National University of Singapore

Institute of System Science, Intelligent System

The design of a real-time sign language recognition system based on YOLO

Group 13

Full Name	Student ID
Yan JiaHuan	A0261968M
Shu WanYang	A0261754B
Xiao ChangWei	A0226757U

Project Supervisor: Dr. Zhu Fangming

Contents

1 | Introduction

Sign Language is a medium for communication used primarily by people who are either deaf or mute. People use it to communicate their thoughts to the world. There are different types for sign language based on geography and context of spoken language such as American Sign Language, British Sign Language, Japanese Sign Language, etc. The emphasis of this research is on American Sign Language (ASL).

Communication through sign language can be orchestrated in a variety of ways. There are certain words of the spoken language that can be directly represented and interpreted through simple gestures. Words like Hello, Mom, Dad, etc. have designated signs or gestures. However, there are certain words that don't have pre-defined signs. In this case, a technique called "Fingerspelling" is used to spell the word out using sings for individual alphabets. Typically, fingerspelling is used when someone is trying to convey their name.

The most common technique in this field is using Image Processing Algorithms to extract features from orchestrated gestures and then using Neural Networks to learn these features and increase utility. Advances in deep learning have led to the creation of Object Detection Algorithms such as You Only Look Once(YOLO).

In this project, the purpose is to propose a system that can accurately identify the orchestrated gesture, map it to the desired word or alphabet in the sign language vocabulary and try to generate a reasonable sentence by this words. Because we want the system light enough to be easily deployed in device with low storage, 2We adopt YOLO-Fastest-v2 as the training model. YOLO-Fastest-v2 model is very light, the weight file after training is only about 1MB, perfectly match our need.

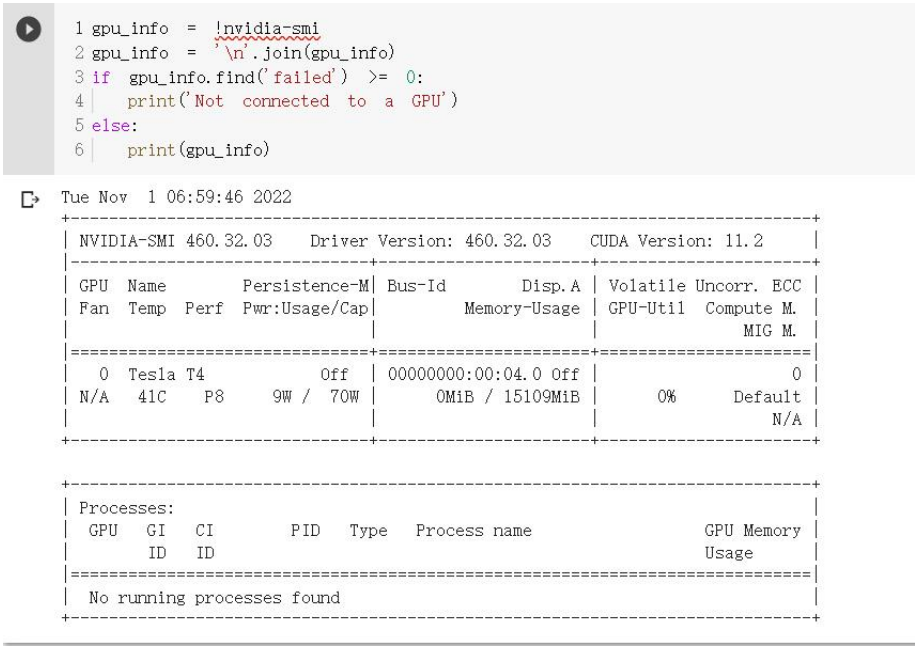
People using sign language often need to rely on a translator to convey what they are trying to say to a person that does not understand sign language. With the help of our system, people use sign language without depending on another person can really help them be independent and ignite the confidence to present themselves to the world without any fear.

2 | Tools and introduction

2.1 | The environment of projects:

2.1.1 | Google Colaboratory:

Considering the information carriers in this project are images of hand gestures, the bytes in every individual data representation will be quite large. Our team members temporally aren't equipped with high-performance Pcs with GPU. Hence, we decided to implement the processing of raw data and the procedure of training on Google Colabortory (Colab). Colab is a large-scale cloud computing platform for machine learning and deep learning programs by providing computational hardware. One of the largest advantages of Colab is that you don't have to bother the package dependencies problems (Some versions of tensorflows will have mismatch errors with other packages or editions of python, especially python 3.9+). Colab provides Tesla T4 GPU and a high ram for running the machine learning programs, which has been shown below.



```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
Tue Nov  1 06:59:46 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   41C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 2.1: The GPU version in Colab

2.1.2 | Anaconda+Python 3.9:

The ideal final implementation of our system is on local, which means we need to build a lcoal enviroments to apply our after-trained model. The anaconda3 is used in our projects as both the enviroment and python compiler. We export the after-trained weights from Colab and utalize it to run the local scripts to detect the sign language from local image, video or web camera. The packages need to be installed on local has been listed on below.

- Python 3.9.1
- Tensorflow 2.1.0
- keras 2.1.0
- opencv-python 4.6.0
- pillow 9.3.0

Some of the package contains the normal machine learning functinal packages like scikit-learn, matplotlib and numpy. Hence, just install the aforementioned 4 packages with the correct edition of python, the overal local enviroment setup will meet the requirements. Tensorflow.keras is a TensorFlow high-level API for building and training deep learning models. We build our training model and detection script based on keras fucntions.

2.1.3 | OpenCV:

OpenCV is the full name of Open Source Computer Vision Library. OpenCV is an distribution-based cross-platform computer vision library that runs on Linux, Windows, and Mac OS operating systems. It's lightweight and efficient – a collection of C functions and a handful of C++ classes. It also provides interfaces to Python, Ruby, MATLAB, and other languages, and implements many common algorithms in image processing and computer vision.

OpenCV aims to be a standard API that simplifies the development of computer vision programs and solutions. OpenCV is dedicated to real world real-time applications, and its execution speed is improved through optimized C code writing.

OpenCV has many application scenarios. Including image recognition, object detection, image segmentation, style transfer, image reconstruction, super resolution, image generation, face recognition and so on. OpenCV can help engineers with most computer vision tasks.

2.1.4 | MediaPipe:

MediaPipe is a framework for building machine learning pipelines for processing video, audio, and other time series data. This cross-platform framework works on desktop/server, Android, iOS, and embedded devices such as Raspberry Pi and Jetson Nano.

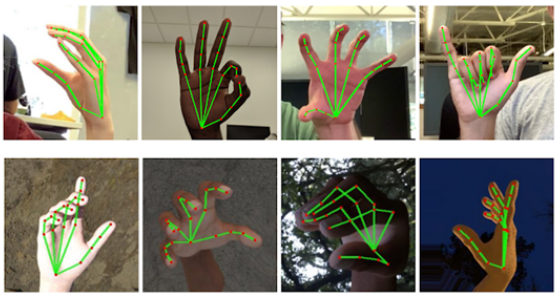


Figure 2.2: Hand lines in mediapipe

MediaPipe powers revolutionary products and services we use every day. Unlike resource-intensive machine learning frameworks, MediaPipe requires minimal resources. It's so tiny and efficient that even embedded it devices can run it. When MediaPipe was publicly released in 2019, it opened a whole new world of opportunities for researchers and developers.

The ability to perceive the shape and motion of hands can be a vital component in improving the user experience across a variety of technological domains and platforms. For example, it can form the basis for sign language understanding and hand gesture control and can also enable the overlay of digital content and information on top of the physical world in augmented reality. While coming naturally to people, robust real-time hand perception is a decidedly challenging computer vision task, as hands often occlude themselves or each other (e.g., finger/palm occlusions and handshakes) and lack high contrast patterns.

MediaPipe Hands is a high-fidelity hand and finger tracking solution. It employs machine learning (ML) to infer 21 3D landmarks of a hand from just a single frame.

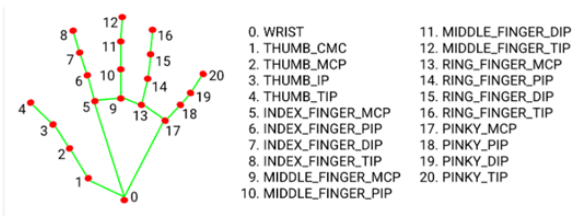


Figure 2.3: 21 joint positions

2.1.5 | Wordninja:

Wordninja is a Python library for splitting consecutive English strings.

It is based on the frequency of the wiki corpus, and in English it is reasonable to assume that the distribution follows Zipf's Law, which builds a lexicon from the highest to the lowest probability of occurrence, for the purpose of word separation.

2.1.6 | PySpellCheck:

Pure Python Spell Checking based on Peter Norvig's blog post on setting up a simple spell checking algorithm.

It uses a Levenshtein Distance algorithm to find permutations within an edit distance of 2 from the original word. It then compares all permutations (insertions, deletions, replacements, and transpositions) to known words in a word frequency list. Those words that are found more often in the frequency list are more likely the correct results.

