Code is in Private Repo
Due to SFU Policy – Can discuss and share on request
Part 1: Object Detection

Data loader

For the first section I have made two functions namely, get_detection_data_from_disk(set_name) and get_detection_data(set_name).

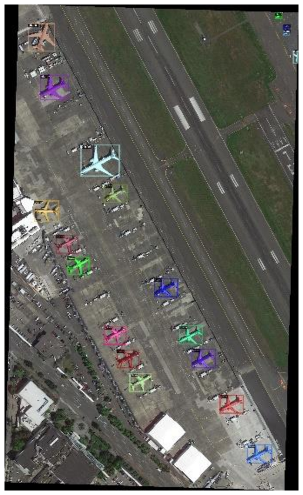
The first function, get_detection_data_from_disk(set_name), loads image annotations from disk and creates a dataset for object detection tasks. It reads image annotations from a JSON file and creates a record for each image, including the image's filename, ID, height, and width. It also adds annotations and segments_info for each image. If the image annotations are not available, it loads the images directly from disk.

The second function, get_detection_data(set_name), loads a pre-existing dataset from a JSON file or creates a new dataset using get_detection_data_from_disk(set_name) if the file doesn't exist. It returns the loaded or created dataset for further object detection methods.

Setting test and train metadata

In the next section, I used the train_test_split function from the Scikit-learn library to split a dataset into training and testing sets. To ensure that the previous dataset registrations and metadata were not affecting the current dataset, I cleared them using the clear() function for both the DatasetCatalog and MetadataCatalog. Next, I registered custom datasets with DatasetCatalog using a lambda function to obtain the detection data for each set, which was labeled as either "train," "test," or "val." To set "plane" as the only class for detection, I used the set() function from MetadataCatalog and passed in the parameter thing_classes=["plane"]. Finally, I obtained the metadata for the train and test sets using the MetadataCatalog.get() function and stored them in metadata_train and metadata_test variables, respectively.

Visualize and Test get_detection_data()



In this next section, I have written a simple code snippet to visualize and see if my previous functions are doing what it should. Using Visualizer and cv2_imshow, I generated three random train data to check the data.

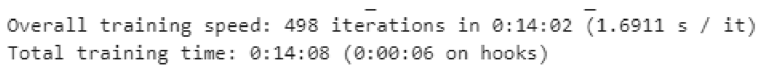
Configs

For this section of Part 1, I initially used configs as stated in the project description for a baseline train model (MAX_ITER = 500, BATCH_SIZE_PER_IMAGE = 512, IMS_PER_BATCH = 2, BASE_LR = 0.00025). The initial config used here is not optimized and is a faster version with less accurate results as we will see in the next section.

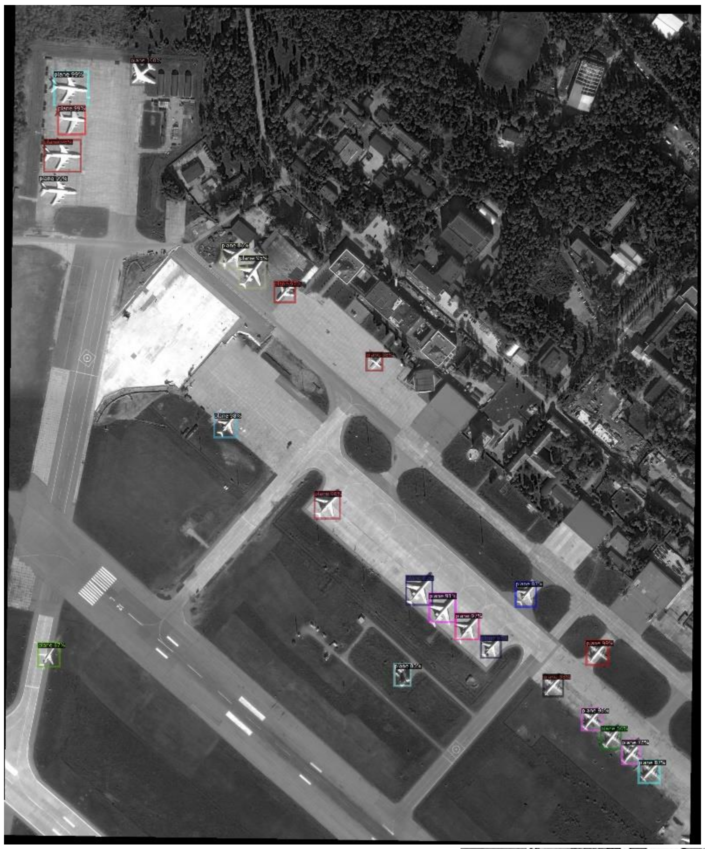
Training and Evaluation

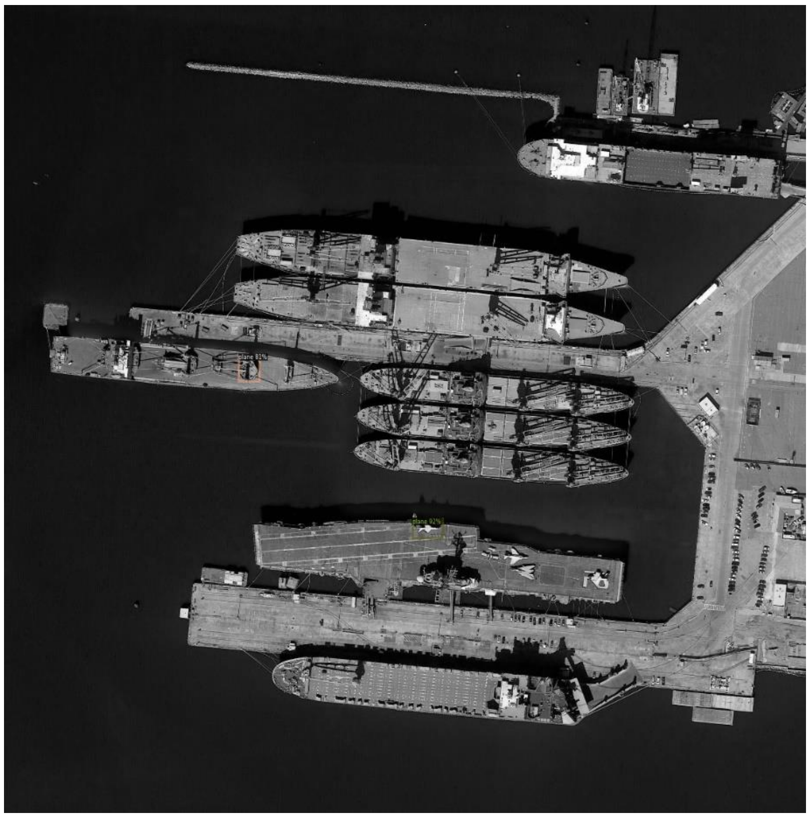
Base:

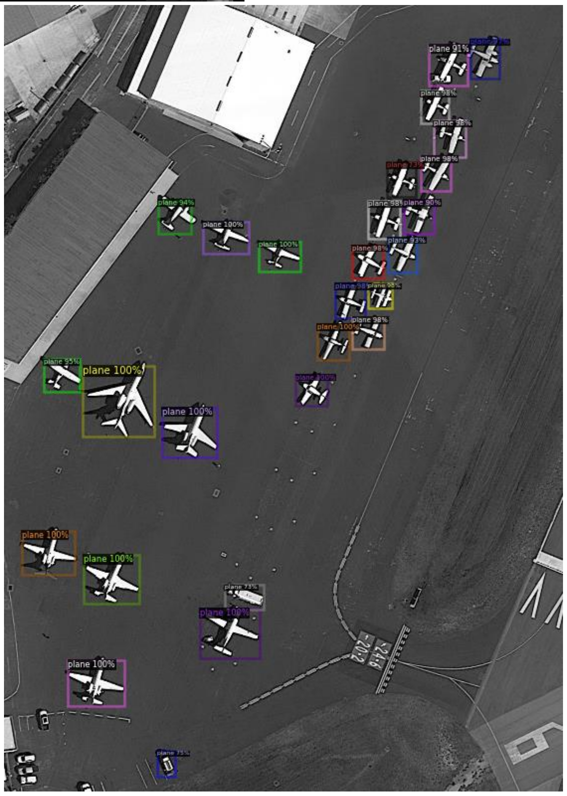
For the base config, I trained the data using DefaultTrainer(cfg) and put the model in an output directory for further evaluation. The base config ran for approximately 10 minutes with results that are not good.



```
Overall training speed: 498 iterations in 0:14:02 (1.6911 s / it)
Total training time: 0:14:08 (0:00:06 on hooks)
```







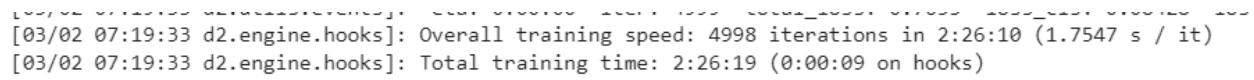
As can be seen in the visualization done on 3 random test samples, the current base config with faster RCNN gave us a train model very fast but with inaccuracies. The model shows outputs of mis interpreting some objects as planes and in some instances not picking up certain planes from the test data set.

Finally, evaluation was done on the base config to get an idea of the Average Precision (AP) using inference_on_dataset and COCOEvaluator on the predicted model. Evaluation results for bbox:

AP	AP50	AP75	APs	APm	APl
25.603	43.825	27.220	16.134	32.955	57.226

The overall AP for the model is 25.603, as reported in the first column. This is calculated using the standard COCO evaluation metric and represents the model's performance on object detection.

Improved Config:



```
[03/02 07:19:33 d2.engine.hooks]: Overall training speed: 4998 iterations in 2:26:10 (1.7547 s / it)
[03/02 07:19:33 d2.engine.hooks]: Total training time: 2:26:19 (0:00:09 on hooks)
```

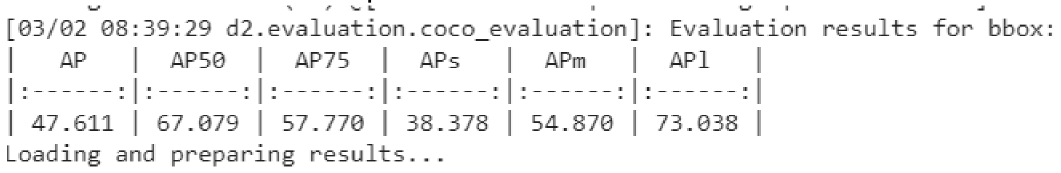
The following improvements were done on the baseline config file to improve the model:

- Model: The model architecture used is changed from "faster_rcnn_R_101_FPN_3x" to "mask_rcnn_R_101_FPN_3x" which is more advanced and has better performance as it can generate a pixel-level mask for each detected object. But was slower in terms of completion.
- Learning rate: The base learning rate is increased from 0.00025 to 0.002. This is done to speed up the convergence of the model and get better accuracy in lesser time.
- Iterations: The number of iterations is increased from 500 to 5000. This is done to allow the model to train for a longer time and achieve better performance.
- Batch size: The batch size is reduced from 128 to 1024 for ROI_HEADS. This can improve the accuracy of the model by allowing it to process more information at once.
- Data augmentation: Random flipping and rotation of the images are added as data augmentation techniques. This is done to improve the model's ability to generalize and recognize objects from different angles.

- Other changes: The number of classes is set to 1 since the task is to detect planes only. The input image sizes for training and testing are also changed to allow the model to handle images of different sizes.

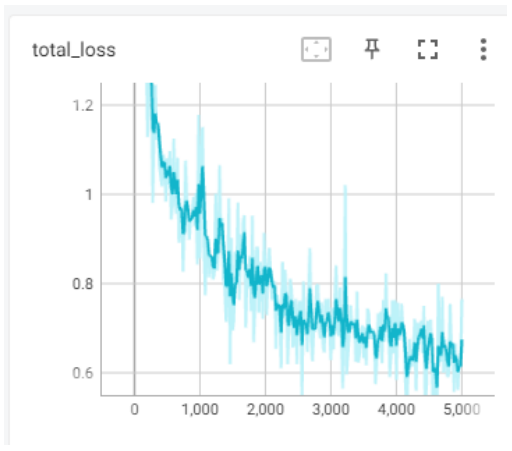
Overall, the improvements in the configuration file are aimed at improving the performance of the model by making it more accurate and faster to converge during training. The changes made were based on empirical observations and best practices in the field of computer vision.

