COMP 4211 Project 1: Face Mask Detection

Wenting Yang, 20583242, wyangaz@connect.ust.hk
Yizhe ZHAO, 20582834, yzhaocj@connect.ust.hk

1. Dataset

1.1 Face mask dataset on Kaggle

We use the given Face mask dataset on Kaggle. This dataset contains 853 images belonging to the 3 classes, as well as their bounding boxes in the PASCAL VOC format. The classes are: 1) With mask; 2) Without mask; 3) Mask worn incorrectly.

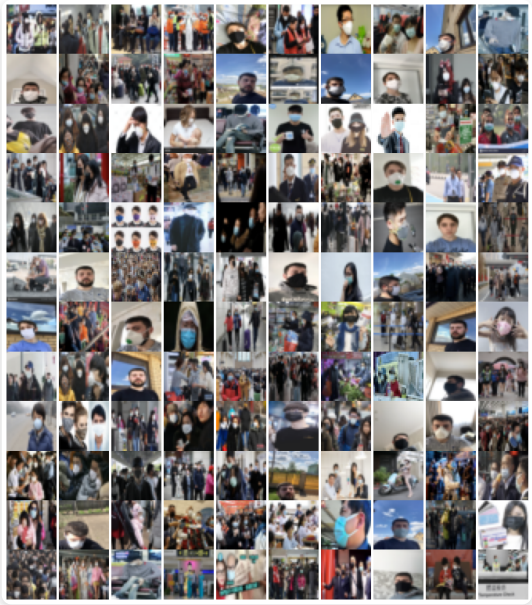


figure 1. Examples of the face mask dataset

1.2 Clipped image dataset

To do the classification task, we clip the face images from the original dataset using given annotation boxes and we finally got 4072 clipped images with three classes: 1) With mask; 2) Without mask; 3) Mask worn incorrectly.

figure 2. Examples of clipped image dataset

1.3 Image preprocessing

For the face detection + classification task, we have three steps to do image preprocessing. The image processing of object detection with YOLO will be introduced later.

1. Save all the data into dataframe or list format for the sake of convenience

We refer to the notebook "Pytorch FasterRCNN by Daniel" in Kaggle to write 3 functions `generate_box()`, `generate_label()`, `generate_target()` to read data from `.xml` files. We convert important data into the format of dataframe or list and save them into local `.csv` files. There are 852 images in total. Examples of images with labels from the original dataset are shown below.

figure 3. Example 1 of images with labels from the original dataset

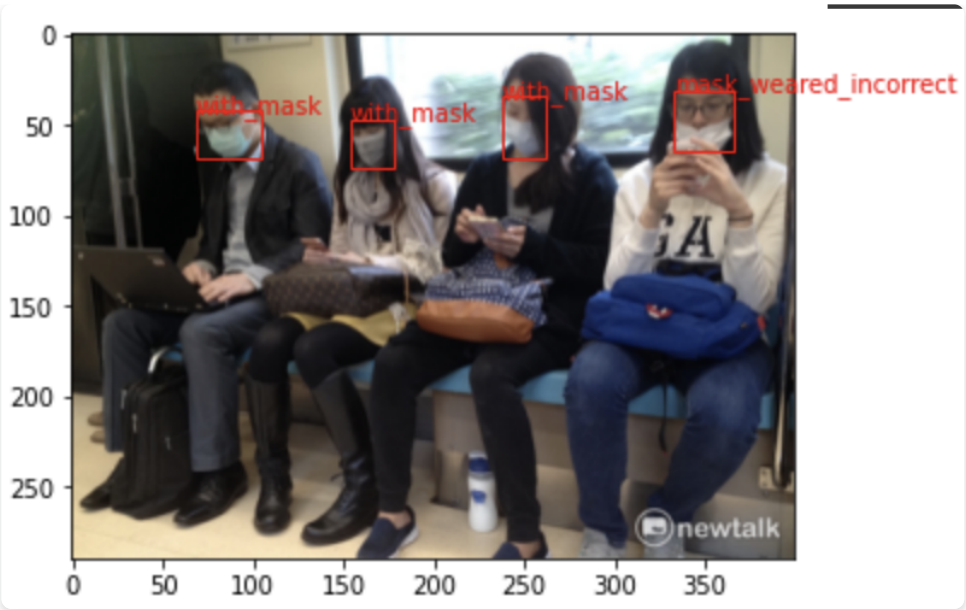


figure 4. Example 2 of images with labels from the original dataset

2. Clip the image

To do the classification task, we clip the face images from the original images using given annotation boxes and we finally got 4072 clipped images

3. Build Face dataset and dataloader

We try two methods to set up Face dataset and two methods give large difference results. We will show the different parts in experiment section.

- Use `train_test_split()`

  We build the corresponding dataset and dataloader for the clipped images Dataset. We use `train_test_split()` function to derive the train set and valid set. Here is the parameters:

  ```
  train, valid = train_test_split(df_dataset, test_size=0.2, random_state=4211)
  ```

  In this method, the clip images is shuffled and there can exist clipped images in same pictures. Therefore, the similarity of training and valid sets can be high.

- Split the data directly

  We just assign the clipped images in first 177 pictures of original Kaggle dataset (first 814 clipped images 20% of the dataset) to validation set and the remainings are for training set.

  ```
  train, valid = df_dataset.iloc[814:], df_dataset.iloc[0:814]
  ```

After we split the dataset, We transform clipped images into tensors and size 32x32. For the baseline model, we only use grayscale and change to RGB channel in later improvement.

```
transf = transforms.Compose([transforms.Resize((32,32)),
transforms.ToTensor(),transforms.Grayscale(num_output_channels=1)])
```

The batch size for training set is 64 and 1 for validation set.

2. Task

The main machine learning task we performed on this dataset is to detect mask, no mask and incorrectly worn masks.

To be specific, we take images as inputs, and output the bounding boxes together with the classes they belong to. The three classes are 1) with mask; 2) without mask; 3) mask worn incorrectly.

We mainly use F score as the performance metrics, and mAP(@0.5, @.5 : .95) as the auxiliary metrics.

The F score is calculated as follows:

- Precision: $P = TP/(TP + FP)$
- Recall: $R = TP/(TP + FN)$
- F score: $F1 = (2 * P * R)/(P + R)$

Here TP is the number of correct objects given, FP is the number of incorrect objects given, FN is the number of missed objects. Correct objects means 1) predicted bounding box has $\text{IoU} > 0.5$ with ground truth; 2) predicted class of the object is the same as the true class.

3. Method

We approach this task through two different paths, Face Detection + Classification and YOLOv5 object detection model.

3.1 Face Detection + Classification

3.1.1 Face Detection

In this task, we use a pretrained Retinaface model to do the Face Detection part. Retinaface is a robust single stage face detection model first proposed in 2019 in the paper “RetinaFace: Single-stage Dense Face Localization in the Wild”. The model we use is:

```
model = get_model("resnet50_2020-07-20", max_size=2048)
```

3.1.2 Classification

First, we train our classification model using the clipped image dataset mentioned above. The models include simple CNN model(our baseline), vgg16, resnet18 and we save the best pretrained model in terms of validation accuracy. Then, we use these pretrained classification model to classify the images from Retinaface model's prediction.

3.2 YOLOv5 object detection model

In this path, we train YOLOv5 on our dataset and analyze performance metrics. Different methods such as hyperparameter tuning, using deeper model, data augmentation, model ensembling and test time augmentation are used to improve the model performance.

4. Experiment

4.1 Face Detection + Classification

4.1.1 Classification

We have a baseline model which is a simple CNN model + FC layer. And we try three methods to improve the baseline performance: Optimization setting, Different Architecture and Data augmentation.

Model name	Setting	Training loss	Validation loss	Validation Accuracy
Baseline	optim.Adam(model.parameters())	0.6352	0.6618	0.8882
Baseline Optimization1	optim.Adam(model.parameters(),lr = 0.001,weight_decay=0.0001); num_epochs = 25; bz =128	0.6257	0.6524	0.8993
Baseline Optimization1	optim.Adam(model.parameters(),lr = 0.002,weight_decay=0.0001); num_epochs = 30; bz =128	0.6240	0.6463	0.9017
VGG16	torchvision.models.vgg16(pretrained=True)	0.2446	0.7708	0.8796
VGG16 Fine-tune	torchvision.models.vgg16(pretrained=True)	0.1067	0.1225	0.9644
Resnet18	torchvision.models.resnet18(pretrained=True)	0.4110	0.5011	0.8391
Resnet18 Fine-tune	torchvision.models.resnet18(pretrained=True)	0.0397	0.1265	0.9693
Baseline Augment1	transforms.RandomHorizontalFlip(p=0.9)	0.6063	0.6099	0.9398
Baseline Augment2	transforms.RandomHorizontalFlip(),transforms.RandomRotation(degrees=(10, 90)	0.6093	0.6067	0.9435
Baseline Augment3	transforms.RandomHorizontalFlip(),transforms.ColorJitter(brightness=.5, contrast=.5, saturation=.5, hue=.5)	0.6453	0.6391	0.9140

Other common setting:

Loss function -- nn.CrossEntropyLoss()

num_epochs -- 20

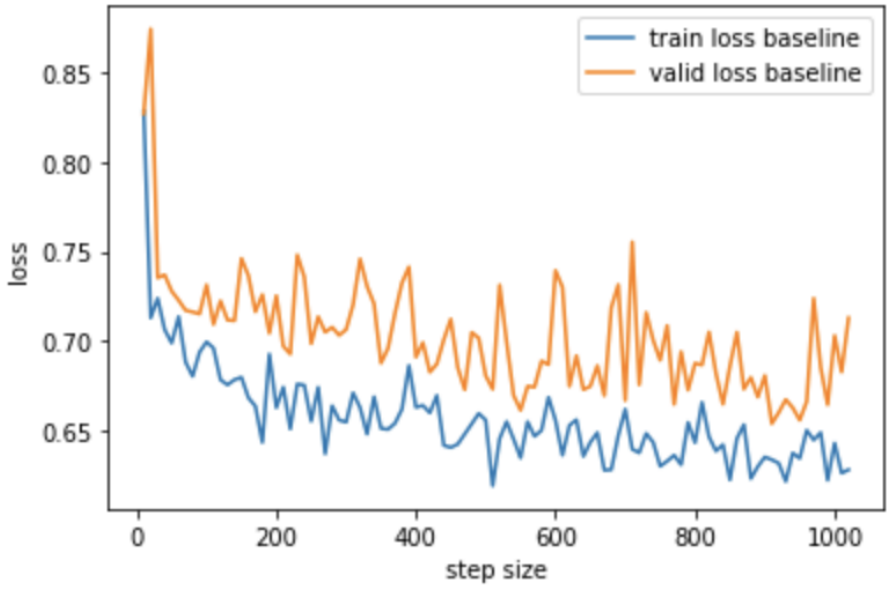
bz = 64

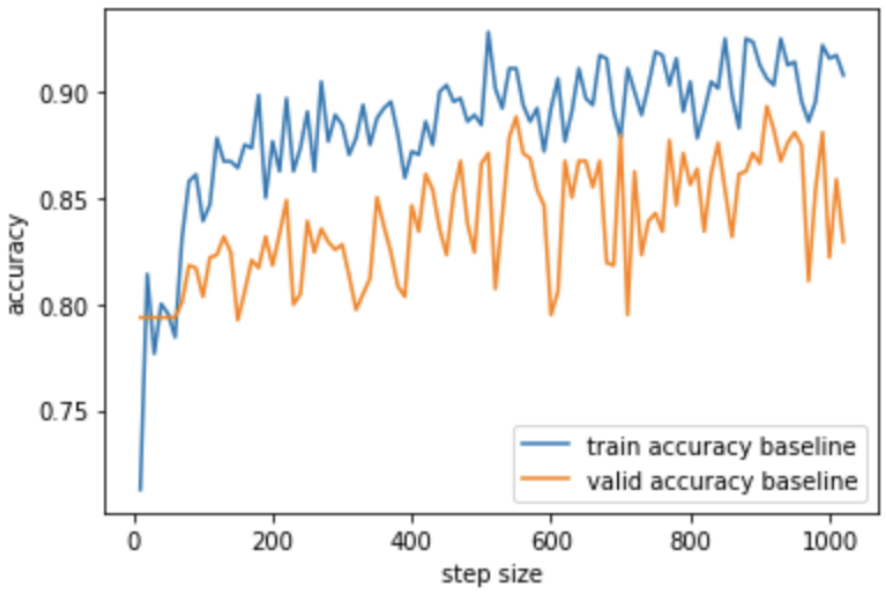
4.1.1.1 Baseline

For Baseline model, we use grayscale images to train the model. Here is the baseline architecture.

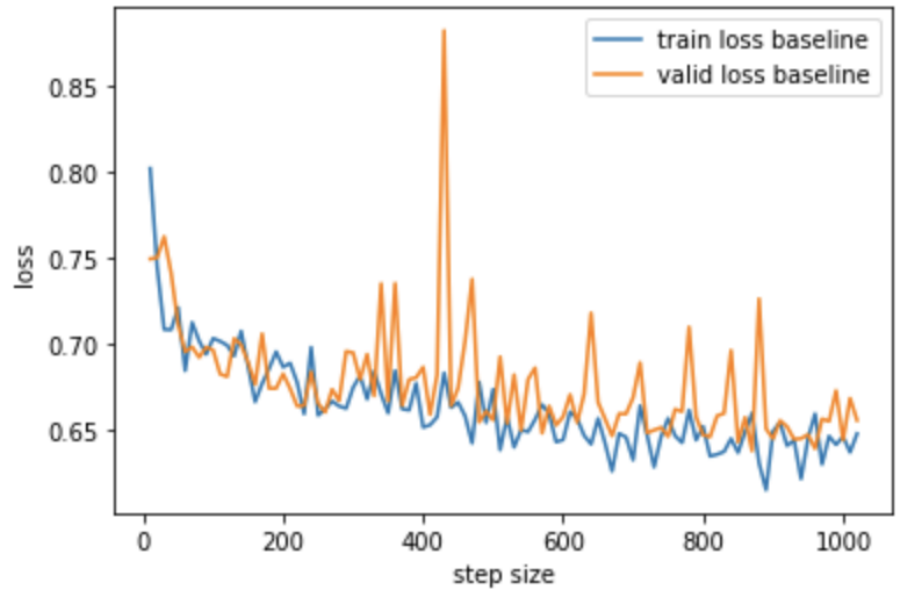
```
Baseline(
  (cnn_model): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): ReLU()
    (10): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU()
    (13): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (15): ReLU()
    (16): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): ReLU()
    (19): AvgPool2d(kernel_size=16, stride=16, padding=0)
  )
  (fc_layer): Sequential(
    (0): Linear(in_features=512, out_features=512, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=512, out_features=3, bias=True)
    (4): Sigmoid()
  )
)
```

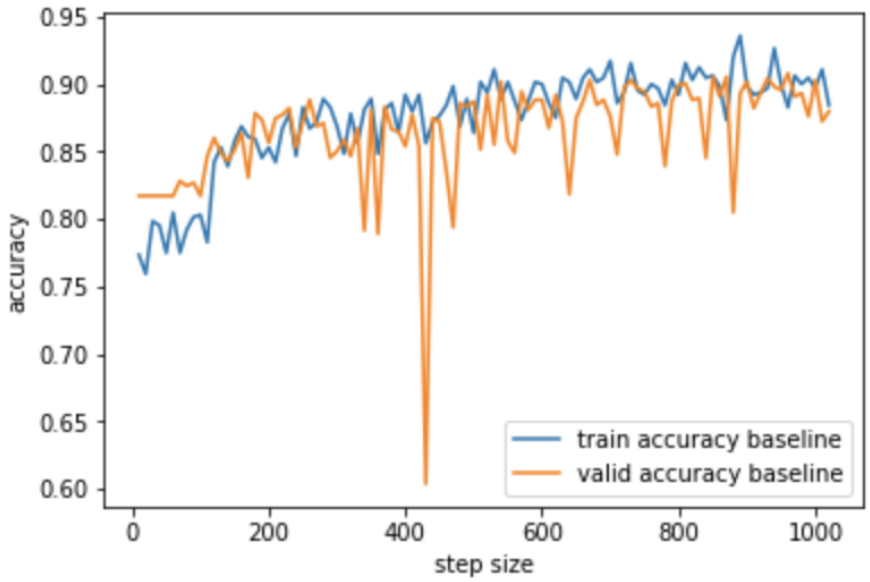
Here is the loss and accuracy when we split the data directly(without shuffle the clipped image and split the data by original pictures). As we can see from the graph, there is a gap between training loss and validation loss, which implies overfitting. This may be caused by small training set, too much training epoches. Besides, the validation accuracy does not improve much in later epoch, which means that we can stop training earlier.





Here is the loss and accuracy when we split the data randomly(shuffle the clipped image and split the data in clipped image dataset). The gap here is quite small and the reason may be that the clipped images in same picture can exist in both training set and also validation set and, thus, two set can have higher similarity due to clipped image shuffle. In this case, the validation set can have large bias on reflecting the true performance of the model on new data. Therefore, we think this split data method is not good but can be a great chance for us to do the comparison and explore the reasons behind some abnormal situation.



