Machine Learning Engineer Nanodegree

Capstone Project

FC Su

July 1st, 2018

I. Definition

Project Overview

Amblyopia, also called lazy eye, is a disorder of sight due to the eye and brain not working well together It results in decreased vision in an eye that otherwise typically appears normal.Amblyopia begins by the age of five In adults, the disorder is estimated to affect 1–5% of the population While treatment improves vision, it does not typically restore it to normal in the affected eye.[1]
I am the one who suffer this disease with two eye, my vision is really bad. I have no glass, the only thing I can do to clearly see something is using magnifier for near thing and telescope for far thing. I can't easily see something in front of me. In class, I can't easily see the text on the blackboard. In work, as a firmware engineer, I can't easily see other colleague's code on their monitor. I really need something that can help me see something! I think this is the common requirement for low vision people or elder. A software magnifier can detect things, auto zoom in to things or manually zoom in or out.
There are many magnifier app in App Store, such as SuperVision+ Magnifier [2]. Most of them emphasis on zoom in or out feature, But none of them can real time detect or predict

what it is, For non-mobile solution, eSight [3] should be a good one. But it need another electronic device, and it can't auto detect object. For the academic research, although augmented reality and real time object detection algorithm growths rapidly recently. But they seems like not focus much on low vision or magnifier field.
In this project, I created an iOS App that is capable of real time detecting things, such as person, laptop, dog, banana, …, and hand. This application use a classifier trained by COCO [4] dataset to detect everyday things and EgoHands [5] dataset to detect hand, And it also provide magnifier functionality to zoom in/out to more clearly identify things.

Problem Statement

In this project, I would like to develop a magnifier application that can detect object with my iphone, and zoom in to what I am interesting in.
With this feature, if colleague would like to share their code with me and shake hand on their monitor I can easily catch what they want me to focus on. I can even see far things even I don't have glass.

My goal is to create a real time object detection magnifier in iphone. Below are some steps involved.

1. Download and preprocess the COCO and EgoHands data.
2. Train a real time object detection classifier that can determine if an image contains things and hand.
3. Make the classifier run on iOS.
4. Make the app detect hand and COCO and mark it up.
5. Make the app capable of zoom in/out and speak the predicted things aloud.

The final application is expected to be useful for real time detecting hand and COCO.

Metrics

Real time object detection always use mAP and FPS to compare their performance. So, I will also use them to compare in the this project. Below are some precision, recall, IoU, mAP and FPS definition.

- Precision: how accurate is your predictions. i.e. the percentage of your positive predictions are correct.
- Recall: how good you find all the positives. For example, we can find 80% of the possible positive cases in our top K predictions.
- IoU (Intersection over union): measures how much overlap between 2 regions, This measures how good is our prediction in the object detector with the ground truth (the real object boundary).

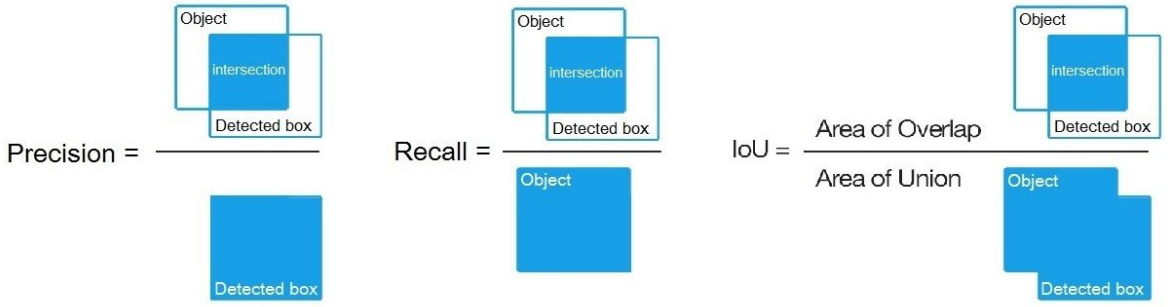


Fig. 1 Precision, Recall and IoU

- Mean Average precision (mAP): average precision
- Frame Per Second (FPS): number of detected box per second

So, if we have higher mAP, recall, IoU and FPS, it means that we have more detected box fit with the ground truth object in one second. And it is what we want.

II. Analysis

Data Exploration

COCO

COCO is a large-scale object detection, segmentation, and captioning dataset. COCO has several features:

- Object segmentation
- Recognition in context
- Superpixel stuff segmentation
- 330K images (>200K labeled)
- 1.5 million object instances
- 80 object categories
- 91 stuff categories
- 5 captions per image
- 250,000 people with keypoints

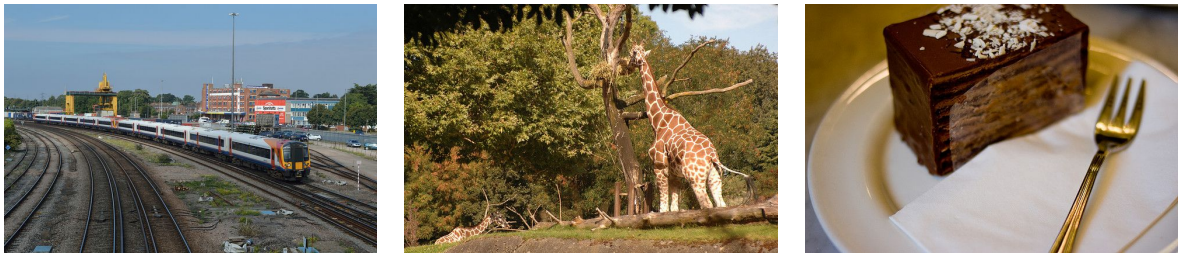


Fig.2 COCO images

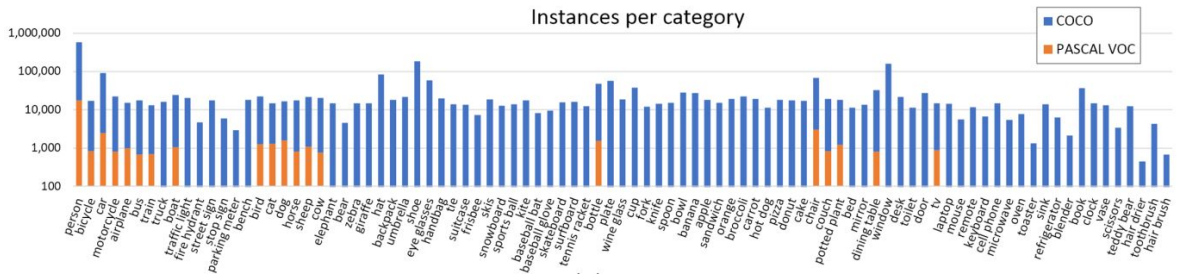
EgoHands

This dataset works well for several reasons. It contains high quality, pixel level annotations (>15000 ground truth labels) where hands are located across 4800 images. All images are captured from an egocentric view (Google glass) across 48 different environments (indoor, outdoor) and activities (playing cards, chess, jenga, solving puzzles etc).[6]

Fig.3 EgoHands images

Exploratory Visualization

Below are the data distribution captured from COCO, we can find all categories should have around 10k images and more than 70% of instance less than 10% of image size, Which means that instances are not so big.



