Canteen Queue Monitoring System

Liu Yuzhou(21100602d)

Responsable of the experiment or other information

1. Abstract

Queue monitoring is an essential task nowadays. By identifying the number of people in a queue, we can know the degree of congestion in a place. Then, people may use the result of detection to optimise the layout or rules in order to have a better plan to make the place work smoothly. In this report, we divide our task into two parts.
The first part is to identify the number of people who are queuing in an image. The second part is the **bonus** part, which hasn't been mentioned in the task specification. It also includes 2 parts, **part 1** is to identify the number of queues in an image (Maybe a place can have multiple machines or servers). **part 2** is to achieve tracking, to assign each person a unique ID.
Keywords-Queue monitoring, detection, tracking, computer vision

Figure 1. *Detect people who are queuing*

In this report, we will introduce:
(1)what general method our selected to train and test
(2)what and how the objects are detected, and how the features are extracted.
(3)how the model is trained, validated and tested, and how it performs in a real-world scenario; here we captured a video for testing
(4)what problems I have encountered and how I solve them.

2. Set up

2.1. Problem definition

Our task is to detect the people included in the service window and classify whether they are queuing or not. At the same time, we also try our best to detect the number of queues inside an image, because that will be useful for estimating **waiting time** (You will have chance to choose from a queue). To address the queue monitoring system as a computer vision task, we mainly have 2 ways: **segmentation** and **detection**. As there are a lot of methodologies, such as Faster-RCNN, Mask-RCNN and yolov8... which are state-of-the-art models we can apply to solve the problem. Feature extractor built based on the VGG network can also be helpful to enhance the performance.

In order to achieve count the number of queue2, we basically could have 2 different approaches: **(1)**: Use the segmentation method (**Mask-RCNN**) to segment different queues in the image. **(2)**: Count the number of machines or servers in the image. Though **(2)** may not be effective; as for the VA canteen, there could be multiple servers for one window, but as there are more servers, the serving time should be shorter? So I think it still makes sense.

By the way, we can also try to use a tracking method to assign IDs for different people who appeared in the video; this can help us to view the state of different people efficiently, here we choose to use **deep SORT** (Stands for Simple Online and real-time tracking with a Deep Association Metric) method to achieve tracking.

Computer vision should be suitable for this task. Because it is powerful for feature extraction and object detection, recent work has already proven that these models achieved state-of-the-art results. We can regard our task as detecting the state of people standing near the service window. I believe we could achieve a decent result on this task by combining these methods.

2.2. Dataset

2.2.1. Method for labelling.

We choose to use **Roboflow** to label the dataset. Labelling for the dataset is really expensive because we cannot give a single label to an image. Instead, we need to label each object by drawing bounding boxes. And we have built **three different datasets**.

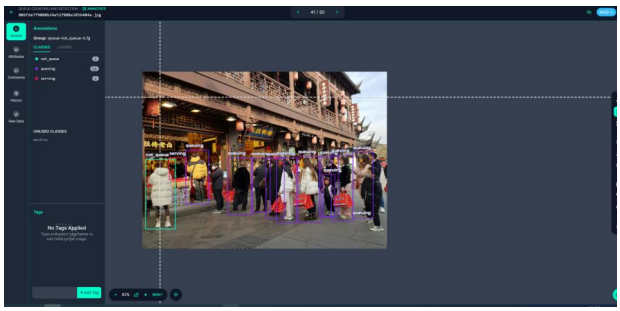

Figure 2. *Manually label the dataset*

Thoughts on labelling:

We have encountered some confusion when drawing bounding boxes. What if two people have some overlapping with each other? Then we considered: Maybe we could change our task to labelling people's heads? As it reduces the bounding box area, which could also reduce the area of overlapping. Finally, we decided to directly include the whole body of the person, no matter how much it overlapped with others, because we thought that might include some context. For people standing in a queue, the overlapping is more likely to happen. The following **Figure 3** shows the part of our dataset.

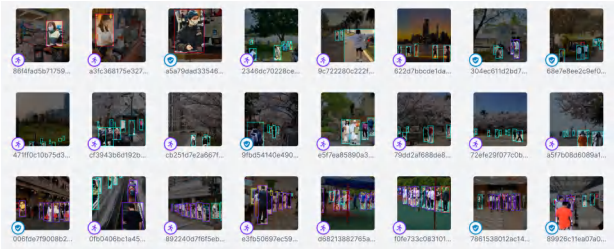

Figure 3. *Part of dataset(Classification&counting)*

2.2.2. Classification&counting dataset.

This dataset was created for the classification task to detect people's states: Queuing or not (**Later, for better time estimation purposes, also try to classify the serving people**).

