

# Interactive Information Wall with Android Lantern

by

WONG Yin Wai, Natalie

Submitted in partial fulfillment of the requirements for the  
degree of

Bachelor of Science (Honours)  
in Computer Science

Hong Kong Baptist University

April, 2020

## Declaration

I hereby declare that all the work done in this Final Year Project is of my independent effort. I also certify that I have never submitted the idea and product of this Final Year Project for academic or employment credits.

---

WONG Yin Wai, Natalie

Date: \_\_\_\_\_

Hong Kong Baptist University  
Computer Science Department

We hereby recommend that the Final Year Project submitted by WONG Yin Wai, Natalie entitled “Interactive Information Wall with Android Lantern” be accepted in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science.

\_\_\_\_\_

Dr. CHOY, Martin Man Ting

Supervisor

\_\_\_\_\_

Dr. HUANG, Xin

Observer

Date: \_\_\_\_\_

Date: \_\_\_\_\_

# Abstract

Nowadays, Augmented Reality (AR) has become popular and many of the innovative projects across the industries have been applying the AR technology to enhance user experience for a task or product. In the coming future, taking the advantage of Projection-based AR, screens are no longer needed. Instead, computer-generated graphics are projected onto real-world surfaces and human are allowed to directly interact with the virtual graphics.

The current Interactive Information Wall which uses a Kinect Sensor and computer system to detection hand motions could be replaced by the Android Lantern, a portable interactive device which consists of a Raspberry Pi 3 single-board computer, a laser projector and a camera module turning any surface into a fully interactive user interface. Users are able to “click” on the virtually projected graphics using their hands and receive real-time response from the Lantern.

Hand detection machine learning models are trained using TensorFlow Object Detection API to recognize human hands in images captured by the camera module embedded in the Lantern. An Android application is deployed on the Raspberry Pi 3 to process the images, trigger the “click” event and provide the user interface for the Interactive Information Wall.

In view of the low applicability of the existing hand dataset and time-consuming manual image annotation procedures, an auto CSV and TFRecord files generator for hand detection model training is developed in order to build a hand dataset which tailors to the needs of the Interactive Information Wall for certain gestures to achieve the “click” event and automates the procedures of annotating the collected images by using

OpenCV API for hand detection.

# Table of Contents

Abstract	i
Abbreviations	
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.1.1 Augmented Reality (AR) . . . . .	1
1.1.2 Projection-based AR . . . . .	2
1.1.3 Current Interactive Information Wall using Xbox Kinect	2
1.2 Objective . . . . .	2
1.2.1 Problems of the Current Interactive Information Wall	3
1.2.1.1 <i>System - Slow Response Time</i> . . . . .	3
1.2.1.2 <i>Kinect Sensor - Insensitive to User's Gestures</i>	3
1.2.2 Android Lantern - an Alternative to the Kinect Sensor and current Computer System . . . . .	3
<b>2 Application Overview</b>	<b>5</b>
2.1 Application Architecture . . . . .	5
2.1.1 Hardware . . . . .	5
2.1.2 Software . . . . .	5
2.1.2.1 User Interface (UI) . . . . .	6
2.1.2.2 Interactive Feature brought by Computer Vision and Machine Learning . . . . .	6
2.2 Existing Methods for Object Detection . . . . .	9
2.2.1 ML Kit's Object Detection and Tracking (ODT) API	9
2.2.1.1 Problems with the ML Kit's ODT model .	9
2.2.2 Hand Detection Models trained using TensorFlow Ob- ject Detection API . . . . .	10

<b>3</b>	<b>Method</b>	<b>14</b>
3.1	Proposed Method . . . . .	14
3.1.1	State Diagram of the Android Lantern Interactive Information Wall . . . . .	14
3.1.1.1	Workflow of Hand Detection Process for the Implementation of Interactive Feature . . .	15
3.2	Implementation . . . . .	18
3.2.1	User Interface of Android application . . . . .	18
3.2.1.1	Fragments . . . . .	18
3.2.1.2	Data Binding with Observable Data Objects and MVVM Architecture . . . . .	19
3.2.1.3	Fragment Transition for Slideshow in Gallery page . . . . .	24
3.2.2	Hand Detection Model Training Process . . . . .	28
3.2.2.1	Problems of University of Oxford's Hand Dataset . . . . .	32
3.2.2.2	Success in Using EgoHands dataset for Hand Detection Model Training . . . . .	35
3.2.3	Hand Detection in Android application . . . . .	37
3.2.3.1	Image Capturing in Android application . .	37
3.2.3.2	Essential Procedures for Processing an Image to Detect a Hand . . . . .	40
3.2.3.3	Hand Position Tracking on UI . . . . .	42
3.2.3.4	Custom MotionEvent for Click Action . . .	44
3.3	Additional Python Scripts for Hand Detection Model Training	46
3.3.1	Auto CSV and TFRecord files Generator for TensorFlow Hand Detection Model Training . . . . .	46
3.3.1.1	Motivations . . . . .	46
3.3.1.2	Modification of the Python scripts in Real-Time Hand Gesture Detection project . . .	48
3.3.1.3	Limitations . . . . .	52
3.3.1.4	Possible Ways to Overcome the Limitations	54
<b>4</b>	<b>Performance Results</b>	<b>56</b>

4.1	Comparison of Pre-trained Object Detection Models' Performance . . . . .	56
4.2	Performance of Inference using different Hand Detection Models	58
4.2.1	Re-trained from Quantized SSD MobileNet V1 . . . . .	58
4.2.1.1	Total Training Time of the models . . . . .	59
4.2.1.2	TensorBoard Records for Model Training and Evaluation . . . . .	59
4.2.1.3	Performance of inference on Samsung Note 3	60
4.2.1.4	Performance of inference on Raspberry Pi 3 B+ Model through Android application . .	61
4.2.1.5	Problems of using <code>object_detection/model_main.py</code> for model training . . . . .	61
4.2.2	Re-trained from Quantized SSD MobileNet V2 . . . . .	61
4.2.2.1	Total Training Time of the models . . . . .	62
4.2.2.2	TensorBoard Records for Model Training and Evaluation . . . . .	62
4.2.2.3	Performance of inference on Raspberry Pi 3 B+ Model through Android application . .	63
4.2.2.4	Performance of inference on Samsung Note 3	64
4.2.2.5	Observation of the Confidence Level among the Detection Results . . . . .	65
4.2.3	Problem for All the Models . . . . .	65
4.2.4	Discovery regarding the use of Python Scripts for Object Detection Model Training . . . . .	65
4.2.5	Conclusion of the Hand Detection Models' Performance	66
4.2.5.1	Criteria of a Quality Hand Dataset . . . . .	68
4.2.5.2	Evaluation on the Quality of the Datasets .	68
4.3	Using Android OS instead of Android Things . . . . .	68
4.3.1	Problems of using Android Things on Raspberry Pi 3	69
<b>5</b>	<b>Discussions</b>	<b>70</b>
5.1	Limitations . . . . .	70
5.1.1	Dependency on Color Features for Hand Detection Model . . . . .	70



5.1.1.1	Weaker Computational Power of Raspberry Pi . . . . .	72
5.1.2	Black Background Color for Android application UI to Smoothen Hand Detection . . . . .	72
5.1.3	Single Hand Detection in Android application . . . . .	73
5.1.4	Latency for Hand Detection on Raspberry Pi . . . . .	73
5.2	Further Developments . . . . .	74
5.2.1	Implementation . . . . .	74
5.2.1.1	Using libGDX API to improve the speed for the conversion of image format from YUV to RGB . . . . .	74
5.2.1.2	Coral USB Accelerator for Hand Detection Model Inferencing . . . . .	76
5.2.1.3	Re-train a hand detection model which supports gesture recognition . . . . .	76
5.2.1.4	Improvement on the Presentation of UI Components . . . . .	77
<b>6</b>	<b>Conclusion</b>	<b>78</b>
 <b>Appendix 1: User Guide of Auto CSV and TFRecord files</b>		
	<b>Generator for TensorFlow Hand Detection Model Training</b>	<b>80</b>
6.0.1	Environment . . . . .	80
6.0.2	Dependencies . . . . .	81
6.0.3	Label for TensorFlow Object Detection model . . . . .	81
6.0.3.1	Structure of files and directories created after the execution of the programs . . . . .	81
6.0.4	Essential Operations . . . . .	82
6.0.4.1	Run <code>python HandDetection.py</code> , the main program . . . . .	82
6.0.4.2	Run <code>python split_train_test_csv_images.py</code> afterwards . . . . .	83
6.0.4.3	Run <code>python generate_tfrecord.py</code> at last . . . . .	83
6.0.5	Optional Operation . . . . .	84
	<b>References</b>	<b>85</b>

# Abbreviations

API	Application Programming Interface
ML	Machine Learning

# Chapter 1

## Introduction

### 1.1 Background

#### 1.1.1 AUGMENTED REALITY (AR)

The definition of AR is that views of physical real-world environments are augmented with superimposed computer-generated images, hence enhancing users' current perception of the reality.

AR is gaining popularity in these days. Many companies and organizations are developing innovative projects based on AR technology and it has been widely applied in the fields of gaming, education, healthcare, retail and architecture etc.

With vivid visual overlay added to the view of the real world, user experience for a task or product is greatly enhanced, bringing enormous positive impact to our daily lives as well as the industries.

We are able to predict the AR technology will continue growing rapidly in the coming future.

### 1.1.2 PROJECTION-BASED AR

Projection-based AR is one of the categories of AR technology. It can be done by using a projector, a camera and a computer.

Computer-generated graphics are projected onto real-world surfaces using a projector. Interaction between human and the projected graphics are allowed by using a camera to recognize real-world objects and detect movements, and a computer to make responses.

Projection-based AR provides a future vision of without the use of screens. It tries to merge the virtual graphics onto the environment in the reality. “Projection Mapping” is a related technology which relies on the use of computer vision to 3D scan the complex scenes in the environment, thus turning any real-world object into a screen for image projection.

### 1.1.3 CURRENT INTERACTIVE INFORMATION WALL USING XBOX KINECT

The Interactive Information Wall located in the Sir Run Run Shaw Building (RRS) on the 7/F at HKBU is currently using the projection-based AR technology to project information on the wall.

- A projector is mounted on the wall to project content on the wall.
- A Kinect sensor is used to provide interaction between human and the projected computer graphics. It senses user’s gestures and transmits data to a desktop application running on a computer.

## 1.2 Objective

This project aims to enhance the Interactive Information Wall at HKBU by further utilizing the technology of Projection-based AR.

## 1.2.1 PROBLEMS OF THE CURRENT INTERACTIVE INFORMATION WALL

The current Interactive Information Wall has the following 2 major problems.

### 1.2.1.1 *System - Slow Response Time*

In the index page, user has to raise up their hand to summon the menu. After that, however, the system takes an unknown time, usually more than 3 minutes to respond.

### 1.2.1.2 *Kinect Sensor - Insensitive to User's Gestures*

User has to move the palm in the air to control the cursor. However, the Kinect sensor is a bit insensitive to the palm's movements, and sometimes a cursor lag will happen. User may encounter difficulties in controlling the cursor and move it to the desired position on the projected screen.

There is another issue regarding the current design of the "click" event. User has to hold their hand in the air still for about 3 seconds to complete the "click" event.

## 1.2.2 ANDROID LANTERN - AN ALTERNATIVE TO THE KINECT SENSOR AND CURRENT COMPUTER SYSTEM

Inspired by the Android Things Lantern developed by Nord Projects, the current Kinect sensor and the system could be replaced by the Android Lantern which can transform any surface into a projected user interface in the hope of shortening the response time by developing an Android application to enhance user experience and allowing user interacting with the virtually projected graphics in a real-time manner by making use of Computer Vision and Machine Learning.



Figure 1.1: Android Things Lantern - Hardware

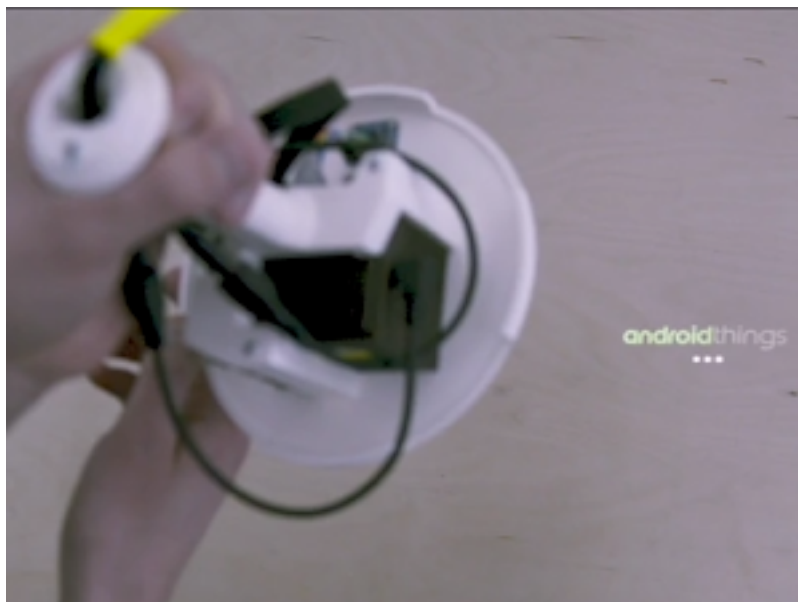


Figure 1.2: Android Things Lantern - Projection

# Chapter 2

## Application Overview

### 2.1 Application Architecture

#### 2.1.1 HARDWARE

Android Lantern is a portable interactive device comprised of

- a single-board computer — Raspberry Pi 3 Model B+
- a palm-sized laser projector, and
- a Raspberry Pi camera module.

The Raspberry Pi camera module is the Computer Vision used for the implementation of interactive feature on the Information Wall.

#### 2.1.2 SOFTWARE

An Android application is developed to provide the user interface and implement the interactive feature of the Interactive Information Wall.

Android 10 is installed on the Raspberry Pi to enable the deployment of the Android application.

Kotlin and Java are the main programming languages for the Android application development. Android Studio is used as the tool for building the application.

### 2.1.2.1 User Interface (UI)

The Information Wall displays three kinds of information, which are

- News,
- Gallery, and
- Department Staff Information.

Each of the above information has its own landing page showing thumbnails of the available items of such information, and inner pages (except Gallery) showing the details of each item. There is a home page which displays three titles corresponding to the above information, allowing users to navigate in the application.

	News	Gallery	Department Staff Information
Landing page			
Item page			

### 2.1.2.2 Interactive Feature brought by Computer Vision and Machine Learning

On the Information Wall displayed by the Android Lantern, users can interact with the virtually projected graphics by “clicking” them using a hand.



The Raspberry Pi camera module is an essential component enabling the Lantern to “see” human hands by capturing images from the real world. The technique of object detection from ML is applied to train a model which can recognize human hands in the images captured by the camera module.

The Android application contains all the programming logic to perform the “click” event and update the UI.

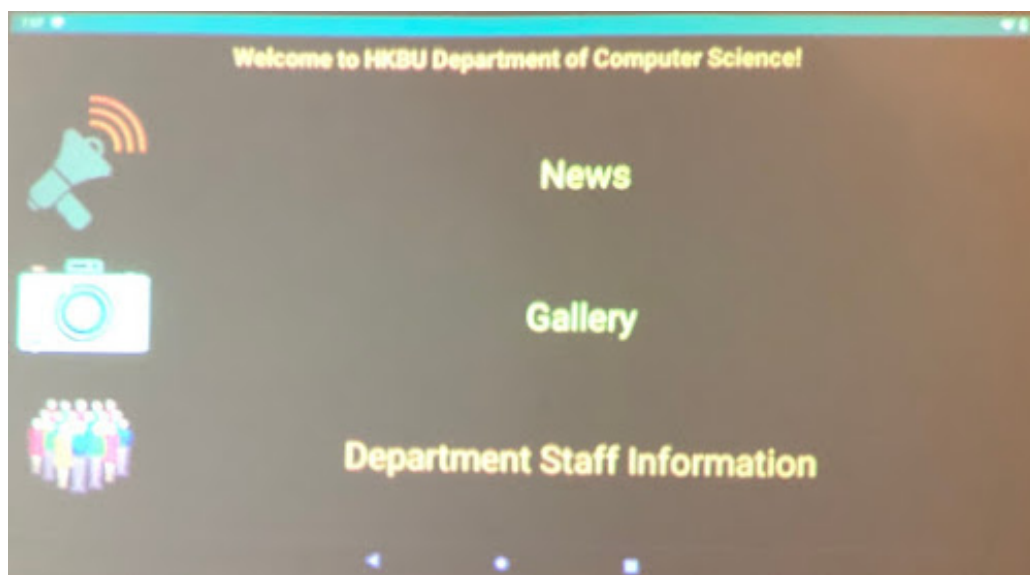


Figure 2.1: Interactive Information Wall with Android Lantern - Home page

## 2.2 Existing Methods for Object Detection

I had researched into different machine learning approaches for the implementation of hand detection in Android application.