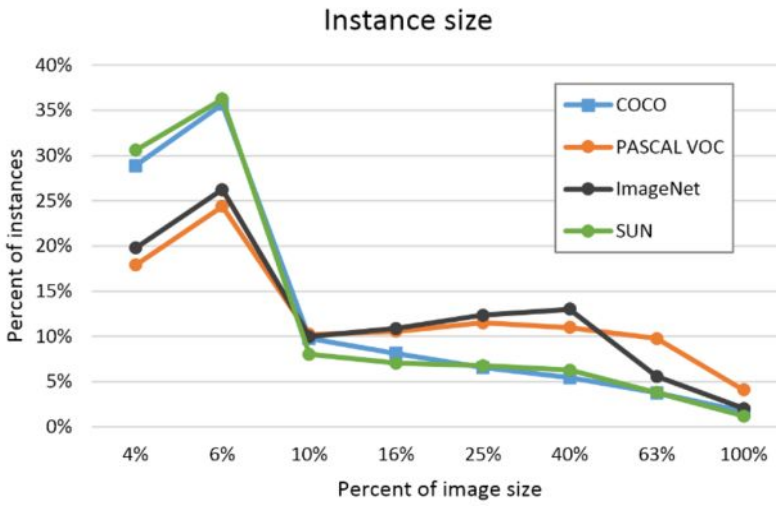


Fig. 4 COCO data distribution. [4]

For EgoHands, the figure captured from Analysis of Hand Segmentation in the Wild [7] as below. We can find hand almost appear at the bottom side and right handed as the heatmap, and almost all hand size less than 10% of the image as the histogram. Also, compare to COCO and EgoHands, we can find that EgoHands' amount is relatively small with respect to any one of COCO category.

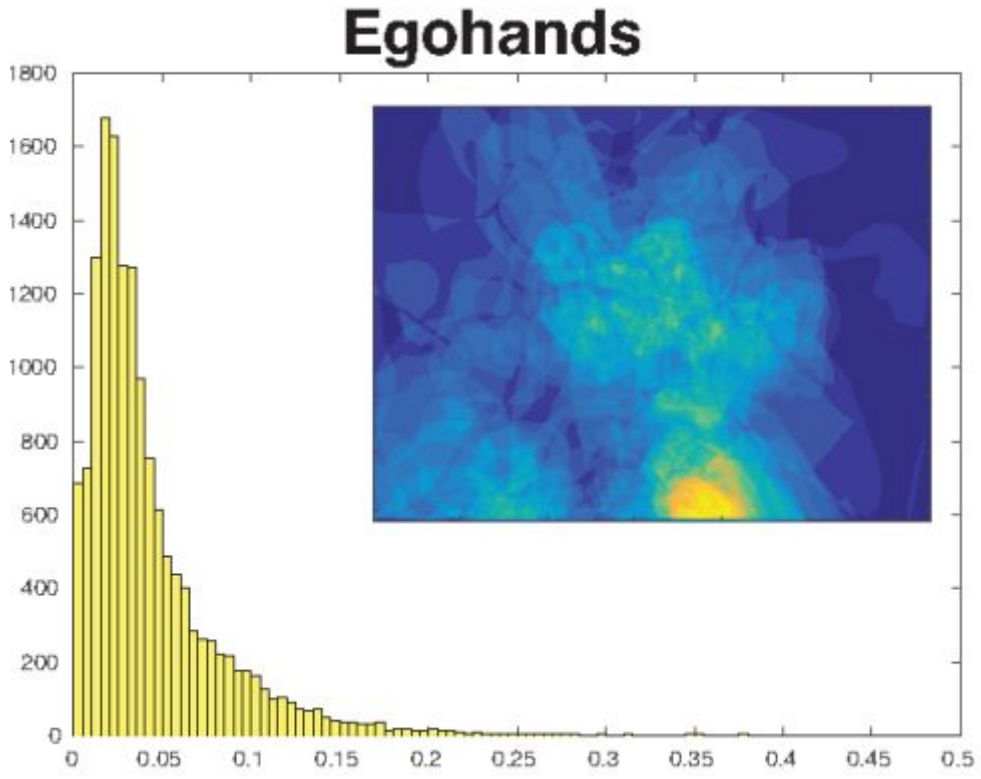


Fig. 5 Visualizing the heatmaps for spatial hand occurrence in EgoHands dataset along with the histogram for bounding box area of hand relative to the image size. Histograms tell about the distribution of hand size (in terms of area) in this datasets. [7]

Algorithms and Techniques

For real time object detection classifier I would like to pick the famous SSD [8] as my candidate. The paper about SSD: Single Shot MultiBox Detector (by C. Szegedy et al.) was released at the end of November 2016 and reached new records in terms of performance and precision for object detection tasks, scoring over 74% mAP (mean Average Precision) at 59 frames per second on standard datasets such as PascalVOC and COCO. [9]