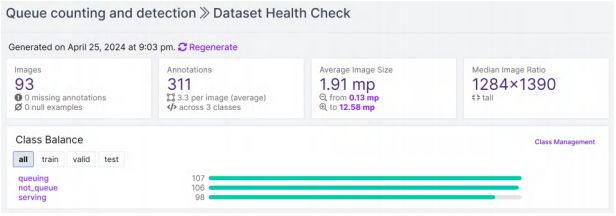

Figure 4. *Counting and classification dataset*

Three different classes:
(1)not_queue: refers to people not in the queue.
(2)serving: The staff who is serving in front of the queue.
(3)queuing: the people who are queuing.

2.2.3. Machine detection dataset.

This dataset is for machine detection, which helps train the model to identify machines that could help people with payment.

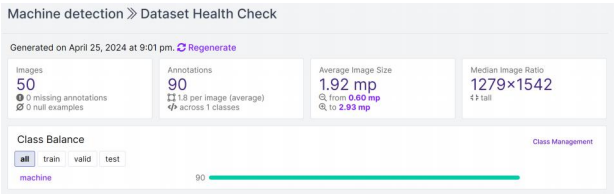

Figure 5. *Machine detection dataset*

Single class:
(1)machine: The machine which is used to serve the people.

2.2.4. Queue counting dataset.

This dataset is a trial on **segmentation method**. To label a dataset in order to perform segmentation is pretty troublesome, so we only labelled four images. Because the performance is kind of poor so we give up on this method (Explained in **section 4**).

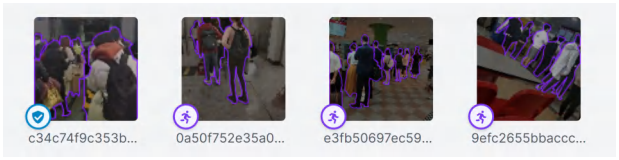

Figure 6. *Queue counting dataset*

Single class:
(1)queue: The group of people stand in the same queue.

3. Queue counting Methodology

3.1. Overall

In this part, we mainly attempt **YOLOv8**, **Faster-RCNN** to do the detection job **(**Basically two ways, **way 1** is to use the detection model also to do classification, **way 2** is to use a feature extractor to predict the bounding box detected by detector**)**. After these, we will try to use **deep-sort** method to achieve tracking based on the yolov8 model and, at the same time, use the feature extractor to detect the state of people (queuing, not queuing or serving).

During the project, we have **2 stages**. In the **first stage**, we only try to detect the queuing and not queue people in the image. In the **second stage**, in order to have a more reasonable way for **waiting time estimation**, we take into consideration the servers and machines that appear in the image. So **Section 3** will all talk about **stage one**, while **Section 4** is **about stage two**.

3.2. Faster-RCNN

3.2.1. Model explanation.

We start with the **Faster-RCNN** model, as it is a state-of-the-art model for object detection nowadays.

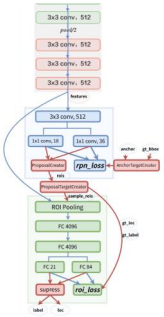

Figure 7. *Structure of Faster-RCNN*

Figure 7 shows the structure of this model. The general working process of this model is basically three stages: **First** is to do feature extraction by some architecture pretty similar to VGG-net. **Second** is to select key points (binary classify the anchor is important or not) and generate proposals of objects. **Finally**, do the detection and classification.

Here, my training method is to load the pre-trained

Faster-RCNN model and then tune it on our dataset. Hopefully, the model can efficiently classify different people.

Before jumping to training, I first choose an image from the validation set to see how the model performs on this image; later, we can compare it with the training result. The result shown in **Figure 8**, as we can see, the majority of the people are detected unless they are too far from the camera. But it is fine, as the people who stay too far are more likely out of our interest.