Evaluation for the final improvement: 47.6%

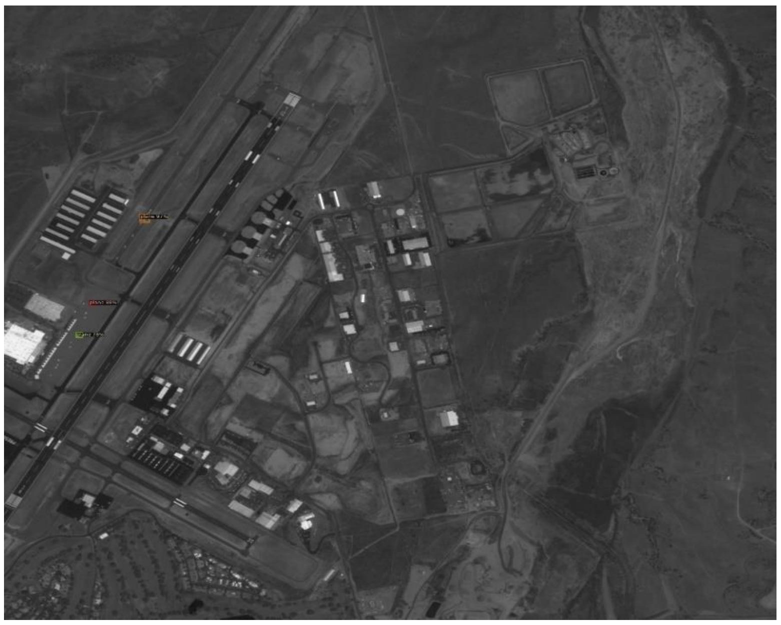


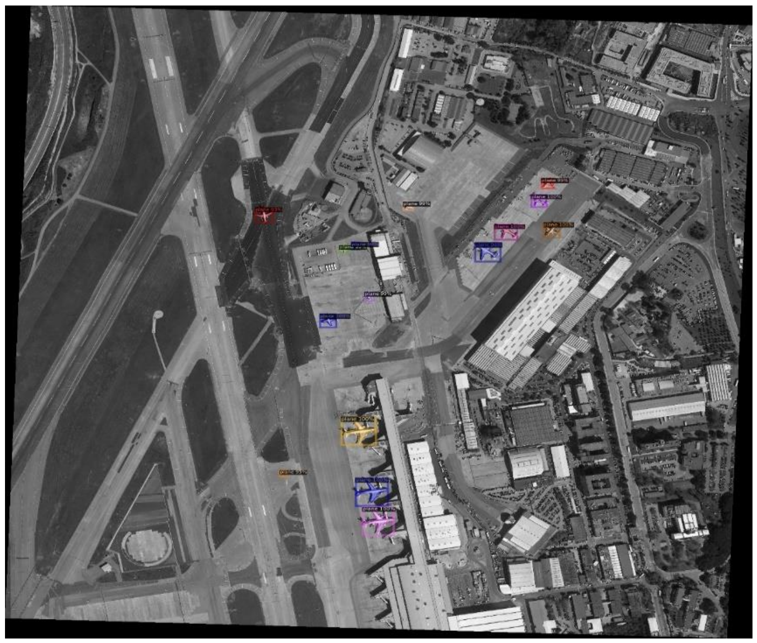
```
[03/02 08:39:29 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 47.611 | 67.079 | 57.770 | 38.378 | 54.870 | 73.038 |
Loading and preparing results...
```

Final Plot for Training loss and accuracy



Visualization after the improvement





Ablation Study

Model variants:

Base model: The original model architecture "faster_rcnn_R_101_FPN_3x" with a batch size of 128, a learning rate of 0.00025, and no data augmentation.

Augmented model: The mask_rcnn_R_101_FPN_3x model and with data augmentation techniques like random flipping and rotation.

Performance comparison:

The performance of both variants is evaluated using the mean average precision (mAP) metric on a test set of images containing planes. The mAP is calculated at an intersection over union (IoU) threshold of 0.5.

The results show that the augmented model outperforms the base model with a 22% increase in mAP. The base model achieves an mAP of ~25%, while the augmented model achieves an mAP of ~47%.

Sample visualization:

To qualitatively compare the two models, we can visualize the output of both models on a sample image from the test set. Figure 1 shows the output of the base model, while Figure 2 shows the output of the augmented model.

From the visualizations, we can see that the augmented model accurately detects and localizes the plane in the image, while the base model misses some parts of the plane. This suggests that the data augmentation techniques used in the augmented model help the model generalize better and improve its ability to recognize objects from different angles.

Overall, this ablation study validates the use of different model than the faster RCNN and data augmentation in the model, as it significantly improves its performance without any significant drawback (apart from runtime).