### 2.2.1 ML KIT'S OBJECT DETECTION AND TRACKING (ODT) API

**ML Kit** is a mobile SDK developed by Google. It is one of the best tools for programmers who are new to ML compared to other ML frameworks because of its provision of the ready-to-use API. This enables faster mobile application development progress.

Bounding box surrounding a detected object in each image can be obtained through the following lines of code.

```
for (obj in detectedObjects) {  
    ...  
    val bounds = obj.boundingBox // returns a Rect object  
    ...  
}
```

I can make use of the bounding box to obtain the location of a user's hand on the projected area of the Lantern so as to implement a click event allowing the user to interact with the virtually projected graphics.

#### 2.2.1.1 Problems with the ML Kit's ODT model

The ready-to-use API provided by ML Kit hinders variations during on-device inference.

The existing ODT model can only classify objects into 5 coarse categories which are `FASHION_GOOD`, `FOOD`, `HOME_GOOD`, `PLACE`, `PLANT` and `UNKNOWN`. Therefore, I cannot directly apply this model for hand detection. I have

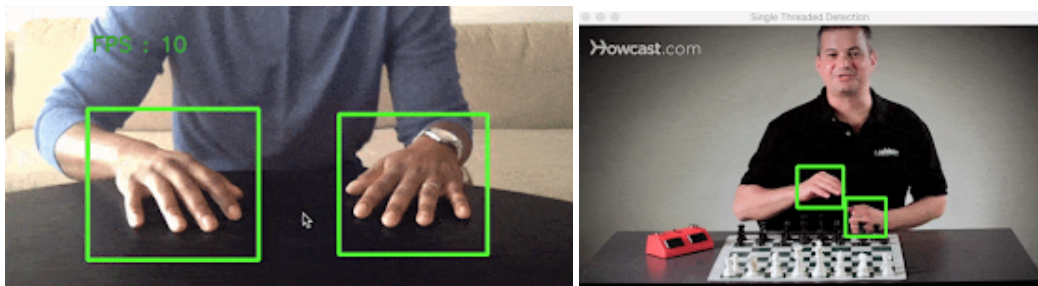
to train a custom hand detection model by myself using TensorFlow Object Detection API.

## 2.2.2 HAND DETECTION MODELS TRAINED USING TENSORFLOW OBJECT DETECTION API

### Real-time Hand Detector using Neural Networks (SSD) on Tensorflow by Victor Dibia

The hand detection model trained in this project produces excellent results in tracking hand, with an average precision of 96.86%. It is a re-trained model based on a pre-trained standard SSD MobileNet V1 model with 200,000 training steps using the technique of Transfer Learning. Standard SSD MobileNet V1 model was selected because it is one of the fastest pre-trained object detection model, with a speed of 30 milliseconds for inference. The training process was run on a cloud GPU machine which takes around 5 hours to complete it.

The hand detection process is targeted to be run on videos or personal computers with a web camera capturing video stream using a custom Python script in the project. Multiple hands can be detected in each frame of the video or video stream from the web camera. Bounding boxes would be drawn in each frame to visualize the detection results.

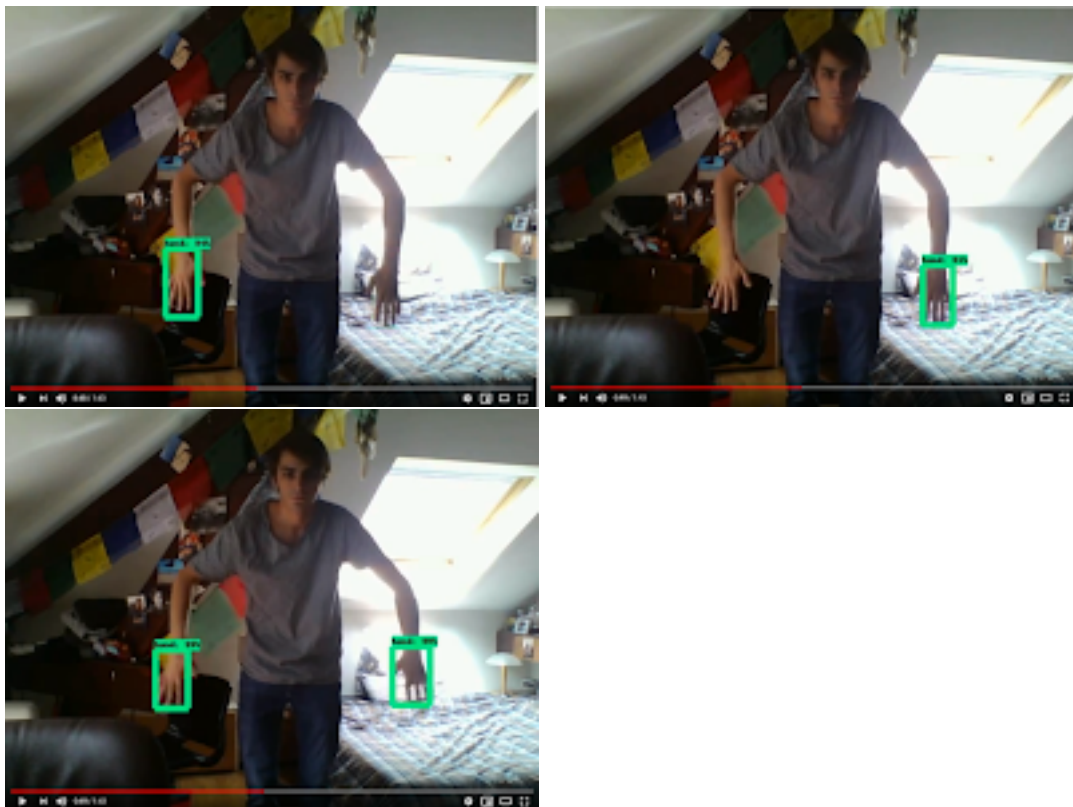


The dataset being used to train this model is EgoHands prepared by Indiana University. Hands were annotated and assigned to 4 classes, **own left**, **own right**, **other left**, and **other right**. In the project, the 4 classes are merged into one class — **hand** so there is just one output label in the detection results.

## Hands Detection in Video Stream by Loïc Marie

The hand detection model in the project was re-trained from a pre-trained Faster R-CNN model using the Hand Dataset prepared by the University of Oxford. All hands in the images of this dataset were labeled as **hand** only. The model was trained on the Google CloudML.

From the project's demo video, hands in each frame captured by web camera were detected and bounding boxes were drawn in each frame to visualize the detection results. The average precision of this hand detection model is 99%, however, the hand detection process was a bit slow and not smooth enough.



This is a tradeoff of the Faster R-CNN model. Although the average precision of the detection results is improved, the detection speed is at least the double of a SSD MobileNet model, ranging from 58 to 1,833 milliseconds [1].

## Hand Tracking (GPU) in Google MediaPipe framework

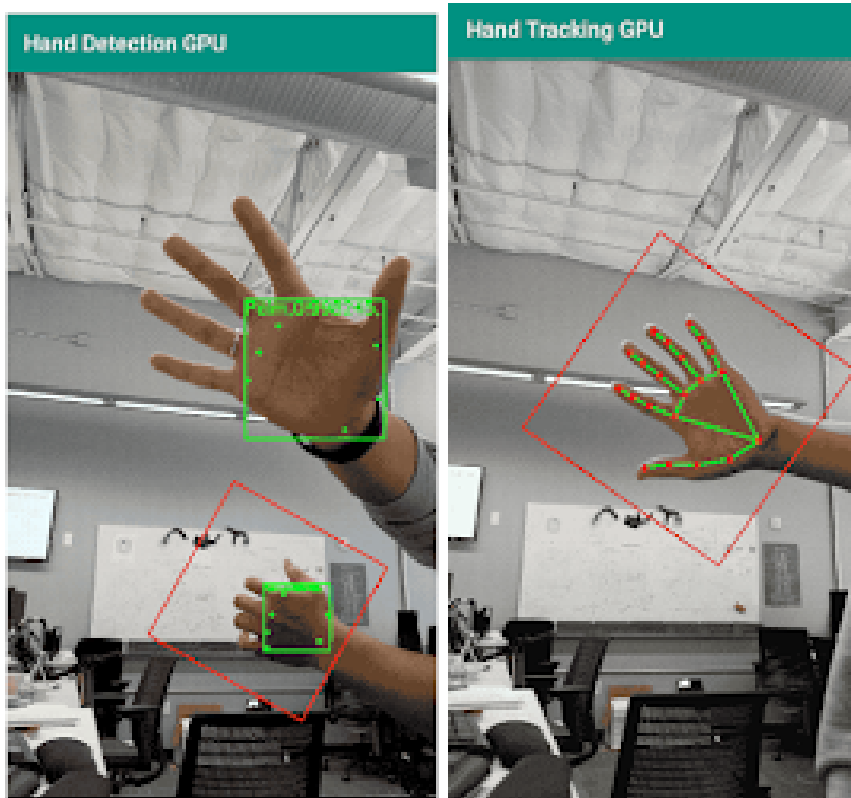
This project is developed and maintained by Google. This project is a bit different than the other hand detection project as it uses two TensorFlow

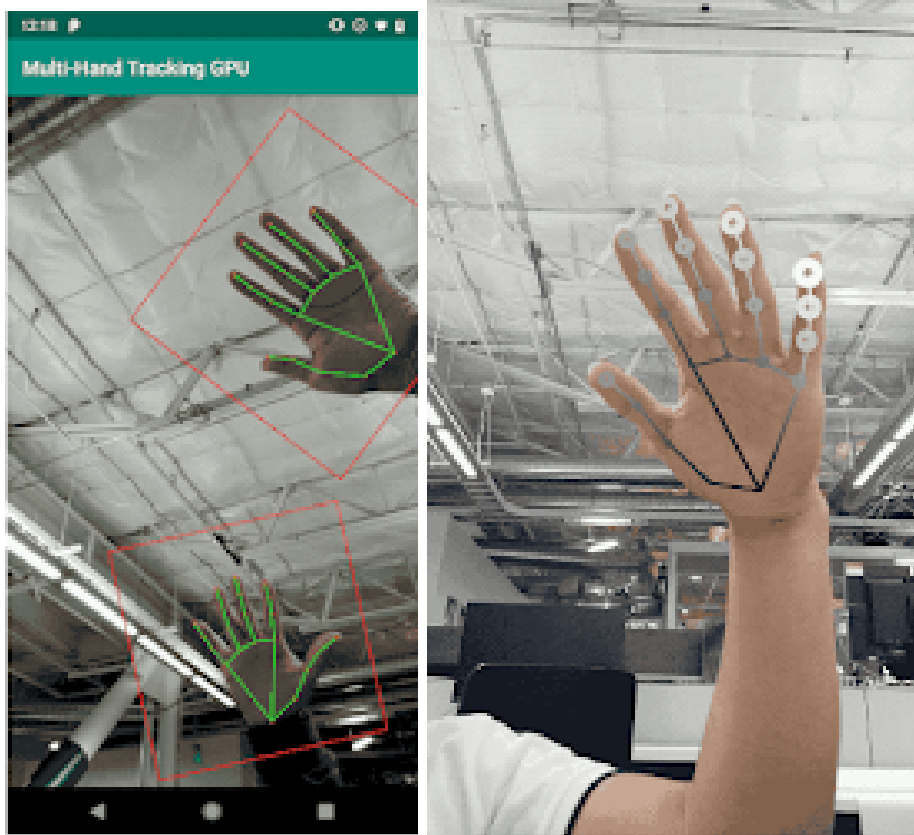
Object detection in TFLite format, one for palm detection, and another one for hand landmark detection which visualizes the hand skeleton.

The following is a brief hand detection process in the project.

1. `palm_detection.tflite`, the palm detection model, is being used first. It defines a cropped image region of a hand.
2. Then, `hand_landmark.tflite`, the hand landmark model, is being used to operate on the cropped image region and visualize the key-points of a hand skeleton.

This Hand Tracking project aims to detect only one hand. There is another Multi-Hand Tracking project performing multi-hand detection and tracking. The `hand_landmark_3d.tflite` supports visualizing hand landmarks in 3D.





The hand detection models in the project are custom models and trained from scratch. The palm detection model has an average precision of 95.7% [2]. The hand landmark model is trained based on a dataset of around 30,000 real-world images and a number of synthetic hand images rendered by computers. All the images were annotated with 21 3D coordinates showing the keypoints of a hand skeleton. The hand landmark model could be used for gesture recognition based on the predicted hand skeleton.

# Chapter 3

## Method

### 3.1 Proposed Method

#### 3.1.1 STATE DIAGRAM OF THE ANDROID LANTERN INTER-ACTIVE INFORMATION WALL

After the Lantern is powered on, it launches the Android application and shows the Home page of the Information Wall. There is 3 items with the title of “News”, “Gallery” and “Department Staff Information”.

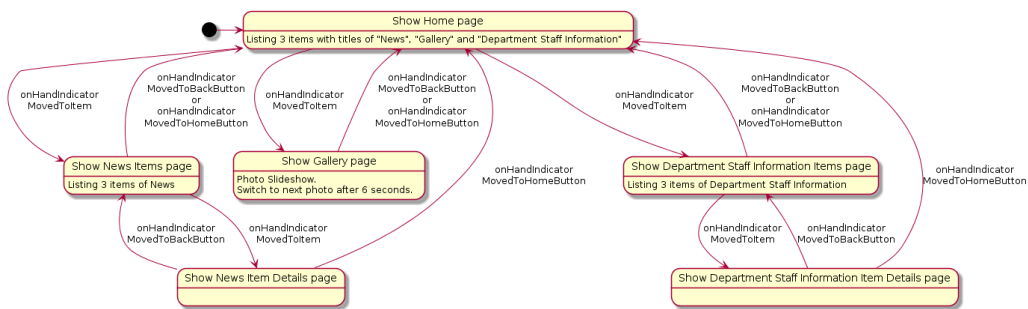


Figure 3.1: State Machine Diagram of Lantern Information Wall



### **3.1.1.1 Workflow of Hand Detection Process for the Implementation of Interactive Feature**

The camera module keeps capturing image. For each image captured, the MainActivity passes it to an Interpreter for hand detection.

A hand detection ML model is trained to recognize a hand. The interpreter utilizes the model to conduct the inference in each image and returns the detection results to the MainActivity after the inference process completes. If there is more than one results, the MainActivity selects the best result and records the result's location on the projected area.

When a user's hand moves into the projected area of the Lantern, a hand indicator is shown on the UI which visualizes the location of the hand on the projected area for the user to navigate the pages in the Android application. When the user's hand moves away from the projected area of the Lantern, the hand indicator disappears.

To trigger a "click" event, the user first controls the hand indicator by waving his/her hand, then moves the hand indicator to the desired position on the UI and finally moves away his/her hand from the projected area of the Lantern. The "click" event is triggered based on last position where the hand indicator appeared on the UI.

In the Home page, if the last position of the hand indicator was shown on the News item, the News item is clicked and the Android application navigates to the News landing page where there are 3 titles of the news articles shown on the page. An item of the news articles is clicked if the last position of the hand indicator was shown on it, bringing user to the details page of the selected news article. The user can navigate up to the previous page or Home page by moving the hand indicator to the "Back" button or the "Go Home" button on the top navigation bar. This "clicking" and navigation processes are also applied to the pages of Department Staff Information.

The Gallery is a page for photo slideshow only. There are a total of 6 photos and each of them stays on the UI for 6 seconds before it is switched to another photo. User can "click" on the "Back" button or "Go Home"

button to return to the Home page.

Google Cloud Firestore is being used to be the database of the Android application storing all the data of the New, Gallery and the Department Staff Information pages. It is a NoSQL database where data is called document and referenced using a path. It also provides API that could be used in the Android application to retrieve data with ease.