The whole project is divided into 4 modules illustrated as figure. Module 1 is the get key frame module, which is aimed to get the frame of hand from every frame from video Capture and resize to the size required by YOLO detect. Module 2 is the YOLO training module, which is aimed to get the weight needed by YOLO detect Model. Module 3 is the YOLO detect Model, which is aimed to get the label by the key frame. Module 4 is the subtitle generate model, which is aimed to generate a semantically sound sentence and add them on the video.

2.2 | Pretrained weights

One of the main reason of why pre-trained model has been prequently in recent deep learning models is the scarcity of valid labeled data. Only a very small amount of relevant training data exists for the particular task, so the model cannot learn to draw useful patterns from it. In our project, we applied a Yolo pre-trained model which has been trained from COCO 2017 training images (contains around 200k images and around 80 classes). We wish to obtain the fundamental classification ability from the pre-trained weights and transplant the ability to our own models. By learning the commonalities through this pre-training method, then transplanting the commonalities into a task-specific model and fine-tune it with a small amount of labeled data from our American sign language dataset, so that the model only needs to "learn" the "special" parts of that particular task from the "commonalities".

2.3 | Dataset

This project is determined to solve the sign language recognition task. According to our research, there are more than 140 sign languages around the world. It could be a temporally unsolvable problem if we want to build a system to recognize all the sign languages. Because some sign languages are mainly used by minorities compared to all groups of sign language users, and the valid datasets haven't been collected yet. Also, the diversity of hand gestures is limited, and the same hand gestures can represent different meanings in different sign language. Hence, we determined to solve the sign language recognition task on American sign language. The language components are not the daily communication phrases often used like other sign languages, but the 26 English letters. There will be 26 classes on our dataset, and each represents a letter, then the semantics will be constructed by amalgamating the letters to words separately.

Figure 2.4: American Sign Language

In this project, we choose an ASL public dataset from roboflow collected by David Lee. Click here to the public dataset. This dataset contains total 1728 images, each letter has around 60 samples. It is obviously a small dataset, but all the classes have almost same scale of samples, which means it is nearly balanced.

2.3.1 | Data Augmentation

The image augmentation techniques are applied in our data processing procedure. We didn't choose the methods that can cause displacement, as the information of boxes (x,y,w,h) will be changed accordingly. Our strategy is to randomly add noises into the image. By applying this image augmentation, the overfitting problems can be alleviated.

Figure 2.5: Left: image before augmentation. Right: image after augmentation

2.3.2 | Raw data labeling

There is mislabeling issue in this dataset. Some letters like 'z' outnumber all other samples, and letters like 'N' and 'M' are much rarer than others. Hence, we need to make self-labeled data to compensate the imbalance issue.
We further use Labelimg tool to collect our own dataset to try to counteract the imbalance. We manually collected more than 500 reliable samples. The camera will save the image according to to set up sampling intervals when we twist our hands, the same hand gestures from different angles will be used. The reliable images are collected from the total raw data and labeled by three candidates. The label each represents the class of hand gesture, the coordinate of the center of box, and the height and with of the box.

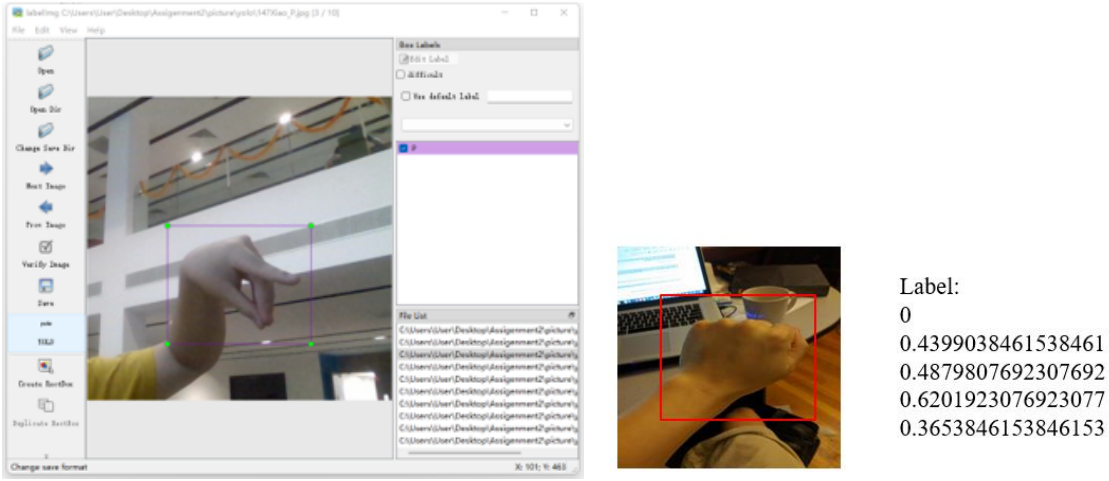


Figure 2.6: self-labeled data use labelimg tool

3 | System Design

3.1 | Overview Architecture:

The whole project is divided into 4 modules illustrated as figure. Module 1 is the get key frame module, which is aimed to get the frame of hand from every frame from video Capture and resize to the size required by YOLO detect. Module 2 is the YOLO training module, which is aimed to get the weight needed by YOLO detect Model. Module 3 is the YOLO detect Model, which is aimed to get the label by the key frame. Module 4 is the subtitle generate model, which is aimed to generate a semantically sound sentence and add them on the video.

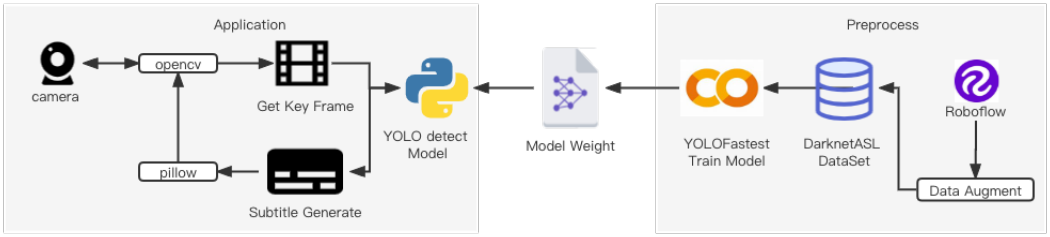


Figure 3.1: Architecture

3.2 | Get Frame:

The first step of the whole project is to use the camera to obtain images of our hands, and then transmit them to the subsequent YOLO model for sign language recognition. Call camera, read image, store image and a series of computer vision related work involves the application of OpenCV.

First OpenCV calls the camera using the "VideoCapture(0)" function. The camera on the laptop will be turned on and begin to capture live images.