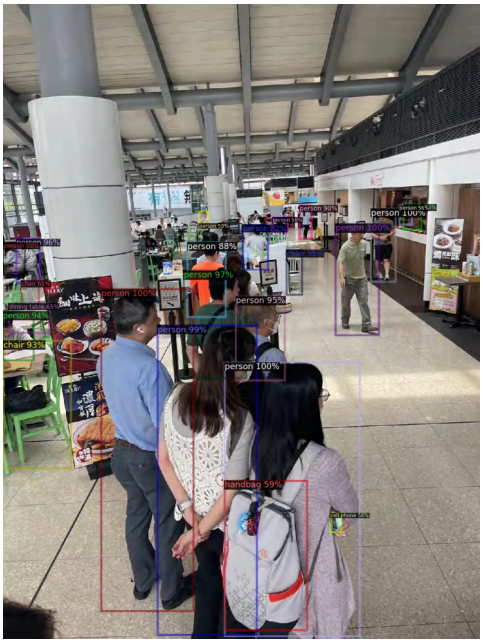

Figure 8. *Performance of Faster-RCNN (Before training)*

3.2.2. Illustration on performance.

Then, we directly tune the model based on the dataset we have labelled. The training takes a pretty long time, as the model is so complex. We can see its performance from **Figure 9**. We can see that the model adjusts the learning rate while training, but it seems it has stepped into a bad position, and the performance is quite poor.

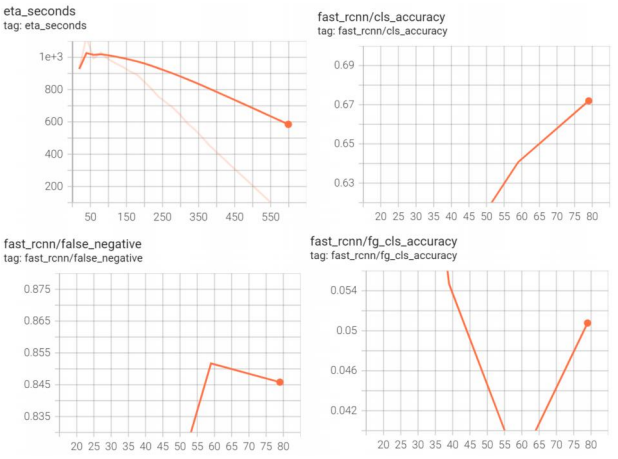

Figure 9. *Performance of Faster-RCNN (Initial learning rate: 0.00025)*

I think that might be because the learning rate was too low initially, which made our model converge slowly. After searching, I found that the model uses a scheduler to combine the warm-up and step-wise learning rate decay. So, I think setting a larger learning rate might be helpful in increasing accuracy (Maybe for warm-up, we need a larger step). So, I changed the learning rate to 0.0025, and then I found the performance of the model boosted. The improved performance shows in **Figure 10**.

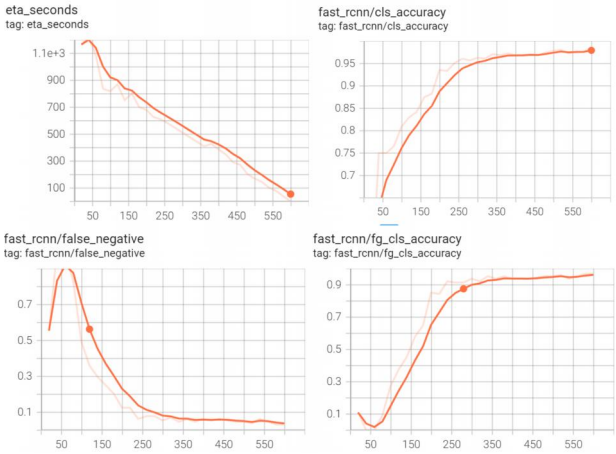

Figure 10. *Performance of Faster-RCNN (Initial learning rate: 0.0025)*

We can see the detection result is quite decent as it shows in **Figure 11**. We can see it accurately detects the people who are standing in the queue.

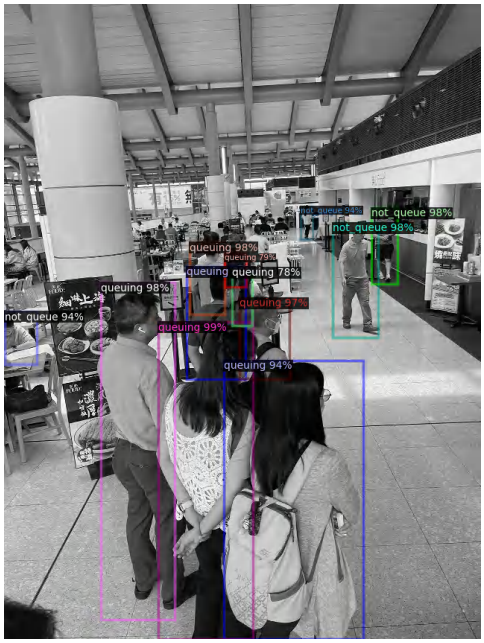

Figure 11. *Detection of people queuing or not)*

3.2.3. Difficulties&drawbacks.

1. (**Long time for training**) The training speed is too slow for this model as it includes a lot of layers, so to train 600 epochs takes around 30 minutes. (On T4 GPU)
2. (**hard to adjust learning rate**) The learning rate makes a huge difference in training. When the learning rate is too low, the performance is poor. I cannot figure out the best learning rate for the model (Though it performs quite well now). Maybe just start at a higher learning rate, as it will decay in order to fit the dataset.