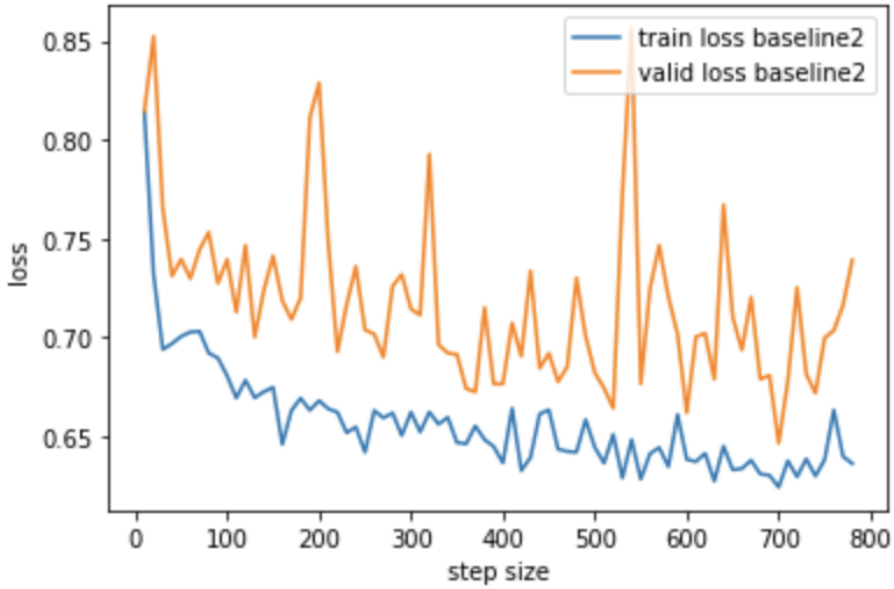


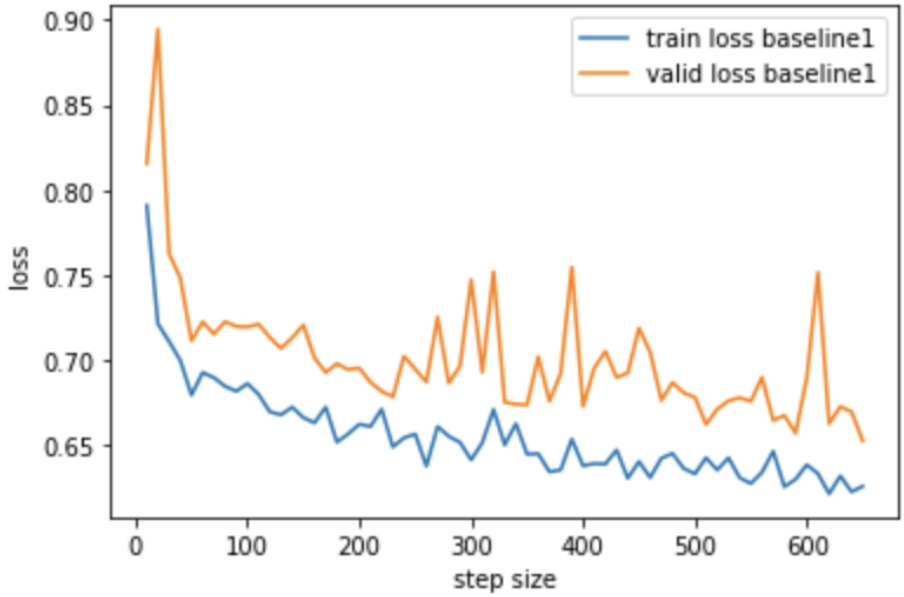
4.1.1.2 Improvement

1. Optimization setting

We use two settings and each one has a relatively better performance on validation set.

1. optim.Adam(model.parameters(),lr = 0.001,weight_decay=0.0001); num_epochs = 25; bz =128
2. optim.Adam(model.parameters(),lr = 0.002,weight_decay=0.0001); num_epochs = 30; bz =128





Name	Validation Accuracy
Baseline	0.8882
Improvement1	0.8993
Improvement2	0.9017

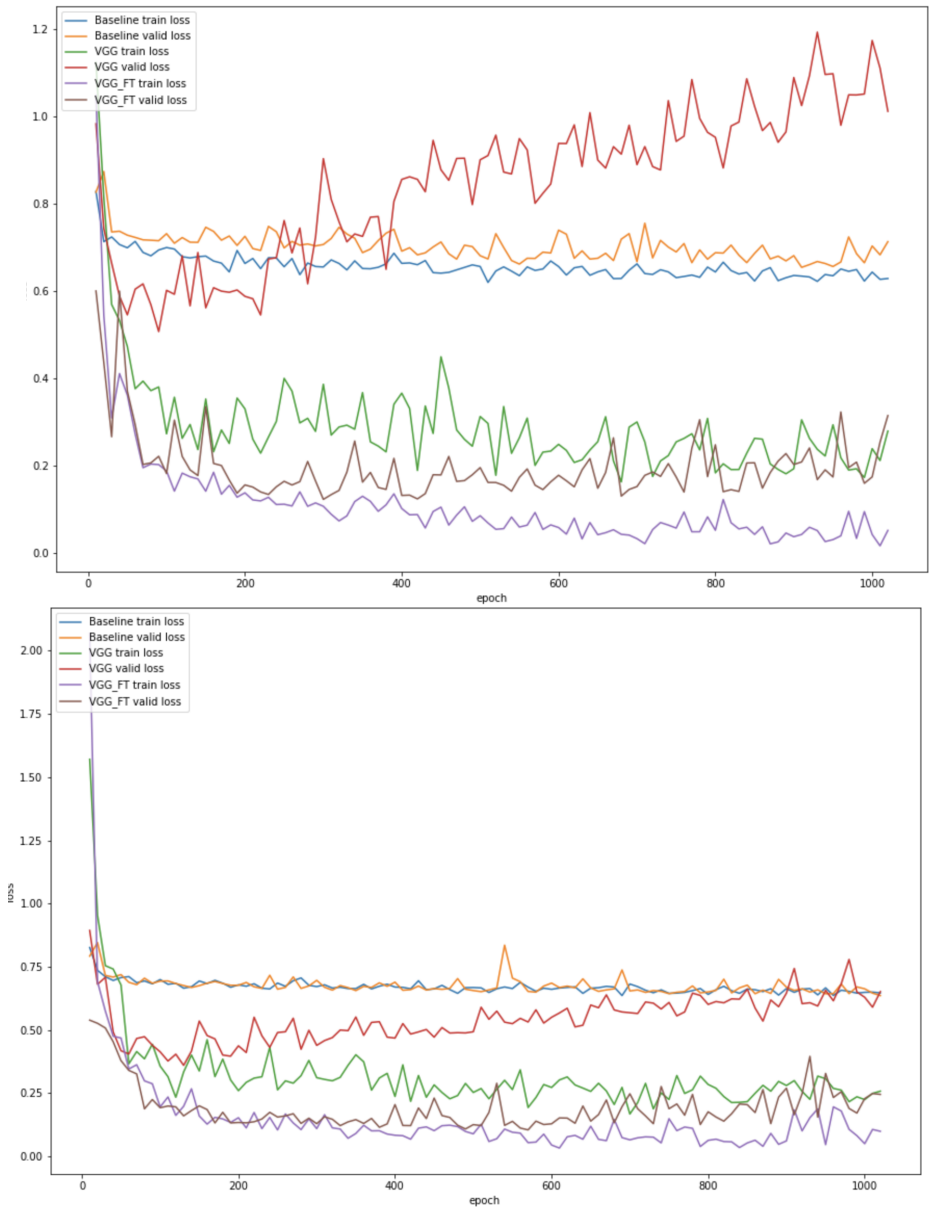
2. Different Architecture

In this method, we use two pretrained model: vgg16 and resnet18 to do the classification part and the fintune models actually do a quite well job compared to other improvement.

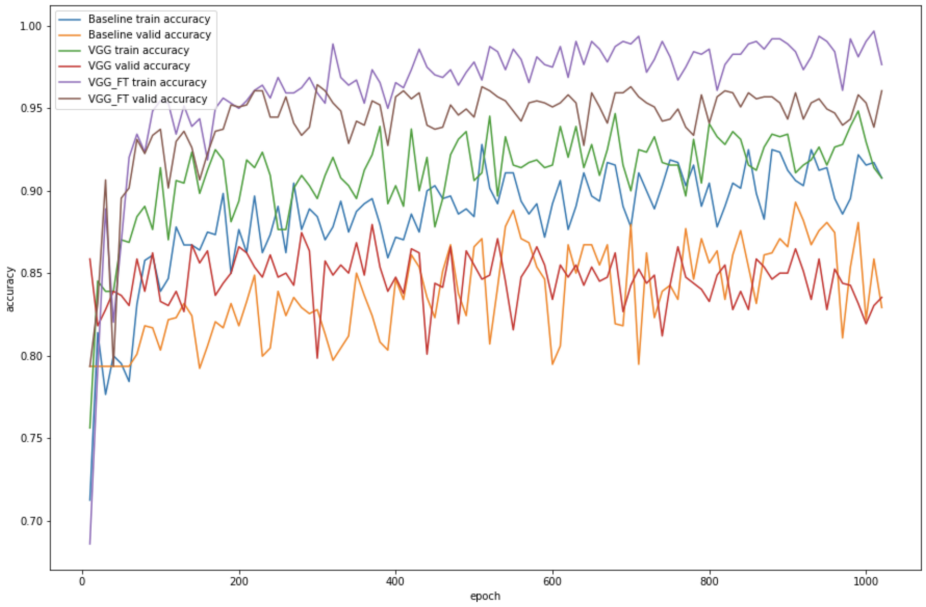
Take VGG16 net as an example

The first loss graph is trained by the direct split dataset and second one is from random split dataset. From both graphs, we can find that VGG finetune model do the best among three methods. This is because VGG is pretrained from a larger dataset and can have a better ability to extract features from the images. Besides, since VGG is pretrained from other type of images instead of mask images, finetune the model can enable VGG to better fit our mask dataset and thus has a better performance.

One thing we notice from both graph is that the gap between the freezed vgg's training loss and validation loss is quite large. Since no matter the split dataset methods we use, the gap is large, we can infer that the gap is because the freezed vgg model itself cannot predict the validation data well and cause the overfit but not the dataset problem.



Here is the accuracy graph, we can know that the validation performance of Baseline and freezed VGG net is similar while VGG fine-tune has the highest accuracy.



Above all, compared to freezed vgg model, fine-tune model have a better performance in both validation loss and accuracy. The resnet models also have a similar result.