```
#Read the OpenCV library
import cv2

#Turn on the laptop's built-in camera
cap = cv2.VideoCapture(0)

"""
ret: True or False indicates that the image is not read
frame: Represents a frame of an image captured
"""
ret,frame = cap.read()
```

Figure 3.2: OpenCV code

After acquiring the initial image, the system cannot easily recognize the hand posture. Because the hand is only part of the image, the whole image contains a lot of useless information, such as the person's face, arms, upper body and so on. MediaPipe is helpful when we want to get the hand position. MediaPipe can provide us with their specific coordinates through the recognition of 21 joints in the hand, to locate the hand position.

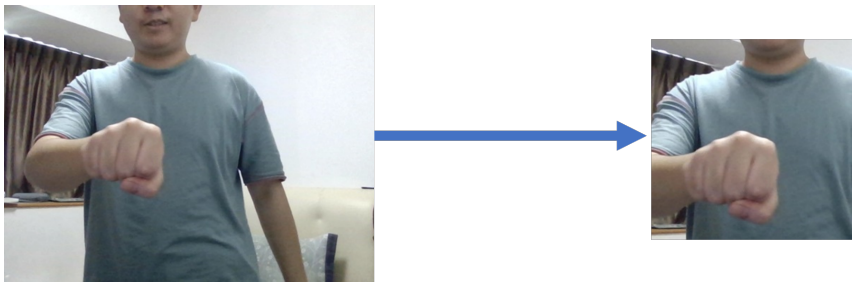


Figure 3.3: left before screenshot, right after screenshot

The whole process of obtaining key frames is as follows:

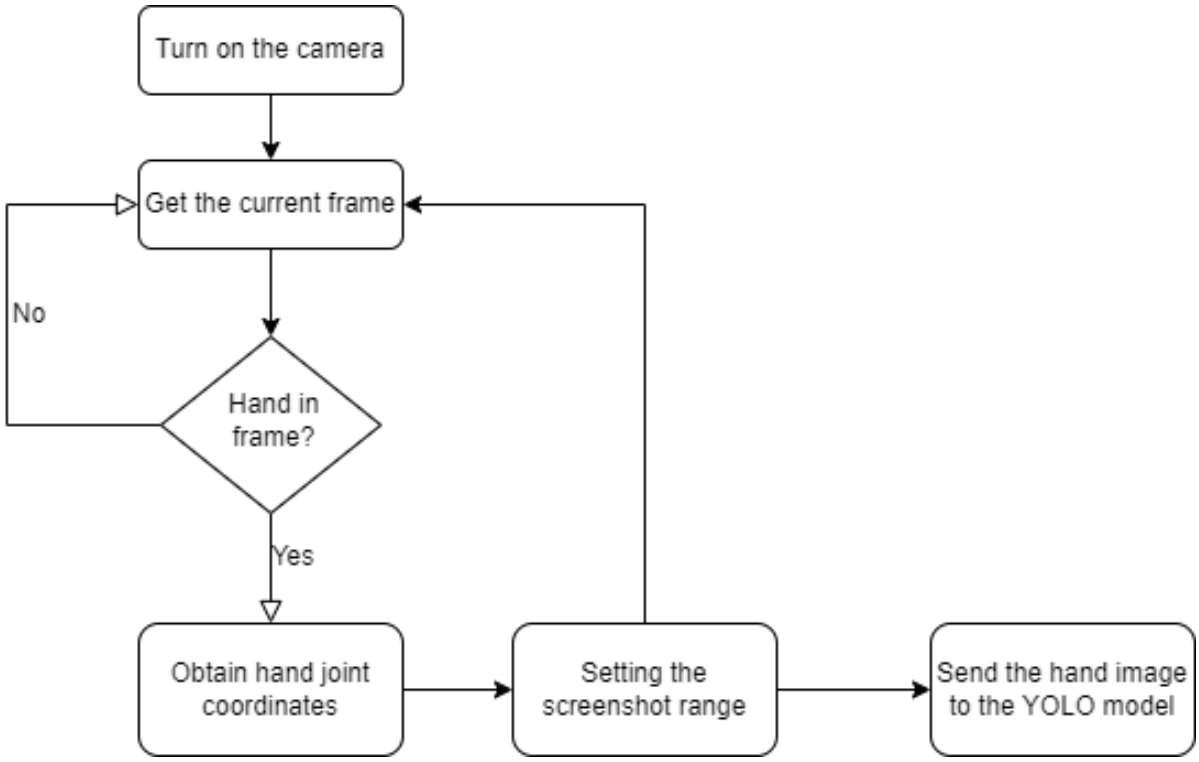


Figure 3.4: process of obtaining key frames

3.3 | Training:

The training procedure is the core part in our project, and the total training can be devided into several sub-components. Firstly, we will apply data pre-processing techniques to make sure the inpur data follows certain format (same size and channels). The data that are mislabeled will be filtered (There is no mislabeled data in our dataset according to the result). Secondly, the anchor data and classes data are read into memory. Thirdly, the hypermaters are determined (batch size, number of epoch, learning rate, optimization machine). Fourthly, the original initialed weights are replaced by pre-trained weights from COCO. Then start training, the loss are computed according to alogrithm. The losses are cyclely backpropgrated to update the weights. The final weights will be saved to Google Drive.

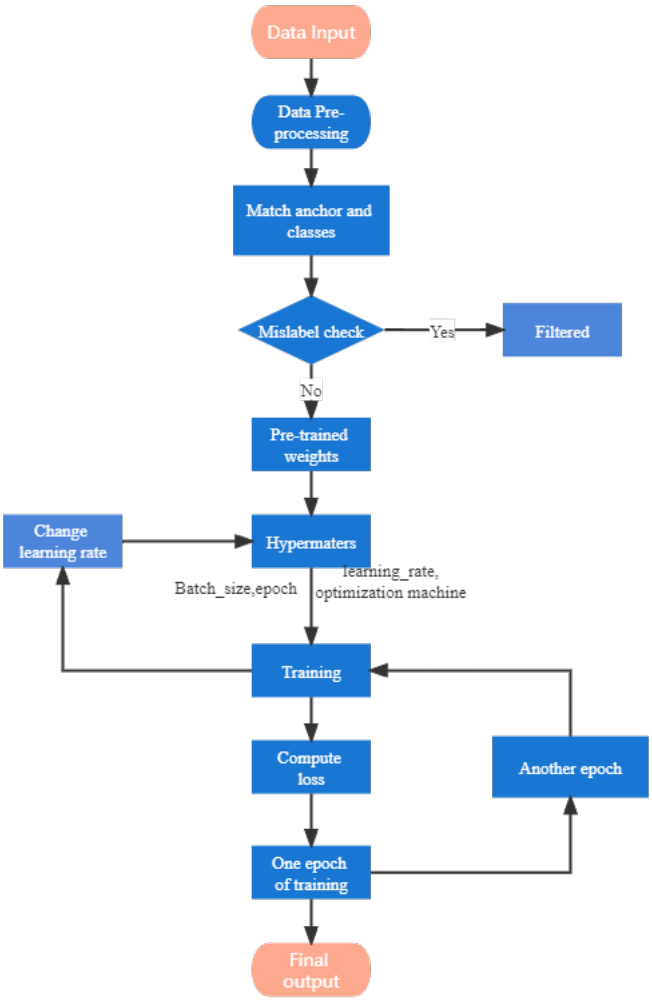


Figure 3.5: The flowchart of training procedure

3.4 | YOLO