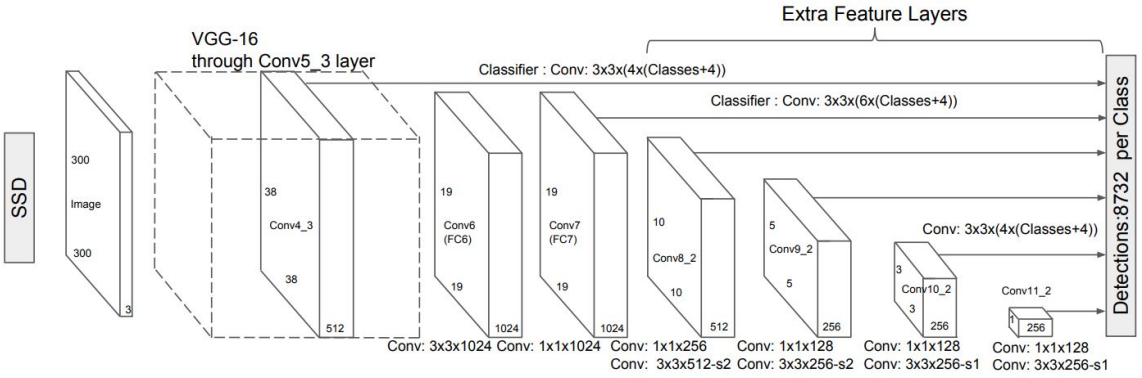


Fig. 6 Architecture of Single Shot MultiBox detector (input is 300x300x3)

As you can see from the diagram above, SSD's architecture builds on the venerable VGG-16 architecture, but discards the fully connected layers. The reason VGG-16 was used as the base network is because of its strong performance in high quality image classification tasks and its popularity for problems where transfer learning helps in improving results. Instead of the original VGG fully connected layers, a set of auxiliary convolutional layers (from conv6 onwards) were added, thus enabling to extract features at multiple scales and progressively decrease the size of the input to each subsequent layer. [9]

Making multiple predictions containing boundary boxes and confidence scores is called multibox.[10]

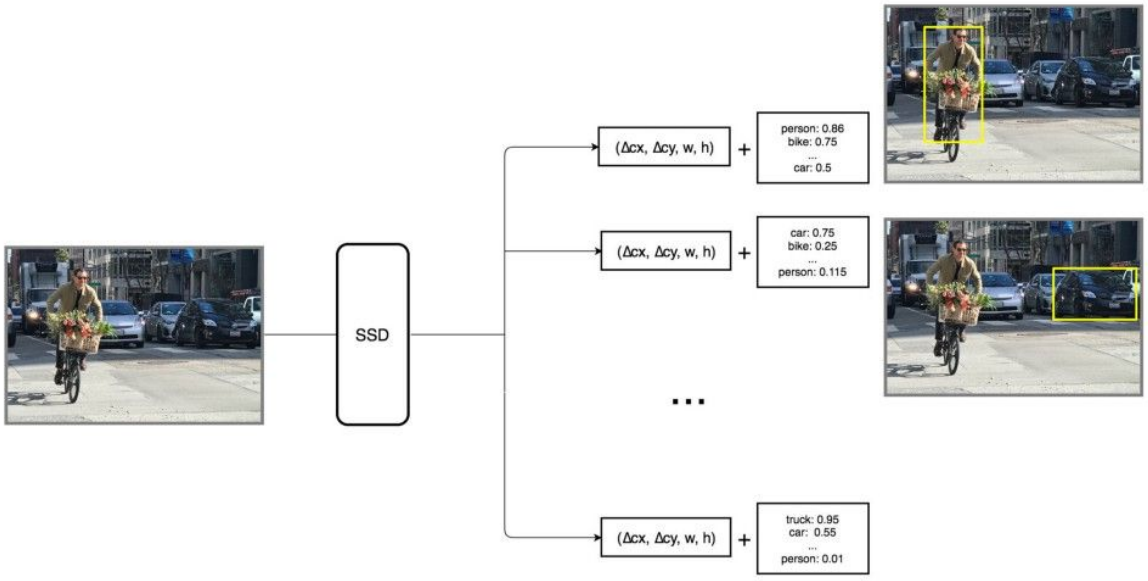


Fig. 7 Each prediction includes a boundary box and 21 scores for 21 classes (one class for no object).[10]

Above is some example run on PascalVOC [11], which has total 20 classes.

Benchmark

From the YOLO website [12], we can find it has some comparison between SSD and YOLO as below. YOLOv2 has higher mAP and SSD has higher FPS. The mAP would be decided during the training stage, we can't change. How about the FPS? Due to the original paper run the SSD and YOLO on a fat desktop GPU, and how about the FPS on iPhone? I would like to run SSD and YOLO on iPhone and compare their FPS. For real time detection, I hope the FPS would be larger than 5 FPS or more.

Performance on the COCO Dataset

Model	Train	Test	mAP	FLOPS	FPS
SSD300	COCO trainval	test-dev	41.2	-	46
SSD500	COCO trainval	test-dev	46.5	-	19
YOLOv2 608x608	COCO trainval	test-dev	48.1	62.94 Bn	40
Tiny YOLO	COCO trainval	test-dev	23.7	5.41 Bn	244

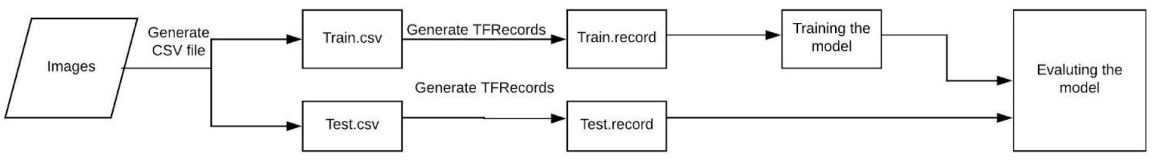
Fig. 8 SSD vs YOLO on COCO dataset

III. Methodology

Data Preprocessing

In order to decrease the training time, We will use pre-trained COCO weight from SSD and YOLO. So, we will ignore COCO data processing. For the EgoHands portion, since we use Tensorflow Object Detection API [13], we first need to convert data to tensorflow data record. I follow the deeptraining_hands [14] to build my training environment. Below are some pre-processing steps.

- Download pretrained SSD and YOLO weight
    - SSD weight
        - We will use ssd_mobilenet_v1_coco [15] from Tensorflow detection model zoo [16].
    - YOLO weight
        - We will use YOLOv2 [17] and Tiny YOLOv2 [18] from YOLO web page.
- Prepare for Tensorflow Object Detection
    - Basic flow



    - The egohands_setup.py will do following things.

- Downloads the egohands datasets
    - The Egohands data zip file, with labelled data, contains 48 folders of locations where video data was collected (100 images per folder).