3. (**Worsen detection result**) We can easily see that the detection performance has worsened a little bit. As we can see originally, the model can detect the part of people sitting far away from the camera. However, after tuning our dataset, the number of people it can detect is deducted somehow.

3.2.4. Reflection.

The labelling method we are using seems to be useful, as it includes the **context** surrounding the people, so we should not only label people's head, instead, label the whole person, Which could be helpful in detection, I guess that's the reason why the performance of this method is quite decent. But we still need to try to find some way to be more efficient for training.

3.3. Faster-RCNN with feature extractor(VGG-net)

3.3.1. Model explanation.

As I mentioned in **Section 3.2**, there are three problems: the training speed is too low for such a large model. And it is hard to find the best parameter for such a model. Also, the tunning on our own dataset worsens the performance. Then, what about not touching the original Faster-RCNN model?

So, in this section, we tried to crop those labelled bounding boxes from the image (As it shows in **Figure 12**). Then, an extractor is trained based on those cropped images in order to classify the states of different peoples. Then, during the model utilisation stage, **detection** and **classification** are completed separately. The Faster-RCNN model will first detect the bounding boxes, and then we will crop and reshape the bounding box and send it to the feature extractor to do the classification.

Figure 12. *Cropped bounding boxes*

Train feature extractor:

First, I am going to introduce how my extractor is built. As shown in **Figure 7**, the feature extraction layer of Faster-RCNN is built based on a structure pretty similar to the structure of **VGG-net**. Also, as VGG-net is trained on a large image dataset and is well known for its ability to do object classification, I believe it is most suitable here.

I first take the feature extraction layer of **VGG-net**, and then connect it with three **fully connected linear layers**. Instead of directly using the output of the VGG-net to make the prediction, I chose to take the output of the second linear layer first. As shown in **Figure 13** (Hopefully, it can be regarded as a strong representation of the image) and then feed it to some traditional machine learning models to make the prediction.

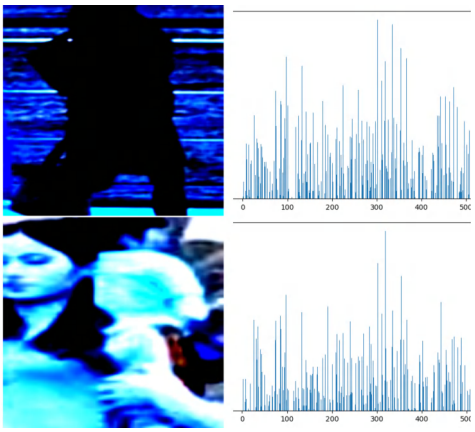

Figure 13. *Feats extracted by VGG intermediate linear layer*

Following **Table 1** shows the classification result of the feature extractor, and we can see that **SVM** model perform best, so we are going to test it on our dataset.

Model/Classifier	Accuracy
VGG + SVM	86.36%
VGG + (Random Forest)RFC	81.82%
VGG + (KNeighborsClassifier)KNN	77.27%

Table 1. *Performance for feature extractor*

3.3.2. Illustration on performance.

It is hard to evaluate whether the model performs well or not based on accuracy metrics, so I just compare the performance of both models on the part of the validation set. As we can see from **Figure 14** and **Figure 15**, both model makes some mistakes, but the accuracy for training on Faster-CNN directly is generally better than classifying each bounding box separately by using the feature extractor. But I think it is acceptable because the training time for the feature extractor is much shorter compared to tunning the Faster-RCNN model directly.

Figure 14. *Result for Faster-RCNN*

Figure 15. *Result for Faster-RCNN+feature extractor*

3.3.3. Difficulties&trials on improvement.

1. (**Accuracy not enough**) Although training such a model is much cheaper, the accuracy is not as high as directly tuning the Faster-RCNN model.
2. (**Good extend ability**) The feature extractor can be extended to any of the detection models, as we could always crop the bounding boxes and feed it into the feature extractor, and then we can predict people's state (So later on we extend it to the yolov8 model, mentioned in **section 3.4**).
3. (**Mis-classification of the server**) We can see that both models have some problems recognising the servers that serve in front of the queue. So later, we decided to add a class separately called **server** in order to have better classification results.
4. (**Data augumentation**) Because all the images are labelled by ourselves, it must be limited, so we decided to use the rotation technic to enlarge the dataset. That means for images in the training set, we **rotate them 15 degrees** to obtain a new image in order to obtain a new image and add it to the training set. This is helpful because when we are taking images, we are more likely to take them from different angles.
5. (**Code changing of detectron2**) As the Faster-RCNN model doesn't allow us to modify the predicted output directly, I had to modify the code in order to modify the output object.