The YOLO algorithm is designed to to extract features from the image to create a corresponding feature map that can be used to identify objects in a given image. As the feature map is populated with more information regarding the subtle nuances of each object, the network gains more confidence in predicting objects and their classes in targeted areas. YOLO can generate grids according to the size of the image and segment the whole image into several sub-areas. Each area contains a portion of the image that may or may not contain an object. Typically, an object will occupy more than one cell in the grid. In such a scenario, only the grid cell that hosts the centre of an object is said to contain that object. This grid cell is thus responsible for predicting the object. This concept is realized by assigning bounding boxes to objects that are present in the image. Each bounding box has a confidence score associated with it.
We initially applied a nerual network structure based on Darknet53. The layers are combined with convolution, batch-normization and residual block architectures.

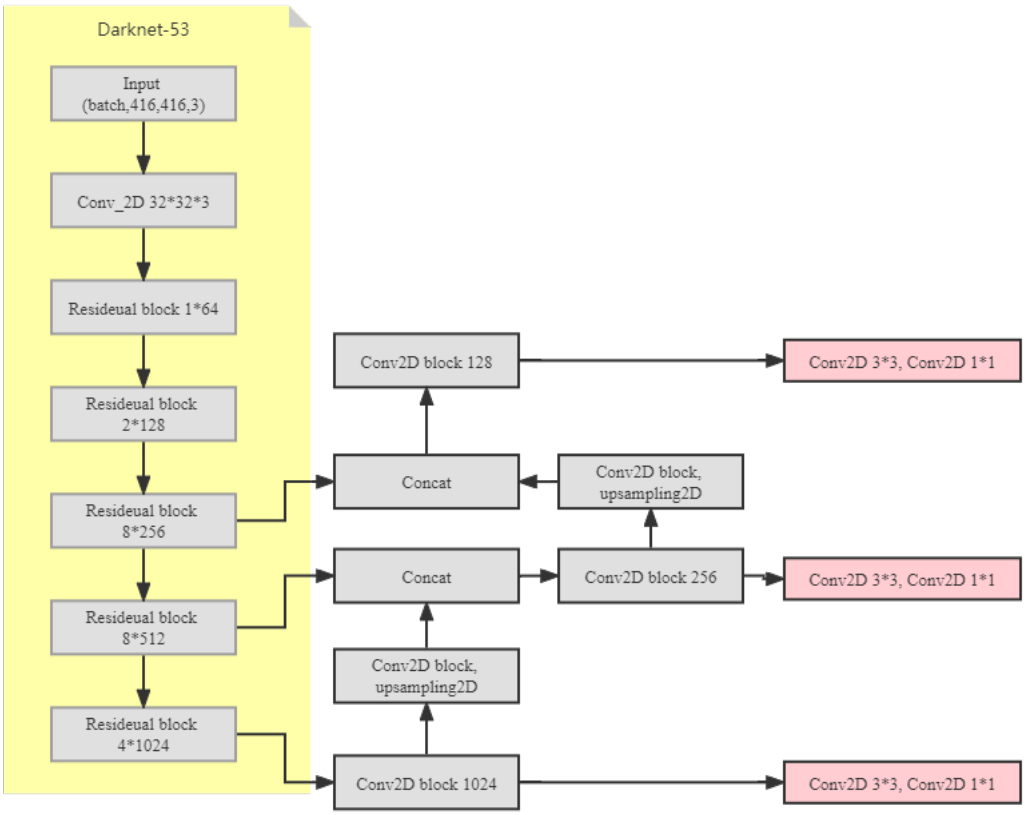


Figure 3.6: The structure of our model

3.5 | Subtitles Generate

In the module, we will implement the function to generate semantically sound sentences from the label obtained by the yolo model and add subtitles to the returned video.

The process flow of how to generate the sentences and add subtitles is like below:

1. Get returnlabel
2. Check if returnlabel is equal to preValue
3. If is check if the lable is repeated 10 times
4. If is append the label to text
5. Use Wordninja to participle the text
6. Use Spell correction to correct all words in text
7. Use PIL draw the text on the bottom of image
8. Show the image

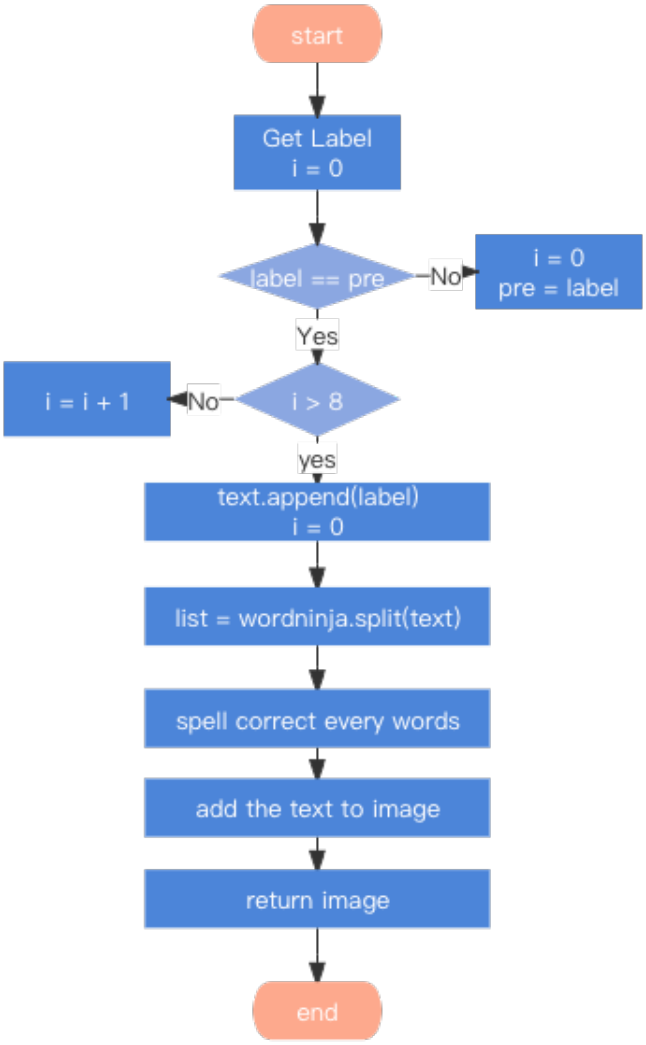


Figure 3.7: Subtitle Generate

4 | System Performance

4.1 | The performance on recognition