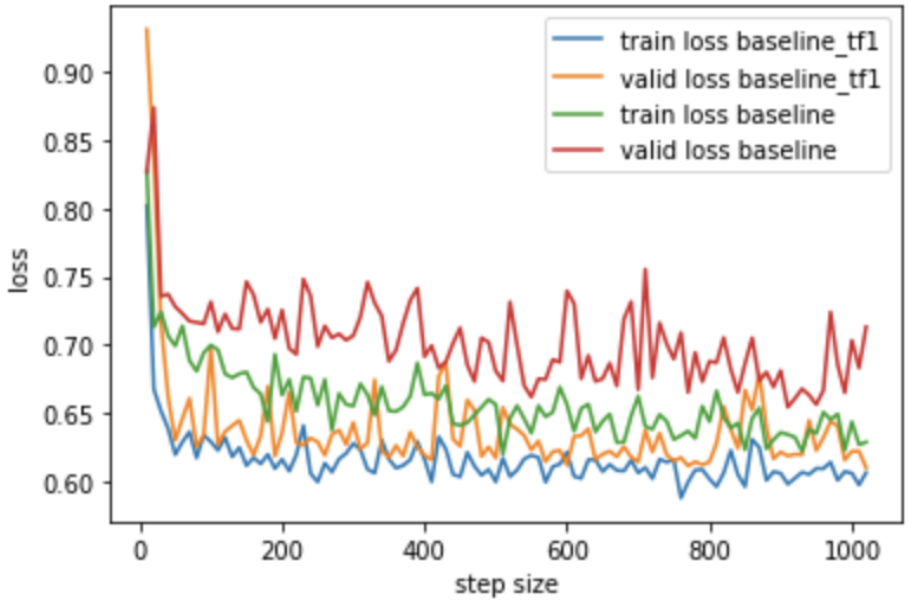
3. Data augmentation

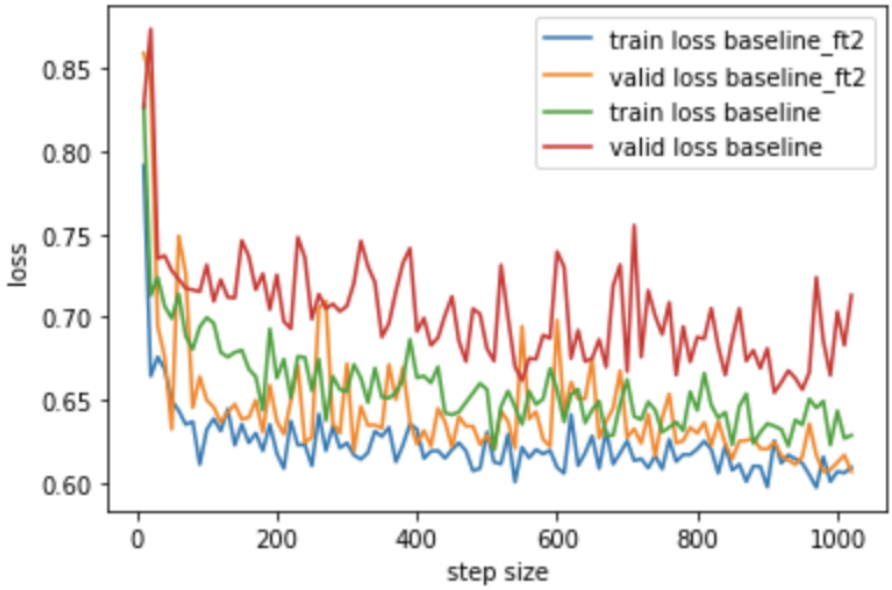
We use three data augmentation method to prevent overfitting and the performance is quite good.

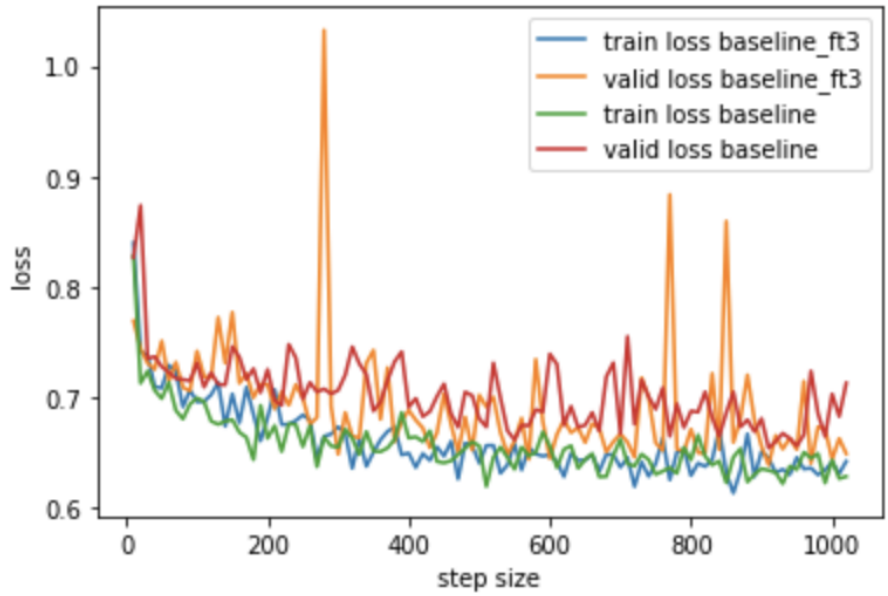
Here are three method:

1. `transforms.RandomHorizontalFlip(p=0.9)`
2. `transforms.RandomHorizontalFlip(),transforms.RandomRotation(degrees=(10, 90))`
3. `transforms.RandomHorizontalFlip(),transforms.ColorJitter(brightness=.5, contrast=.5, saturation=.5, hue=.5)`

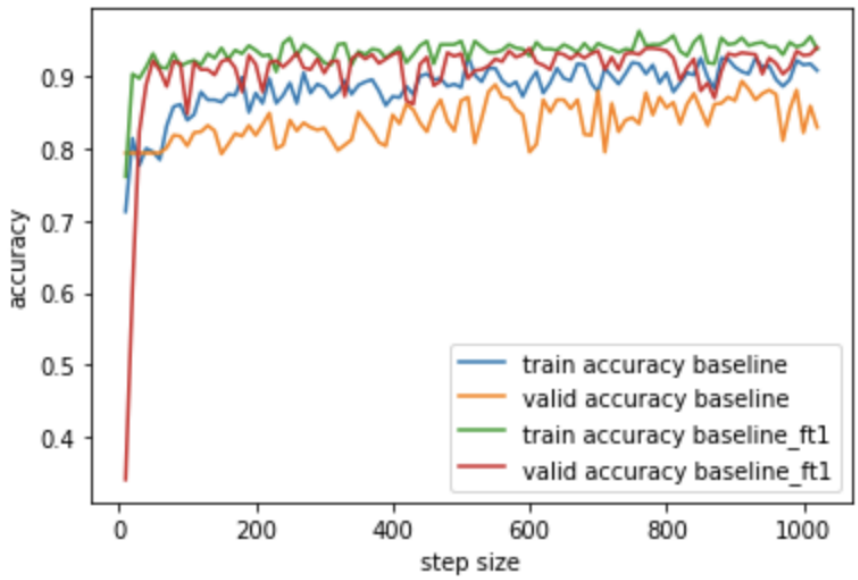
   All the methods above are trained with RGB data.

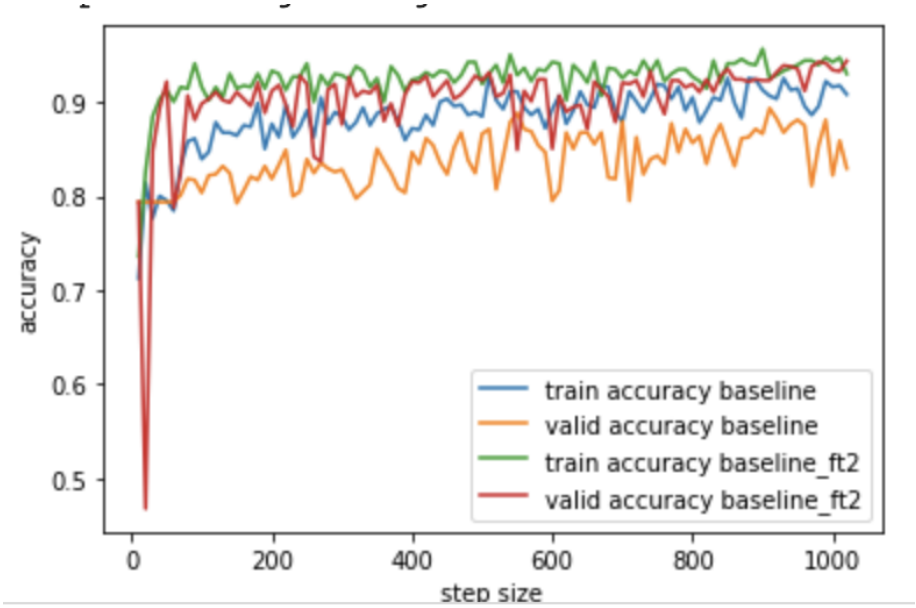


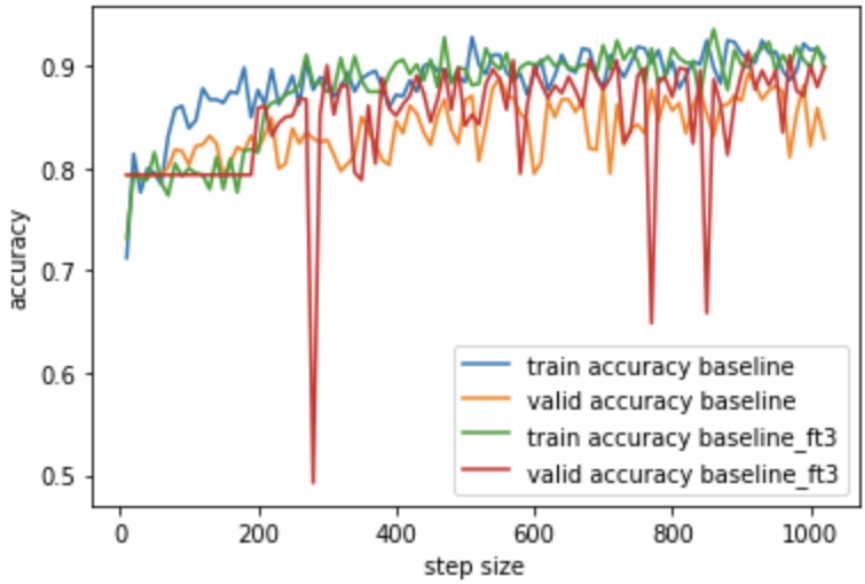




From three loss graphs above, we can see the loss gap become small because data augmentation add some randomness to training data set and prevent overfitting to some extend. Besides, the first two methods are better than the third method regarding the validation loss, this may be because the color of the images is quite important in this task since faces and masks follow some standard rules so add the randomness to the color is not a good idea.