3.4. Yolov8(Bonus: Tacking)

3.4.1. Model explanation.

As **Yolov8** is also a state-of-art model, we can see its structure from **Figure 16**. It can generate **three types** of representations for objects: middle-sized, large-sized, and small-sized. Which could be helpful for detecting objects of different sizes.

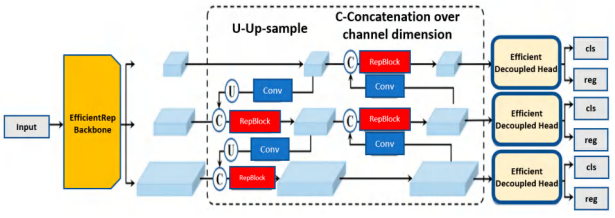

Figure 16. *Structure of Yolov8*

Tracking is about assigning an ID to people who are in the queue. And keep track of the people by using that ID. To achieve this, we choose to use **deep sort** combined with the yolov8 model. The following is the workflow. We can see that the idea is somehow similar to the idea of **feature extractor** we built in **section 3.3**; it is about "memorising" the object cropped and then doing a query among the object that appears during a period of time. If it is similar to a previous object, they will be assigned the same ID.

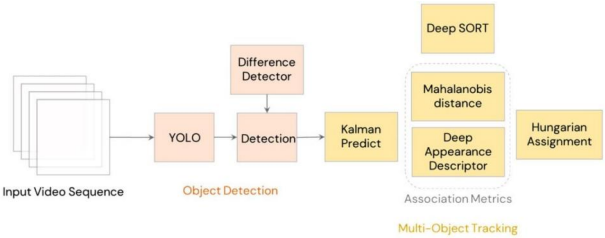

Figure 17. *Architecture for deep-sort*

3.4.2. Illustration on performance.

Though the deep sort has already improved a lot compared to the original SORT(Stands for Simple Online and real-time tracking) method, especially for ID switching problems, but we can see it still occurs. Below **Figure 18** shows a screenshot obtained by the tracking model.

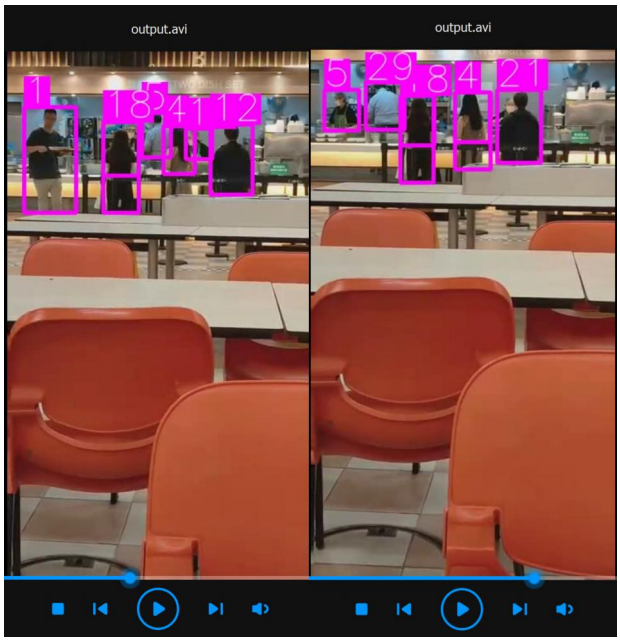

Figure 18. *Tracking result*

4. Waiting time estimation

Note: Result for this section will be demonstrated in section 5

4.1. Overall

I think estimating time-based on the number of queuing people is not reasonable. In an image, there might be **multiple queues**, so you will have a chance to choose one queue and join in. And the number of people serving is also

related to the waiting time. So, our group determined to estimate the time based on the number of servers and the number of machines (Two types of queues).

During this section, I tried a few methods to identify the number of queues in an image, and some of them failed, some of them succeeded. I will show 2 types of time: first is the original time based on the number of people, and second is after it divides the number of queues. Basically, we assume that the time for serving one people is 5 seconds.

4.2. Mask-RCNN

4.2.1. Model explanation.

As Mask-RCNN is a state-of-art model for segmentation. The working flow of this model is shown below in **Figure 19**. It does segmentation and object detection together.