The test performance of this YOLO-fastest has been visualized. The overall performance is beyond 60%on precision, we understand that it is not a great score in computer vision. But we require our model to run well in users' personal devices which means our model must be lightweight and can reach a certain level of responding speed. That's why we cut down the model's complexity to improve the whole system's responding time.

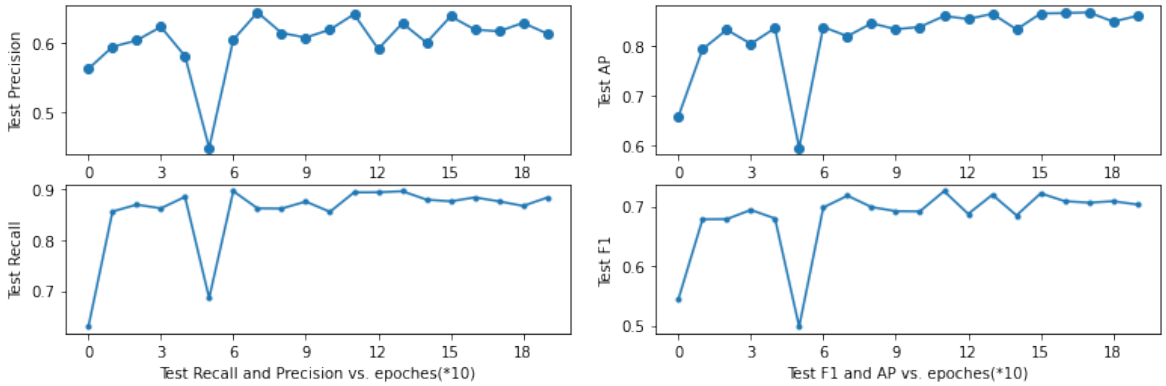


Figure 4.1: The original performance: precision, recall, average precision, F1 score

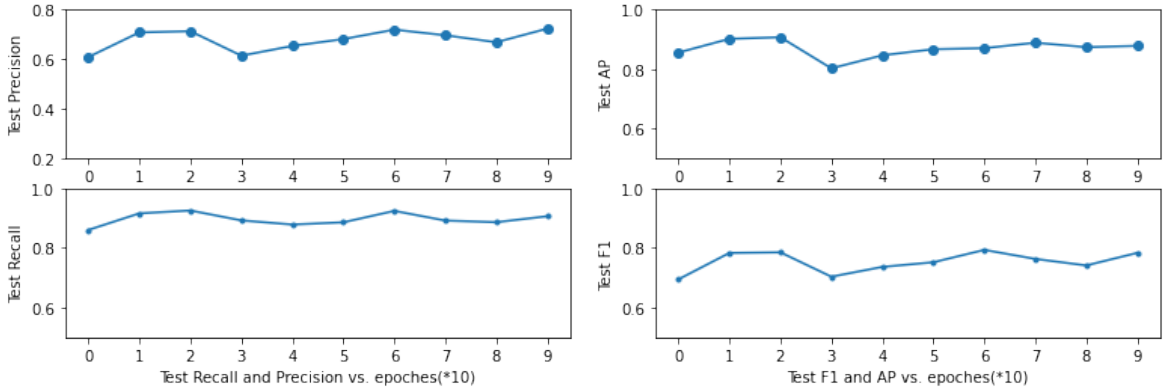


Figure 4.2: The model's performance on test dataset after fine tuning

We fine-tune the hypermaters, and we found the best statistically performance on testset when we set the training hyperparameters like:

- epochs–200
- steps–[150.0, 250.0]
- batch size–8
- anchors–[120.55, 97.22, 148.54, 151.12, 185.12, 214.38, 228.47, 147.84, 259.63, 210.48, 268.54, 291.81]

4.2 | The performance on recognition of each letter

The letters evaluated for accuracy were divided into three groups, those that were easy to classify, those that were similar, and those that were difficult to identify.

The first group is the letters could easily be identified and totally not similar to each other which include B,C,D,F,G,H,I,K,L,O,P,Q,R,U,V,W,X,Y.

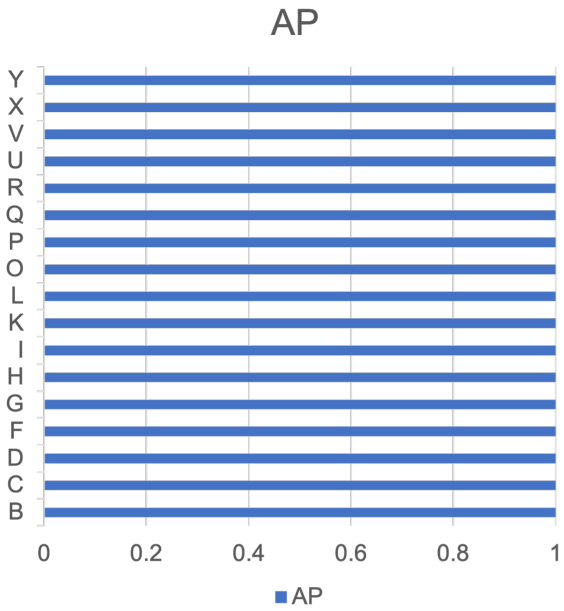


Figure 4.3: letters easy to identify and classify

The second group is the letters very similar to each other which include A,E,M,N,S,T.

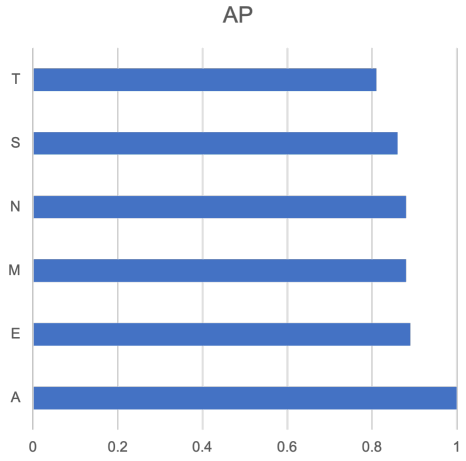


Figure 4.4: letters hard to classify

The third group is the letters very hard to identify which include J,Z.

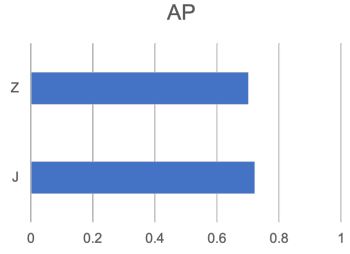


Figure 4.5: letters hard to identify

4.3 | The performance on responding speed

Another very important performance metric is the FPS at which the system runs, as the goal is to display the user's sign language translation in real time, we need to maximize the FPS of the video stream to ensure system availability.

When the system using CPU to handle the detect task, the FPS is shown to be around approximately 4-7 fps:

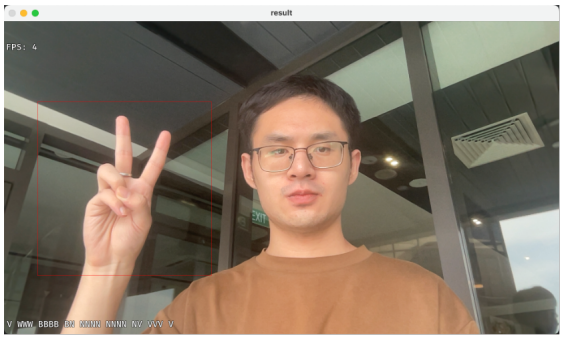