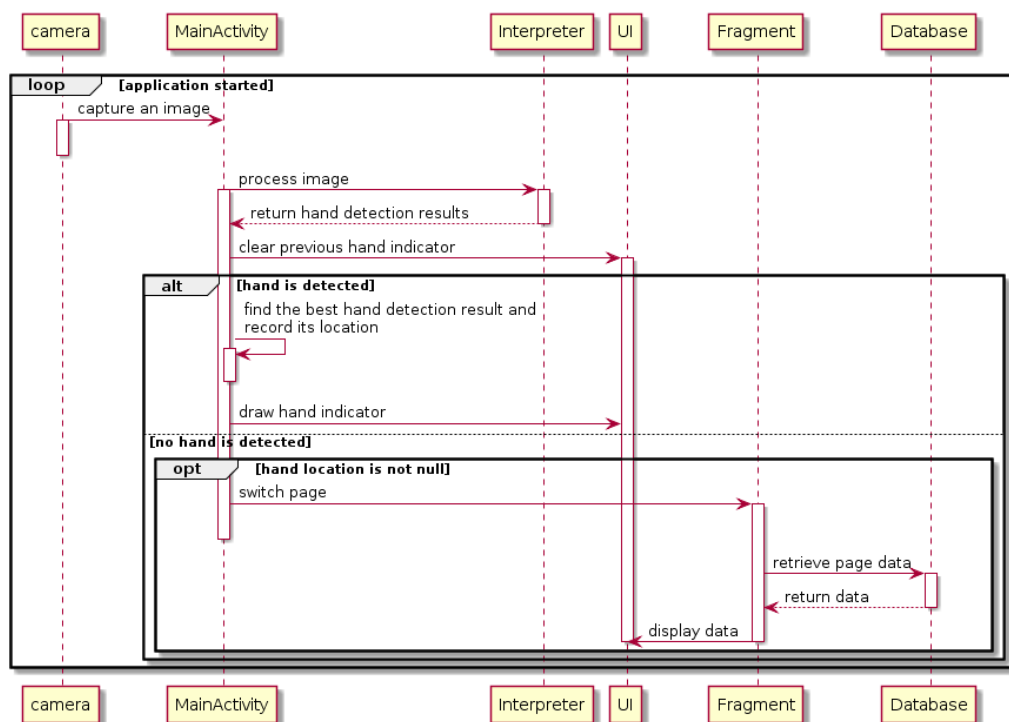


Figure 3.2: Sequence Diagram of Workflow for Interactive Feature

## 3.2 Implementation

### 3.2.1 USER INTERFACE OF ANDROID APPLICATION

#### 3.2.1.1 Fragments

The News page, Gallery page, Department Staff Information page and their inner pages exist as fragments in the Android application. Figure 3.3 shows a graphical version of the navigation graph in `mobile_navigation.xml`.

In `activity_main.xml`, all the layout of the fragments defined in `mobile_navigation.xml` will be connected to the fragment layout below so that user can navigate around the Android application to view different pages.

```
<fragment
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    ...
    app:defaultNavHost="true"
    app:navGraph="@navigation/mobile_navigation" />
```

The use of fragments allows easy implementation of navigation in the Android application. In `mobile_navigation.xml`, I can add an `action` tag to link up two fragments. Here is an example showing that the home fragment is linked up with the New, Gallery as well as the Department Staff Information pages' fragments.

```
<fragment
    android:id="@+id/nav_home"
    android:name="com.example.lanterninfowall.ui.home.HomeFragment"
    tools:layout="@layout/home_fragment">
    <action
        android:id="@+id/action_nav_home_to_nav_gallery"
        app:destination="@id/nav_gallery" />
```

```

<action
    android:id="@+id/action_nav_home_to_nav_news"
    app:destination="@id/nav_news" />
<action
    android:id="@+id/action_nav_home_to_nav_staffinfo"
    app:destination="@id/nav_staffinfo" />
</fragment>

```

In MainActivity.kt, calling

```
findNavController(R.id.nav_host_fragment).navigate(R.id.nav_news)
```

will bring the user to the News page's fragment, and calling

```
findNavController(R.id.nav_host_fragment).navigateUp()
```

will bring the user back to previous page's fragment.

### 3.2.1.2 Data Binding with Observable Data Objects and MVVM Architecture

Data binding is a popular alternative approach to the standard `findViewById()` method for binding data to UI components using a declarative format.

In `staff_info_item_fragment.xml`,

```

<TextView
    ...
    android:text="@{staff.name}"
    ... />
<TextView
    ...
    android:text="@{staff.title}"
    ... />

```

The MVVM stands for Model-View-ViewModel. The main advantage of using the MVVM architecture in mobile applications is to decouple the UI's logic from the main program's logic, making the application easier to manage.

In the Android application, I have created ViewModel classes which declare all the relevant data to be shown in the page layouts. For each page, say the News landing page, there are one fragment class, a XML layout file and a ViewModel class to achieve data binding.

I have applied one-way data binding in the Android application. The fragment class contains all the programming logic including retrieving data from the database but it will not directly interact with the UI components in the layout. Instead, a ViewModel class instance is created in the fragment. The ViewModel class, together with a fragment-specific data binding class, can be seen as the middle persons responsible for the communication between the fragment class and the XML layout file.

For example, in `NewsFragment.kt`,

```
class NewsFragment : Fragment() {
    ...
    private lateinit var viewModel: NewsViewModel

    // fragment-specific data binding class
    private lateinit var binding: NewsFragmentBinding
    ...
    override fun onCreateView(...): View? {
        binding = NewsFragmentBinding.inflate(...).apply {
            ...
            vm = ViewModelProvider(this)
                .get(NewsViewModel::class.java)
        }
        ...
    }
}
```

As there are lots of news articles and department staff information in the

database, `RecyclerView` is used in the layout of the News and Department Staff Information landing pages showing the thumbnails which consist of the news headings and the names of the staff respectively.

In `news_fragment.xml`,

```
<data>
    <variable
        name="vm"
        type="com.example.lanterninfowall.ui.staffinfo.NewsViewModel" />
</data>
...
<androidx.recyclerview.widget.RecyclerView
    ...
    app:itemBinding="@{vm.newsItemBinding}"
    app:items="@{vm.newsItems}"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/>
```

In `NewsViewModel.kt`,

```
val newsItems = ObservableArrayList<NewsItemViewModel>()

val newsItemBinding = ItemBinding.of<NewsItemViewModel>
    (BR.news, R.layout.news_item_fragment)
```

A `ViewModel` class has class attributes which refer to all the relevant data to be shown on the UI. With the help of the `ViewModel` class and data binding, I can easily create thumbnails and bind them to the `RecyclerView` by using the binding adapters `items` and `itemBinding` in the XML layout file as well as the `ItemBinding` class in the `ViewModel` class.

Additionally, I have to create another `ViewModel` class and XML layout file for each thumbnail.

In `news_item_fragment.xml`,

```

<data>
    ...
    <variable
        name="news"
        type="com.example.lanterninfowall.ui.home.NewsItemViewModel"/>
</data>
...
<TextView
    ...
    android:text="@{news.heading}"
    ... />

```

In NewsItemViewModel.kt,

```

class NewsItemViewModel : ViewModel() {
    ...
    val heading = MutableLiveData<String>().apply { value = "" }
}

```

The Observable and MutableLiveData classes are used along with data binding to enable automatic real-time updates on the data fields of the UI components.

Back to the NewsFragment.kt, I have instantiated an object of the thumbnail's ViewModel class and assigned values to the class attributes for each data of the news items retrieved from the database.

```

binding.vm?.apply {

    FirebaseFirestore.getInstance().collection("news")
        .addSnapshotListener { querySnapshot, exception ->

            newsItems.clear()
            newsItems.addAll(querySnapshot?.documents?.map { doc ->

```



```

        // instantiate a thumbnail's ViewModel object
        val item = NewsItemViewModel()
        ...
        // assign the value retrieved from the database
        // to the class attribute of the ViewModel object
        item.heading.value = doc.getString("heading")
        ...
    }
}
}

```

For each assignment, the UI components of a thumbnail are notified and they update the data fields to display the newly assigned values. Then, the UI component — **RecyclerView** in the `news_fragment.xml` is notified to display the UI components of the thumbnail. The procedures repeat until the fragment class has done the mapping for all the news item data, therefore, a number of thumbnails are shown on the News landing page. This strategy is also applied to the Department Staff Information landing page.

Taking the advantage of the observable data objects and data binding, I have used the `addSnapshotListener()` callback method of the Google Cloud Firestore API in the fragments so that whenever there is new data inserted into the database, the callback method will be triggered and the UI components as well as their data fields will be automatically updated. This is particularly useful for the pages in which data is constantly changing, like the news articles in the News pages.

If **ViewModel** class and data binding are not used, we have to call `findViewById()` method to bind data to the layout so the main program's logic will be mixed with the UI's logic. Moreover, without using the **MutableLiveData**, we have to manually call `postInvalidate()` for each UI components so as to update the data field. These makes the main program to be lengthy and hard to manage.

### 3.2.1.3 Fragment Transition for Slideshow in Gallery page

In the Gallery page, taking the advantage of data binding with the `FragmentManager.beginTransaction()` and `handler.postDelayed()` methods, I am able to achieve image slideshow by just creating two instances of slideshow item fragment to show many images and their associated information on the UI.

With the help of the ViewModel and data binding classes, I can reuse the two slideshow item fragments so as to keep the program logic to be as minimal as possible. Before the fragment transaction begins, I assigned all the values retrieved from the database to another item fragment's ViewModel class attributes. As the `MutableLiveData` object can notify the UI to update the changes on the data fields, after the `commit()` method for the fragment transition is called, the data fields showing the image and its information in text form are automatically updated.

In `gallery_fragment.xml`,

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/slideshow">
    ...
</androidx.constraintlayout.widget.ConstraintLayout>
```

In `GalleryFragment.kt`,

```
val fragment = itemFragments[itemFragmentIndex++ % 2]
...
GalleryItemFragmentBinding.vm?.let {
    it.photoURL.value = ...
    it.alt.value = ...
    it.qrcodeURL.value = ...
}
```

```

fm.beginTransaction()
    .addToBackStack(null)
    .setCustomAnimations(android.R.anim.slide_in_left,
                        android.R.anim.slide_out_right)
    .replace(R.id.slideshow, fragment)
    .commit()

handler.postDelayed(this, 6000)

```

By repeating the procedures of ViewModel attributes' assignment to another item fragment, the two instances of slideshow item fragment can interchange with each other, showing different images and text each time.

In order to keep the slideshow item staying on the screen for a few seconds, say 6 seconds, I have applied the `handler.postDelayed()` method to add delays between the ViewModel attributes' assignment and the fragment transition

Since the images for the slideshow can be constantly changing, it is impossible to download all the images to the Android application and load them within the application. To resolve this issue, I have used the `load()` method of Coil, an image loading library specialized for Android application.

I have to first create a `BindingAdapters.kt` class.

```

object BindingAdapters {
    @BindingAdapter("app:imageSrc")
    @JvmStatic fun setImageSrc(view: ImageView, src:String) {
        view.load(src)
    }
}

```

Then I am able to assign a URL of the image in `String` form to the binding adapter in the layout file through the corresponding ViewModel attribute.

In `GalleryFragment.kt`,

```
GalleryItemFragmentBinding.vm?.let {  
    it.photoURL.value =  
        "https://www.comp.hkbu.edu.hk/v1/pic/news/971.jpg"  
    ...  
}
```

In `gallery_item_fragment.xml`,

```
<ImageView  
    ...  
    app:imageSrc="@{vm.photoURL}"  
    ... />
```

Finally, the image will be loaded if the URL can be accessed without any problem.

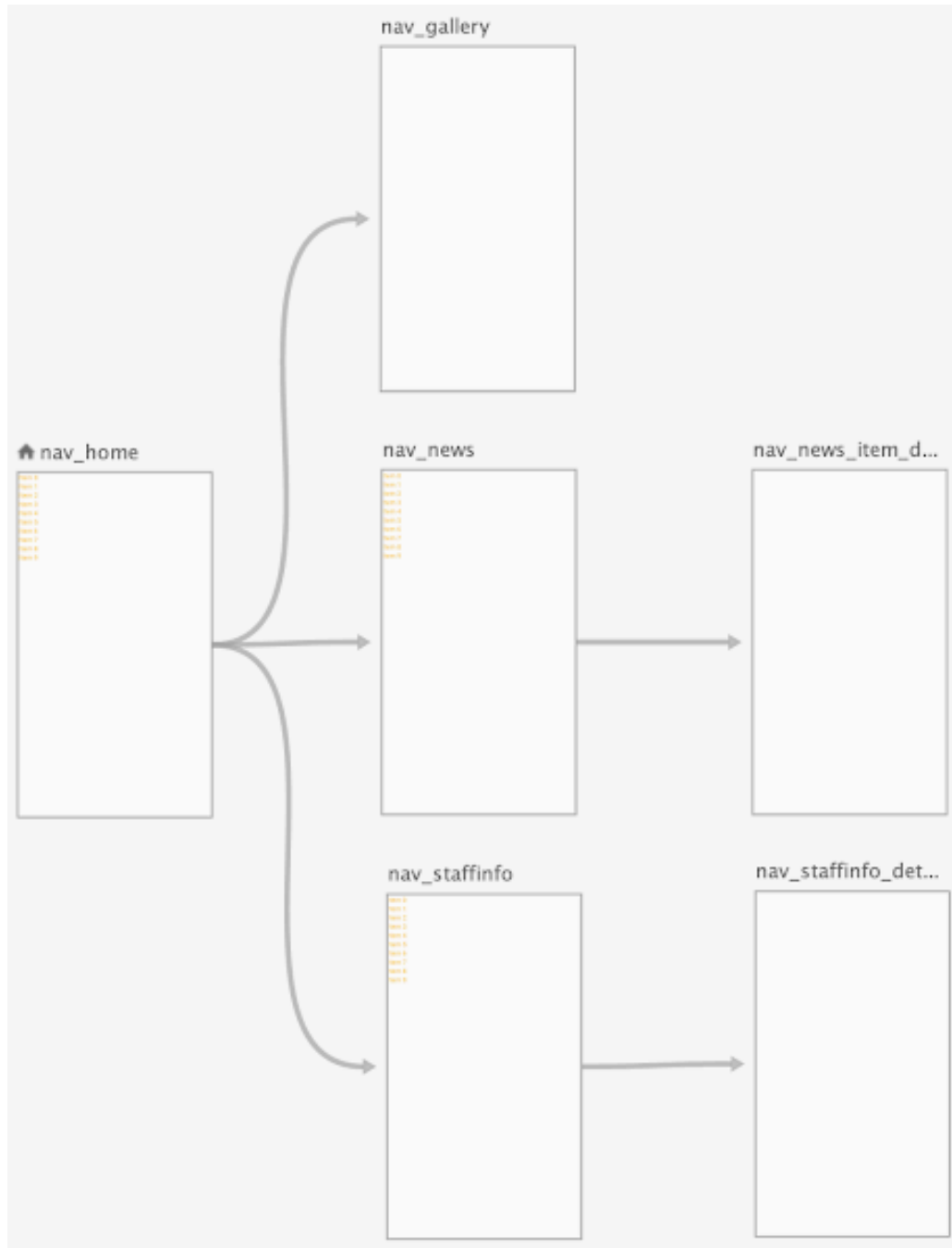
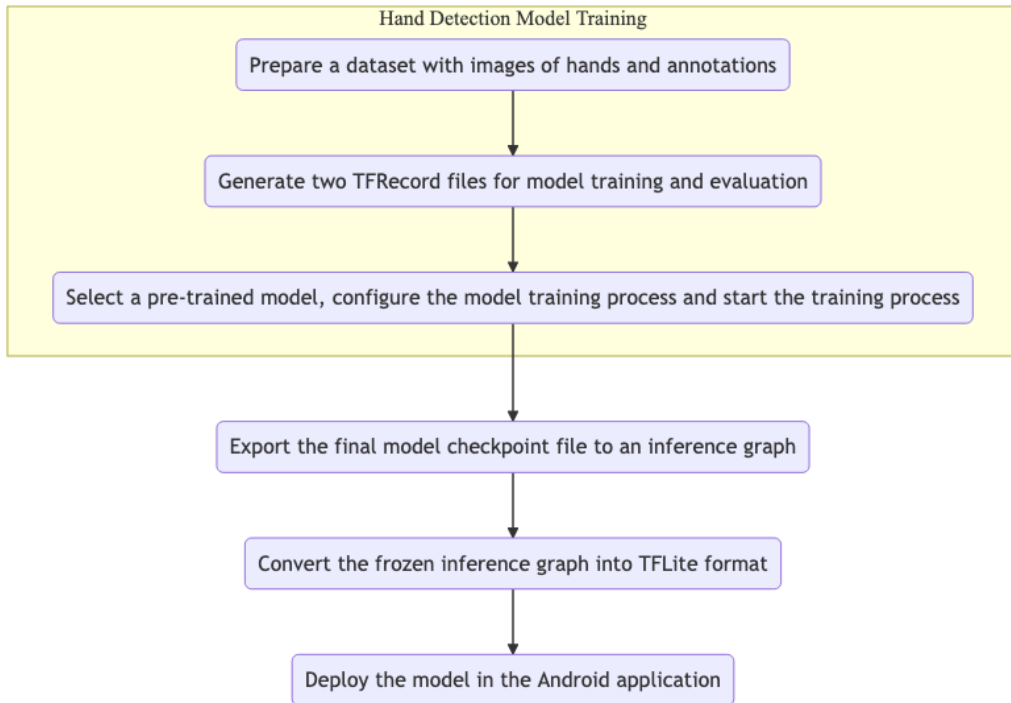


Figure 3.3: Navigation Graph of Fragments

### 3.2.2 HAND DETECTION MODEL TRAINING PROCESS

Training a hand detection model is one of the vital steps to enable a user interacting with the virtual graphic projected on a wall.

I have applied **transfer learning** technique to enable rapid development of a hand detection model. A well pre-trained TensorFlow object detection model is reused and re-trained its final layer of the neural network with minimal training data and less computational power to classify new class label, i.e. hand.



The training process of a hand detection model based on a pre-trained Object Detection model using TensorFlow Object Detection API involves 3 essential steps.

1. Prepare a dataset with images of hands and annotations.

For each image, the locations of hands are annotated. For each location of a hand, there are two coordinates associated with it, which are the minimum value of the x axis and the maximum value of the y axis (**xmin**, **ymin**) as well as the maximum value of the x axis and the

minimum value of the y axis (`xmax`, `ymin`).

All the images in the dataset are of the same size with the image that is going to be recognized by the model. In this project, the Raspberry Pi camera module captures images with a size of 1280 x 720 so the size of the images in the dataset are also 1280 x 720.

The images are separated into two folders, one for model training and another one for model evaluation. In each image folder, there is a CSV file which contains all the essential information about the images under the folder, the coordinates of the hand(s) in each image and the class label - `hand`.

In `train_labels.csv`,