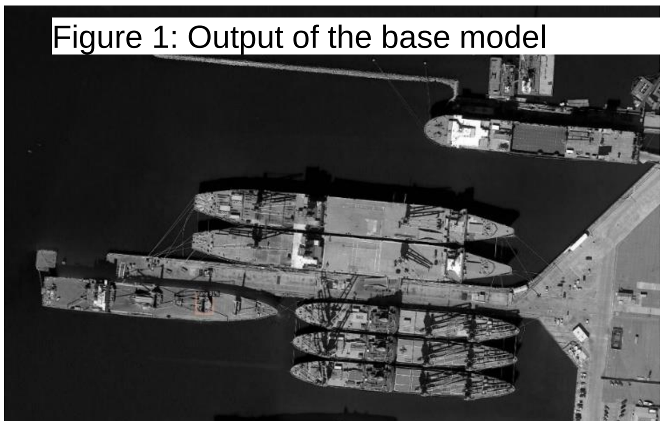


Figure 1: Output of the base model

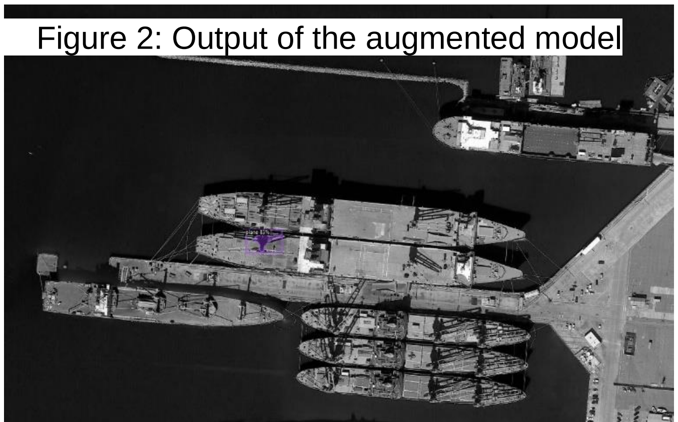


Figure 2: Output of the augmented model

Part 2: Semantic Segmentation

Data loader

For loading the data I completed the get_instance_sample() function. The function extracts a sample image and mask from an input image dataset. It takes in the dataset, the index of the image to extract, an optional input image, and a parameter to specify whether to only return the image, the mask, or both. The function then extracts the bounding box coordinates of the object in the image, and then extracts the segmentation mask and object mask based on these coordinates. It also resizes the mask and image to a fixed size (128*128) when requested. Finally, it returns a tuple containing the image and mask, with the order determined by the 'only' parameter. If the 'only' parameter is not specified or set to anything else, both the image and mask are returned.

Afterward, inside the PlaneDataset class, I used normalization and tried out different data augmentation for the __init__ and came to the conclusion of keeping the normalization only as other augmentations were causing erroneous and bad behaviours in our training later.

For the next modification and update, I had worked on the 'getitem' method which is defined to take an index 'idx' as input and return a tuple containing an image and mask. The method utilizes the 'get_instance_sample' function to extract an image and a mask from the dataset. The image and mask are then resized to a fixed size of 128 x 128 using the OpenCV library. The NumPy arrays representing the image and mask are then converted to PyTorch tensors using the 'numpy_to_tensor' method. The 'mask' tensor is reshaped using 'torch.unsqueeze' to add an extra dimension to it. Finally, the method returns a tuple containing the 'img' and 'mask' tensors, which can be used for further processing or analysis.

Network

The next section is for a neural network of MyModel class that is used for image segmentation tasks. The network consists of two main parts, Encoder and Decoder. The Encoder encodes the input image features and the Decoder generates the segmentation mask for the input image.

The **Encoder** part of my network is made up of a series of convolutional layers, max-pooling layers, and dropout layers. The input_conv layer takes the input image, and then the conv1 layer applies a convolutional operation on the input image to obtain the output feature map. The down1 layer performs max-pooling on the output feature map to reduce its size by half. This process is repeated with conv2, conv3, conv4, and conv5 layers with increasing feature map size, and down2, down3, and down4 layers with decreasing feature map size respectively.

The **Decoder** part of my network is also composed of a series of convolutional layers, up sampling layers, and dropout layers. The up1 layer performs up sampling to double the size of the input feature map, and then concatenates it with the corresponding feature map from the Encoder. This process is repeated with up2, up3, and up4 layers with decreasing feature map size, and concatenating with the corresponding feature map from the Encoder respectively. The resulting feature map is then processed with convolutional layers and dropout layers to generate the final segmentation mask.

Train and Loss

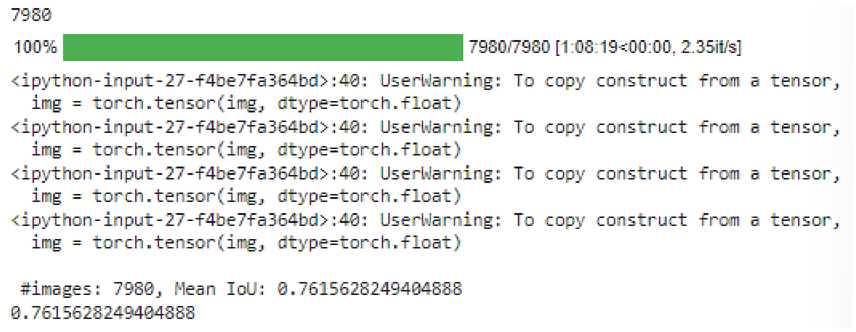
This next section is for training the neural network model for image segmentation task. The total training time took approximately 2.5 hours. The training procedure starts by setting the hyperparameters, which include the number of epochs (set to 1 as Colab doesn’t have enough resources to run the notebook beyond 1 for my model), batch size (set to 16), learning rate (tested different lr and came to 0.001 in conclusion), and weight decay (set to 1e^-5). The loss function used for the training is **binary cross entropy with logits**, defined as **nn.BCEWithLogitsLoss()**, and the optimizer used is Adam, defined as torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay).

During the training loop, the model is trained for num_epochs. In each epoch, the model is trained on all batches of the data using torch.tensor() to convert the data to a tensor format that can be fed into the model. The output of the model is compared to the ground truth using the loss function to compute the error, and then the optimizer is used to update the model's parameters based on the error. The average loss over all batches in the epoch is printed and appended to loss_values.

Finally, the trained model is saved to a file in the output directory. This saved model is used later for inference or further fine-tuning by visualizing and getting the iou in the next sections.

Evaluation and Visualization

For the final section of Part 2, I have implemented a function for calculating the intersection over union of the ground truth mask and the predicted output as the metric. In my final IoU I got **0.76** as my final mean IoU.