      -- LOCATION_X
      -- frame_1.jpg
      -- frame_2.jpg
      ...
      -- frame_100.jpg
      -- polygons.mat // contains annotations for all 100 images in current folder
      -- LOCATION_Y
      -- frame_1.jpg
      -- frame_2.jpg
      ...
      -- frame_100.jpg
      -- polygons.mat // contains annotations for all 100 images in current folder
- Renames all files to include their directory names to ensure each filename is unique
- Splits the dataset into train (90%) and eval (10%) folders.
- Reads in polygons.mat for each folder, generates bounding boxes and visualizes them to ensure correctness..
- Once the script is done running, you should have an data/train/images and data/eval/images folder. Each of these folders should also contain a csv label document each - data/train_labels.csv, data/eval_labels.csv that can be used to generate tfrecords.

Implementation

The whole implementation step is as below. I will describe each step in this following section.

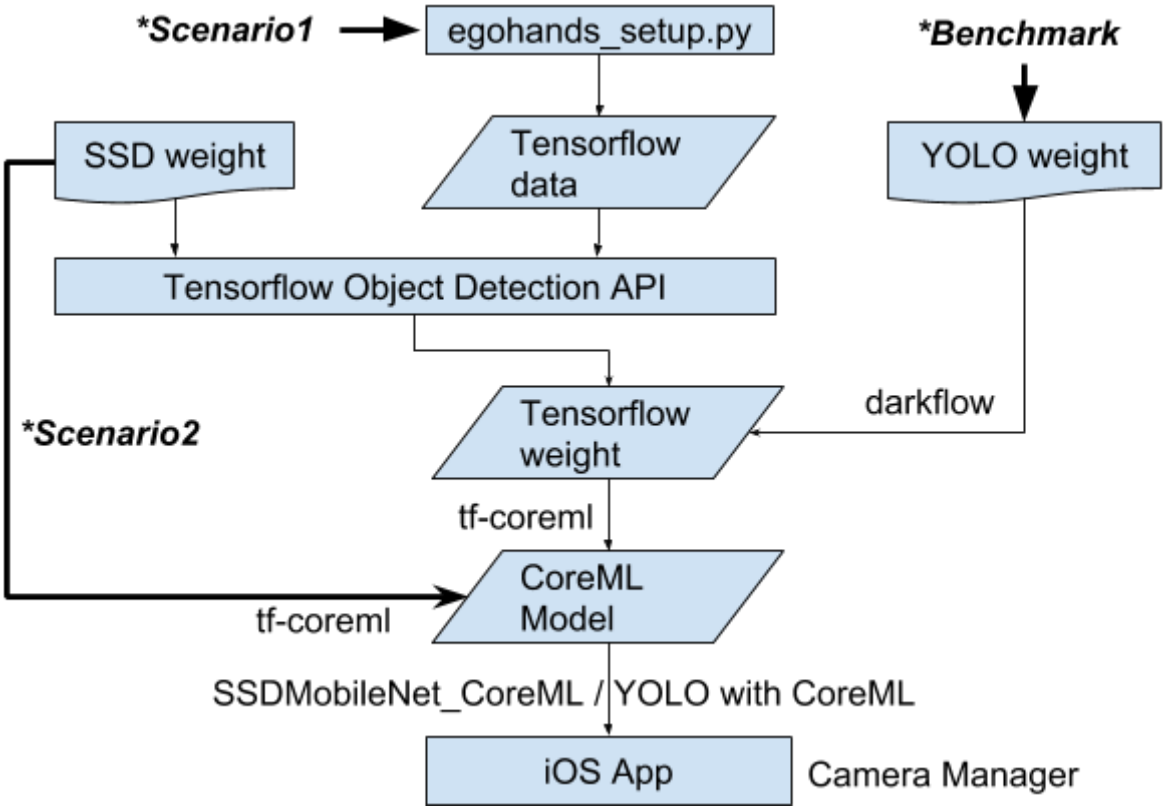


Fig. 9 Implementation flow

Since we have two scenario and one benchmark, and the specific implementation steps is as below.

*Scenario1: SSD with EgoHands

- Run egohands_setup.py to download EgoHands data and construct deeptraining_hands environment and get tensorflow data record as previous section mentioned.
- Download pretrained SSD weight from Tensorflow detection model zoo.
- Apply transfer learning of SSD weight and EgoHands tensorflow data record with Tensorflow Object Detection API to get Tensorflow weight.
- Convert Tensorflow weight to CoreML model with tf-coreml [19].
- Create iOS App following SSDMobileNet_CoreML [21] and apply CoreML model to predict hand.
- Import Camera Manager [23] module to provide zoom in/out feature.

*Scenario2: SSD with COCO

- Download pretrained SSD weight from Tensorflow detection model zoo.

- Convert SSD weight to CoreML model with tf-coreml.
- Create iOS App following SSDMobileNet_CoreML and apply CoreML model to predict COCO.
- Import Camera Manager module to provide zoom in/out feature.

*Banchmark: YOLO with COCO

- Download pretrained YOLOv2 and Tiny YOLOv2 weight from YOLO.
- Convert YOLO weight to Tensorflow weight with darkflow [20].
- Convert Tensorflow weight to CoreML model with tf-coreml.
- Create iOS App following YOLO with CoreML [22] and apply CoreML model to predict COCO.
- Import Camera Manager module to provide zoom in/out feature.

Below are some summary for the implementation

- Tensorflow Object detection API
    - Follow the TensorFlow Object Detection API tutorial—Training and Evaluating Custom Object Detector [24]. It should be fine for me.
- CoreML Model Conversion
    - SSD to CoreML
        - Follow the SSDMobileNet_CoreML, it should be fine for me.
    - YOLO to CoreML
        - Since darkflow have some bug during the conversion process, we need to change below line to fix it.
            - Convert YOLOv2 weight
                - Line 121 of darkflow/darkflow/utils/loader.py
                    - self.offset = 16
            - Convert Tiny YOLOv2 weight
                - Line 121 of darkflow/darkflow/utils/loader.py
                    - self.offset = 20
- iOS App CoreML Model integration
    - Pay attentation to the output array of SSDMobileNet_CoreML. The SSD will output two arrays, array_0 should be box coordinate and array_1 should be confidence scores.

Refinement

For the SSD transfer learning of tensor object detection API, I have not change too much on the hyperparameters, except some portion to meet my dataset characteristics. The default hyperparameters would be fine for me.
By observing the tensorboard graphs for total loss(see image below), it should be possible to get an idea of when the training process is complete (total loss does not decrease with further iterations/steps). I ran my training job for 200k steps (took about 2 days) and stopped

at a total Loss (errors) value of 1.359. For the hand detector trained here, the mAP value was 0.9306@0.5IOU. mAP values range from 0-1, the higher the better.