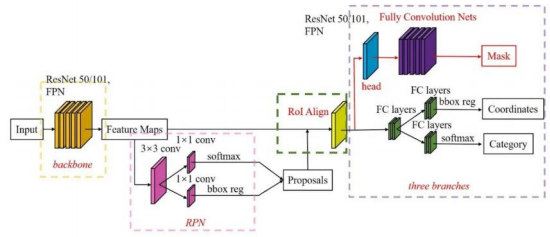

Figure 19. *Structure of Mask-RCNN*

I tried to use it to identify queues that appeared in an image directly. As shown in **Figure 20**.

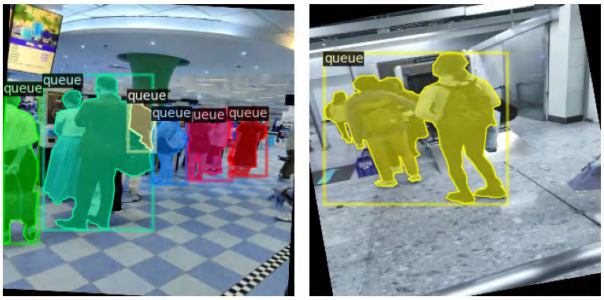

Figure 20. *Segmentation on queue dataset*

4.2.2. Illustration on performance.

But this method fails, as the model cannot segment queues even on the training set. I guess there are 2 potential reasons that could lead to such a result:

1. **Reason 1**: Segmentation is based on the similarity of pixels, so such kind of labelling is meaningless, because there could be adjacent people but not standing in the same queue (Then how is the model going to differentiate them?).
2. **Reason 2**: The training set is too small; as such, a model needs to feed a huge amount of images for training in order to achieve a good result.

4.3. Estimate by number of machine and servers

4.3.1. Method explanation.

In this section we are not going to introduce any new model to do the task. Instead, we just add a class called server to the original dataset. Also, as the original Faster-RCNN doesn't have a class to represent the machine which could help people to make payments, we built our own dataset to try to make the model able to detect the machine (**Machine detection dataset**).

4.3.2. Illustration on performance.

Part 1: Sever detection

Here, I tried both methods:

-**Method 1**: train directly use Faster-RCNN

(Mention in **Section 3.2**)

As we can see from **Figure 21**, it seems that the accuracy is quite high.

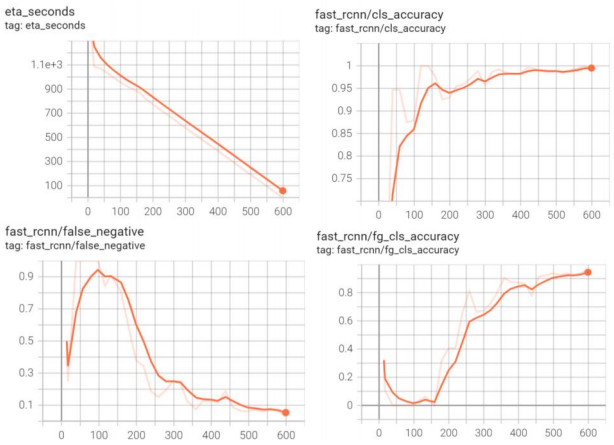

Figure 21. *Train result for Faster-RCNN*

However, after taking a look at the classification result, I found the detection performance has been destroyed. Perhaps that is why the classification accuracy increased.

Figure 22. *Detection and classification example (Faster-RCNN)*

Then I wanted to find the reason why it is worse. So, I take a look at the box loss. Then I found from **Figure 23** that the box loss kept varying and could not converge, which is why it failed.

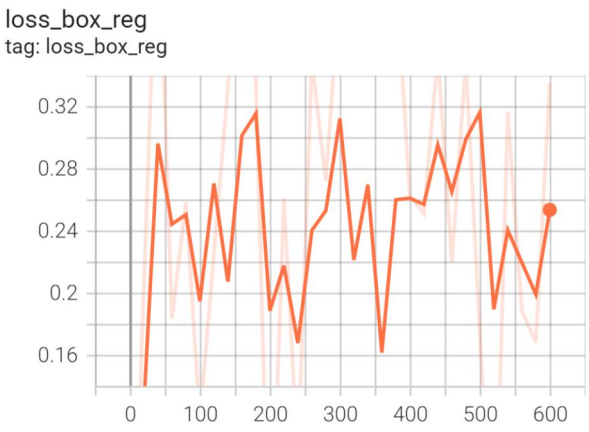

Figure 23. *Structure of Mask-RCNN*

-Method 2:Doing classification using feature extractor (Mention in **Section 3.3**)
As we increased the amount of dataset and added the **server** class, the feature extractor didn't perform that well, but it is much better than random guessing. From **Table 2**, we can see that the VGG+SVM model is the best model we can obtain here.