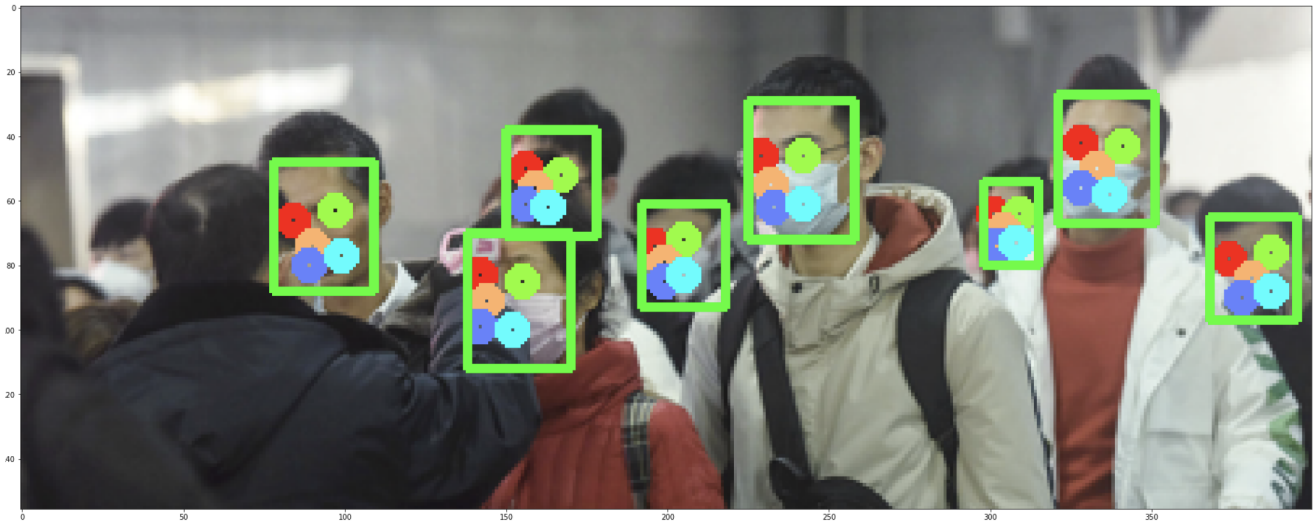


The accuracy graphs also follow my analysis of loss graphs, except method 3, the first two methods have a relatively high validation accuracy because of the use of data augmentation methods and RGB channels.

4.1.2 Face Detection

4.1.2.1 Face Detection using Retinaface

In this section, we first use the pretrained Retinaface model to do the face detection. Here is an example and we get the box annotations of every images in Kaggle face dataset.



4.1.2.2 Face Detection using Retinaface

We use the pretrained classification model in section 4.1.1 to predict the label of images clipped from Retinaface box annotation. We use best saved models -- VGG16-Finetune and Resnet18-Finetune to classify the images and then we use F1 score[1], recall and Precision to evaluate the performance.

We first use the whole dataset (training + validation set) to calculate the F1 score and recognize that this is unfair and is kind of cheating. Therefore, we do the task again by only using validation set data. Here are the result:

For all clipped image data

Name	F1 score	Precision	Recall
VGG16-Finetune	0.6664	0.6498	0.6839
Resnet18-Finetune	0.5688	0.5546	0.5837

For validation clipped image data

Name	F1 score	Precision	Recall
VGG16-Finetune	0.5810	0.5562	0.6081
Resnet18-Finetune	0.6268	0.6000	0.6560

The result is worse than YOLO model, the reason may be because we haven't train RetinaFace model on our dataset and the face bounding boxes are different from mask bounding boxes, so the detection part may not do very well.

4.2 YOLOv5 object detection model

YOLO (You Only Look Once) is an one-stage object detection methods which utilize one CNN network to directly predict the class and location of different objects. One major advantage of YOLO over the two-stage algorithms, such as RCNN, is that it is faster when predicting. Considering that the potential application scenario of this model may be a real-time recognition system, YOLO is a relatively good choice for this task. We use the Ultralytics open source implementation of YOLOv5 to train out custom model on the face mask dataset.

All the codes are run on Google Colab, with all the dependencies specified in requirement.txt.

4.2.1 Prepare Training Data

In order to train YOLOv5 on custom data, data should be arranged in required format.[2]

Create Labels

The annotations in the original dataset are in the PASCAL VOC format, while YOLOv5 requires **YOLO format** with one `*.txt` file per image. The `*.txt` file specifications are:

- One row per object
- Each row is `class x_center y_center width height` format.
- Box coordinates must be in **normalized xywh** format (from 0-1).
- Class numbers are zero-indexed.

We take with mask as class 0, without mask as class 1, and mask worn incorrectly as class 2. Since the objects in original dataset are represented in the PASCAL VOC format, e.g. `x1 y1 x2 y2`, where (x_1, y_1) is the top left corner and (x_2, y_2) is the lower right corner of the bounding box, we calculate the center (x, y) and the object width and height. Then divide `x_center` and `width` by image width, and `y_center` and `height` by image height to determine the normalized `xywh`.

Following is an example label file

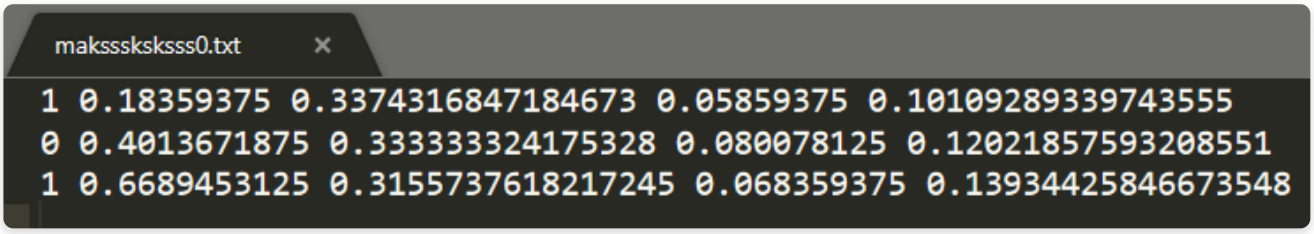


figure. Example label file

Organize Directories

The total number of images in the dataset is 853. We use `train_test_split` to split the dataset into train, valid and test sets in the ratio of $8:1:1$. YOLOv5 locates labels automatically for each image by replacing the last instance of `/images/` in each image path with `/labels/`, so we store train images in `yolov5/Dataset/images/train`, train labels in `yolov5/Dataset/labels/train`. Similarly for valid set and test set.

Create facemask.yaml

`data/facemask.yaml` shown below, is the dataset configuration file that defines 1) a path to a directory of training images; 2) the same for our validation images; 4) the number of classes; 5) a list of class names:

```
train: Dataset/images/train

val:   Dataset/images/val

nc : 3

names: ['With_Mask', 'Without_Mask', 'Incorrect_Mask']
```

4.2.2 Train YOLOv5s

Configurations

We first use the the smallest and fastest model available, namely YOLOv5s, with 7.3M parameters to train on our face mask dataset. We use `--weights yolov5s.pt` to obtain and start training from the pretrained weights. The default configuration of YOLOv5s is adopted as shown in models/yolov5s.yaml. The following figure illustrates the structure of YOLOv5s.