Figure 4.6: FPS powered by CPU

After we call Cuda for acceleration, the FPS is shown in the figure and it is around 10-15:

Figure 4.7: FPS powered by GPU

4.4 | Model lightweight processing

The original weights file is around 230MB, and can not be easily deployed in our system. The computing memory can be easily occupied and run the 'Out of memory error. Also, The FPS is magnificently influenced by the large model. To solve the problem, we decided to narrow down our model to 10% even 1% of the origin.

The solution is to reduce the number of neurons in each layer, thousands to hundreds, hundreds to dozens and we also use the average function to update the batch normalization layer instead of backpropagation to speed up the training. The final model is only 1MB. The parameters have been reduced from 61,581,727 to 242,146.

Of course, by doing this, the performance of correct recognition will be reduced. But it is a necessary trade-off. A powerful model which is too large to operate on any device is indeed trash in the industry. We believe the choice is worth making. Indeed, the response time is much shorter. And the memory issue has been alleviated.

Figure 4.8: The comparison of original model and model after lightweight processing

5 | Findings and discussions

5.1 | OpenCV compatibility issues:

OpenCV currently supports multiple platforms and multiple programming languages. But we still encountered some problems in the process of using it. We initially used macOS to program the project but installing OpenCV was complicated and there were error messages when using it. We ended up using multiple platforms, including codelab online, windows, and macOS.

5.2 | Only localized code:

5.2.1 | Issue:

At present, our project can only run-on PC. If users want to use the sign language recognition system, they need to install some code library on their computers. This can be difficult for users who are not computer science graduates, especially the deaf and mute.