```
filename,width,height,class,xmin,ymin,xmax,ymax
CHESS_COURTYARD_BT_frame1038.jpg,1280,720,hand,771,638,1031,718
JENGA_OFFICE_HT_frame1805.jpg,1280,720,hand,290,367,414,704
PUZZLE_COURTYARD_HT_frame1237.jpg,1280,720,hand,525,268,672,419
...
```

2. Generate two TFRecord files from the CSV files by running `generate_tfrecord.py`.

A TFRecord file contains all the information stated in a CSV file as well as the images' bytes. After the TFRecord files are generated, the images and CSV files are no longer needed.

3. Download a pre-trained Object Detection model, configure the model training process and start the model training by running Python scripts.

I have selected quantized SSD MobileNet V1 and V2 pre-trained object detection model from Tensorflow detection model zoo for the re-training process. This is because currently only SSD MobileNet models can be converted into TFLite format, which is optimized to be executed efficiently on mobile and embedded devices with limited computational and memory resources like mobile phones and Raspberry Pi.

I have to modify a few lines in `pipeline.config` under the pre-trained model folder so as to customise the training pipeline and strategy suitable for my dataset.

- `num_classes`: 1. There is only one class label - hand.
- Set the `num_steps`: to the total number of training steps for the hand detection model.
- Set the `num_examples`: to the total number of evaluation steps for the hand detection model.
- Set the `fine_tune_checkpoint`: to the path of `model.ckpt` under the pre-trained model folder.
- Set the `input_path`: under the `train_input_reader`: and `eval_input_reader`: sections to the location of TFRecord files for model training and evaluation respectively.
- Set the `label_map_path`: to the location of a label map in `.pbtxt` format. I have created a `hands_label_map.pbtxt` for the hand detection model training process.

```
item {
  id: 1
  name: 'hand'
}
```

To start training a hand detection model, I have run the `object_detection/model_main.py`, or `object_detection/legacy/train.py` plus `object_detection/legacy/eval.py` under the TensorFlow models' object detection repository. The `model_main.py` is the newer Python script for training of object detection model. This program enables the model training and evaluation to be done at the same time. In another words, the training of model starts first, then after a certain period of time, the training process is stopped and evaluation of the model starts. After the evaluation is done, the training process starts again. The training and evaluation processes are done alternatively throughout until the final step of the model training is reached. If legacies are used for the hand detection model



training, I have to run the `train.py` first, following by the `eval.py` so as to achieve training and evaluating the model at the same time.

After the training process finished, I have to export the final model checkpoint file to an inference graph by running `object_detection/export_tflite_ssd_graph.py` and `tflite_graph.pb` and `tflite_graph.pbtxt` are generated. The frozen inference graph is then converted into TFLite format by using the command `tflite_convert`.

```
tflite_convert \
--output_file="../../../hand_detect.tflite" \
--graph_def_file="../tflite_graph.pb" \
--inference_type=QUANTIZED_UINT8 \
--input_arrays=normalized_input_image_tensor \
--output_arrays='TFLite_Detection_PostProcess',
                 'TFLite_Detection_PostProcess:1',
                 'TFLite_Detection_PostProcess:2',
                 'TFLite_Detection_PostProcess:3' \
--mean_values=128 \
--std_dev_values=128 \
--input_shapes=1,300,300,3 \
--change_concat_input_ranges=false \
--allow_nudging_weights_to_use_fast_gemm_kernel=true \
--allow_custom_ops
```

It is notable that the `input_shapes` of the pre-trained SSD MobileNet model is set to be 300 x 300, meaning that an image which is going to be recognized should have the same size. As my hand detection model is re-trained from a pre-trained model, I have to follow the configuration of the pre-trained model for my hand detection model.

### 3.2.2.1 Problems of University of Oxford's Hand Dataset

I have first tried using the hand dataset prepared by University of Oxford to train a hand detection model. This dataset has 4,070 images for model training and 822 images for model evaluation.

In Loïc Marie's Hands Detection project, the `create_inputs_from_dataset.py` helps me to download the dataset from the website, read the hand annotation files in `.mat` format and generate TFRecord files. I have used the `hands_train.record` and `hands_val.record` for my hand detection model training. I have chosen the SSD MobileNet V1 pre-trained model and started the training by running the `object_detection/model_main.py`

#### *Errors encountered:*

I first started the model training process in a Docker machine installed on Macbook Pro through VirtualBox. The target number of training steps is 6,000 and Python 2.7 was used to execute the Python script. The training was started without any error, however, during the model evaluation, I found that the detection precision was always zero. VirtualBox is a software used for virtualization, hence making the Docker container becoming a virtual machine instead. As hardware is virtualized under virtual machines, I believed that there were some problems or errors occurred during the accessing of the computer's hardware resources.

After I had started the training using Python 3.6 on Macbook Pro, the `TypeError: object of type <class 'numpy.float64'> cannot be safely interpreted as an integer` occurred. I suspected that this is caused by a few problems. - The arithmetic operation of division is different between Python 2 and Python 3. In Python 2, the division operation discards the decimal places and result is in the type of `int`.

```
```python
1 / 200
# The result is 0.
```
```

In Python 3, the result division operation is in the type of `float`.

```
```python
1 / 200
# The result is 0.005.
```
```

As I was using `Python3.6` to run the model training Python script, this error

- The `numpy` version being used. I have installed the latest `numpy` version (1.18.0), however, the `float64` is not supported starting from `numpy` version 1.12.0.
- TensorFlow Lite operations target at `float32` for floating-point inference but perhaps `numpy.float64` had been used to generate TFRecord files.

Besides the problems that different versions of libraries and dependencies are being used, I have discovered some problems regarding the quality of images in this hand dataset.

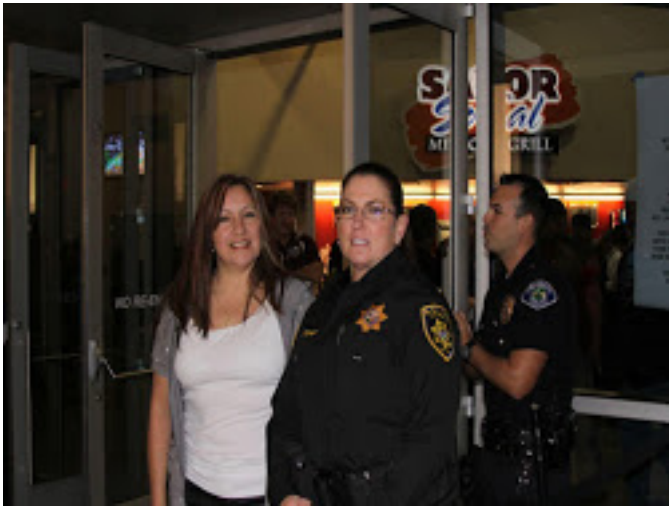

- Hands in the images are quite small which cannot be seen clearly.
- Images vary in size. The dimensions of the images are any combination ranging from 90 x 100 to 1280 x 1024 pixels.
- The resolution of most of images that are used for training are low. Many of them have dimensions below 500 x 500 pixels.



612 x 704



305 x 321

| Images   | Dimensions |
|--|------------|
|  | 1280 x 960 |
|  | 131 x 200  |

### 3.2.2.2 Success in Using EgoHands dataset for Hand Detection Model Training

From Victor Dibia's hand detector project, I learned about the EgoHands dataset prepared by Indiana University. This dataset has 4,401 images for model training and 401 images for model evaluation and I found that this dataset is in high quality.

- Hands in the images are big and can be seen clearly. This is because

all the images are captured from an egocentric view by Google Glasses.

- All images are in the same size of 1280 x 720 pixels, the HD standard format.



The `egohands_dataset_clean.py` provided in Victor Dibia's hand detector project helps me to download the dataset from the website, clean the files in the dataset and generate two CSV files, `train_labels.csv` and `test_labels.csv`. The cleaning process includes

- reading the `polygons.mat` files which contains the annotations of the hand bounding boxes in all the images,
- renaming all the filename of the images to make sure each filename is unique, and
- splitting the dataset, i.e. images, into two folders, around 83% for training and 10% for testing.

Then, I have to run `generate_tfrecord.py` to generate two TFRecord files, `train.record` and `test.record` for model training and evaluation respectively. The hand detection model training and evaluation run smoothly without any error using Python 2.7 and the SSD MobileNet V1/V2 pre-trained model. I am also able to frozen the inference graph of the model and convert it in TFLite format.

### 3.2.3 HAND DETECTION IN ANDROID APPLICATION

To perform hand detection in the Android application, I have to add the hand detection model file in TFLite format to the Android application project's **assets** folder, capture images from the camera module and call suitable methods provided by TensorFlow for inference using the model.

#### 3.2.3.1 Image Capturing in Android application

To capture images from the camera module, I had used Camera2 API to take photos with 1 millisecond delay.

```
val imageCaptureRunnable = {  
    mCamera.takePicture()  
}  
...  
handler.postDelayed(imageCaptureRunnable, 1)
```

The method `OnImageAvailableListener` will be called whenever an image is captured from the camera module. From this method, I am able to obtain the image in JPEG format for hand detection.

```
private val imageAvailableListener =  
    ImageReader.OnImageAvailableListener { reader ->  
        ....  
        // Obtain the image captured from the camera  
        val image = reader.acquireLatestImage()  
        val imageBuffer = image.planes[0].buffer  
        val imageBytes = ByteArray(imageBuffer.capacity())  
        imageBuffer.get(imageBytes)  
  
        // Obtain a bitmap of the image  
        val bitmap = getBitmapFromByteArray(imageBytes,  
            image.width, image.height)
```

```

    ...
}

```

However, taking photos with 1 millisecond delay is not fast enough. The Lantern is considered to be an interactive device so it is important to achieve real-time hand detection.

To improve the speed for hand detection, I have added `AutoFitTextureView` which is a custom camera preview view. The `AutoFitTextureView` captures images in real-time, just like a video capturing a number of frames per second. Therefore, I am able to obtain more images than taking photos one by one in a certain period of time, say in a second.

In `activity_main.xml`,

```

<com.example.lanterninfowall.AutoFitTextureView
    android:id="@+id/texture"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

```

In `MainActivity.kt`,

```

textureView = findViewById(R.id.texture)
...
val texture = textureView.surfaceTexture!!
...
val surface = Surface(texture)
previewRequestBuilder = mCameraDevice
    .createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW)
previewRequestBuilder.addTarget(surface)
...
previewReader = ImageReader.newInstance(
    viewWidth, viewHeight,
    ImageFormat.YUV_420_888, 1)
previewReader.setOnImageAvailableListener(

```



```
        imageAvailableListener,mCameraHandler)
previewRequestBuilder.addTarget(previewReader.surface)
...
```

***Problem encountered:***

I have reused the `OnImageAvailableListener` to obtain the images captured by the preview view. However, errors occurred at the line `previewRequestBuilder.addTarget(previewReader.surface)` so I cannot obtain the byte buffer of an image captured by the preview view from the `OnImageAvailableListener`.

Even if `previewRequestBuilder.addTarget(surface)` worked well and there was a `TextureView.SurfaceTextureListener` with the `onSurfaceTextureUpdated(texture: SurfaceTexture)` method, I was still unable to obtain the byte buffer of an image captured by the preview view.

***Solution: Use of Camera API instead of Camera2 API***

After reviewing the TensorFlow Object Detection demo project, I found that I am able to obtain the byte array of an image captured by a preview view from the `onPreviewFrame` callback method provided in `Camera.PreviewCallback` class which belongs to the Camera API.

The parameter `bytes` is the image byte buffer and `addCallbackBuffer(bytes)` enables the callback method being called continuously whenever an image is available on the preview view.

```
val previewCallback = Camera.PreviewCallback { bytes, camera ->
    ...
    camera.addCallbackBuffer(bytes)
}
```

Moreover, I discovered that the custom preview view `AutoFitTextureView` is no longer needed. To obtain an image byte buffer using Camera API, I can just override the `onPreviewFrame` of `Camera.PreviewCallback` class

and do the following procedures, keeping the lines of code in the program as minimal as possible.

```
// camera setup
val parameters = camera.parameters
...
parameters.focusMode =
    Camera.Parameters.FOCUS_MODE_CONTINUOUS_PICTURE
parameters.setPreviewSize(IMAGE_WIDTH, IMAGE_HEIGHT)
camera.parameters = parameters
...
// camera preview view setup
camera.setPreviewCallbackWithBuffer(previewCallback)
camera.addCallbackBuffer(ByteArray(ImageUtils.getYUVByteSize(
    IMAGE_WIDTH, IMAGE_HEIGHT)))
camera.startPreview()
```

### 3.2.3.2 Essential Procedures for Processing an Image to Detect a Hand

As the hand detection model can only accept an image bitmap with a size of 300 x 300 while the image captured from the camera module has a size of 1820 x 720, I have to create a matrix `frameToCropTransform` which shrinks the original size of the image to fit the one required by the hand detection model using a utility method `getTransformationMatrix` provided by the `ImageUtils.java` class in the demo project.

```
private val IMAGE_WIDTH = 1280
private val IMAGE_HEIGHT = 720
private val TF_OD_API_INPUT_SIZE = 300

frameToCropTransform = ImageUtils.getTransformationMatrix(
    IMAGE_WIDTH, IMAGE_HEIGHT,
```

```

        TF_OD_API_INPUT_SIZE, TF_OD_API_INPUT_SIZE,
        0, MAINTAIN_ASPECT)

```

The preview view captured an image in YUV format but the hand detection model only works on a bitmap in RGB format. Therefore, I have to do a conversion on the image format. For each image captured by the preview view, I just passed the byte buffer to another utility method `convertYUV420SPToARGB8888` in `ImageUtils.java` to perform conversion of image format.

```

rgbBytes = IntArray(IMAGE_WIDTH * IMAGE_HEIGHT)
...
val previewCallback = Camera.PreviewCallback { bytes, camera ->
    ImageUtils.convertYUV420SPToARGB8888(bytes, IMAGE_WIDTH,
        IMAGE_HEIGHT, rgbBytes)

    processImage() // Perform hand detection inside this method
    ...
}

rgbFrameBitmap = Bitmap.createBitmap(IMAGE_WIDTH, IMAGE_HEIGHT,
    Bitmap.Config.ARGB_8888)
...

private fun processImage() {
    rgbFrameBitmap.setPixels(rgbBytes, 0, IMAGE_WIDTH, 0, 0,
        IMAGE_WIDTH, IMAGE_HEIGHT)
    ...
}

```

After a bitmap in RGB format is prepared, I have used the `Classifier.java` and `TFLiteObjectDetectionAPIModel.java` class provided by the demo project for hand detection. `recognizeImage()` is the only method that I need to detect a hand in the image. This method accepts a bitmap with a

size of 300 x 300 and returns a list of `Classifier.Recognition` objects if hands are detected in the image.

