CMPT 762

Computer Vision

Assignment 3

Gokul Mohanarangan
gma56@sfu.ca

301436162

Free-Late day: 2 late days
are intended to be used here.

The name on Kaggle : Naruzu
Best accuracy : 0.37045

PART 1 :

i) List of the configs and modifications that you used.

```
cfg.DATALOADER.NUM_WORKERS = 4
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml")
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.003
cfg.SOLVER.LR_SCHEDULER_NAME = "WarmupCosineLR"
cfg.SOLVER.STEPS = []
cfg.MODEL.PIXEL_STD = [57.375, 57.120, 58.395]
cfg.SOLVER.MOMENTUM = 0.9
cfg.SOLVER.MAX_ITER = 2000
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.6
```

No special modifications were used for the pipeline, as improving the configurations improved the AP of the whole training set.

ii) Factors which helped improve the performance.

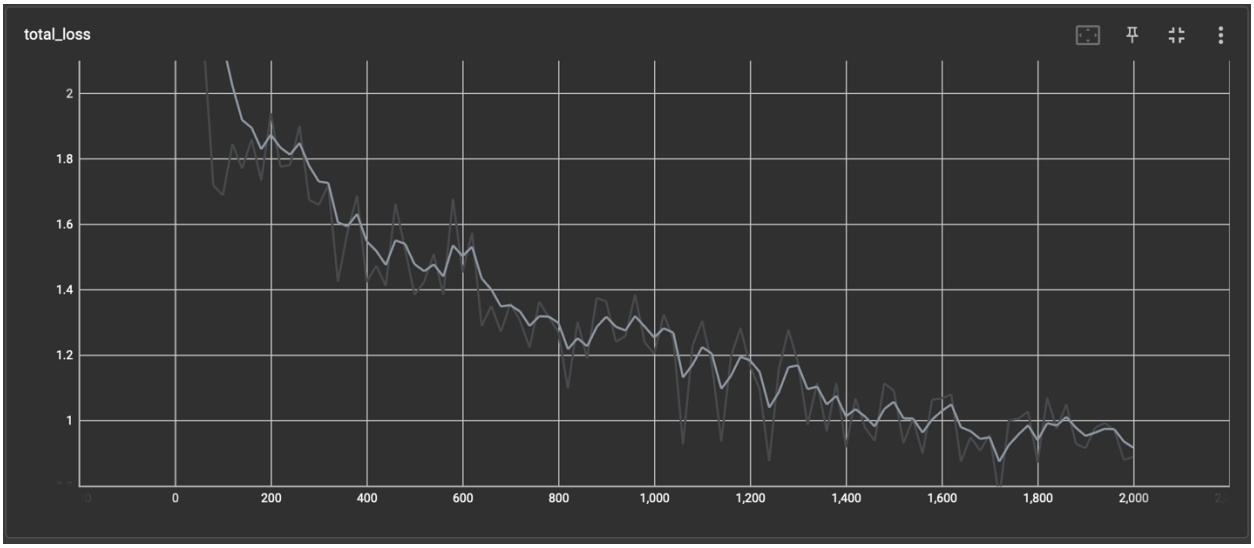
```
cfg.SOLVER.MAX_ITER = 2000
```

With fewer iterations, the loss seemed to stagnate and did not improve after changing a variety of configurations. When the max iter was increased to be big, though the loss was a little fluctuating, the loss definitely decreased ever so slightly but continuously.

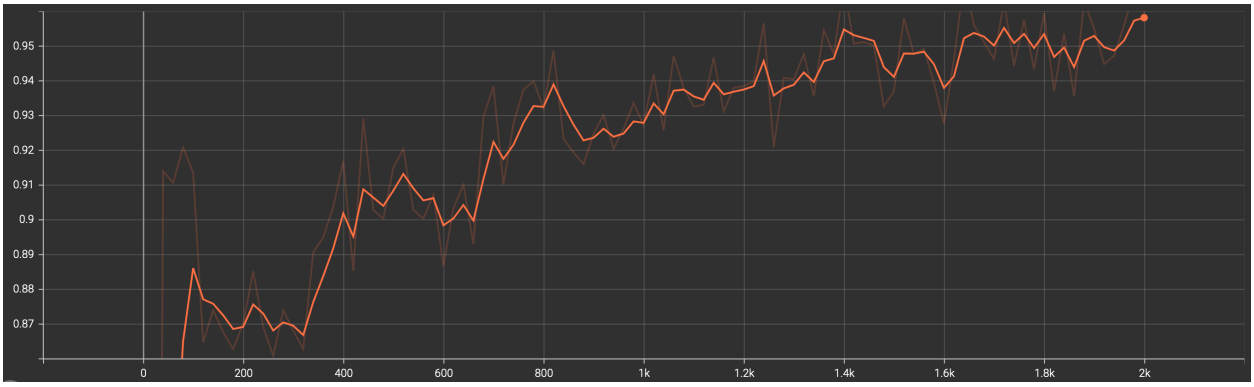
```
cfg.SOLVER.BASE_LR = 0.003
cfg.SOLVER.LR_SCHEDULER_NAME = "WarmupCosineLR"
```

The learning rate was set too high when starting and optimizer seemed to be oscillating around a local minimum. i.e.) Loss is less in one iter and it shoots up at another iter and this pattern continued. When it was too low, the loss did not decrease for several iterations and was stuck. Choosing an optimum lr and using a lr scheduler for this set of configurations gave best results.