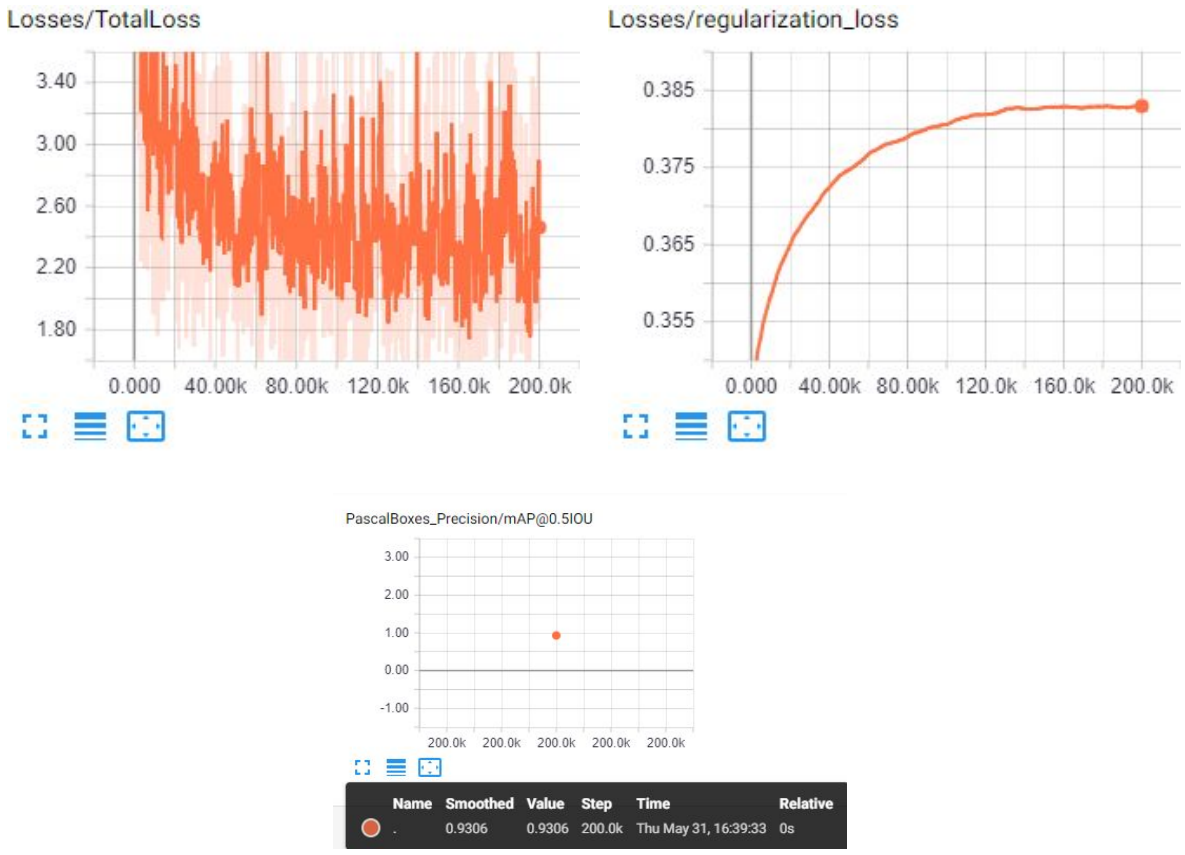


Fig. 10 Losses and mAP of SSD with EgoHands

IV. Results

Model Evaluation and Validation

To test my hand model accuracy, I have tested the model on the Hand Dataset [25] as below. I randomly pick 6 images of the similar characteristics of EgoHands dataset. But

there are only 2 out of 6 images been detected, as left top and right top. Although my model mAP is 93, but the result seems not that good.

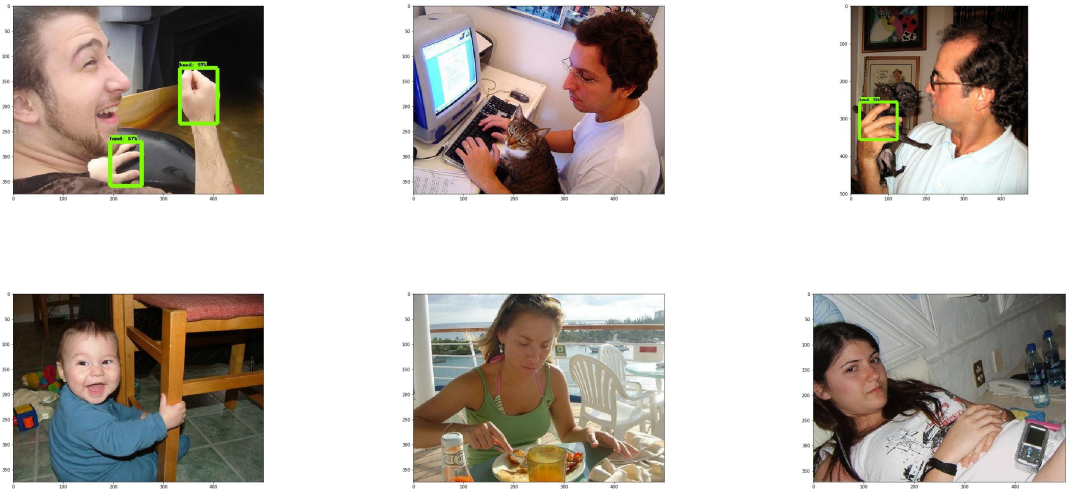


Fig. 11 Hand detection with Hand Dataset

Even it can only detect 2 out of 6 images of these images, but detected box corrected mark up the hand, so I think the model align well with the hand detection solution. Although it is not too robust to detect all hand. But all its detected box correctly mark up the hand, so i think the model should be trusted.