```
private var detector: Classifier? = null

private val croppedBitmap = Bitmap
    .createBitmap(TF_OD_API_INPUT_SIZE, TF_OD_API_INPUT_SIZE,
        Bitmap.Config.ARGB_8888)
    ...

private fun processImage() {
    ...
    // Prepare a cropped Bitmap of an image
    // with a size of 300 * 300
    val canvas = Canvas(croppedBitmap)
    canvas.drawBitmap(rgbFrameBitmap, frameToCropTransform,
        Paint())

    // detect a hand in the image
    val results = detector?.recognizeImage(croppedBitmap)
    ...
}
```

### 3.2.3.3 Hand Position Tracking on UI

I am able to obtain the position of hand in the image by calling the method `getLocation()` for each `Classifier.Recognition` object. To notify the user whether a hand has been detected, I have added a custom view `TrackingView` and draw a red dot showing the location of the user's hand on the UI by overriding the `onDraw()` method.

In `activity_main.xml`,

```
<com.example.lanterninfowall.TrackingView
    android:id="@+id/tracking_view"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent">
```

In TrackingView.kt,

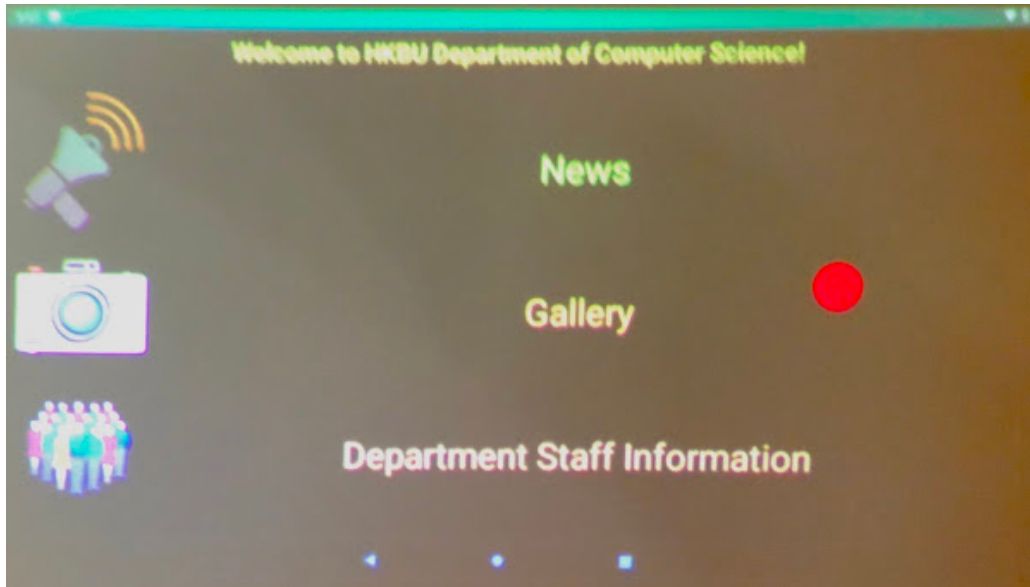
```
fun drawHandSpot(x: Float, y: Float) {
    ...
}

override fun onDraw(canvas: Canvas) {
    ...
    canvas.drawCircle(x, y, 30f, paint)
    ...
}
```

In MainActivity.kt, since there may be many results returned from `recognizeImage`, I choose the best result, i.e. the one with the highest confidence, to be the location of hand being shown on the UI.

```
private fun processImage() {
    ...
    trackingView.postInvalidate()
    ...
    for (i in handResults.indices) {
        if (handResults[i].confidence > bestHandResult.confidence)
            bestHandResult = handResults[i]
    }
    ...
    val location = bestHandResult.location
    ...
    trackingView.drawHandSpot(location.centerX(),
        location.centerY())
}
```

The method `getLocation()` (written in `.location` for Kotlin) returned a `RectF` object and I decided to draw the red dot, i.e. the hand position indicator on UI, using the center x and y of the `RectF` object.



Since the image has been resized to 300 x 300 for hand detection, I have to resize the image back to its original size of 1820 x 720 by calling the `invert()` method of the `Matrix` class. Similarly, the center x and y coordinates of the `RectF` object have to be transformed so that the location of the hand can be correctly shown on the UI. This can be done by calling the `mapRect()` method of the `Matrix` class.

```
frameToCropTransform.invert(cropToFrameTransform)
...
cropToFrameTransform.mapRect(location)
```

### 3.2.3.4 Custom MotionEvent for Click Action

After I have obtained the location of the hand in an image, I can implement the feature enabling the user to “click” the virtual graphics with the hand in a simple way.

In `activity_main.xml`, I have added a `ConstraintLayout` which covers the entire UI to ensure all the items shown on the UI are clickable.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/content_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <!-- toolbar layout -->
    <!-- content page layout -->
</androidx.constraintlayout.widget.ConstraintLayout>
```

In `MainActivity.kt`, I have created a custom `MotionEvent` using the location of hand in an image and called the `dispatchTouchEvent()` method of the `ViewGroup` class.

```
contentView = findViewById(R.id.content_view)
...
private fun addMotionEvent(x: Float, y: Float, act: Int) {
    ...
    val motionEvent = MotionEvent.obtain(..., act, x, y,...)

    // Dispatch touch event to view
    contentView.dispatchTouchEvent(motionEvent)
}
```

If a hand can be detected across the frames in the preview view, `MotionEvent.ACTION_DOWN` takes place. At the moment when a hand can no longer be detected in the current frame captured by the preview view, i.e. user moves the hand away from visible area of the camera module, `MotionEvent.ACTION_UP` takes place in order to trigger the “click” action.

```
private fun processImage() {
    ...
```

```

    if (handResults?.size!! > 0) {
        ...
        addMotionEvent(centerX, centerY, MotionEvent.ACTION_DOWN)
        ...
    } else {
        ...
        if ((centerX >= 0f) && (centerY >= 0f)) {
            // hand previously being detected has moved away
            // trigger click action
            addMotionEvent(centerX, centerY, MotionEvent.ACTION_UP)
            ...
        }
    }
    ...
}

```

### 3.3 Additional Python Scripts for Hand Detection Model Training

#### 3.3.1 AUTO CSV AND TFRECORD FILES GENERATOR FOR TENSORFLOW HAND DETECTION MODEL TRAINING

The Generator is a number of Python scripts specialized for automatically detecting **a hand per frame** from a MP4 video using OpenCV API, saving the frames in JPEG format and generating CSV and TFRecord files for hand detection model training and evaluation.

##### 3.3.1.1 Motivations

###### *Flaws in Egohands Dataset*

For the Egohands dataset, the hand gestures captured in most of the images do not exactly suit my need for hand detection on the Interactive Information



Wall. These gestures include:

- holding the cards while playing card games
- holding the chess pieces while playing chess
- holding wooden blocks while playing Jenga
- holding and assembling the puzzles



With these gestures, a few fingers disappear from the first person view and fingers are curled inward. It turns out that a gesture that is holding something in the hand has a higher confidence, i.e. easier to be detected as hand. However, images with gestures of clicking something are what I am looking for as I want to train a hand detection model to enable the user clicking the virtually projected graphics on the wall. Therefore, I decided to collect images of open hands with either index fingers or all fingers pointing upward.

### ***Get Rid of Manual Annotation of Hands by using OpenCV API for Hand Detection in Video Frames***

At present, we have to manually annotate the hands in each image by using image annotation tools like LabelImg and this is time consuming.

Inspired by the Real-Time Hand Gesture Detection project written in Python, I learnt that the OpenCV API supports hand detection in video

frames and bounding box is drawn for each frame in which a hand is detected. Thus, I have modified the project's `HandDetection.py` and `hand.py` scripts in order to build my own dataset, that is collecting images of hands from video frames and generating CSV files for hand detection model training and evaluation.

### 3.3.1.2 Modification of the Python scripts in Real-Time Hand Gesture Detection project

In `Hand.py`, I am able to obtain the two coordinates of the annotation of a detected hand in each video frame, i.e.  $(x_{min}, y_{max})$  and  $(x_{max}, y_{min})$  from the function `get_roi_to_use(self, frame)` which calculates the Region of Interest (ROI) of a detected hand in a video frame.

```
def _track_in_frame(self, frame, method="camshift"):
    ...
    if self._ever_detected:
        roi_for_tracking = self.get_roi_to_use(frame)

        x, y, w, h = roi_for_tracking

        xmin = x
        ymin = y
        xmax = x + w
        ymax = y + h

        if self._debug:
            # visualize the bounding box of
            # the detected hand in the frame
            cv2.imshow("DEBUG:_track_in_frame(frame_roied)", roi)
        ...
    return xmin, ymin, xmax, ymax
```

The `HandDetection.py` is the main program. It first reads each frame from

a video and then detect a hand in the frame.

```
# `source` is a path to a MP4 video
self.capture = cv2.VideoCapture(source)
...
def capture_and_compute2(self):
    ...
    while self.capture.isOpened():
        # get video frame one by one
        ret, frame = self.capture.read()

        if ret is True:
            self.add_hand2(frame) # detect a hand in the frame
            ...
```

`_track_in_frame()` from `Hand.py` is called to see if a hand is present in the frame or not. If a hand is present, `save_frame()` is called to save the frame in JPEG format and write information about the image, i.e. filename, frame size, the class label, i.e. `hand` as well as the coordinates of the annotation to a CSV file.

```
def add_hand2(self, frame, roi = None):
    new_hand = self.hand_from_frame(frame, roi)

    xmin, ymin, xmax, ymax = new_hand._track_in_frame(frame)

    if new_hand.valid: # if a hand is detected in the frame
        ...
        self.save_frame(frame, xmin, ymin, xmax, ymax)
```

The size of each frame should be the same of the one captured by the Raspberry Pi camera module which is 1280 x 720. If the frame size of the input video is not in 1280 x 720, I have to resize the frame using the `resize()` function provided by the OpenCV Python library as well as re-calculate the coordinates of the annotation.

```

def save_frame(self, frame, xmin, ymin, xmax, ymax):
    ...
    if needToResize:
        frame = cv2.resize(frame, (0,0), fx=ratio_w, fy=ratio_h)

        xmin = xmin * ratio_w
        ymin = ymin * ratio_h
        xmax = xmax * ratio_w
        ymax = ymax * ratio_h
    ...

```

The values for `ratio_w` and `ratio_h` are calculated in `capture_and_compute2()` function.

```

self.desiredFrameWidth = 1280
self.desiredFrameHeight = 720
...
def capture_and_compute2(self):
    ...
    self.frameWidth = int(self.capture.get(cv2.CAP_PROP_FRAME_WIDTH))
    self.frameHeight = int(self.capture.get(cv2.CAP_PROP_FRAME_HEIGHT))

    if (not self.frameWidth == self.desiredFrameWidth) and
        (not self.frameHeight == self.desiredFrameHeight):

        needToResize = True

        # initialize the ratio for resizing the frames
        self.ratio_w = self.desiredFrameWidth / self.frameWidth
        self.ratio_h = self.desiredFrameHeight / self.frameHeight
    ...

```

Noted that at this stage, the CSV file is the temporary one and all the frames are saved in a temporary folder. They are neither for model training nor for model evaluation. Additional Python scripts, namely

`split_train_test_csv_images.py` and `generate_tfrecord.py` have to be executed to build a complete dataset for the hand detection model training and evaluation.

In `split_train_test_csv_images.py`, the data in the temporary CSV file is split into 2 parts, 80% for model training and the rest for model evaluation. This operation is done by using `train_test_split()` function provided by the `scikit-learn` library.

```
imgdf = pd.read_csv("hand_labels.csv")

x = imgdf.iloc[:, :].values
y = np.ones(imgdf.shape[0]) # dummy values

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.2, random_state=4)
```

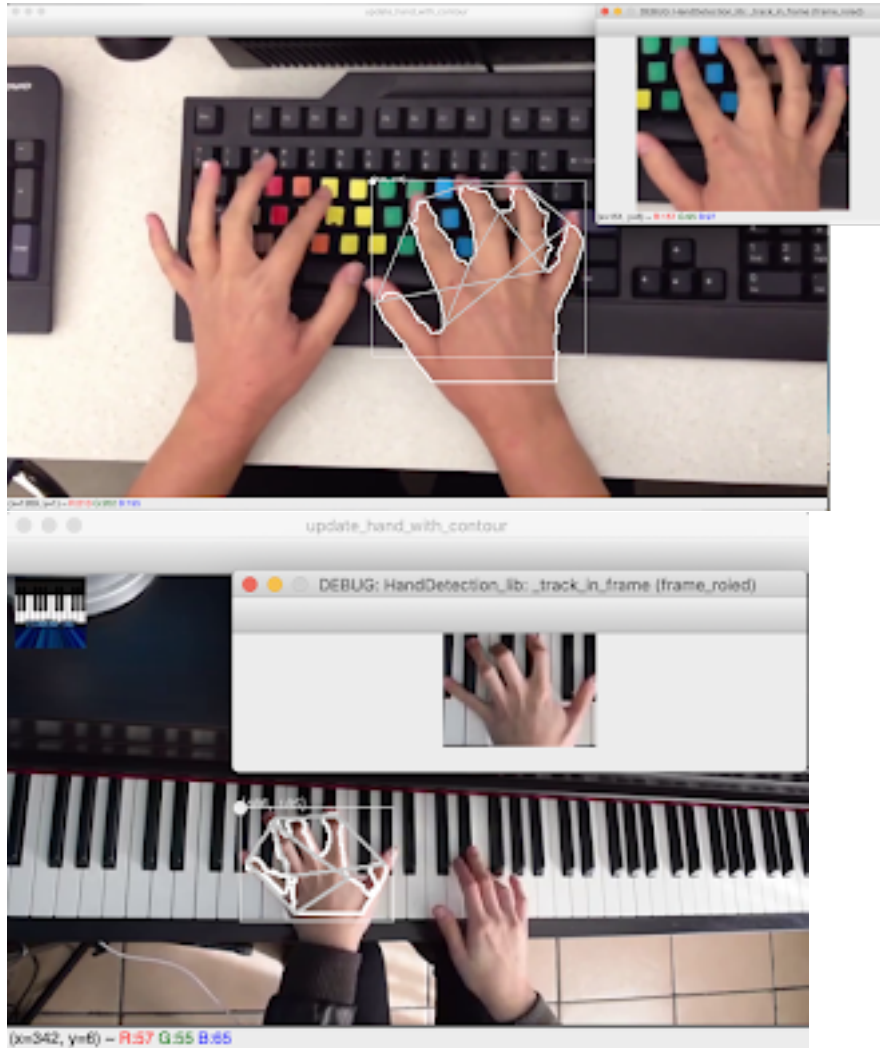
After that, two CSV files, one for model training and another for model evaluation, are generated. Then, images in the temporary folder will also be split into two folders according to the filename of the images stated in the two CSV files.

Finally, I have to run `generate_tfrecord.py` to generate two TFRecord files for the input of the hand detection model training.