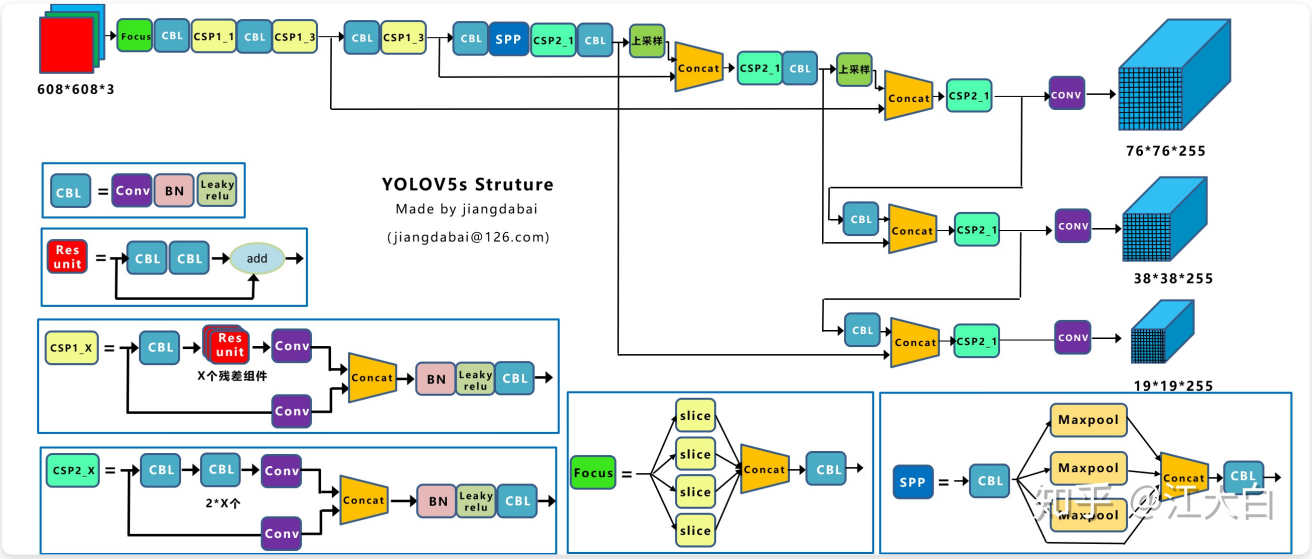


figure. YOLOv5s Structure

We use image size of 320 and batch size of 16 to train the model for 300 epochs. The hyperparameters used for training are set as follows,

```
lr0=0.01, lrf=0.2, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0,
warmup_momentum=0.8, warmup_bias_lr=0.1, box=0.05, cls=0.5, cls_pw=1.0, obj=1.0,
obj_pw=1.0, iou_t=0.2, anchor_t=4.0, fl_gamma=0.0, hsv_h=0.015, hsv_s=0.7, hsv_v=0.4,
degrees=0.0, translate=0.1, scale=0.5, shear=0.0, perspective=0.0, flipud=0.0, fliplr=0.5,
mosaic=1.0, mixup=0.0
```

Result Analysis

The train loss plots and valid loss plots are shown below. There are three kinds of loss calculated,

- `cls_loss`: classification loss
- `obj_loss`: confidence loss
- `box_loss`: `xywh` loss, i.e. the loss caused by bounding box

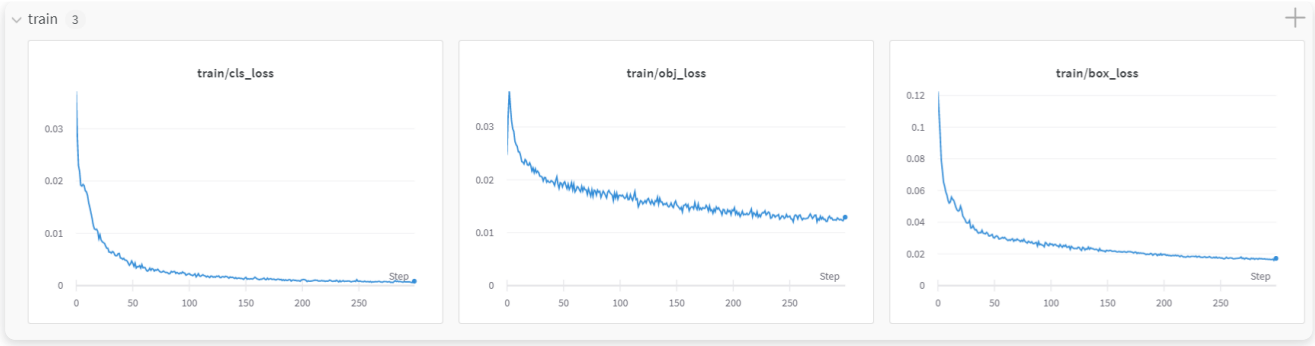


figure. YOLOv5s train loss

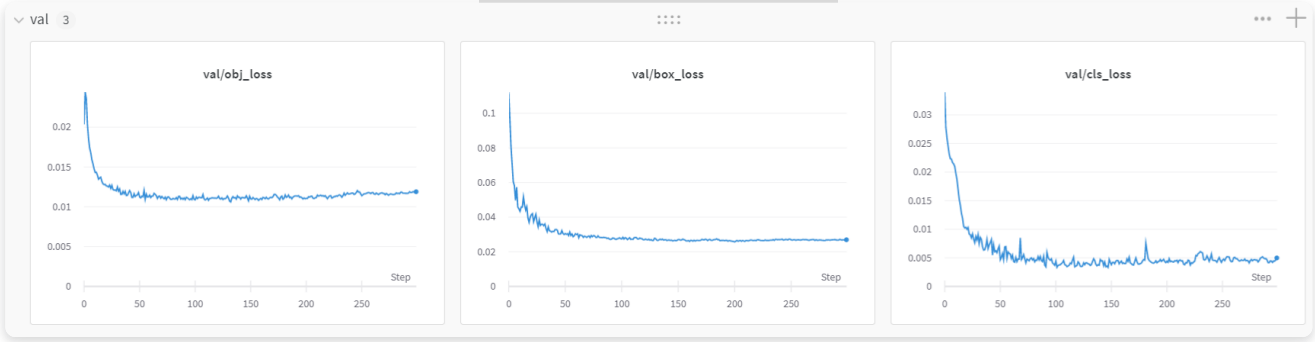


figure. YOLOv5s validation loss

Browse here to see detailed panels of train loss and val loss.

As illustrated in the loss plot, the validation loss actually converges at around 150 to 200 epoch. Loss drops really fast in the first 50 epochs.

Following are the metrics evaluated on validation set,

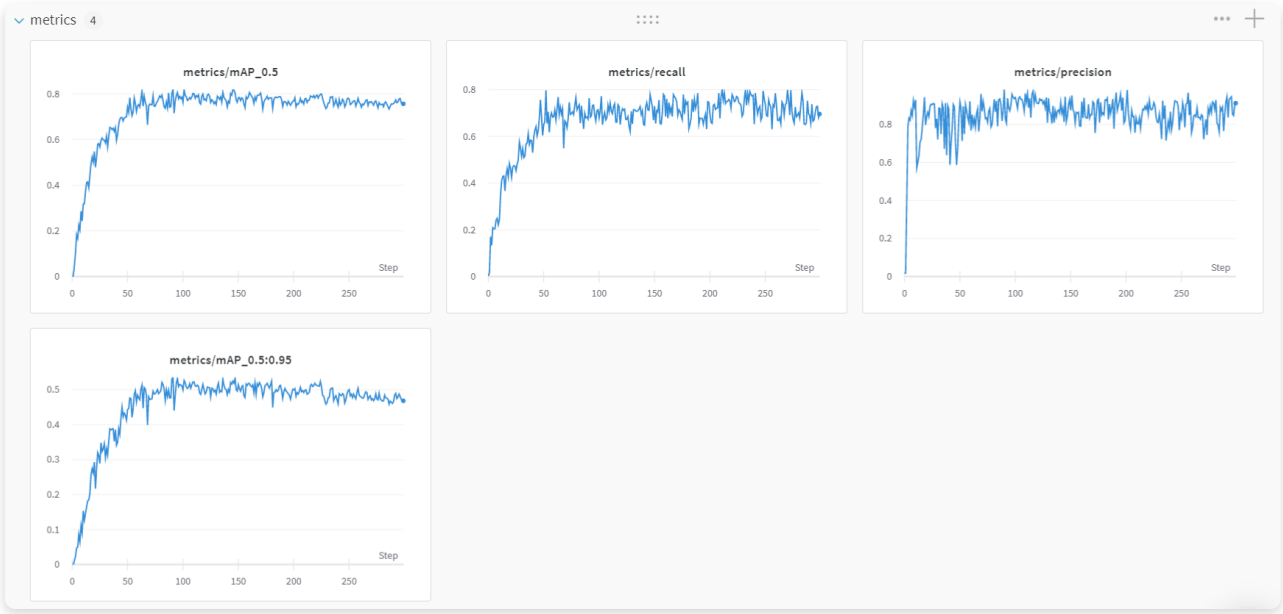


figure. YOLOv5s metrics

We can see from the figure that the model actually converges at around 100 epoch, where all the metrics reach their best value.

Using the best model obtained from the training process, we plot the F1 curve, PR curve, P curve and R curve to analyze the model performance.

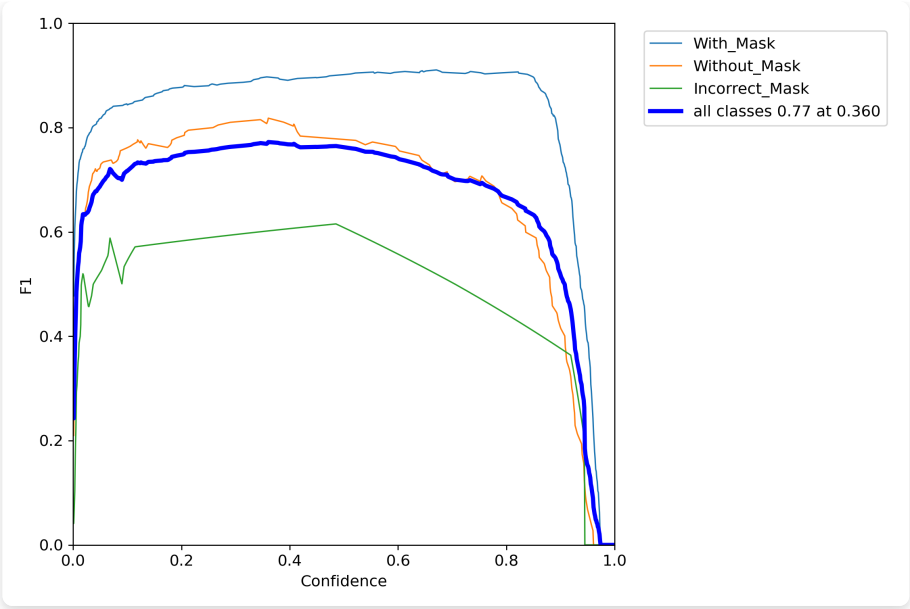


figure. YOLOv5s F1_curve

From the F1 curve, we can see that the F1 score of all classes is 0.77 at confidence of 0.360 which is a good performance. By analyzing the F1 score in each class, we can find that our model performs best on with mask class, worse on without mask class and the worst on incorrect worn mask class. This is probably because we have the largest number of data in with mask class and the smallest number of data in incorrect worn mask class. We can see that the F1 curve of incorrect worn mask class is not smooth, which also illustrates that we have very little data in incorrect worn mask class.