Justification

Below are the results run on my iPhone 7. Since mAP already decided in the training phase, there are three out of four mAP are captured from original author. Compare SSD with EgoHands and COCO, we can find that EgoHands have larger mAP and FPS. it should be acceptable, because EgoHands' size and category is much smaller than COCO. Compare SSD and YOLOv2 with COCO, why YOLOv2 slow 4 times (9/2) than SSD? I think that may be original author use good desktop GPU. Besides, the size of YOLOv2 model is 216 MB and SSD is 27 MB, which means YOLOv2 model size is 8 times larger than SSD. I think this may probability the root cause. I think YOLOv2 may not be suitable to run on resource restricted device. But how about SSD and Tiny YOLOv2 on COCO, Although Tiny YOLOv2 is faster than SSD. But in practice, It's really not easy to train Tiny YOLOv2. Among my experience, I use CPU to train EgoHands with SSD would take 2 days to complete 200k steps. But with Tiny YOLOv2, it will take 1 week to run 2k steps, and YOLOv2 need 3 week to run 2k steps. The training time of YOLO is really slow on CPU. My CPU is 2 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz (10 Core).

Model	mAP %	FPS	model size - MB
SSD with EgoHands	93	9.x	27
SSD with COCO	21 [16]	6.x	27
YOLOv2 with COCO	48.1 [12]	2.x	216
Tiny YOLOv2 with COCO	23.7 [12]	12.x	45

Fig. 12 Perforance comparison on iPhone7

To see how well the result is, I will show some results of my implementations in Fig. 13, 14 and 15 as next section.

My original purpose is to make a magnifier with trace object/hand functionality, I think the result would be somehow match my purpose.

V. Conclusion

Free-Form Visualization

Although EgoHands instance are a little right handed and almost less than 10% of the image. But when I integrate the trained model in an iOS App, I think the result are not so bad as following.
Below is the real iOS magnifier app I created for detecting hand. It should be what I want. In normal situation, as left image, I can use it to detect hand. You can see I zoom in 1x with a FPS of 9.835, and the detected hand with a probability of 0.96 or 96%. I can even zoom in to see what person what me to focus on, as in the right image. You can see I zoom in 3x to some code portion.

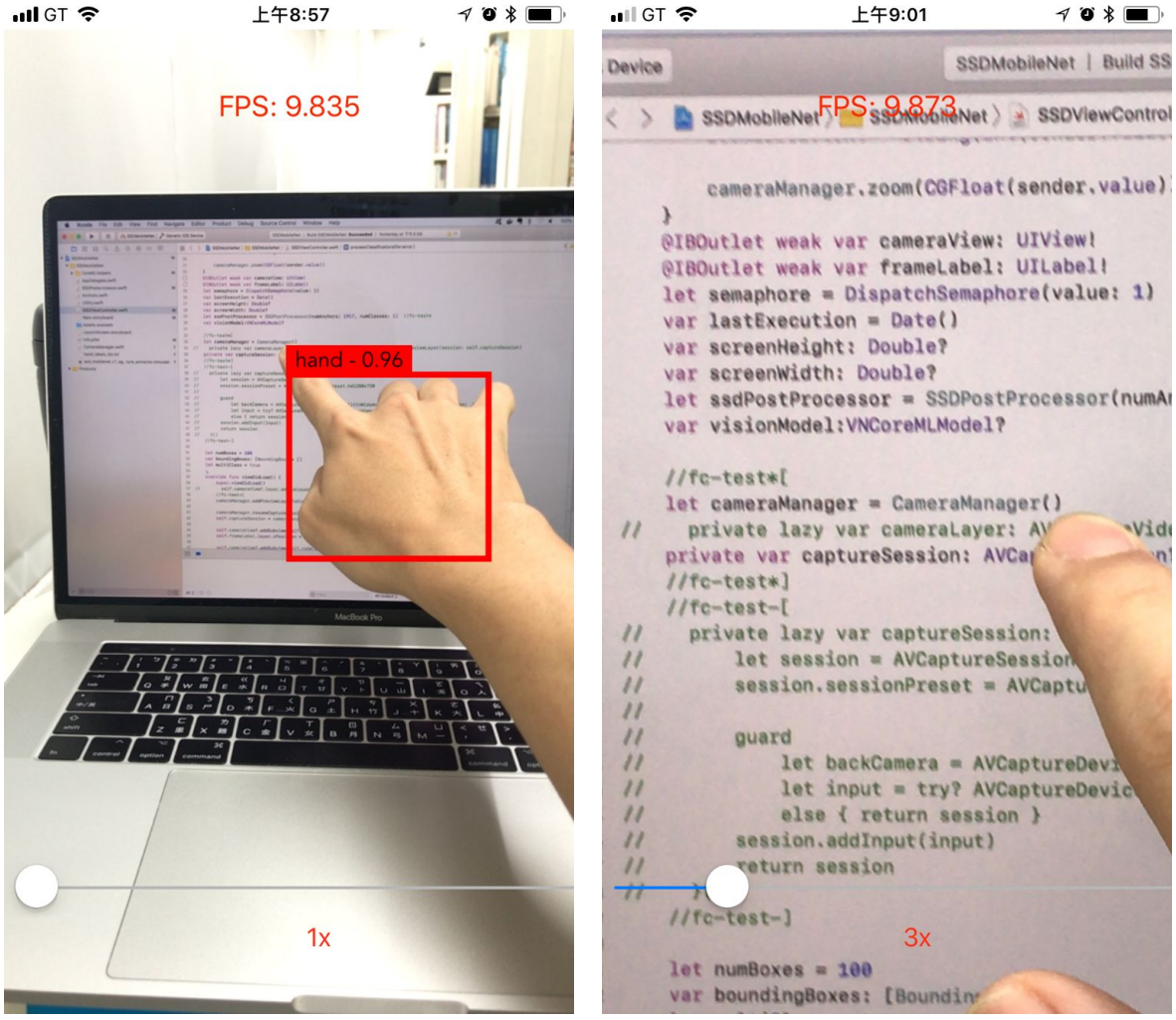


Fig. 13 iOS App - magnifier for detecting hand and zoom in to some code portion

Below is an iOS magnifier App using SSD to detect COCO. In the left, I use it to detect COCO with a FPS of 5.452. In the right, I can zoom in 3x to some area and it can detect more detail things with a FPS of 5.732.

