max\_vol = volume\_range[1]  
  
vol\_bar = 400  
vol = 0  
vol\_percentage = 0  
  
  
# height and width of the screen  
width\_cam = 640  
height\_cam = 480  
  
capture = cv2.VideoCapture(0)  
capture.set(3, width\_cam)  
capture.set(4, height\_cam)  
  
previous\_time = 0  
  
detector = htm.handDetector(detection\_confidence=0.7)  
while True:  
  
 success, image = capture.read()  
 image = detector.find\_hands(image)  
 landmark\_list = detector.find\_position(image, draw=False)  
  
 if len(landmark\_list) != 0:  
 print(landmark\_list[4], landmark\_list[8])  
  
 x1, y1 = landmark\_list[4][1], landmark\_list[4][2]  
 x2, y2 = landmark\_list[8][1], landmark\_list[8][2]  
 # Center of the line  
 cx, cy = (x1 + x2) // 2, (y1 + y2) // 2  
  
 cv2.circle(image, (x1, y1), 10, (255, 0, 0), cv2.FILLED)  
 cv2.circle(image, (x2, y2), 10, (255, 0, 0), cv2.FILLED)  
 cv2.line(image, (x1, y1), (x2, y2), (255, 0, 0), 3)  
 cv2.circle(image, (cx, cy), 10, (255, 0, 0), cv2.FILLED)  
  
 length = math.hypot(x2 - x1, y2 - y1)  
  
 vol = np.interp(length, [50, 250], [min\_vol, max\_vol])  
 vol\_bar = np.interp(length, [50, 300], [400, 150])  
 vol\_percentage = np.interp(length, [50, 300], [0, 100])  
  
  
  
 print(int(length), vol)  
 volume.SetMasterVolumeLevel(vol, None)  
  
 if length < 50:  
 cv2.circle(image, (cx, cy), 10, (0, 255, 0), cv2.FILLED)  
  
 cv2.rectangle(image, (50,150), (85, 400), (255, 0, 0), 3)  
 cv2.rectangle(image, (50,int(vol\_bar)), (85, 400), (255, 0, 0), cv2.FILLED)  
 cv2.putText(image, f'{int(vol\_percentage)}%', (40, 450), cv2.FONT\_HERSHEY\_COMPLEX, 1, (255, 0, 0), 3)  
  
  
  
  
 current\_time = time.time()  
 fps = 1 / (current\_time - previous\_time)  
 previous\_time = current\_time  
  
 cv2.putText(image, f'FPS: {int(fps)}', (40, 40), cv2.FONT\_HERSHEY\_COMPLEX, 1, (255, 0, 0), 3)  
  
 cv2.imshow("Image", image)  
 cv2.waitKey(1)

**Further Improvements**

Right now as we know, out hand\_detection\_module is using GoogleNet. But I would also like to try the ResNet architecture and notice the improvement or deterioration in the performance.

In addition to this, I would also like to train it to modulate other aspects of my pc, like minimizing or closing a window, just by hand gesture.

Finally I would like to combine this Hand Detection module with Face Detection and Iris Detection module.

**References**

<https://google.github.io/mediapipe/solutions/hands.html>

<http://www.no-free-lunch.org/>.

Witten, I. H., and Eibe Frank. Data Mining: Practical Machine Learning Tools and Techniques. Amsterdam: Morgan Kaufman, 2005