iii) Final plot for total training loss and accuracy

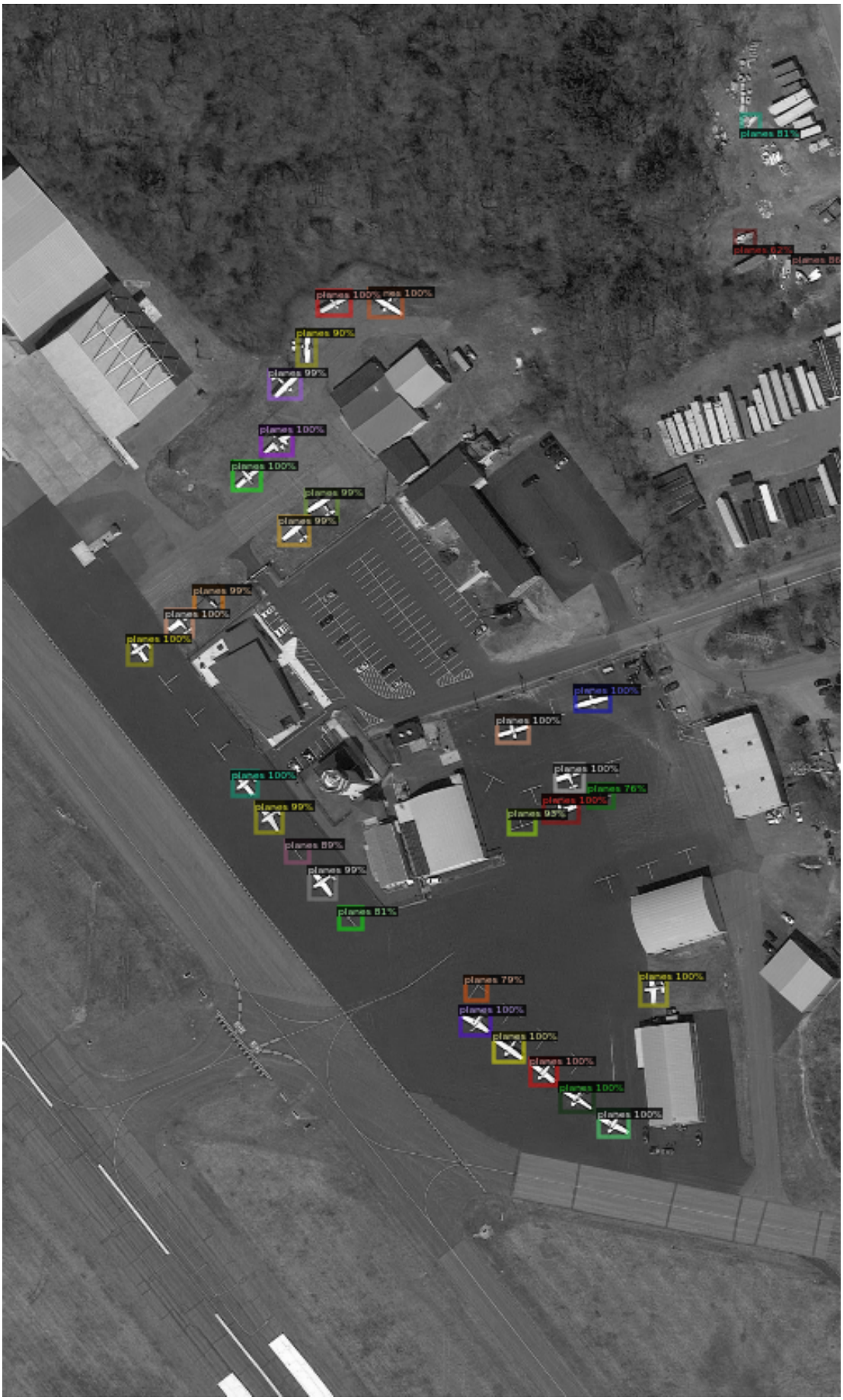


Iterations



Faster RCNN Accuracy vs iterations

iv) The visualization of predictions from 3 test samples :


planes 81%
planes 82%
planes 100%
planes 100%
planes 90%
planes 99%
planes 100%
planes 100%
planes 99%
planes 99%
planes 99%
planes 100%
planes 100%
planes 100%
planes 100%
planes 100%
planes 100%
planes 76%
planes 100%
planes 93%
planes 100%
planes 99%
planes 89%
planes 99%
planes 81%
planes 79%
planes 100%
planes 100%
planes 100%
planes 100%
planes 100%
planes 100%

v) Ablation study

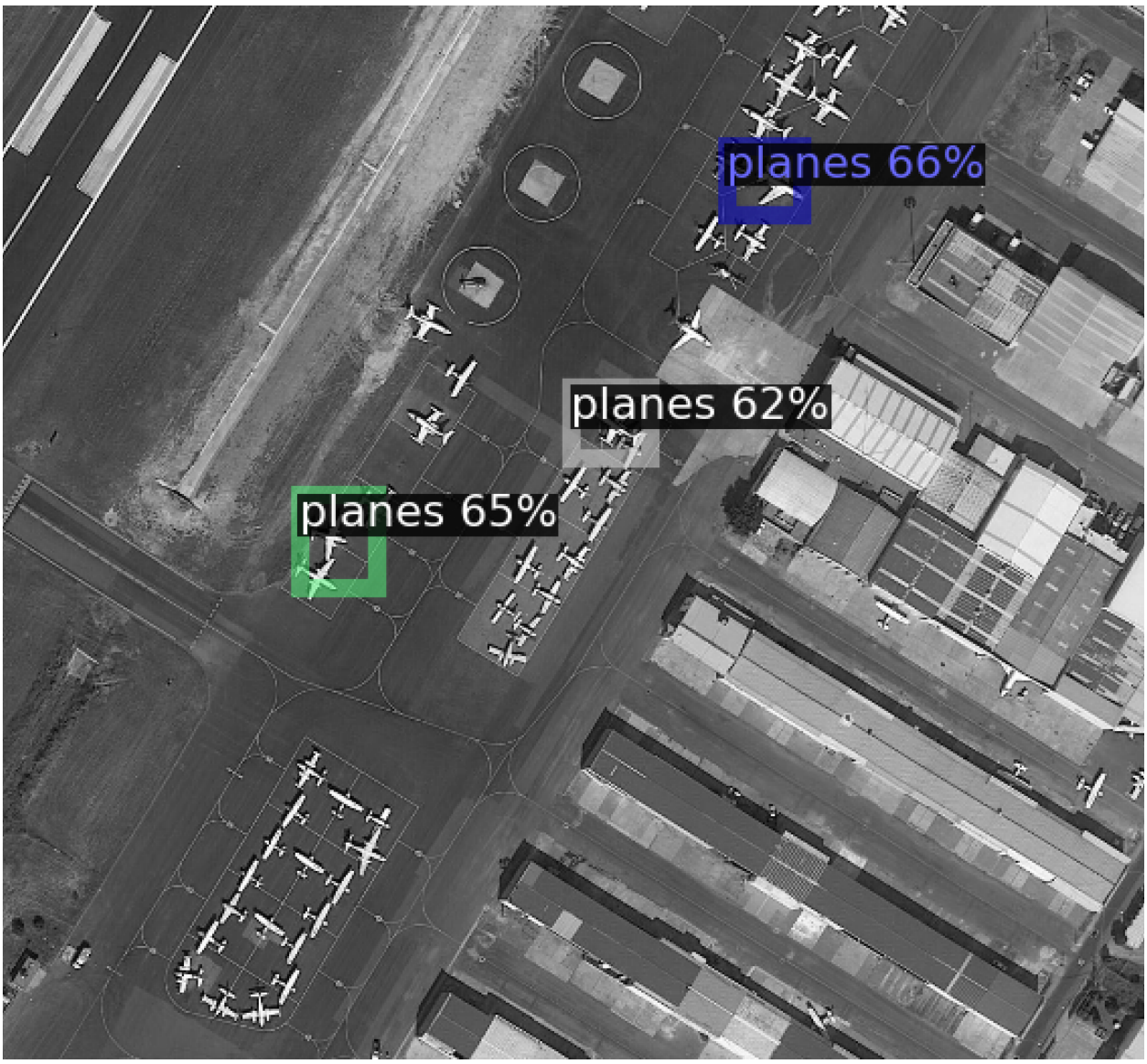
With a few changes to the configurations for training, the loss at epoch 0 was 0.61 and this did not seem to decrease for over 100 iterations. The model was not learning at every iteration. The best loss was at 0.58 which led to the final AP metric to be 0.19 which was affecting the test predictions. But, after performing the above set of configurations, the loss at epoch 0 was 0.30 (almost 50% better than previous training routine already). This time, the loss seemed to decrease gradually and the idea is that with enough resources, we can run longer iterations and it will give us better results slightly. The loss ended less than 0.20 and the final AP was 0.44 which was decent. The Kaggle score with and without the edit are 0.37 and 0.35 respectively.