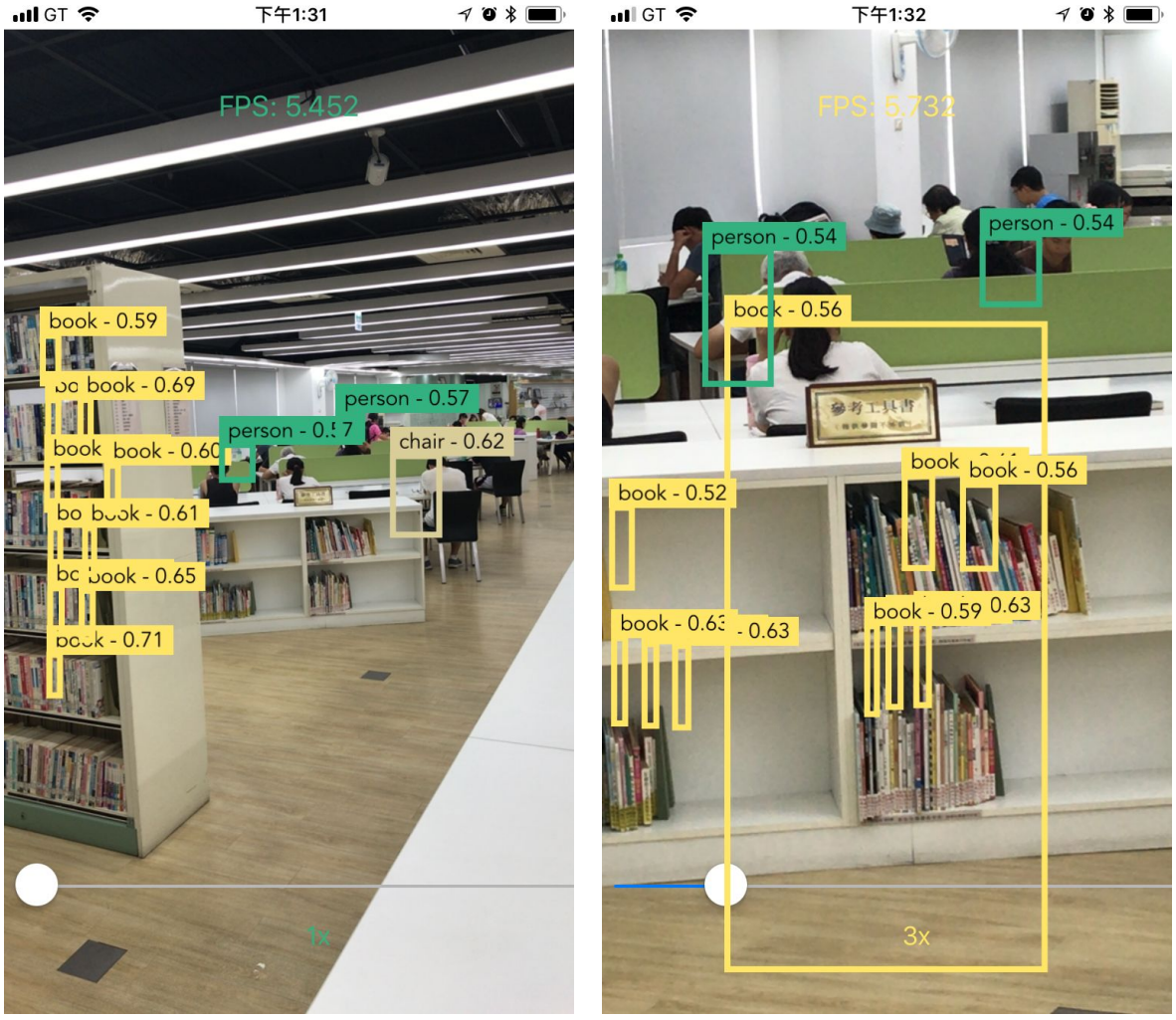
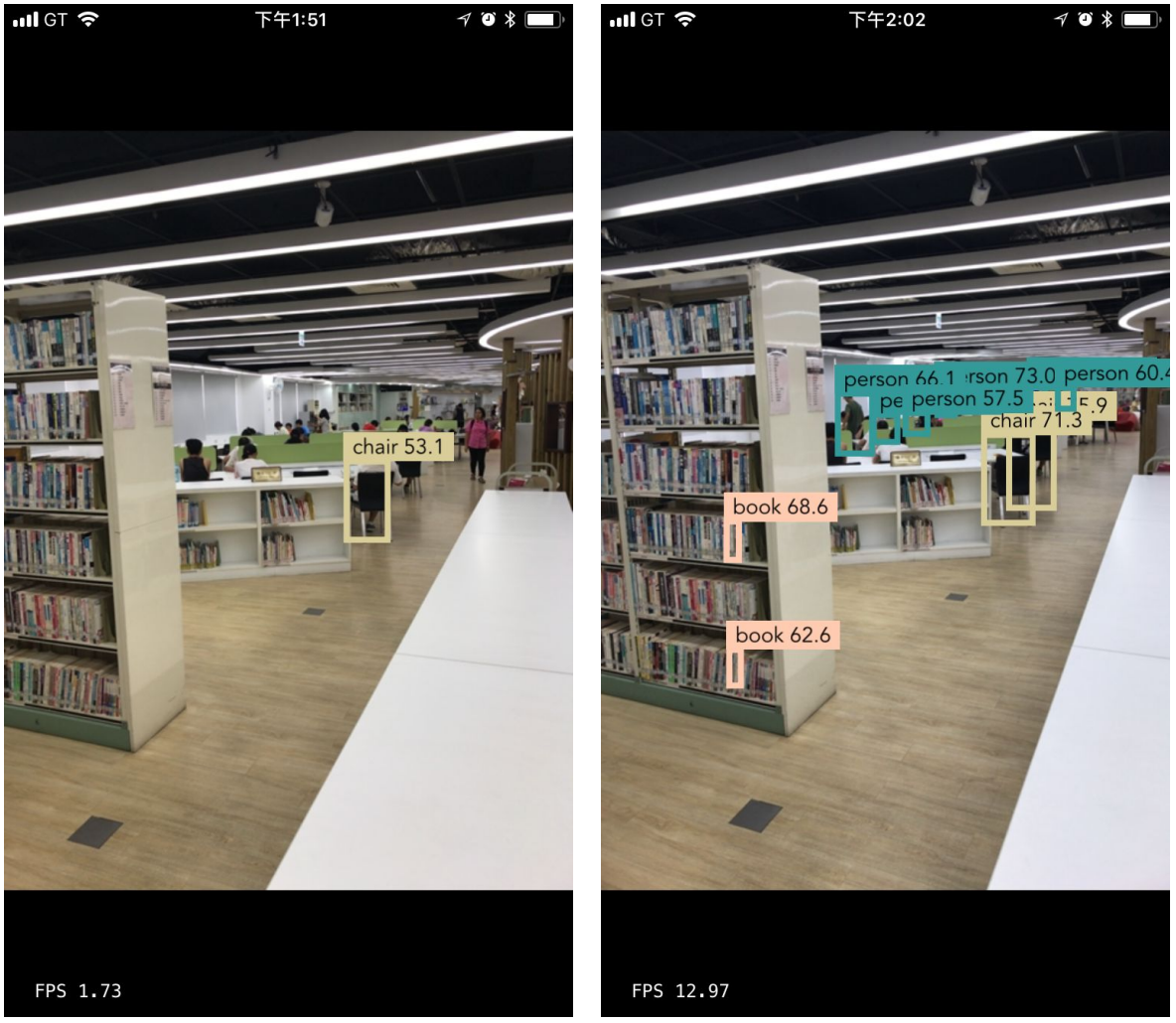