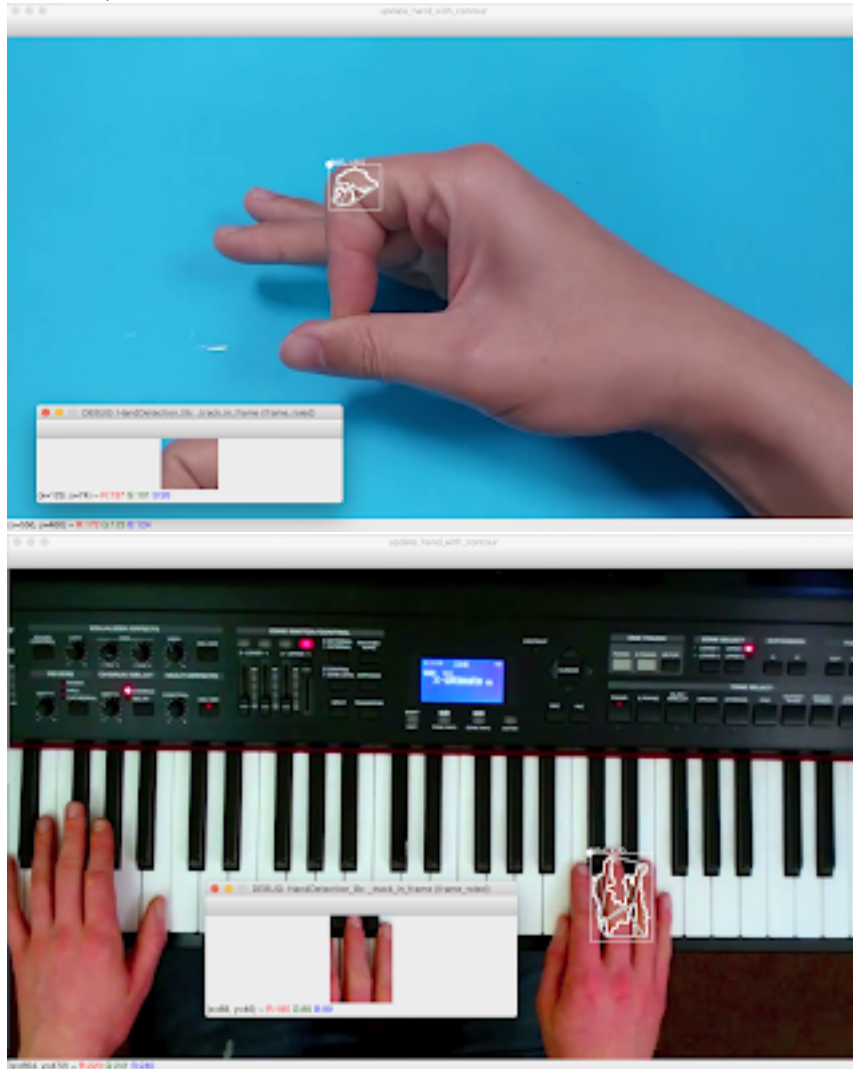
### 3.3.1.3 Limitations

I have found several limitations while the Python scripts were executing.

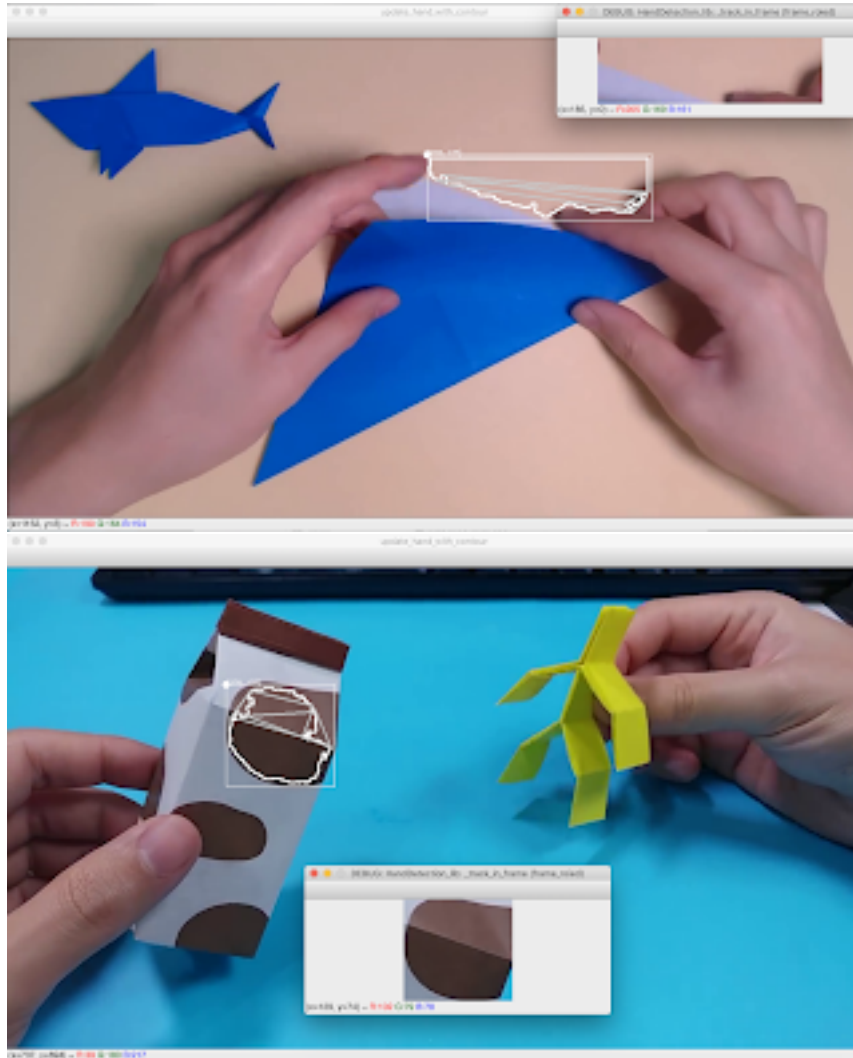
- Only one hand can be detected in each frame.



- Hands which are too big (i.e. hands which are too close to the camera) can hardly be detected or only fingers can be detected.



- Other objects appeared in the frame having a similar color with skin will be wrongly classified as hand.



### 3.3.1.4 Possible Ways to Overcome the Limitations

I have not yet figured out the solutions for detecting multiple hands, however, to deal with the last two limitations, I can collect images from the following types of videos so as to gather better training data.

- People playing the piano
- People typing words on the keyboard, and
- People folding Origami models.



The gestures in these videos are the one that I am looking for, i.e. open hands with either index fingers or all fingers pointing upward.

# Chapter 4

## Performance Results

### 4.1 Comparison of Pre-trained Object Detection Models' Performance

There are many choices of pre-trained model in the TensorFlow detection model zoo but not all models can be converted into TensorFlow Lite format which can be used for object detection on mobile devices. It is because only a small set of TensorFlow operations are compatible with those of TensorFlow Lite. Operations of TensorFlow Lite target at floating-point (float32) and quantized (uint8, int8) inference. At present, only the MobileNet models can be converted into TensorFlow Lite format and the quantized MobileNet models can be run on Coral USB Accelerator, an on-board Edge TPU co-processor for faster machine learning inference on mobile devices.

The following two scatter diagrams show the differences among the TensorFlow models in terms of size, accuracy and latency. MobileNet models which are optimized to run on mobile devices has a relatively small size and low latency, however, the tradeoff is the decreased accuracy of inference. Moreover, the quantized models have an up to 4 times reduction on the size of their original models as the float weights are turned into 8-bit only, allowing the models to run faster on mobile devices.

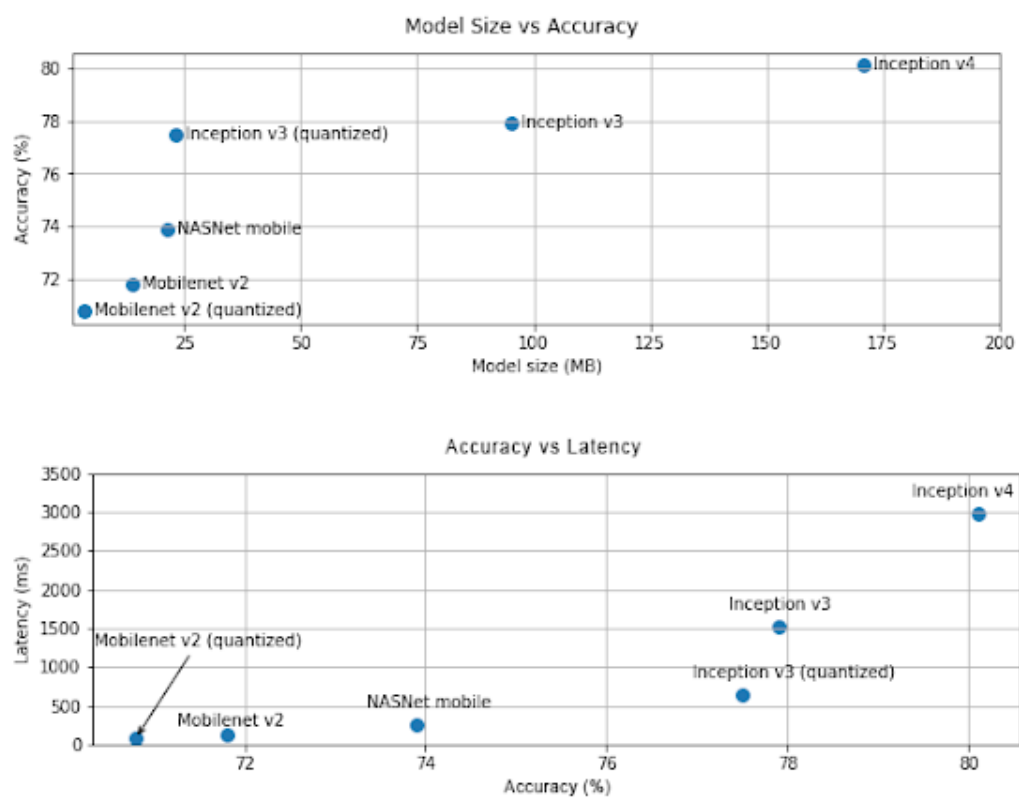


Figure 4.1: TensorFlow models - comparisons

## 4.2 Performance of Inference using different Hand Detection Models

I have successfully trained several hand detection models with different number of training steps based on Quantized SSD MobileNet V1 and V2 using EgoHands dataset prepared by Indiana University as well as the hand dataset prepared by myself using the Auto CSV and TFRecord files Generator.

### ***TensorBoard — a visualization tool for object detection model training***

TensorBoard is an web interface visualizing the metrics for a model training and evaluation. After each training, I have accessed to the TensorBoard to learn more about the learning process of the trained model. The average precision (mAP) and the total loss are the metrics that I was looking at to see if a model was successfully trained. The mAP shows the model's accuracy of prediction and the total loss indicates the errors made by the model during the evaluation.

### ***Testing Hand Detection Models on Samsung Note 3 smartphone***

I have installed the official TensorFlow Object Detection Demo application for Android on the smartphone and used it to try out the freshly trained hand detection model first before adding the model to this project's Android application.

The models' performances of inference are compared and the summaries are as follows.

#### 4.2.1 RE-TRAINED FROM QUANTIZED SSD MOBILENET V1

5 hand detection models were trained based on the EgoHands dataset.

The model with 500 training steps was trained on Macbook Pro using one CPU core. The model with 1,500 training steps was trained on Macbook Pro using a quad-core processor. The rest of the models were trained on a desktop computer using one CPU.

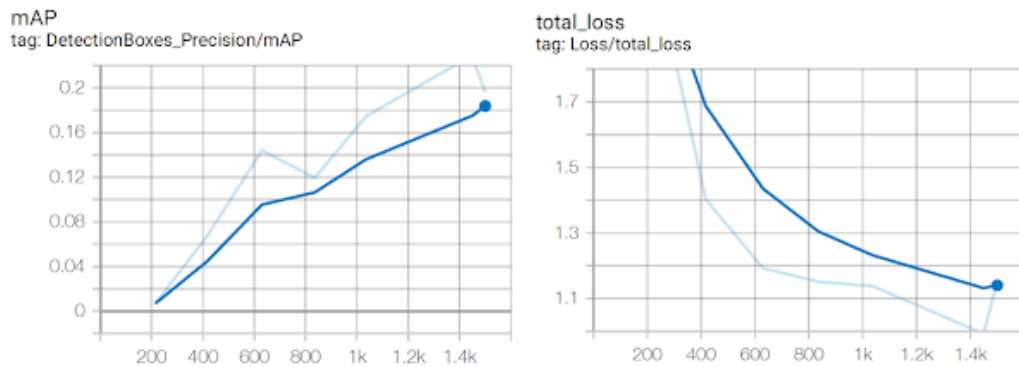
The Python script used for the training and evaluation process is `object_detection/model_main.py` and it was executed using Python 2.7.

#### 4.2.1.1 Total Training Time of the models

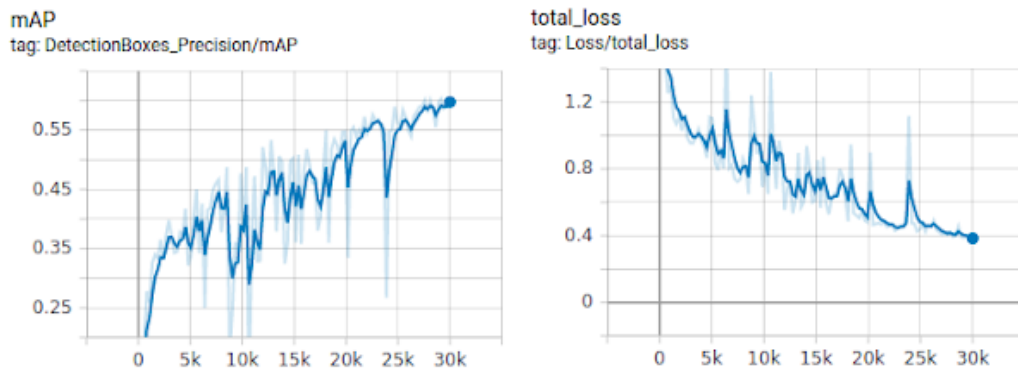
| 500 steps  | 1,500 steps       | 15,000 steps           | 30,000 steps            | 50,000 steps              |
|------------|-------------------|------------------------|-------------------------|---------------------------|
| 61 minutes | 1 hour 20 minutes | 11 hours (667 minutes) | 19 hours (1135 minutes) | 31.7 hours (1900 minutes) |

#### 4.2.1.2 TensorBoard Records for Model Training and Evaluation

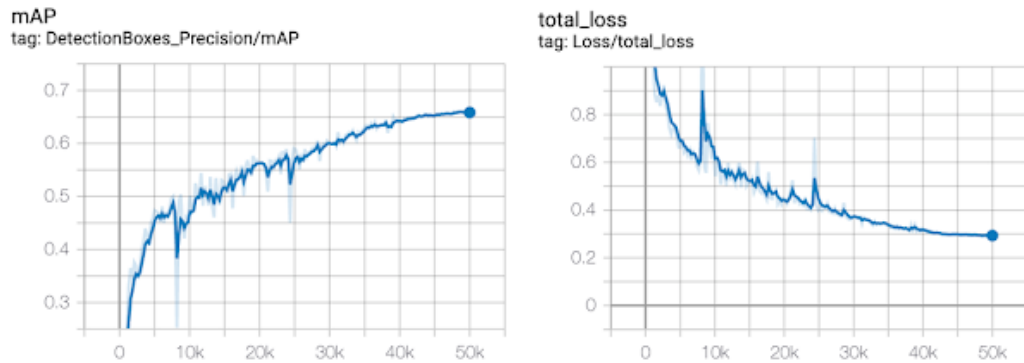
##### *Model trained with 1,500 steps*



##### *Model trained with 30,000 steps*



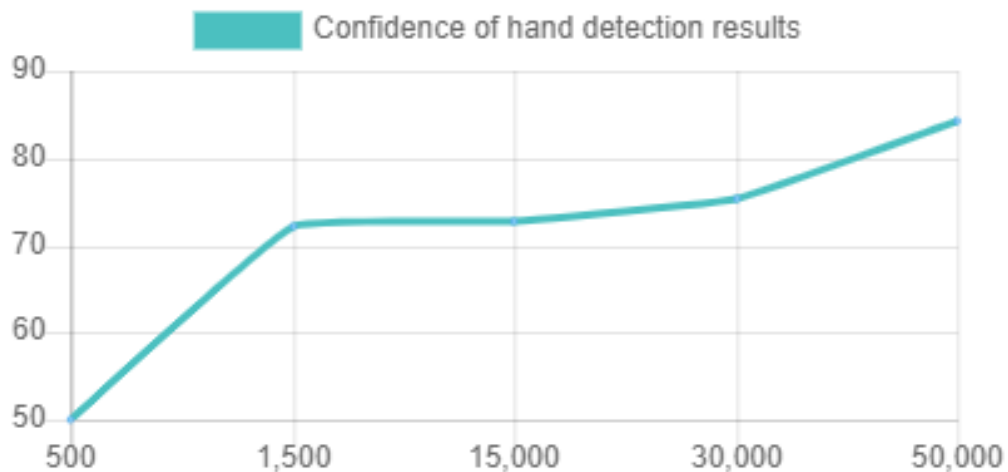
### *Model trained with 50,000 steps*



The value of mAP climbed and the value of the total loss decreased gradually throughout the training processes, indicating that hand detection models were well trained.

#### **4.2.1.3 Performance of inference on Samsung Note 3**

The following chart shows the confidence of hand detection results (in %) corresponding to the models' total number of training steps. Higher confidence means that the model has a higher accuracy in the detection of a hand.