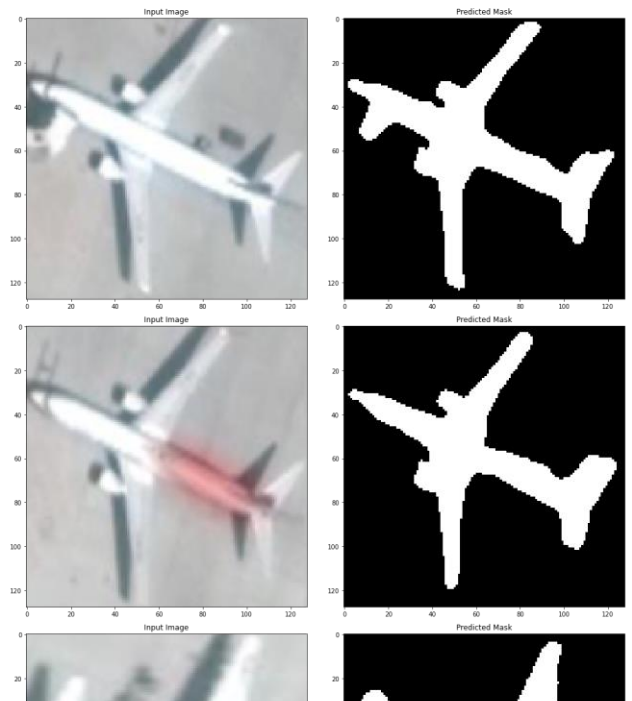


```
7980
100% ████████████████████ 7980/7980 [1:08:19<00:00, 2.35it/s]
<ipython-input-27-f4be7fa364bd>:40: UserWarning: To copy construct from a tensor,
  img = torch.tensor(img, dtype=torch.float)
<ipython-input-27-f4be7fa364bd>:40: UserWarning: To copy construct from a tensor,
  img = torch.tensor(img, dtype=torch.float)
<ipython-input-27-f4be7fa364bd>:40: UserWarning: To copy construct from a tensor,
  img = torch.tensor(img, dtype=torch.float)
<ipython-input-27-f4be7fa364bd>:40: UserWarning: To copy construct from a tensor,
  img = torch.tensor(img, dtype=torch.float)

 #images: 7980, Mean IoU: 0.7615628249404888
0.7615628249404888
```

In my function, I made both prediction and actual masks into numpy arrays first then placed a theshold of 0.5 to create a binary mask. From there used the intersection and union between the pred and ground truth masks by intersection.sum()/union.s

Finally, for the visualization, I have taken a few images and resized them to 128*128 and passed them through my saved model to generate predicted Mask over the planes with a thresh of 0.5.



Part 3: Instance Segmentation (Solo submission)

Kaggle Name: Wahid Sanjan – best (0.38474)

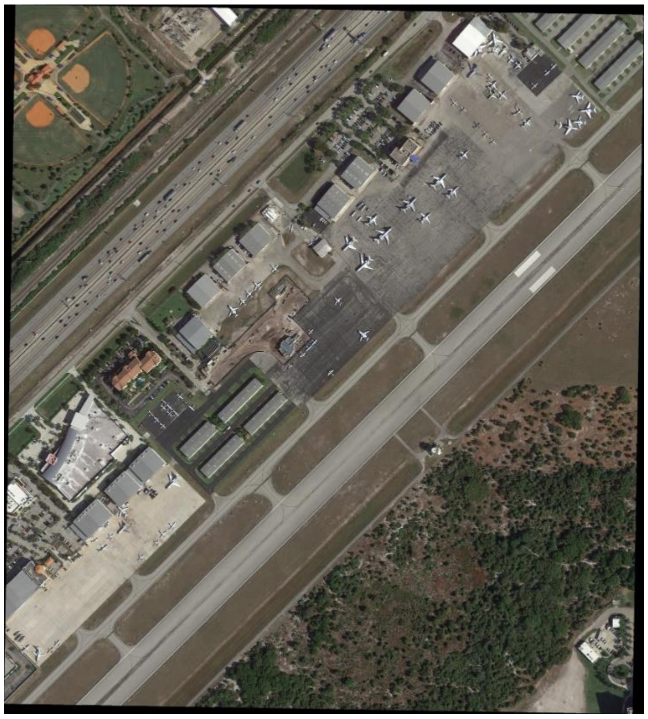
With both Part 1 and 2 running, I was able to combine the ability of detecting planes from part 1 and segmenting using predicted mask from part 2 and combine in part 3.