Fig. 14 iOS App - magnifier for detecting COCO

Below is an iOS app using YOLO to detect COCO. We can find that since Tiny YOLOv2 is much more faster than YOLOv2, so the Tiny YOLOv2 detected object is much more than YOLOv2.



YOLOv2 Tiny YOLOv2

Fig.15 iOS App - YOLO detect COCO

Reflection

I would like to summarize the entire process I used in this project as below.

1. Declare the problem and find the related dataset, and project.
2. Download the related dataset and project and survey the training model.
3. Data pre-processing for the training model.
4. Train the classifier and verify the result.
5. Survery how to import the classifier to the iOS app.
6. Download related iOS project and apply the classifier to the app.
7. import some additional functional module to the app.
8. Vesualize and test the result.

I think the most interesting part of this project is step 6 and 7. I hadn't even touched iOS app programming ever. With this project, I not only need to learn how to program iOS app, but also need to learn iOS GUI and camera programming. It's kind of challenge and interesting.

The most difficult part of the project should be step 3 and 4. Although there are many proven training process published in the internet, but the training process would be so long. I use 2 intel(R) Xeon server level CPU to train SSD to detect hand in 2 days of 200k steps. But, when I want to train YOLO to detect hand, I found that it seems like not what I can do in the CPU. The training time to detect hand for YOLOv2 is 2k step in 3 weeks, and Tiny YOLOv2 even need to run 1 week for only 2k steps. Originally, I would also like to provide my YOLO classifier. But, finally I give up my YOLO hand classifier. I think it is not what I can do in the CPU. It should be GPU level task.
The final result of SSD detect hand is 9 FPS, and even detect COCO it can still run to 6 FPS. Although it is not that real time of 30 FPS, but in everyday life, I think it should be enough.

Improvement

Below are some improvement I think I can do further.

1. Apply Tiny YOLOv2 to detect hand
    - From the Fig. 12, I think the performance of Tiny YOLOv2 would be better than SSD. But since I have no good GPU, I can't have a try. If someday I have a good GPU. I definitely want to give it a try.
2. Detect little change of the image and mark it up
    - In this project, I make a magnifier to detect hand. But I think I can improve this app by detecting motion with image subtraction of OpenCV. In many case, even though we can't detect object, but we still can mark it up. I think this should be useful.
3. Combine hand and COCO into one model
    - Due to resource limitation, I train one model to detect hand and use one pretrained model to detect COCO. If I have good GPU, I think I can retrain the model to detect COCO and hand, and not need to separately run two app. But I think it should be a long time even with good GPU.
4. Enhance inference speed
    - Apple announced new ARKit 2.0 recently, which focus on performace improvement. My implementation is based on AVFoundation. I think if I change to ARKit, the performance should have some improvement.
5. Enlarge hand data
    - Because EgoHands dataset emphasis on hand egocentric point of view. During the test phase, I found that I can only detect some hand. I think I can detect more hand with larger hand data. I think Hand Dataset would be a good one.

VI. Reference

[1] Amblyopia (https://en.wikipedia.org/wiki/Amblyopia)
[2] SuperVision+ Magnifier (https://itunes.apple.com/us/app/supervision-magnifier/id691435681?mt=8)
[3] eSight (https://www.esighteyewear.com/)
[4] Microsoft Common Objects in Context (COCO) (http://cocodataset.org/#home)
[5] EgoHands: A Dataset for Hands in Complex Egocentric Interactions (http://vision.soic.indiana.edu/projects/egohands/)
[6] Real-time Hand-Detection using Neural Networks (SSD) on Tensorflow (https://github.com/victordibia/handtracking)
[7] Analysis of Hand Segmentation in the Wild (https://arxiv.org/abs/1803.03317)
[8] SSD: Single Shot MultiBox Detector (https://arxiv.org/pdf/1512.02325.pdf)
[9] Understanding SSD MultiBox—Real-Time Object Detection In Deep Learning (https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab)
[10] SSD object detection: Single Shot MultiBox Detector for real-time processing (https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06)
[11] The PASCAL Visual Object Classes Homepage (http://host.robots.ox.ac.uk/pascal/VOC/)
[12] YOLO: Real-Time Object Detection (https://pjreddie.com/darknet/yolo/)
[13] Tensorflow Object Detection API T(https://github.com/tensorflow/models/tree/master/research/object_detection)
[14] deeptraining_hands (https://github.com/GustavZ/deeptraining_hands)
[15] ssd_mobilenet_v1_coco model (http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v1_coco_2017_11_17.tar.gz)
[16] Tensorflow detection model zoo (https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md)
[17] YOLOv2 Weight (https://pjreddie.com/media/files/yolov2.weights)
[18] Tiny YOLOv2 weight (https://pjreddie.com/media/files/yolov2-tiny.weights)
[19] tf-coreml (https://github.com/tf-coreml/tf-coreml)
[20] darkflow (https://github.com/thtrieu/darkflow)
[21] SSDMobileNet_CoreML (https://github.com/vonholst/SSDMobileNet_CoreML)
[22] YOLO with CoreML (https://medium.com/@syshen/yolo-with-coreml-819799789c11)
[23] Camera Manager (https://github.com/imaginary-cloud/CameraManager)
[24] TensorFlow Object Detection API tutorial—Training and Evaluating Custom Object Detector (https://becominghuman.ai/tensorflow-object-detection-api-tutorial-training-and-evaluating-custom-object-detector-ed2594afcf73)
[25] Hand Dataset (http://www.robots.ox.ac.uk/~vgg/data/hands/)