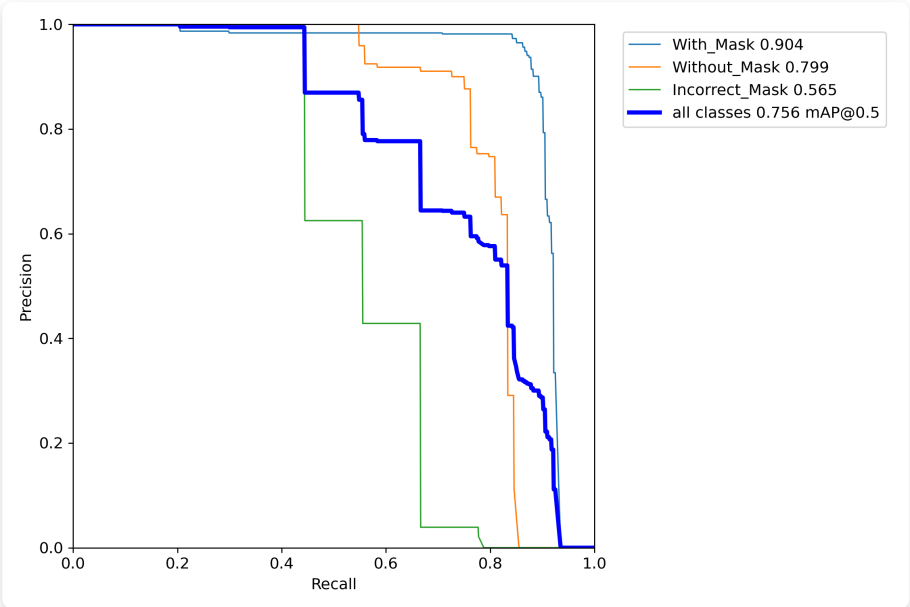


figure. YOLOv5s PR_curve

The PR_curve also demonstrate that our observation is correct. We have 0.756 mAP@0.5 for all classes which is good, and 0.904 for with mask class which is excellent, while only 0.565 for incorrect worn mask which is much worse and .

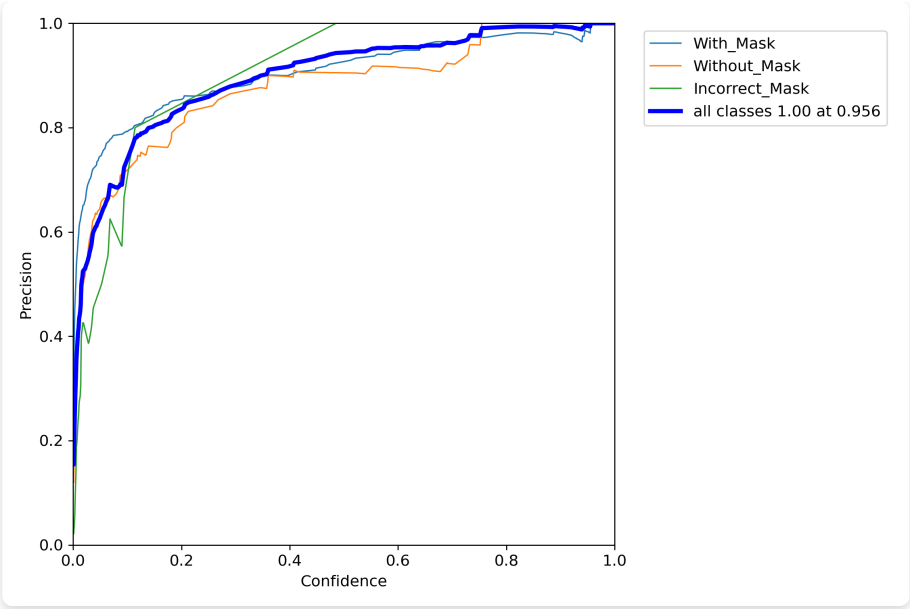


figure. YOLOv5s P_curve

The P_curve turns out to be pretty good for all the three classes and the difference is not significant which can be interpreted as that our model conservative since there are not enough data for the minority class. It will not predict one to be in the minority class unless it is pretty sure.

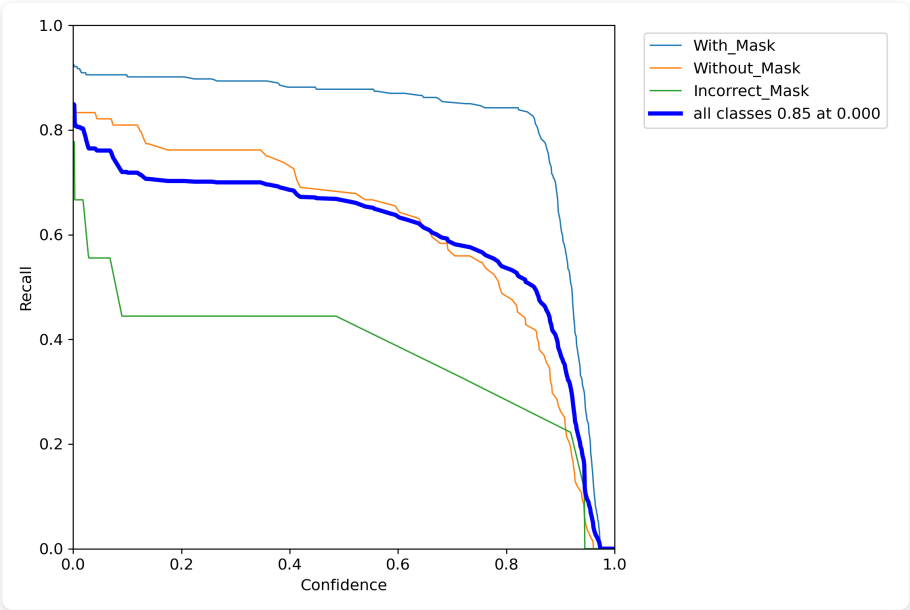


figure. YOLOv5s R_curve

This is illustrated by R_curve too since we have pretty bad recall even with small confidence in the incorrect worn mask class.

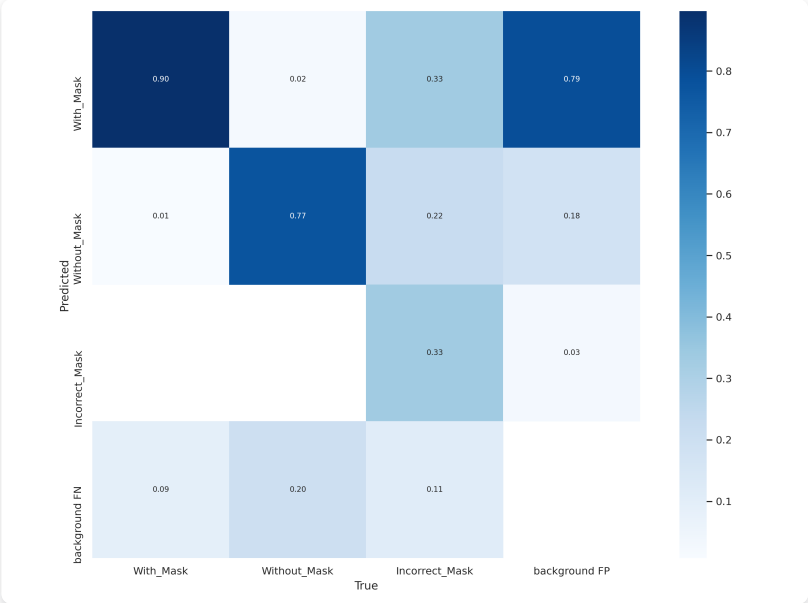


figure. YOLOv5s confusion matrix

Our observation is also shown by the confusion matrix, where most of the objects in incorrect worn mask are misclassified while the objects classified as incorrect worn mask are most true.

In conclusion, our model performs well on the majority classes and bad on minority classes. The overall performance on validation set is good.

4.2.3 Test YOLOv5s

We then test the best model we obtained in the last section on the test set using image size of 320 and IoU threshold of 0.6. The result metrics are as follows,

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95:
all	86	462	0.835	0.628	0.641	0.375
With_Mask	86	335	0.943	0.897	0.93	0.614
Without_Mask	86	111	0.893	0.676	0.688	0.355
Incorrect_Mask	86	16	0.668	0.312	0.305	0.156

figure. YOLOv5s test metrics

The test result supports the observation of validation result. Find the figures of this test result in `runs/test/exp5`.