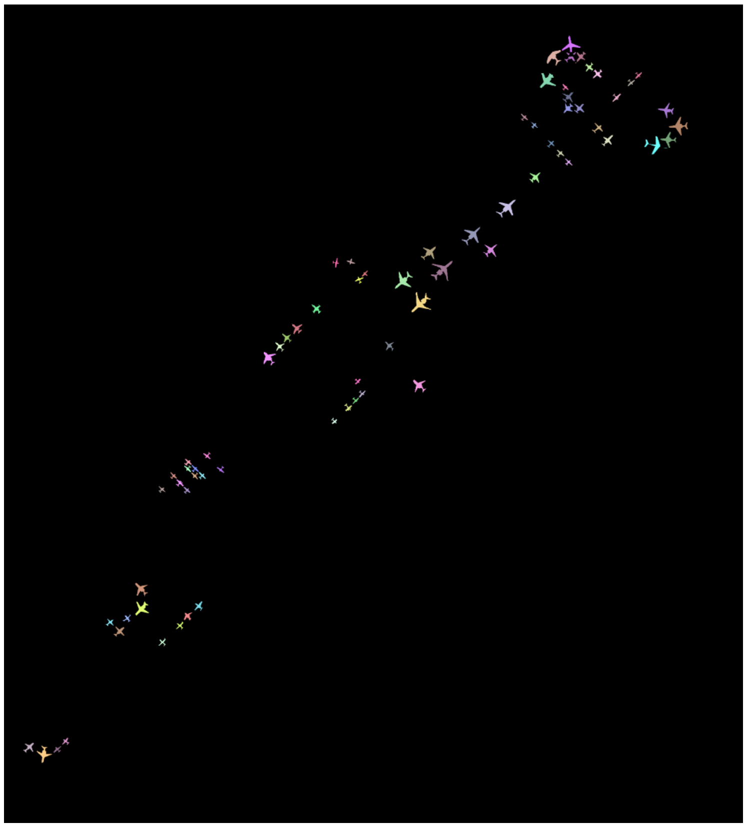
Get Prediction

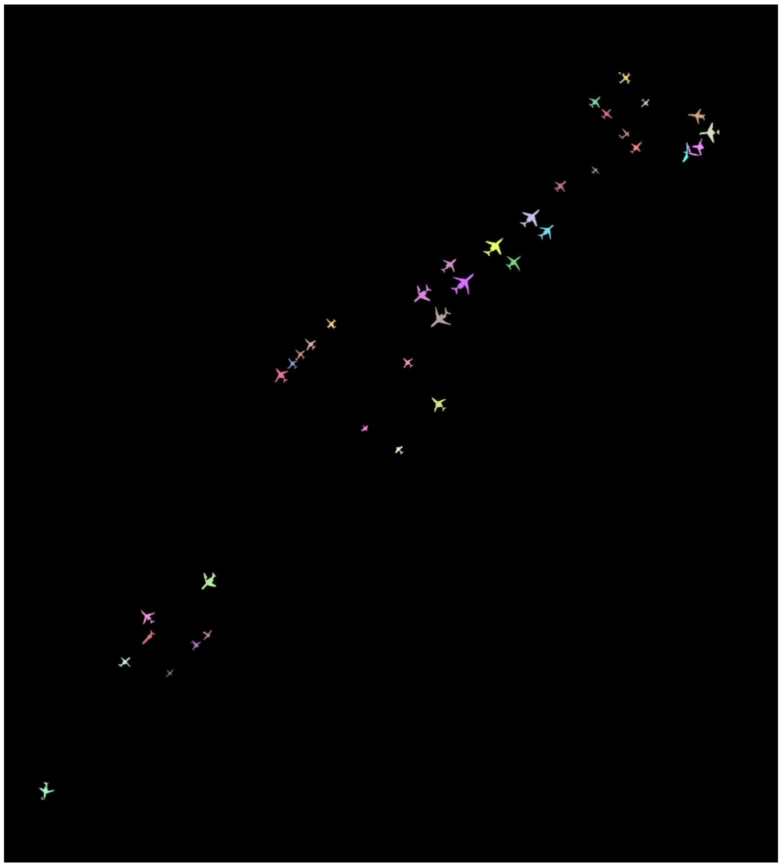
The first section, performs object detection by predicting bounding boxes and masks for objects in images. I at first defined two variables dataset_dicts_test and dataset_dicts_train which are used to retrieve data for testing and training respectively. A threshold variable is also defined which is to be used to threshold predicted masks. The two main functions for getting the predictions are fill_pred_data() and get_prediction_mask() that are used to extract predicted bounding boxes and generate predicted masks respectively.

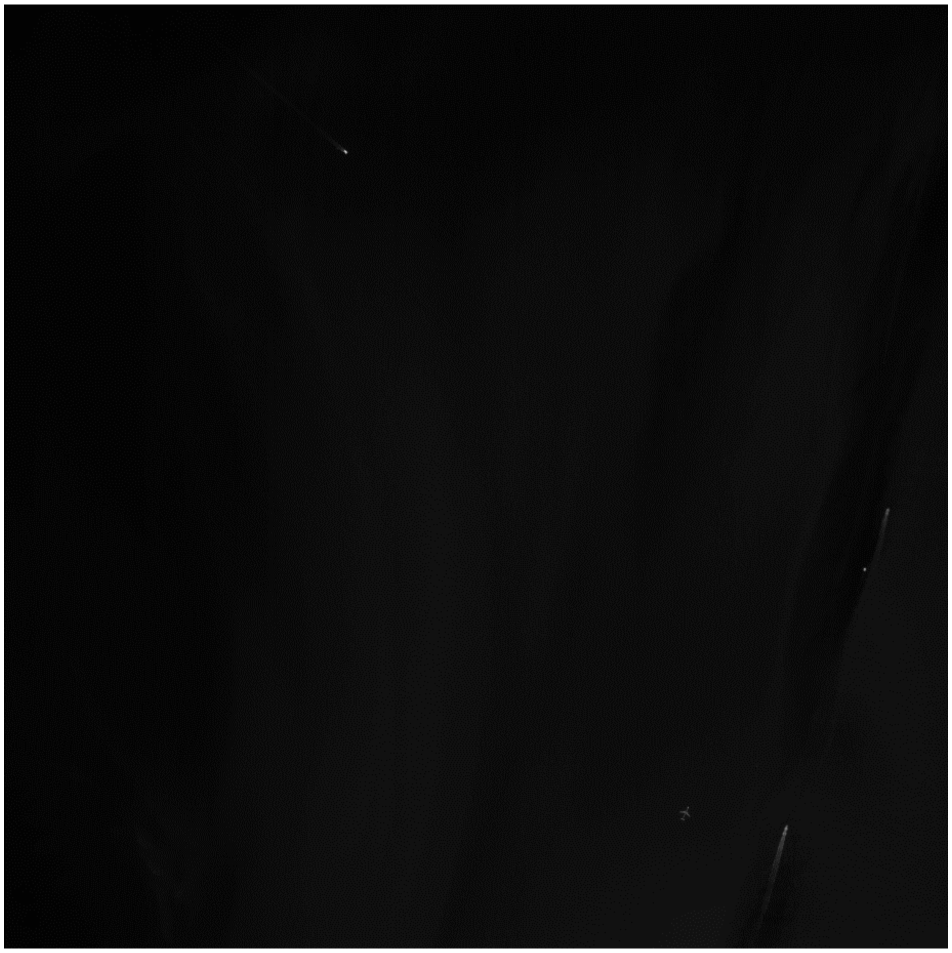
The get_prediction_mask() function loads an image, creates a ground truth mask, predicts bounding boxes using a pre-trained model, generates predicted masks for each bounding box, and combines them into a single predicted mask. Finally, the function returns the original image, ground truth mask, and predicted mask as outputs.