The model with 500 training steps is able to detect a hand, however, the

confidence of the detection results is low. The models with training steps greater than or equal to 1,500 have significant higher confidence. Smoother detection of a hand was achieved using the models with training steps greater than or equal to 15,000, however, there could be multiple detection results for a single.

#### **4.2.1.4 Performance of inference on Raspberry Pi 3 B+ Model through Android application**

Only the model with 50,000 steps was deployed on the Raspberry Pi 3 and the confidence of hand detection results is 71.09%. It could accurately locate a hand across the images but the detection is less smooth, i.e. bumpy detection with around 1 second delay.

#### **4.2.1.5 Problems of using `object_detection/model_main.py` for model training**

I attempted to train a model with 100,000 as the target number of total training steps using the desktop computer with one CPU. The training process ran smoothly until it was killed at step 74,555. The training process using a pre-trained quantized SSD MobileNet V1 model and the Python script `model_main.py` cannot reach 100,000 training steps on the desktop computer.

### **4.2.2 RE-TRAINED FROM QUANTIZED SSD MOBILENET V2**

Two hand detection models were trained based on the EgoHands dataset and one model was trained based on my own dataset.

All the 3 models were trained using one CPU in the desktop computer. The Python scripts used for the training and evaluation process is `object_detection/legacy/train.py` and `object_detection/legacy/eval.py`. They were all executed using Python 2.7.

### 4.2.2.1 Total Training Time of the models

Two models trained based on EgoHands dataset:

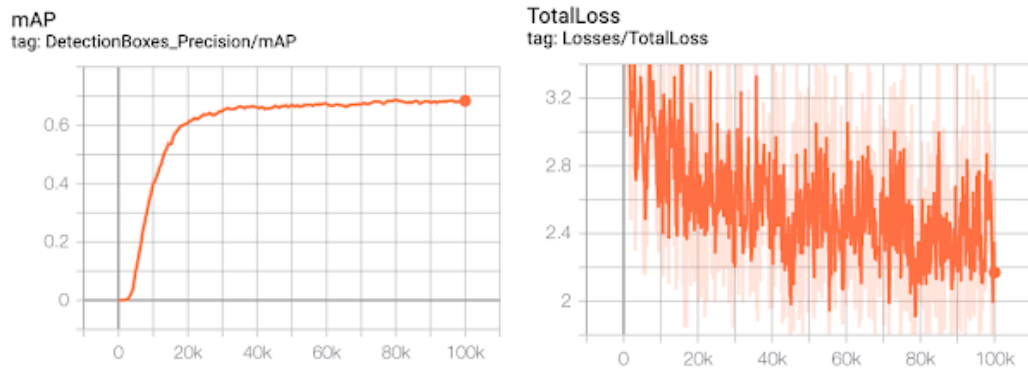
| 50,000 steps              | 100,000 steps           |
|---------------------------|-------------------------|
| 37.1 hours (2228 minutes) | 99 hours (5967 minutes) |

The model trained based on my own dataset:

| 50,000 steps              |
|---------------------------|
| 49.5 hours (2969 minutes) |

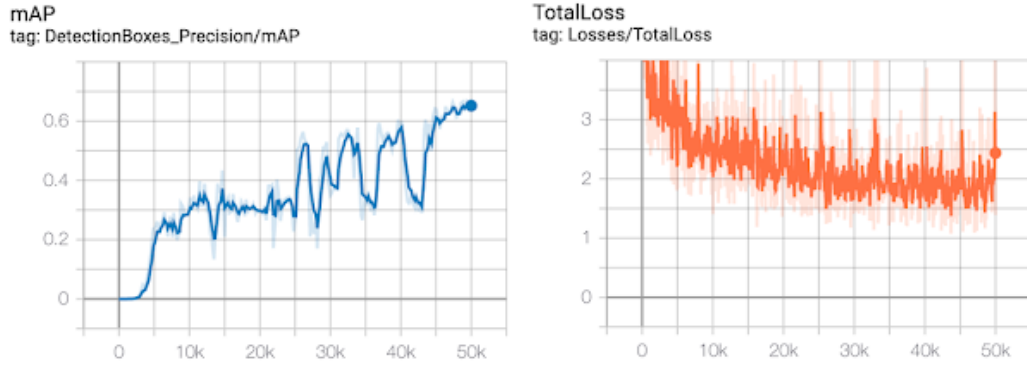
### 4.2.2.2 TensorBoard Records for Model Training and Evaluation

*Model trained based on EgoHands dataset with 100,000 steps*





### *Model trained based on my own dataset with 50,000 steps*



In general, the mAP value rose and the value of the total loss fell throughout the training processes, indicating that hand detection models were successfully trained.

For the model trained with 100,000 steps based on EgoHands dataset, the mAP value surged to 0.6 at step 20,000 and started to level off at step 50,000. On the other hand, for the model trained with 50,000 steps based on my own dataset, the mAP value only climbed to 0.3 at step 20,000 and reached 0.65 at the end of the training process. It seems that the model trained with 100,000 steps based on EgoHands dataset is better than the one trained with 50,000 steps based on my own dataset.

#### **4.2.2.3 Performance of inference on Raspberry Pi 3 B+ Model through Android application**

The following chart shows the confidence of hand detection results (in %) corresponding to the models' total number of training steps. The 2 models were trained based on the EgoHands dataset.



Both of the models have a high accuracy of hand detection. The positions of a hand across the images could be located very accurately with with less than 1 second delay, almost a real-time detection.

For the model trained based on my own dataset, the total training steps was 50,000 and the average confidence of hand detection is only 59.37%, with a maximum confidence up to 93.35%. This model is able to locate a hand across the images, however, the detected position is sometimes inaccurate. Moreover, the detection is unsmooth, i.e. very bumpy detection with a minimum of 2 seconds delay.

#### 4.2.2.4 Performance of inference on Samsung Note 3

The confidence of hand detection for the 2 models trained based on the EgoHands dataset is 99.61%. Both of them could achieve smooth hand detection and only one detection result was shown for a single hand.

The average confidence of hand detection for the model trained based on my own dataset is 82.03%, with a maximum confidence up to 90.63%. Only open hand, i.e. the back of the hand facing upward with all fingers pointing upward, could be easily detected as most of the images in my dataset contains images of this kind of gesture. Nonetheless, the detection is less smooth, i.e. bumpy

detection.

#### **4.2.2.5 Observation of the Confidence Level among the Detection Results**

There was a wide range of confidence among the detection results, from 12.5% or less, up to 99.6%.

#### **4.2.3 PROBLEM FOR ALL THE MODELS**

Other objects which have similar color to skin could be misclassified as hand.

#### **4.2.4 DISCOVERY REGARDING THE USE OF PYTHON SCRIPTS FOR OBJECT DETECTION MODEL TRAINING**

For the hand detection model training process based on the Ego-Hands dataset, I had started the model training using quantized SSD MobileNet V2 using `object_detection/model_main.py` executed by Python 2.7, however, the training process killed abruptly at step 222 as the model evaluation process could not be done. Additionally, I had used the `object_detection/legacy/train.py` and `object_detection/legacy/eval.py` to re-train the quantized SSD MobileNet V1 model for hand detection. Yet, I could not even start the training process.

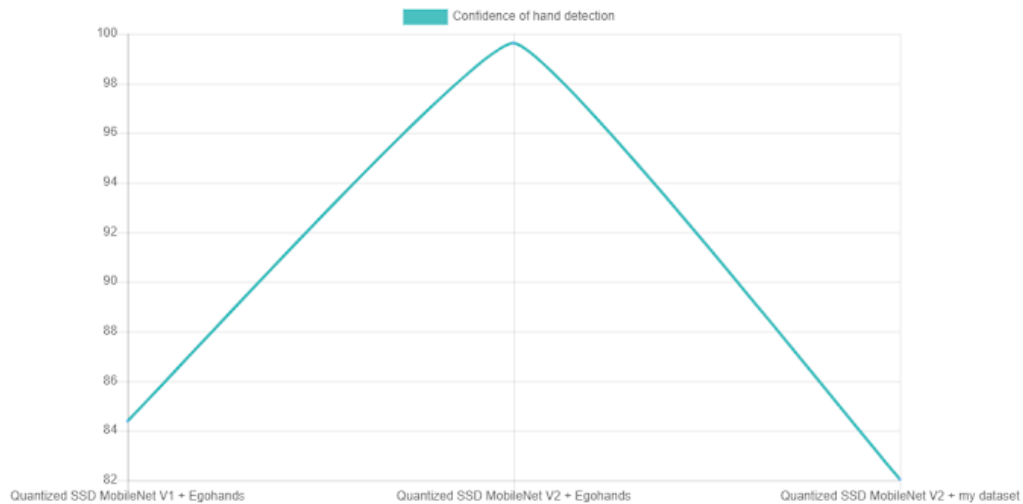
The combination of a pre-trained model and the Python script(s) used for the training process is summarized as follows.

| <code>object_detection/</code>                               | Quantized SSD<br>MobileNet V1  | Quantized SSD<br>MobileNet V2 |
|--|--|-------------------------------|
| <code>model_main.py</code>                                   | Yes (works well with 50,000 as the target no. of training steps but unable to reach 100,000 training steps on a desktop computer with one CPU) | No                            |
| <code>legacy/train.py</code> and <code>legacy/eval.py</code> | No   | Yes                           |

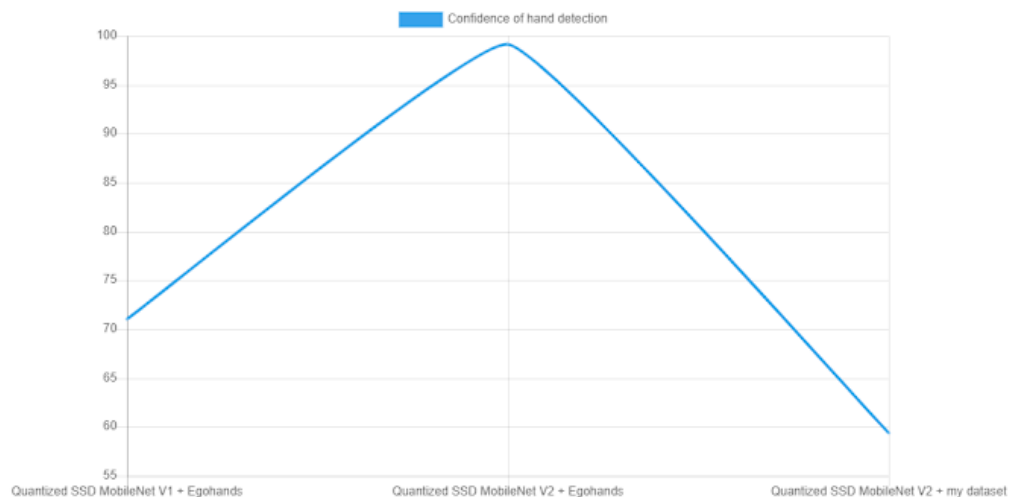
#### 4.2.5 CONCLUSION OF THE HAND DETECTION MODELS' PERFORMANCE

Among all the hand detection models that I have trained so far, the best hand detection model is trained using the EgoHands dataset prepared by Indiana University and the quantized SSD MobileNet V2 pre-trained model. The following graphs compare the hand detection confidence among the best 2 hand detection models re-trained from quantized SSD MobileNet V1/V2 with 50,000 and 100,000 steps respectively using EgoHands dataset as well as the model re-trained from quantized SSD MobileNet V2 with 50,000 steps using my own dataset.

*Comparison of hand detection confidence running on the smart-phone*



*Comparison of hand detection confidence running in the Android application on Raspberry Pi 3 B+ Model*



To conclude, a good hand detection model for mobile application, which is able to locate a hand accurately and with minimal delay or real-time hand detection, can only be generated by using a quality dataset, and re-training a pre-trained object detection model with higher accuracy and low latency

like quantized SSD MobileNet V2.

#### **4.2.5.1 Criteria of a Quality Hand Dataset**

A quality hand dataset should have the following traits.

For the images, - hands can be clearly seen, and - size (or dimension) of all the images should be consistent with the images captured by the camera for inference. It is better to be in HD format already without resizing.

For annotation of hands in the images, - all the hands which can be clearly seen in each image should be annotated to maximize the data being used to train a hand detection model.

#### **4.2.5.2 Evaluation on the Quality of the Datasets**

Even though my dataset has images collected from the videos of people playing the piano, typing words on the keyboard and folding Origami models have more gestures which suit my purpose, the Egohands dataset still outperforms my own dataset. The Egohands dataset has more than 4,401 images (around 83% of total images) for model training. Annotations for all the hands that appear in each image are done. In contrast, for my own dataset, I only have 1,537 images (80% of the total images collected from 7 videos) for model training. Only one hand can be detected and annotated for each image.

### **4.3 Using Android OS instead of Android Things**

At the very beginning of this project, I had followed user guide of the Android Things Lantern's demo project so I had installed the Android Things on the Raspberry Pi. Android Things is an operating system specialized for smart devices to execute Android applications on them.

#### 4.3.1 PROBLEMS OF USING ANDROID THINGS ON RASPBERRY PI 3

When I first deployed a hand detection model re-trained from SSD MobileNet V1 model in both of the Android applications running on Samsung smartphone and Raspberry Pi 3, I found that the hand detection process runs smoothly, achieving real-time inference with the best hand detection result's highest confidence of 84% on the smartphone, however, there is a high latency, with around 2 to 3 seconds delay, for the hand detection process running on the Raspberry Pi 3 and the highest confidence of the best hand detection result is only 72%.

This problem may be associated with the Android Things. The Android Things only supports 32 bits processing but Android OS supports 64 bits processing. As the processor of Raspberry Pi 3 supports 64 bit processing, I have installed the Android OS on the Raspberry Pi 3 in a bid to make fully use of the computational power of the Raspberry Pi 3 so that the hand detection process can be enhanced.

# Chapter 5

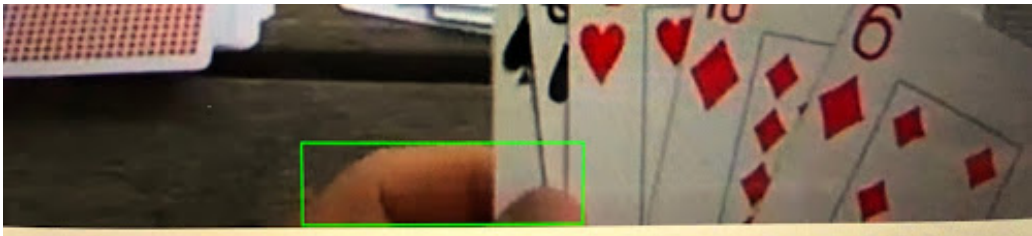
## Discussions

### 5.1 Limitations

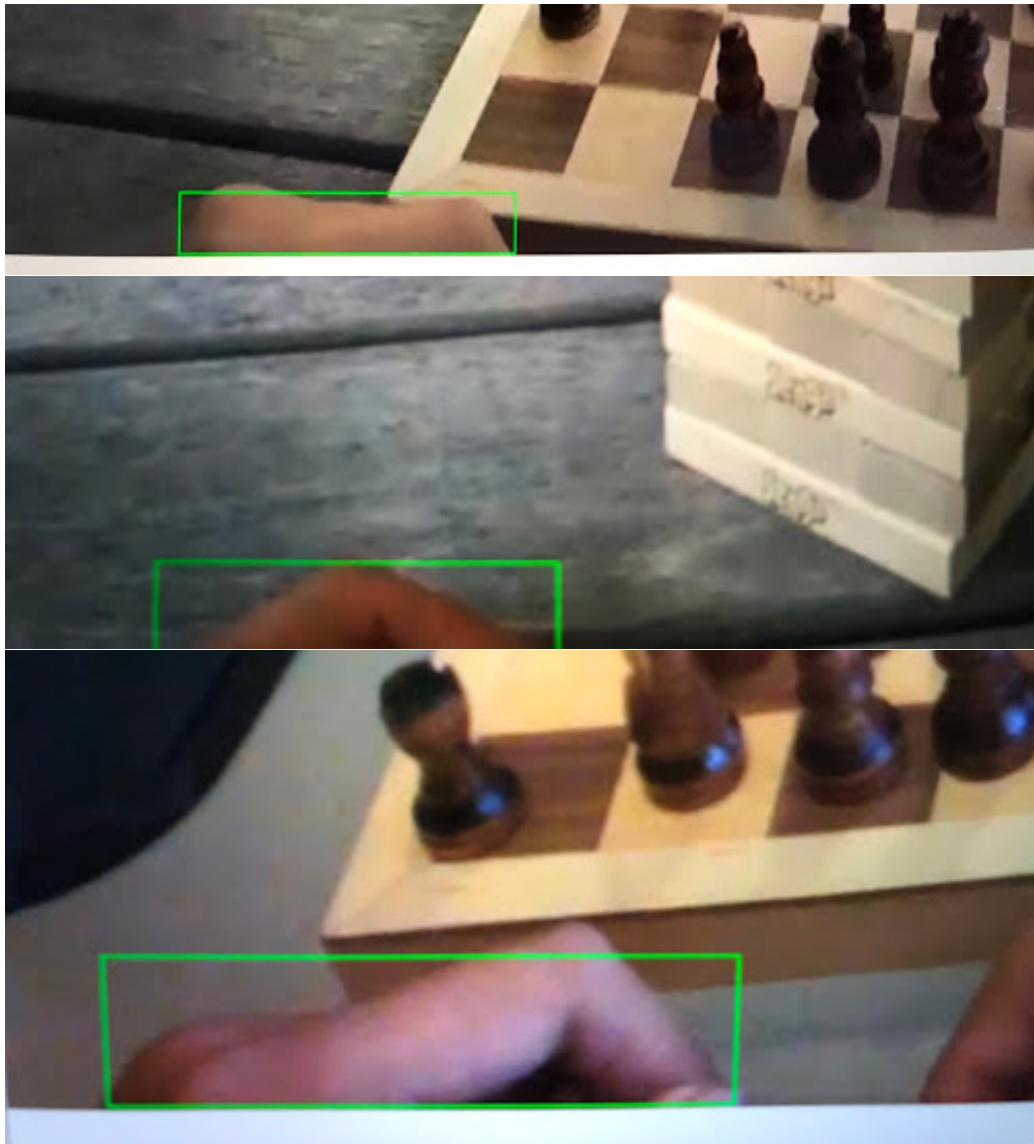
#### 5.1.1 DEPENDENCY ON COLOR FEATURES FOR HAND DETECTION MODEL

Reviewing the problems of the hand detection models that I have trained, I discovered that other objects which have similar color to skin would be misclassified as hand. This shows that the hand detection models rely heavily on color features to detect a hand.