Model/Classifier	Accuracy
VGG + SVM	55.68%
VGG + (Random Forest)RFC	50.00%
VGG + (KNeighborsClassifier)KNN	51.13%

Table 2. *Performance for feature extractor*

Part 2: Machine detection

But unfortunately, the machine detection failed. As we can see from **Figure 24**, the accuracy is stuck at some point and didn't improve. I think the limitation of the dataset should cause the problem (Size is too small), as the Faster-RCNN is trained based on a large amount of data.

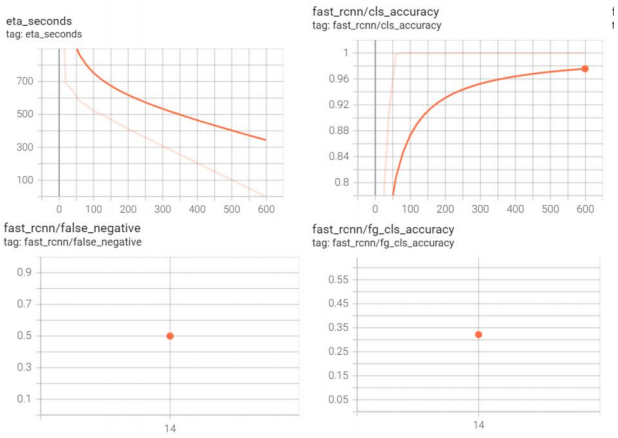

Figure 24. *Training result for machine dataset*

We can see from **Figure 25** that the box loss does not change; it keeps at 0, which means there is no bounding box identified.

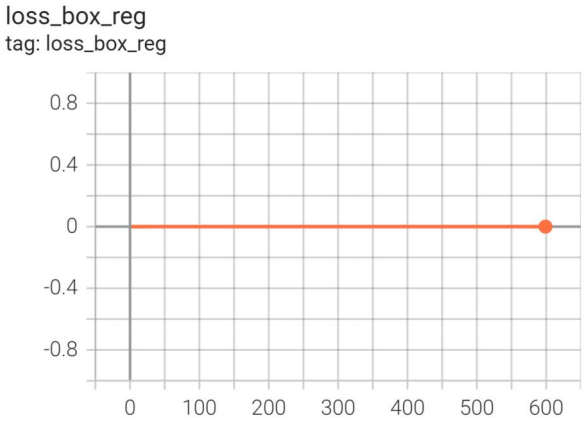

Figure 25. *Box loss for machine set*

4.3.3. Difficulties&trials on improvement.

1. (**Fail for identify machine**) **Potential reason 1**:I think the limitation of the dataset causes the failure in machine identification, the original model doesn't include such a class. The parameter of the original model is used to identify other classes; maybe we need to label more classes and train them together with the labelled machine, then our model may able to find some way to learn some knowledge of machine class based on other classes. But that is infeasible, as we cannot manually label that much data.
2. (**Worsen detection result when adding server**) The potential problem could be similar to what was mentioned above, so a feature extractor could be suitable in such a case.
3. (**Hard to train the extractor**) As we have added too much data to the dataset, it is hard to train the VGG-net. The GPU is easily memory out, so sometimes I need to move things to CPU in order to train the SVM, RFC...
4. (**Not satisfied performance**) The performance is not that good when doing server identification for the VGG feature extractor. Maybe the new class server is not that distinguishable from the queue or not_queue class, which makes our model confusing, so the performance gets worse.

4.3.4. Reflection.

The methods I could come up with to improve the result are too expensive to implement. The large model is hard to train and takes a lot of time; maybe further improvement could be focusing on improving the feature extractor to make it work better.

5. Test (Include waiting time estimate result)

Note: Any concern for the testing please refer to the video folder

5.1. Overall

So, in this section, we mainly take some screenshots of the test videos. And illustrate what those elements shown in the image exactly mean.

The test video is recorded in the VA canteen, and we have a folder to store all of them as is shown in **Figure 26**. **Please read the Readme file before watching the testing videos**.