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.835
[11/01 00:15:59 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 44.160 | 64.445 | 51.552 | 33.455 | 52.312 | 76.199 |
OrderedDict([('bbox', {'AP': 44.16035693077818, 'AP50': 64.44519509407439, 'AP75': 51.552302520455896, 'APs': 33.45517247044965, 'APm': 52.31183993333015, 'APl': 76.19904249769326})])
```

Observations for ablation study :

1 - Missing planes

Sometimes, I observed that with a bad model, many planes were missed from detection. A better model was able to overcome some of this pitfall, especially when there are a bunch of smaller planes (less resolution) grouped together.



2- Each plane detection also comes with a model confidence

A bad model was predicting wrongly with a decent confidence but a better model was able to do a overcome this problem and even if it predicted wrongly, the confidence of the prediction was low and that was a good sign to keep the model configurations based on observations.

PART 2 :

i) Report any hyperparameter settings you used

```
num_epochs = 10
batch_size = 4
learning_rate = 0.008
weight_decay = 3e-5
optim = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)
sch = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=num_epochs)
```

ii) Final architecture

“Conv” Unit :

Layer number	Layer type	Kernel size (For conv layers)	Padding	Input | Output Dimensions	
With activation					
1	Conv2d	3	1	x | x	
2	BatchNorm2d			x | x	

3	Relu			x | x	
Without activation					
1	Conv2d	3	1	x | x	

“Down” Unit :

Layer number	Layer type	Kernel size (For conv layers)	Padding	Input | Output Dimensions
1	“Conv” Unit (above)	-	-	x | x
2	MaxPool2d	-	-	2x | x

“Up” Unit :

Layer number	Layer type	Kernel size (For conv layers)	Stride	Input | Output Dimensions
Bilinear				
1	UpSample (scale factor 2)	-	-	x | 2x
2	“Conv” Unit (above) with activation	-	-	x | x
Non bilinear				
1	ConvTranspose2d	2	2	x | 2x
2	“Conv” Unit (above) with activation	-	-	x | x

Final Network:

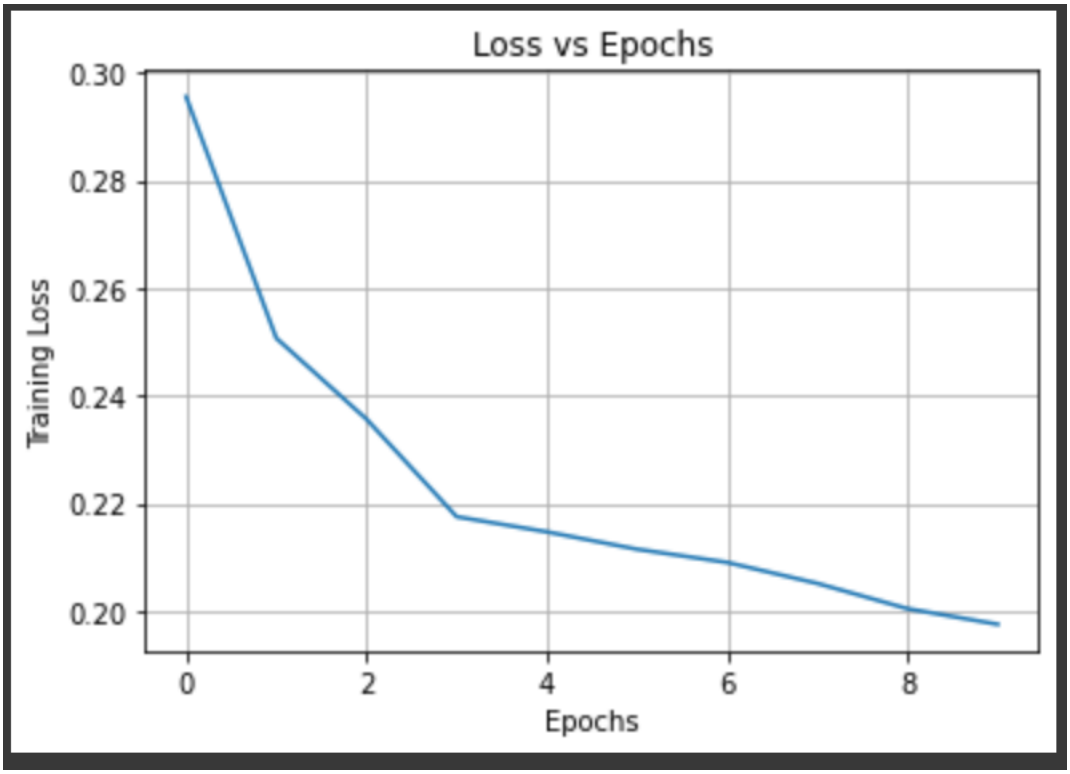
Layer number	Layer type	Input | Output Channels	Input | Output Dimensions
1	Conv Unit	3 | 16	256 | 256
2	Down Unit	16 | 32	128 | 128
3	Conv Unit	32 | 32	128 | 128
4	Down Unit	32 | 64	64 | 64
5	Down Unit	64 | 128	32 | 32
6	Conv Unit	128 | 128	32 | 32
7	Conv Unit	128 | 512	32 | 32

8	Conv Unit	512 | 128	32 | 32
9	Conv Unit	128 | 128	32 | 32
10	Up Unit	128 | 64	64 | 64
11	Up Unit	64 | 32	128 | 128
12	Conv Unit	32 | 32	128 | 128
13	Up Unit	32 | 16	256 | 256
14	Conv Unit	16 | 3	256 | 256
15	Conv Unit	3 | 1	256 | 256

The encoder and the decoder layers were too shallow and the model was not being forced to learn features during the encoding/decoding phase. Hence the model network has to be deeper and though the recommended size was 128 * 128, I went with 256*256 so that the model can do as deep as I want. The limitation here was the training time that comes with it so I could not go deeper than this. But the current network yielded an mean IoU score above 0.55. I also added residuals layers where the input from one of the previous layers was fed back to the layer to the front so that the model "remembers" the features it learnt before. This is similar in concept to skip connections but usually skip connections pan across connecting farther layers. I mimicked the encoder and decoder phase to form an architecture similar to UNet, with the only changing layers were Up and Down.

iii) Loss function and training loss:

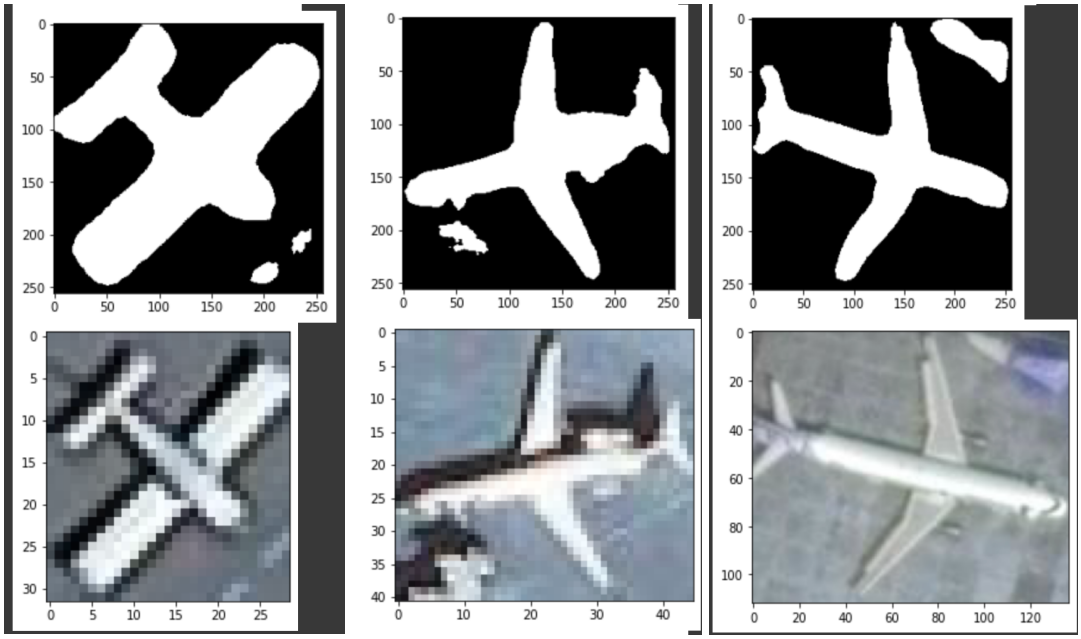
```
crit = nn.BCEWithLogitsLoss()
```



iv) Mean IoU:

```
#images: 7980, Mean IoU: 0.5526990857353405
```

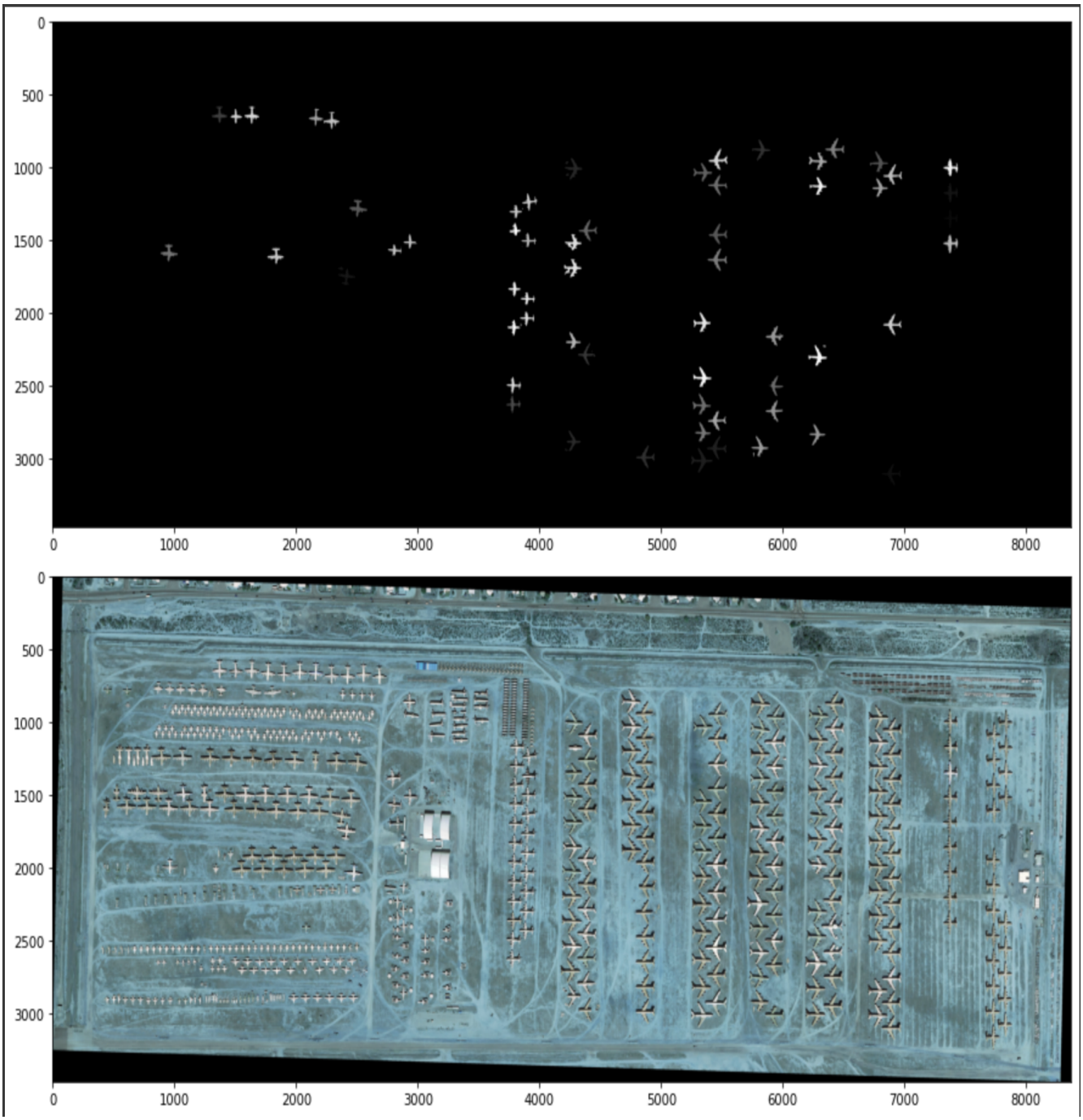
v) Visualize 3 test samples and predicted masks:

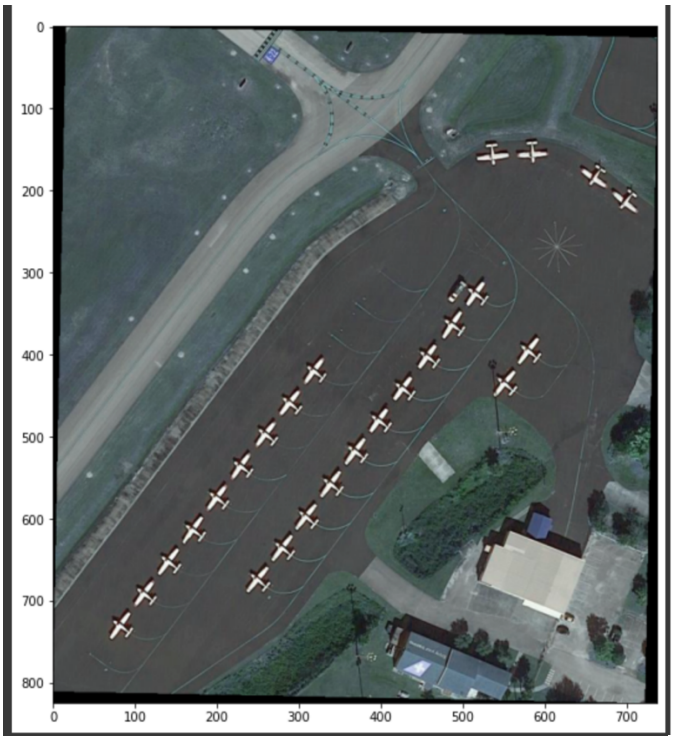
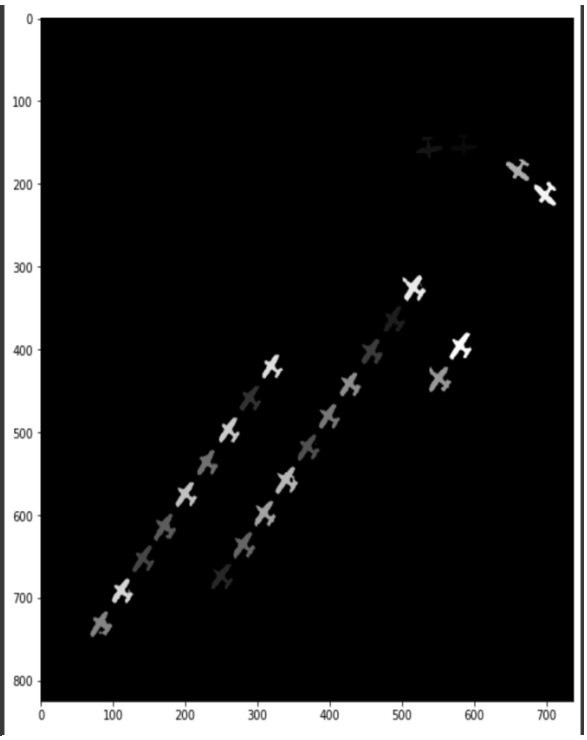
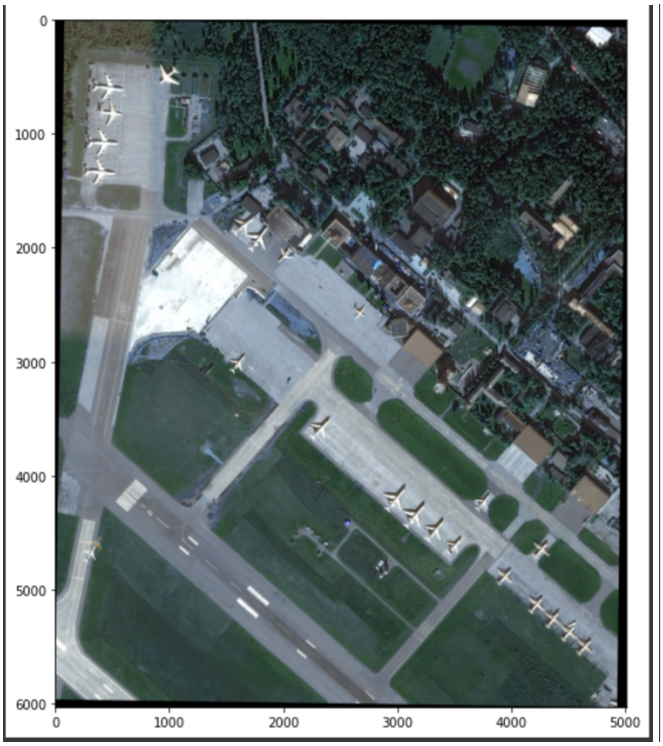
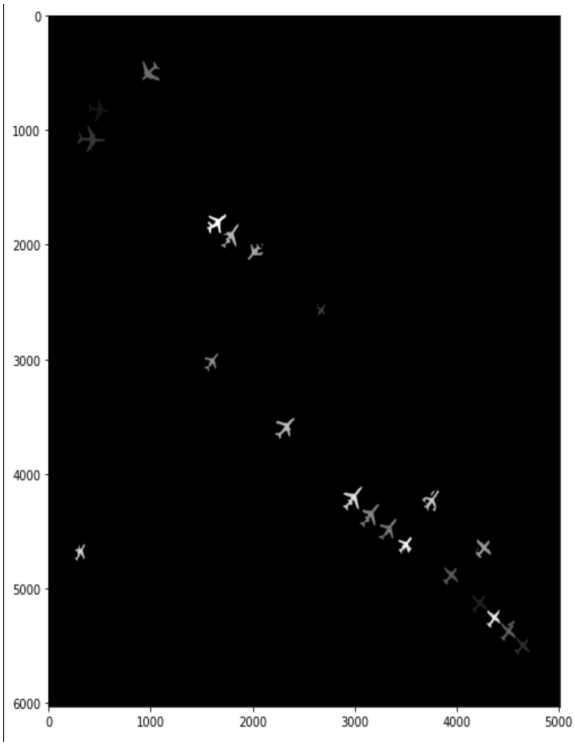


PART 3 :

Kaggle name : Naruzu
Best score : 0.37045

Visualization from test set :

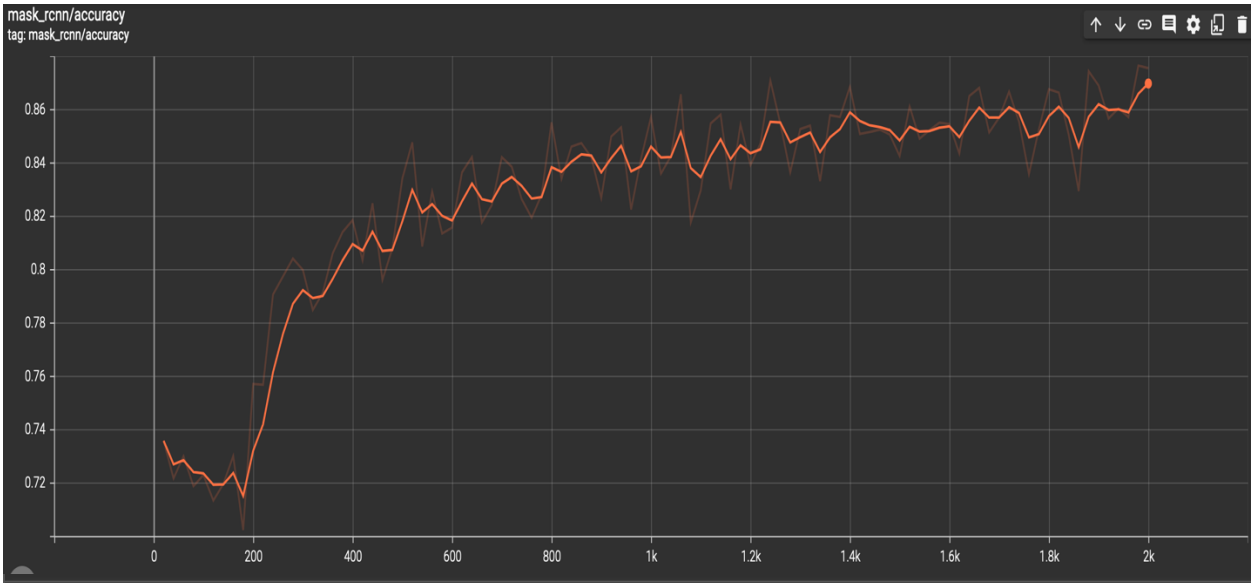

0
500
1000
1500
2000
2500
3000
0
1000
2000
3000
4000
5000
6000
7000
8000
0
500
1000
1500
2000
2500
3000
0
1000
2000
3000
4000
5000
6000
7000
8000

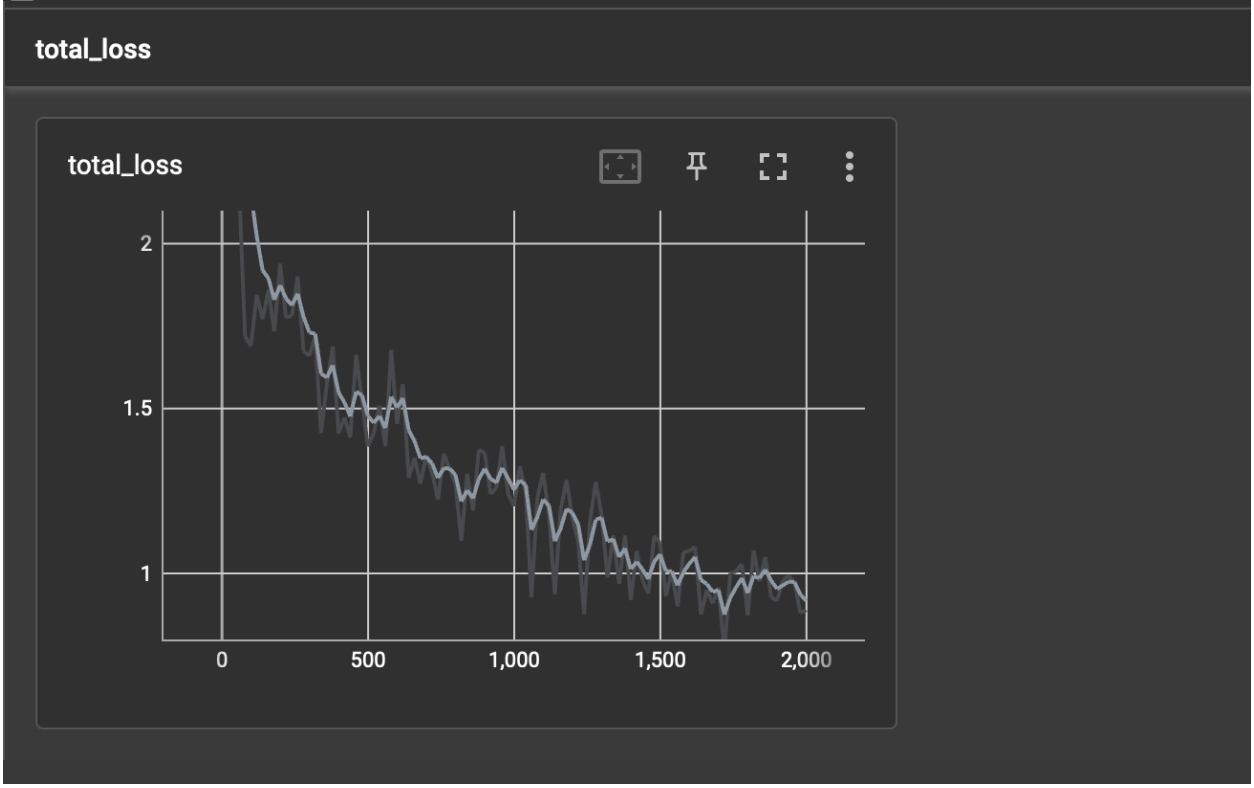





PART 4:

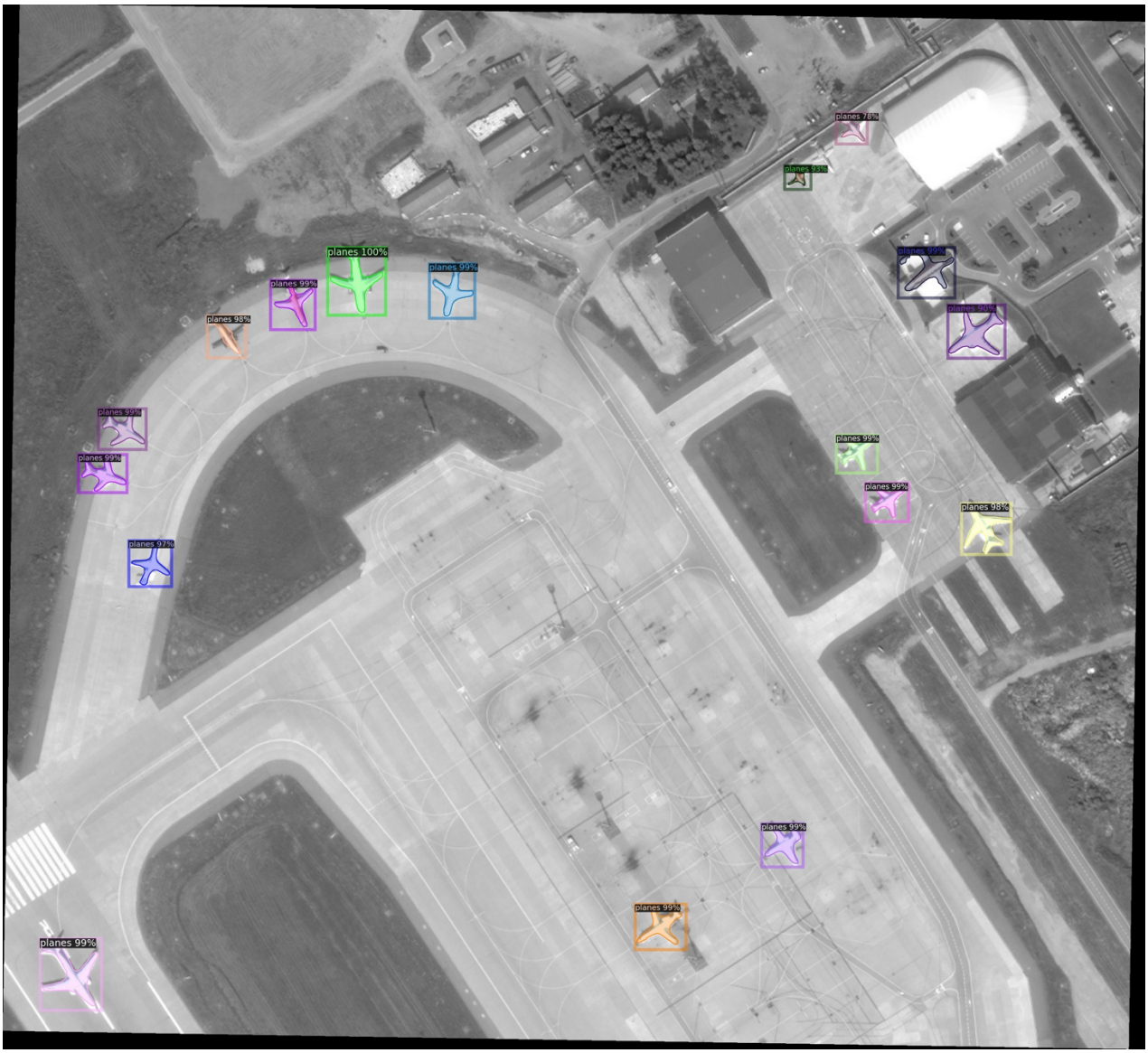
The configurations mentioned in Part 1 have been reused to establish a common ground for comparison between the models. Hence, the configurations and reasoning apply here as well.