5.2.2 | Future:

In the future, we want to deploy the code to cloud servers to provide users with online sign language recognition services through accessible web pages. Of course, we will also deploy to more platforms, such as iOS and Android operating systems, so that users can use sign language recognition system to communicate with deaf people at any time.

5.3 | The video FPS is not high:

5.3.1 | Issue:

Due to the computer performance and the lack of optimization of the current model, the video often appears the problem of slow and high delay in sign language recognition. Users sometimes had to wait several seconds for results, and the videos were not smooth, which severely affected the user experience.

5.3.2 | Future:

We'll optimize the system code to make it more efficient. We will try to use more efficient models or algorithms like yolov7. In the future, we will optimize the video frame rate to more than 30 frames and reduce the time to recognize sign language to less than 1 second.

5.4 | Restrictions on Sign Language Types:

5.4.1 | Issue:

The current sign language recognition system only supports American Sign Language. There are many kinds of sign languages in the world, and each one has many users who need help. Now many deaf people who use other sign languages cannot get help through the system.

5.4.2 | Future:

We will find or build more data sets to include images from other sign languages, such as Japanese sign language or German sign language. We plan to start with other popular sign languages, and in the future the sign language recognition system will support the vast majority of the world's sign language users.

5.5 | Irrationally Translation:

5.5.1 | Issue:

The Spellcheck only correct the origin word to the word with highest frequency and don't consider the context, so incorrect corrections can occur from time to time.

5.5.2 | Future:

We will adopt better NLP solution to improve the rationality of generated Translation