4.2.4 Different configuration of YOLOv5

Observation

By taking a closer look at the images, we can find that many of the objects are quite small. This inspires us to use models more capable to detect small objects and use large image size to enhance detection of small objects.

figure. example of the dataset

Configurations

We use YOLOv5m this time which is a larger model comparing to YOLOv5s, with 21.4M parameters. YOLOv5m is deeper and wider, and thus more capable to detect small objects. We start with the pretrained weight. `--img` is set to 640 this time to have larger image size. Batch size is set to 32 to accelerate training process. The hyperparameter used are as follows,

```
lr0=0.01, lrf=0.2, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0,
warmup_momentum=0.8, warmup_bias_lr=0.1, box=0.05, cls=0.5, cls_pw=1.0, obj=1.0,
obj_pw=1.0, iou_t=0.2, anchor_t=4.0, fl_gamma=0.0, hsv_h=0.015, hsv_s=0.7, hsv_v=0.4,
degrees=0.0, translate=0.1, scale=0.5, shear=0.0, perspective=0.0, flipud=0.0, fliplr=0.5,
mosaic=1.0, mixup=0.0
```

Result Analysis

We can see from the loss plots that the `cls_loss` of YOLOv5m is better on validation set. This indicates that it may have a better performance of classification. The validation `obj_loss` rises after 50 epochs, which means the model overfit in some sense. We should evaluate the metrics to see whether this is a problem.

figure. YOLOv5m vs YOLOv5s loss

It turns out that YOLOv5m over performs YOLOv5s in all metrics, which is exactly what we expected to get. Over fitting in `obj_loss` is not a problem here.

figure. YOLOv5m vs YOLOv5s metrics

Browse here to see detailed panels.

By evaluating the best model we obtained on validation set, we obtain the following metric figures. These figures show that we are gaining better performance overall and on the minority class. The overall performance on all metrics is improved. The gap between the major class and the minority class significantly reduced in terms of F1 score and mAP. However, the recall is still unsatisfactory.

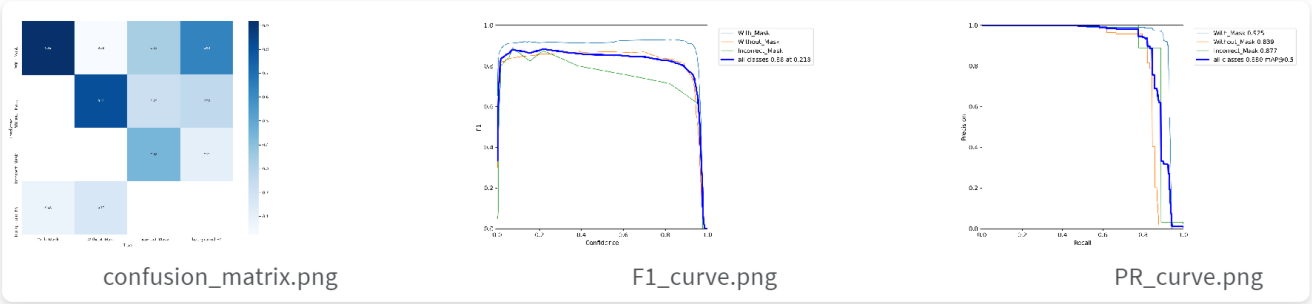


figure. YOLOv5m best metrics 1

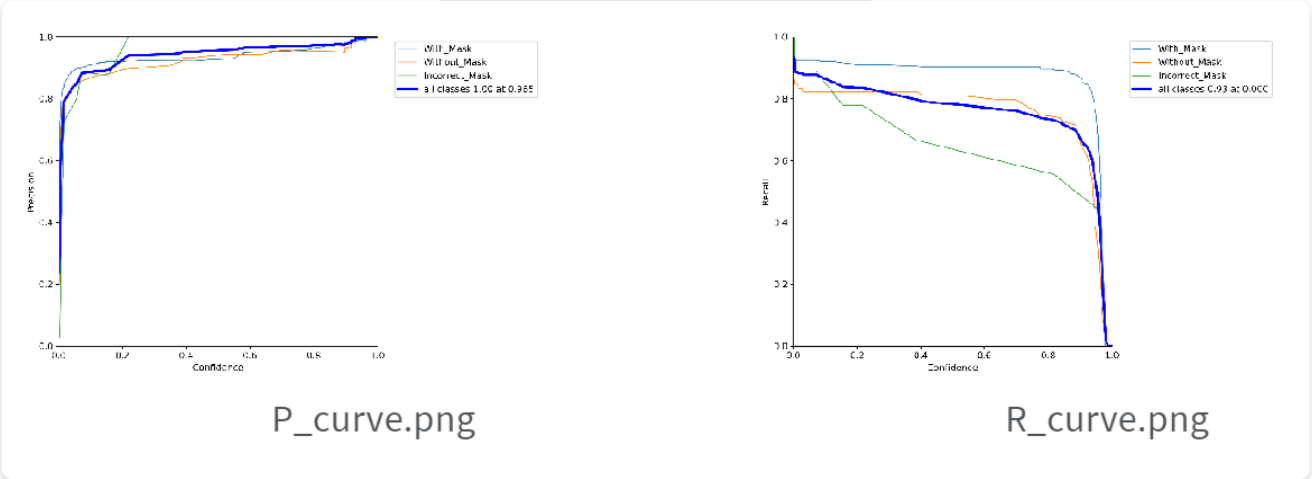


figure. YOLOv5m best metrics 2

Find original figures here or in `runs/train/exp12`.

Test result

We again test the best model obtained on the test set using image size of 640 and IoU threshold of 0.6 . The result metrics are as follows,

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95:
all	86	462	0.897	0.747	0.739	0.498
With_Mask	86	335	0.922	0.955	0.952	0.687
Without_Mask	86	111	0.895	0.847	0.832	0.54
Incorrect_Mask	86	16	0.874	0.438	0.433	0.267

figure. YOLOv5m test metrics

We can tell that the performance of YOLOv5m is all-round better than YOLOv5s. However, the performance on incorrect mask class is still lagging way behind the other two classes.

Find the figures of this test result in `runs/test/exp6`.

4.2.5 Imbalanced Dataset

As shown before, the dataset is an imbalanced dataset with much more with_mask objects and only a few incorrect_mask objects. We try to tackle this problem in this section.

Class weight

Assign different weight to different classes to perform weighted loss is a commonly used strategy to solve imbalanced dataset problem. By checking the source code of YOLOv5, we find that the weighted class is automatically attached by assigning different weights to different classes based on their occurrence. So the results we obtain before are results with weighted loss performed.

Data augmentation

Data augmentation by oversampling is another way to tackle the imbalances set problem. We extract all the images with at least one object in class incorrect_worn_mask in the training set to form a new training set, and train the best YOLOv5m model we obtained in 4.2.3 on the new dataset for 30 epochs with the same configurations. If time allows, we will oversample more images and train the model from beginning. However, considering the training time is long, we use this way to see whether it enhance the performance.

The result of the best model obtained in this model slightly improves from 0.880 to 0.885 in terms of mAP@0.5.

The test result also improves slightly.

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95:
all	86	462	0.862	0.75	0.74	0.499
With_Mask	86	335	0.912	0.955	0.952	0.685
Without_Mask	86	111	0.896	0.856	0.84	0.548
Incorrect_Mask	86	16	0.778	0.438	0.427	0.265

figure. Oversampling test metrics

Find metrics figures here, or in `runs/train/exp22`. Find the figures of this test result in `runs/test/exp8`.

In conclusion, to solve this problem thoroughly, we need more data in the class incorrect_worn_mask.

4.2.6 Improving Model Testing

In this section, we try two methods to improve the model test result using the best models obtained in previous sections.

Model Ensembling

We use model ensembling during testing for improved mAP and Recall. From https://www.sciencedirect.com/topics/computer-science/ensemble-modeling:

> Ensemble modeling is a process where multiple diverse models are created to predict an outcome, either by using many different modeling algorithms or using different training data sets. The ensemble model then aggregates the prediction of each base model and results in once final prediction for the unseen data. The motivation for using ensemble models is to reduce the generalization error of the prediction. As long as the base models are diverse and independent, the prediction error of the model decreases when the ensemble approach is used. The approach seeks the wisdom of crowds in making a prediction. Even though the ensemble model has multiple base models within the model, it acts and performs as a single model.

We ensemble the two models obtained in 4.2.4 and 4.2.5 as a single model and test it on the test set.

The test result improves slightly,

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95:
all	86	462	0.894	0.746	0.745	0.501
With_Mask	86	335	0.919	0.955	0.952	0.682
Without_Mask	86	111	0.889	0.847	0.848	0.553
Incorrect_Mask	86	16	0.875	0.437	0.433	0.267

figure. Ensembling test metrics

Ensembling works probably because the models can sometime find missed detections for each other. Find the figures of this test result in `runs/test/exp9`.

Test Time Augmentation

We can improve the test image size by about 30% for improved results. Using the the best model we obtained in 4.2.4, the test result is as follows,

Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95:
all	86	462	0.941	0.711	0.789	0.513
With_Mask	86	335	0.94	0.955	0.969	0.691
Without_Mask	86	111	0.905	0.865	0.881	0.536
Incorrect_Mask	86	16	0.976	0.312	0.518	0.312

figure. TTA test metrics

The test result is improved as we can detect smaller objects. Find the figures of this test result in `runs/test/exp17`.

4.2.7 Result

The best F1 score we obtained on test set is 0.80 in `run/test/exp6` using model obtained in 4.2.4. The best mAP@0.5 we obtained on test set is 0.789 in `run/test/exp17` using TTA with model obtained in 4.2.4. An example of prediction on test set images is shown below.

figure. example test prediction

1. The function used to calculate IoU is modified from http://www.21cn-home.com/article/381703.html ↩

2. Code in this section is modified from https://www.kaggle.com/rkuo2000/yolov5-facemask ↩