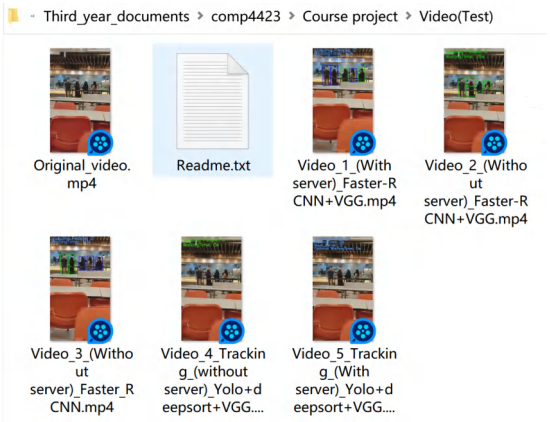

Figure 26. *Test videos*

5.2. Video illustration

Here, I just use **video 5** as an example to illustrate the testing result. You can go to the video folder to view more about the test results of the different models introduced in this report. The following **Figure 27** shows the screenshot from **video 5**. You can see that the words displayed on left up corner of the video. **Queuing people** means the number of people detected in the queue. **Waiting time** equals to the **Queuing people** multiply 5 (seconds). **Number of servers** means the number of serving people detected in the serving counter. **Optimal waiting time** is the waiting time divided by **num servers** (seconds).
The colour above the bounding box (For non-tracking video, just the direct colour of the bounding box) shows the state of people, green means queuing, blue means not queuing, red means serving.
The Number shown on top of the bounding box represents the tracking ID for the person.

Figure 27. *Screenshot of video 5*

5.3. Reflection for testing

Among all the models, I can see that the model trained on Faster-RCNN could be the most stable and accurate one, but it still makes some funny mistakes as is shown in **Figure 28**. It sometimes misclassified the chair as people who are not_queue. Although it doesn't influence our result, we still need to improve it in the feature (Influence of pre-trained parameters), as it only includes people in our training set, so it should not detect chairs in this case.

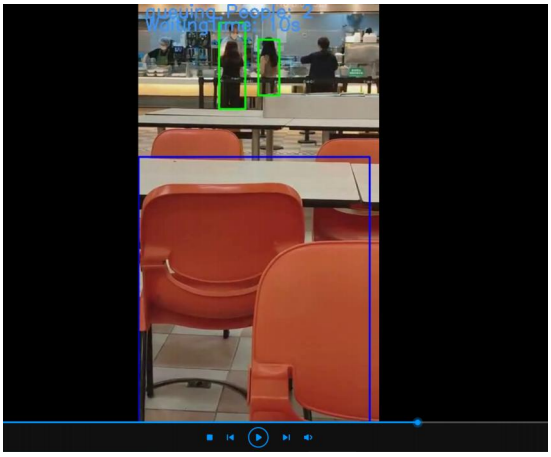

Figure 28. *Mistake of faster-RCNN*

6. General Limitation

The performance of the models in this project is still limited, although the Mask-RCNN have achieved high accuracy in distinguishing whether people is queuing or not, but the performance for identifying servers in queues still needs to be improved. And the model cannot work if different kinds of queues appear in the same image, as to count all the people together is meaningless in that case. The execution time of the models here could also be a problem, as for real-time detection and classification, they should execute faster (May handled by using powerful device).

7. SUMMARY

In this project, we have tried a lot of models. They are **Faster-RCNN, Yolov8, Mask-RCNN, VGG, deep-SORT**. By training based on Faster-RCNN directly, we can obtain the best result for **queuing people counting**, but the training time is the longest among all models. For the estimation of waiting time, we take into consideration the number of servers. Although some of the methods failed during this procedure, I definitely learned a lot from this progress.

8. My Contribution and Role Distinction

In this project, I worked as the leader of the team.
For the dataset part, I helped to take the test video, label a partial of the dataset, and help balance it.
For the model part, I mainly worked on the Faster-RCNN model; after finding the drawback of the original Faster-RCNN, I attempted to use VGG-net to extract features to improve its performance. I also tried to implement the tracking based on the SORT method, but unfortunately, it failed, so later on we implemented it based on deep-SORT together.
Finally, for waiting time estimation, I first tried to use the Mask-RCNN method to identify the queues. After it failed, I tried to identify the servers and machines in the image, but unfortunately, the machine identification failed.
Overall, I think I have learned a lot in this project, and it is nice to collaborate with my group mates.

References

1. Roboflow:
   https://app.roboflow.com/
2. Faster-RCNN:
   https://github.com/chenyuntc/simple-faster-rcnn-pytorch
3. Detectron 2:
   https://github.com/facebookresearch/detectron2
4. Deep sort:
   https://ikomia.ai/blog/deep-sort-object-tracking-guide
5. Mask-RCNN:
   https://zhuanlan.zhihu.com/p/62492064?ref=blog.roboflow.com