Plots for total training loss and accuracy Mask RCNN

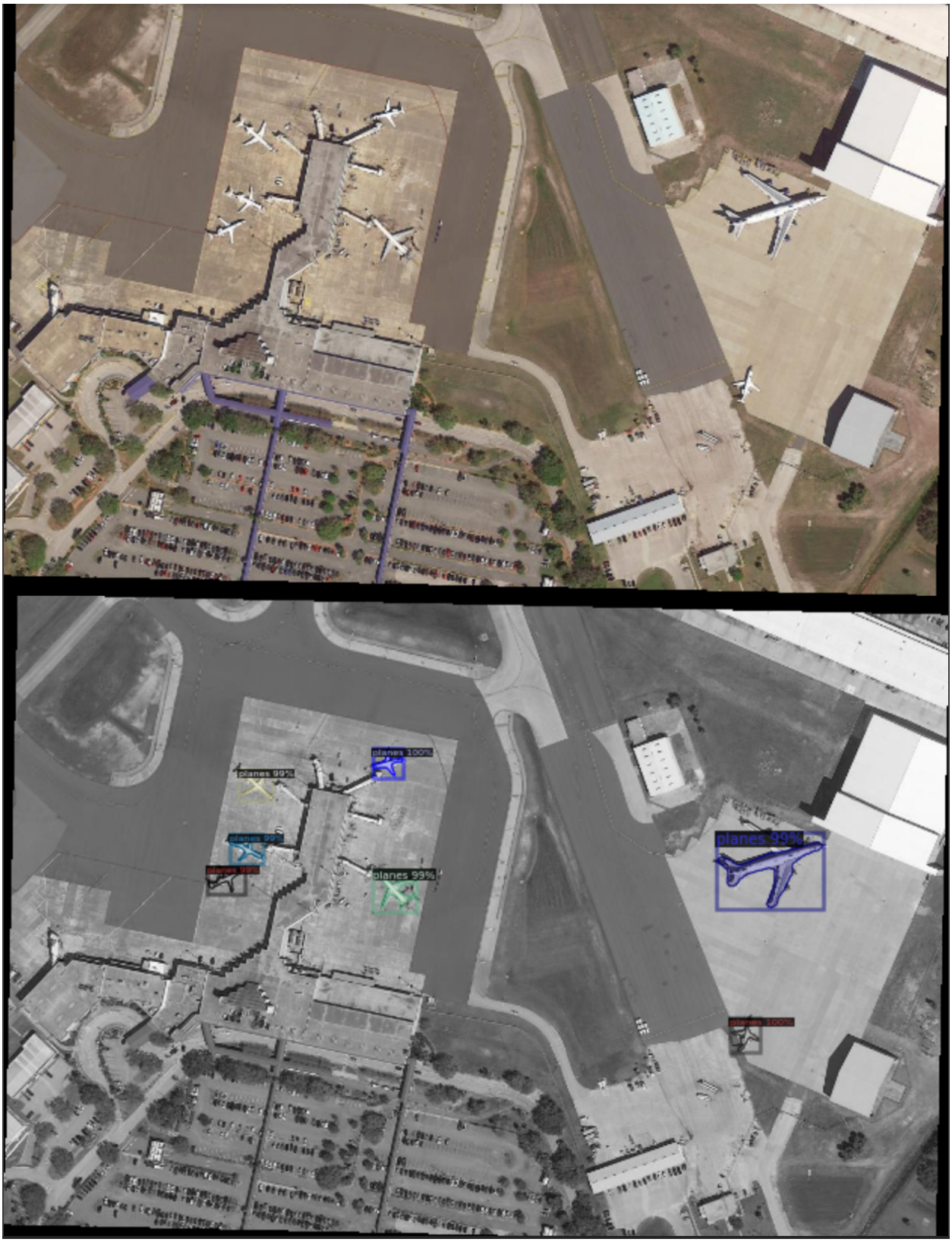




Visualization of 3 test samples :


planes 78%
planes 93%
planes 100%
planes 99%
planes 99%
planes 98%
planes 99%
planes 99%
planes 99%
planes 99%
planes 98%
planes 97%
planes 99%
planes 99%
planes 99%


planes 100%


planes 100%
planes 99%
planes 99%
planes 99%
planes 98%
planes 99%
planes 100%

Comparing both methods:

In the first method, we took a baseline model Faster RCNN and tried to fine tune the model specific to our plane dataset. We built a predictor to determine the bounding boxes for planes in an image. Later we built our own network (with an encoder decoder pattern) to predict the segmentation mask of planes in an image from scratch. In the second method, we used a pre-trained model for both the tasks – detection and seg. mask prediction. The first method allows us to fine