For the image annotation of hands in EgoHands dataset, apart from labeling the whole part of a hand, any part of a hand, like a finger, which could be seen manually in an image was also annotated. (The green non-filled rectangle is the bounding box of the hand.)







Therefore, I assume that this is the reason why the models rely heavily on color features in the detection.

Though, the detection results of the objects which are misclassified as hand usually have a lower confidence than that of a real hand so the possible solution is to filter the detection results by setting a minimum confidence level.

#### 5.1.1.1 Weaker Computational Power of Raspberry Pi

By comparing the hand detection results in the Android applications running on the smartphone and Raspberry Pi, I found that the performance of hand detection on smartphone always outperform the one on the Raspberry Pi when they are using the same model for inference.

There is a tradeoff between the accuracy of the hand detection result and the detection speed for the Android application running on Raspberry Pi. When I am using the hand detection model re-trained from the pre-trained SSD MobileNet V2 model, in the demo project running on the smartphone, I have set the minimum confidence level of the hand detection result to 0.6f in the hope of minimizing the misclassified results and this solution did work, however, in this project's Android application, I can only set the minimum confidence level to 0.1f in order to achieve real-time hand detection.

The implementation for hand detection process are the same as I have just reused the `Classifier.java` and `TFLiteObjectDetectionAPIModel.java` in the demo application for this project's Android application. At this point, I am able to conclude that the computational power of Raspberry Pi is weaker than that of the smartphones and this affects the hand detection performance.

#### 5.1.2 BLACK BACKGROUND COLOR FOR ANDROID APPLICATION UI TO SMOOTHEN HAND DETECTION

Previously, the background color of the UI is white and this raises a color overlaying problem.

The hand detection model relies on color during the inference in a certain extend. That means if an object has a similar color to the skin, the model may wrongly predict it as a hand. Conversely, if a hand has a color which is not similar to that of skin, it may not be detected. The latter is the situation I am facing when the the background color of the UI is white.

As white light beams produced by the projector are projected on the hand,

it makes a hand whiter, having a color which is dissimilar to that of skin. This reduces the accuracy of hand detection on images captured from camera module. A hand cannot keep being detected across the frames in the preview view and thus reduces the user experience on navigation in the application.

By changing the background color of the UI to black, white light beams can be eliminated. The skin color of a hand being captured by the camera module is retained, allowing a hand to be detected smoothly across the frames in the preview view.

### 5.1.3 SINGLE HAND DETECTION IN ANDROID APPLICATION

At present, only the best hand result is selected among the detection results and one hand position indicator, i.e. a red dot, is drawn on the UI based on the location of the best result.

### 5.1.4 LATENCY FOR HAND DETECTION ON RASPBERRY PI

Although I have switched to use Android OS on Raspberry Pi and deployed the best hand detection model re-trained from SSD MobileNet V2 model in the Android application running on the Raspberry Pi, the one-second delay of the hand detection process still persists while I am running the Android application. I doubt that this is another problem raised due to the weaker computational power of the Raspberry Pi.

## 5.2 Further Developments

### 5.2.1 IMPLEMENTATION

#### 5.2.1.1 Using libGDX API to improve the speed for the conversion of image format from YUV to RGB

Currently the image format captured from the preview view is in YUV format, however, the TensorFlow Object Detection API can only support inference on images in RGB format. The conversion of image format from YUV to RGB is now done by `convertYUV420SPToARGB8888()` which can only be executed on the CPU.

The idea of the enhancement is to load the Y and UV channels of an image into GL textures, then draw the textures onto a Mesh by using a custom shader suggested by Ayberk Özgür which performs color space conversion to RGB format.

Two classes - `ShaderProgram` and `Mesh` provided by libGDX API will be used to create GL textures and perform the rendering of images in RGB format. Since the shader can be run on the GPU of the Raspberry Pi, the image format conversion speed is enhanced.

```
shader = ShaderProgram(  
    "attribute vec4 a_position;           \n" +  
    "attribute vec2 a_texCoord;          \n" +  
    "...  
  
    "void main(){                        \n" +  
    "    gl_Position = a_position;        \n" +  
    "    v_texCoord = a_texCoord;        \n" +  
    "}  
    "...
```

```

"void main (void){                                \n" +
"    float r, g, b, y, u, v;                      \n" +

"    y = texture2D(y_texture, v_texCoord).r;       \n" +
"    u = texture2D(uv_texture, v_texCoord).a - 0.5; \n" +
"    v = texture2D(uv_texture, v_texCoord).r - 0.5; \n" +

"    r = y + 1.13983 * v;                          \n" +
"    g = y - 0.39465 * u - 0.58060 * v;           \n" +
"    b = y + 2.03211 * u;                          \n" +

"    gl_FragColor = vec4(r, g, b, 1.0);           \n" +
"}                                                  \n"
)

mesh = Mesh(true, 4, 6, // static mesh, 4 vertices, 6 indices
VertexAttribute(VertexAttributes.Usage.Position, 2,
    "a_position"),
VertexAttribute(VertexAttributes.Usage.TextureCoordinates, 2,
    "a_texCoord"))
mesh?.let {
    it.setVertices(vertices)
    it.setIndices(indices)
}

```

The image format conversion utilizing the libGDX should be continuously running at the background along with the hand detection routines in the MainActivity which is a class extended from the AppCompatActivity. Unfortunately, the current AndroidApplication class provided by the libGDX is not fully compatible with AppCompatActivity, thus my MainActivity class cannot be extended from the AndroidApplication.

A possible solution is to revise the implementation of the AndroidApplication class making it compatible with AppCompatActivity, then compile the whole libGDX project as a .jar file and add it to my Android application project.

### 5.2.1.2 Coral USB Accelerator for Hand Detection Model Inferencing

Coral USB Accelerator is an on-board Edge TPU co-processor designed to significantly accelerate the model inferencing in a power efficient manner on small and low-power devices like smartphones and Raspberry Pi.

The co-processor is able to perform 4 trillion operations per second. The following are the official performance benchmarks produced by C++ benchmark tests [3].

| Model architecture | Desktop CPU* | Desktop CPU* + USB Accelerator (USB 3.0) |
|--------------------|--------------|--|
| SSD MobileNet V1   | 109          | 6.5                                      |
| SSD MobileNet V2   | 106          | 7.2                                      |

Remarks: - Time is measured in milliseconds per inference. -  
The models have an input tensor size of 224 x 224 - \*Desktop  
CPU: Single Intel® Xeon® Gold 6154 Processor @ 3.00GHz

I would like to use the Accelerator in order to improve the speed for the hand detection model inferencing on Raspberry Pi 3. Nevertheless, the current API of the Accelerator only supports running a model inferencing using Python script. I have to write a program acting as an adapter so that I am able to use the Accelerator in Android application.

### 5.2.1.3 Re-train a hand detection model which supports gesture recognition

The current hand detection can only detect the position of a hand in each image captured from the preview view and does not support recognition of any gestures.

In fact, scrolling is one of the essential action for navigation in the pages where there are lots of sub-items. In the news page and department staff information page, there are many news articles and academic staff respectively.

Now there are only 3 items shown on each page. To achieve a better navigation experience, fragment transition could take place when a user waves the hand towards left or right, allowing different items to be shown on the page.

Further research into the TensorFlow Gesture Classification Web App, a demo project showing how to generate TensorFlow Lite models for gesture recognition, can be done so as to enhance the interactive features provided by the Lantern Information Wall.

#### **5.2.1.4 Improvement on the Presentation of UI Components**

There is lots of text in the news page and department staff information page particularly. I think this poses a negative effect towards user experience as this violates the simplicity of a UI.

Re-design on the presentation of the text is therefore needed to reduce the text shown in a page without affecting the content that is essential for the users to understand its meaning.

# Chapter 6

## Conclusion

I have successfully built a portable interactive device — Android Lantern which consists of Raspberry Pi 3 B+ single-board computer, a palm-sized laser projector and a camera module in order to enhance the current Interactive Information Wall by making use of the projection-based AR technology.

The real-time interactive feature provided by the Android Lantern is achieved by a hand detection model, an Android application and the camera module for computer vision. Android OS is installed on the Raspberry Pi 3 for the deployment of Android application.

To allow users interacting with the virtually projected graphics with the “click” event, the Android application plays an important role in obtaining the images captured by the camera module and passing them to the model file for inference. Through using both Camera and Camera2 API to obtain the images, I have discovered the shortcomings of Camera2 API and merits of using Camera API to obtain as many images as possible from a preview view.

For the hand detection model, I have mastered the training process of an object detection model, from generating TFRecord files based on the existing datasets to starting a model training and converting the model in TFLite format which is optimized for mobile applications. To build a dataset which includes more images with gestures suitable for the Interactive Information



Wall, I have created Python scripts to automatically annotate a hand in video frames using OpenCV API for hand detection and generate TFRecord files for model training. Several hand detection models are trained using different datasets as well as pre-trained models and their performances on the inference are compared. A good hand detection model can only be trained from a quality hand dataset and pre-trained object detection model with high accuracy for inference. A quality dataset should have more than 4,400 images in which the bounding boxes are perfectly fit in the size of the hands. SSD MobileNet V2 is the best pre-trained model to high accuracy for hand detection.

The hand detection model in TFLite format is added to the Android application to perform inference on the images captured by the camera module. A “clicking” action can be triggered by a custom MotionEvent based on the location of a hand detected in each image.

The user interface of the Interactive Information Wall is revised. Three kinds of information — News, Gallery and Department Staff Information are shown on the Wall. In the Android application, fragments are used to enable easy navigation in the hierarchy and data binding with MVVM architecture as well as observable data objects are used to achieve real-time updates on the UI components’ data fields and manageable Android application in which UI’s logic and program’s logic are decoupled.

# Appendix 1: User Guide of Auto CSV and TFRecord files Generator for TensorFlow Hand Detection Model Training

This program is specialized for automatically detecting **ONE** hand per frame from a MP4 video using OpenCV API.

During the execution of the program, you have to choose whether you would like to save the frame or not by pressing one of the hotkeys. When the program terminates, a csv file will be generated. Run other consecutive programs to generate other csv files and TFRecord files which can be used for training and testing a TensorFlow hand detection model.

Please refer to the GitHub repository for more details about the source code.

## 6.0.1 ENVIRONMENT

This program has been successfully run using Python 2.7.

## 6.0.2 DEPENDENCIES

Remember to `pip install cv2 numpy csv pandas scikit-learn` to get all the dependencies ready before the execution of the program.

## 6.0.3 LABEL FOR TENSORFLOW OBJECT DETECTION MODEL

Currently only one label - hand is supported. See the line `rowdata = [filename, final_w, final_h, 'hand', xmin, ymin, xmax, ymax]`

### 6.0.3.1 Structure of files and directories created after the execution of the programs

`hand_images/` --> holding all the images to be split into training/testing images

`hand_labels.csv` --> holding all the information of the images and coordinates of the hand bounding boxes in the images

```
hand_my_dataset/
|
|--- hand_test/
|   |
|   --- hand_test_labels.csv
|   |
|   --- all the images for testing
|
| --- hand_train/
|   |
|   --- hand_train_labels.csv
|   |
|   --- all the images for training
```

```
|
--- hand_eval.record
|
--- hand_train.record
```

## 6.0.4 ESSENTIAL OPERATIONS

### 6.0.4.1 Run python HandDetection.py, the main program

This program detects hand in frames from a video source using OpenCV. If a hand is detected in a frame, the frame can be saved in JPEG format under the `image` directory and the coordinates of the bounding box for the hand in the frame will be recorded. After the detection of hand in the video frames finished, a csv file `hand_label.csv` will be generated.

Please change `?.mp4` in the line `hand_detector = HandDetector('resources/?.mp4')` to your video file in MP4 format. You are advised to put all the video sources under the `resources` directory.

After a hand is detected in the frame, you can save the frame by activating any one of the debug window and pressing the `s` hotkey twice. You can also press the `x` hotkey once to discard the frame. Otherwise, after 8 seconds, the frame will be discarded automatically. You may change the time period for making a decision by modifying the number in ms i.e. 8000 in the lines `if cv2.waitKey(8000) & 0xFF == ord('x'):` and `elif cv2.waitKey(8000) & 0xFF == ord('s'):`.

Before the frame is saved, it will be automatically resized to a size of `1280 * 720`. You may change your desired frame width and height by changing the number of the variables `self.desiredFrameWidth` and `self.desiredFrameHeight`.

#### *Customize the program*

You may change the variable `self.image_dir` to create another directory

with a different name for holding all the JPEG images which are going to be saved during the execution of the program. The default name of the directory is `image`.

You may change the variable `self.csvFilename` to create another csv file for recording all the information about the image and the coordinates of the bounding box indicating the position of a hand in the image. The default name of the csv file is `hand_label.csv`.

#### **6.0.4.2 Run `python split_train_test_csv_images.py` afterwards**

This program splits the images in the `image` directory into 2 separated directories and corresponding image data in the `hand_label.csv` csv file into 2 separated csv files. By default a dataset directory called `hand_my_dataset` will be created. Under this dataset directory, there are two subdirectories. `hand_train` is intended to be used for training while `hand_test` is intended to be used for testing during the hand detection model training process.

##### ***Customize the program***

You may change the names of the images, training, testing directories and the csv files by modifying the variables located at the beginning of the program.

#### **6.0.4.3 Run `python generate_tfrecord.py` at last**

This program generates a TFRecord file either based on the train or test csv file, i.e. `hand_train_labels.csv` or `hand_test_labels.csv`.

Run the program twice to generate 2 TFRecord files, one for training and another one for testing. Change the name of the files and directories by modifying values of the variables located at the beginning of the program.

### 6.0.5 OPTIONAL OPERATION

Run `visualize_image_bbox.py` separately to visualize the bounding box for each image saved under the `image`, `hand_train` and `hand_test` directories.

# References

- [1] TensorFlow, “Tensorflow detection model zoo - coco-trained models.” [https://github.com/tensorflow/models/blob/master/research/object\\_detection/g3doc/detection\\_model\\_zoo.md#coco-trained-models](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md#coco-trained-models).
- [2] V. Bazarevsky and F. Zhang, “On-device, real-time hand tracking with mediapipe,” 2019. <https://ai.googleblog.com/2019/08/on-device-real-time-hand-tracking-with.html>.
- [3] Coral, “Edge tpu performance benchmarks.” <https://coral.ai/docs/edgetpu/benchmarks/>.