tune to specific use cases, the way we want to, as we have the freedom to build on top of existing solutions and also configure the parameters to suit our experiments and resources. In the second method, these wont be possible. The second method is light weight, out of the box consumable and doesn't require too many resources and infrastructure whereas the first method does nt have these advantages.

AP: In terms of AP, both methods perform well. The AP for 2nd method was lesser than what we achieved in method 1 (AP 35+) due to the reasons mentioned above.

```
[11/01 19:04:07 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 31.850 | 50.333 | 36.088 | 19.915 | 41.055 | 62.872 |
Loading and preparing results...
```

Process : In our method, we build or finetune 2 diff models and hence to predict the instance segmentation, we need to first call the detection model to get bounding boxes and then use that information to predict the masks for the detected instances. This also gives us the opportunity to perform any intermediate changes between these phases which is not possible with 2nd method. In the second method, this happens internally.

Accuracies: In terms of accuracies from the graphs, they have reasonably same accuracy with method 1 having a slight advantage.

Training time: The 2nd method took far less training times than 1st method.

Results: As far the test samples and the AP scores go, there is not a huge difference between both methods as they are close enough and both have their pros and cons. I personally prefer method 1 if I have the infrastructure as I can fine tune to my needs and its slightly better in performance from the AP scores. If I don't have the infrastructure, then I would definitely prefer method 2.