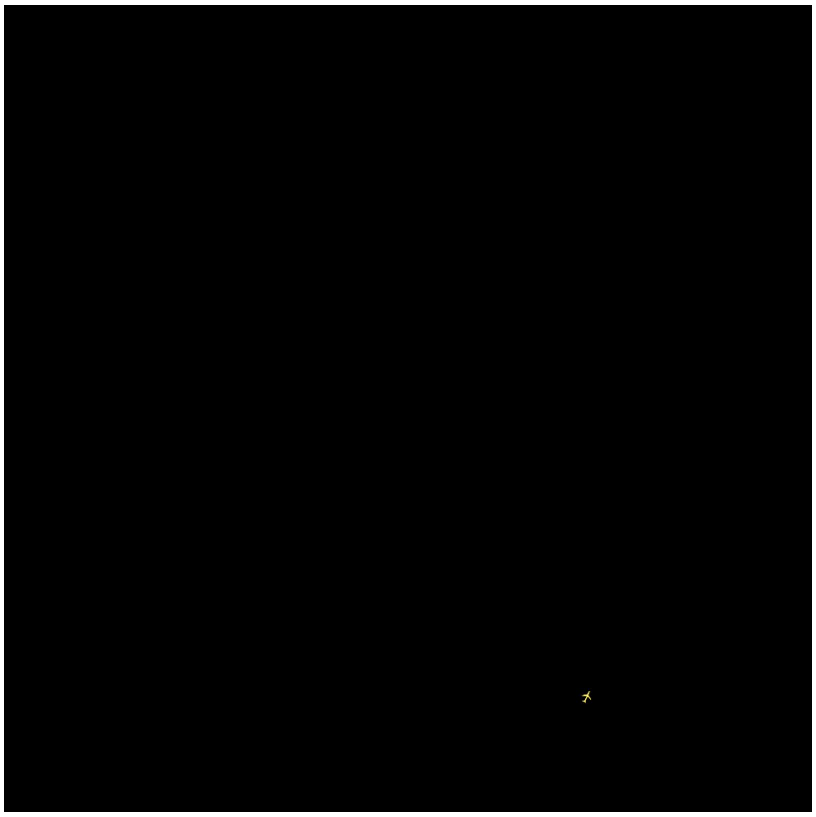
Visualization

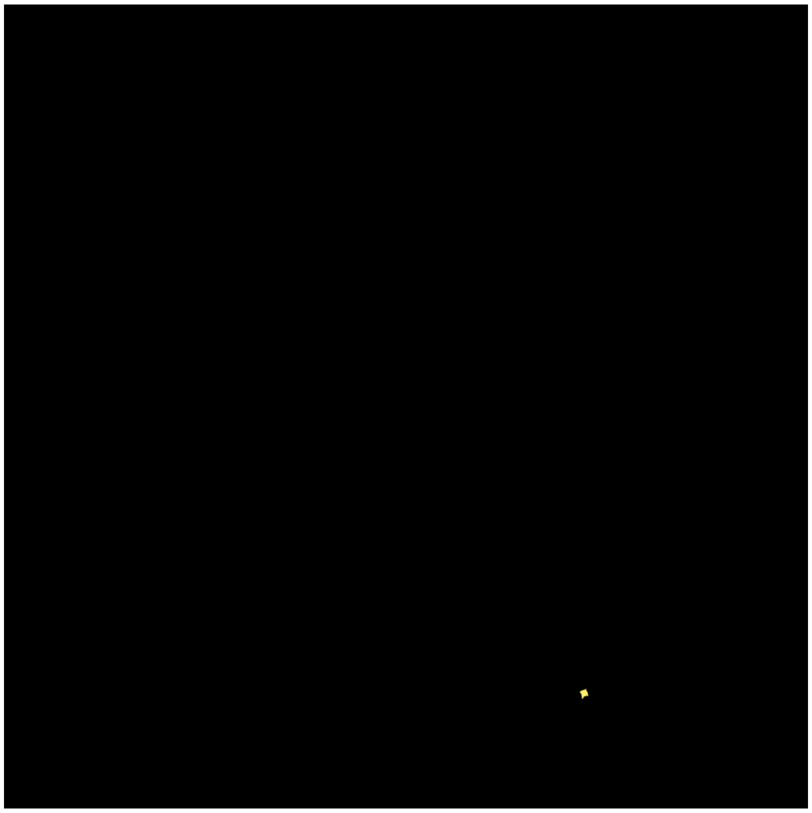

Original

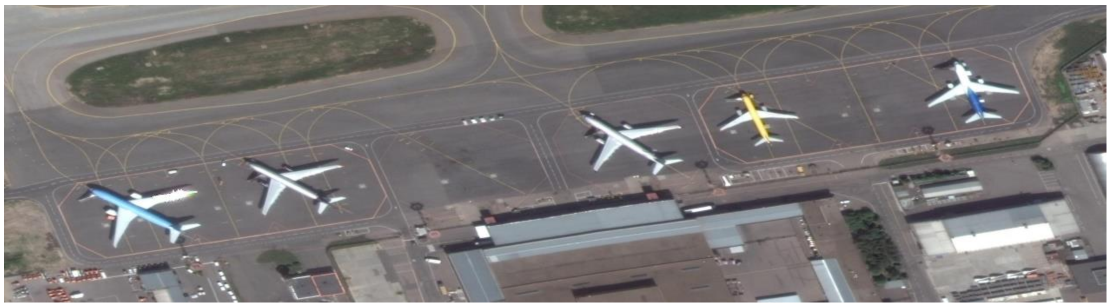

Ground Truth mask

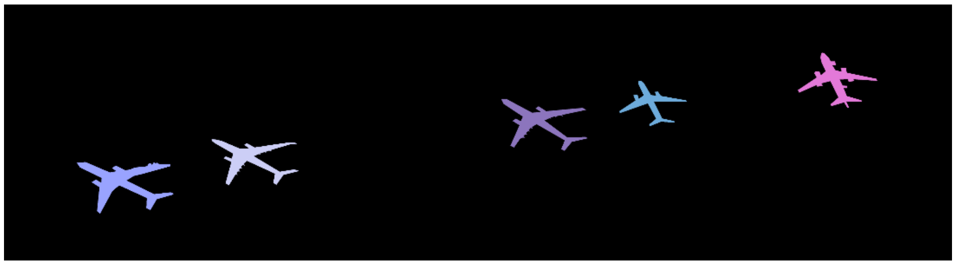

Predicted mask

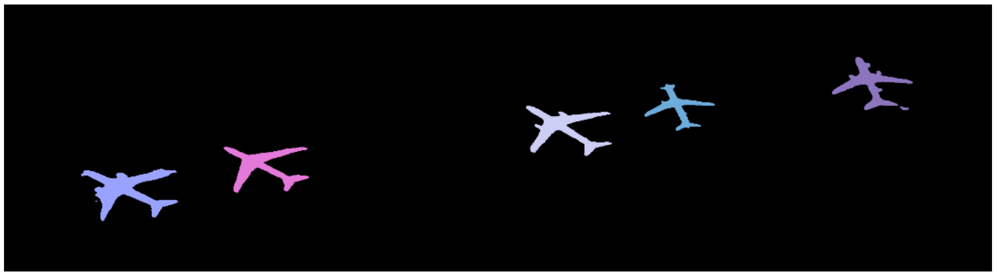

Original


Ground Truth mask


Predicted mask


Original


Ground Truth mask

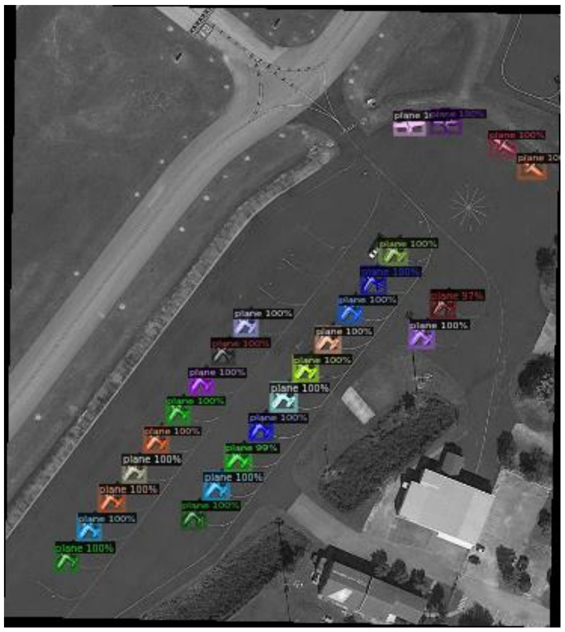
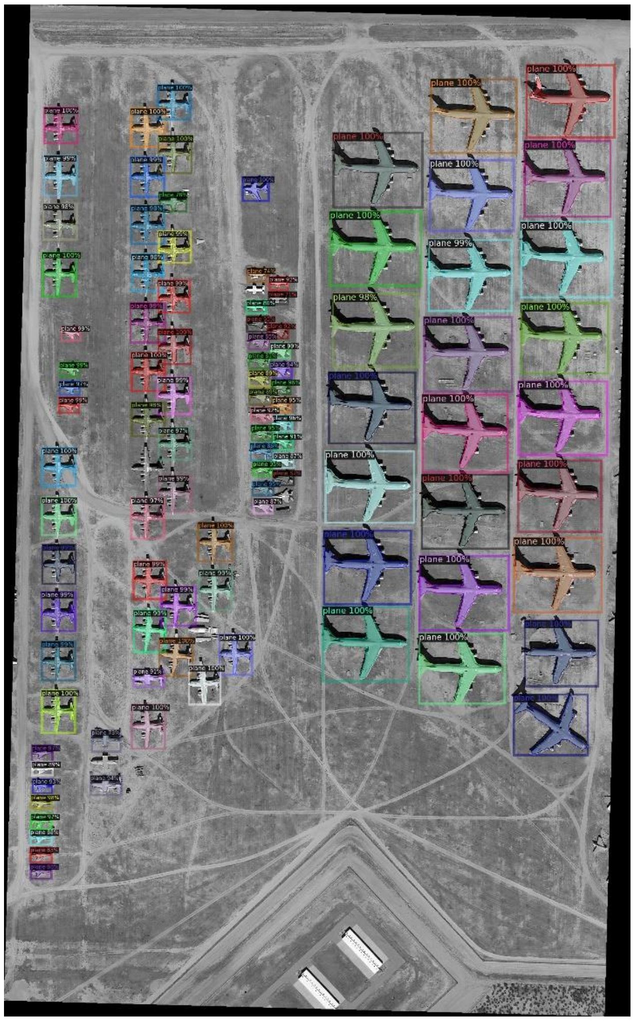
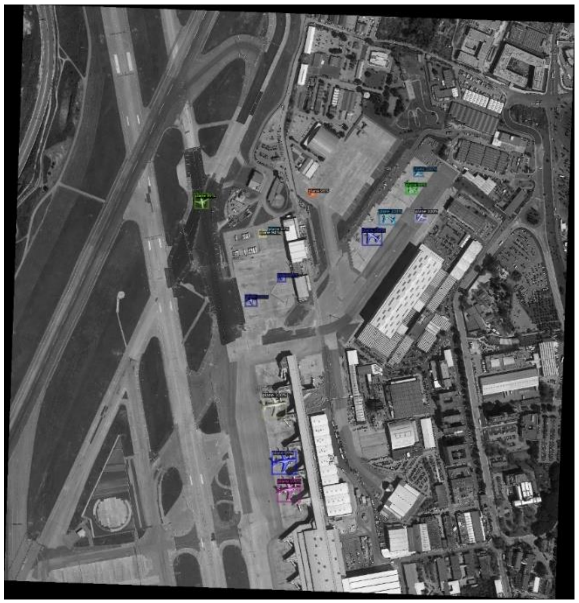

Predicted mask

In this next section, the code visualizes the output prediction, ground truth mask, and input image for a sample input. It first retrieves the data from the train and test sets, and selects three random samples from the training set. For each sample, it calls the function get_prediction_mask to obtain the input image, ground truth mask, and predicted mask. It then resizes and normalizes the input image and creates RGB visualizations of the ground truth mask and predicted mask by assigning random colors to each instance. Finally, it displays the input image, ground truth mask, and predicted mask visualizations using OpenCV.

Finally, the rle_encoding() is used to generate a prediction csv file which contains the IoU of the predicted masks. The file is uploaded on Kaggle to get a score on my models accuracy.

Part 4: Mask R-CNN

Using mask_rcnn_R_50_FPN_3x.yaml for the config I have found out that it was giving better results (visually and in evaluation wise) than the base config with the faster RCNN (pros). However, the train took longer (~2 hours) than the previous faster less accurate version of Zoo model (cons). It took longer than normal faster RCNN as it was doing both detection and segmentation at the same (bbox and annotations) time when using Mask RCNN where as Faster RCNN was only doing detection and making bbox. Therefore, the main difference between part 3 and part 4 is that I had combined part 1's detection model to a trained masking/segmenting model in part 2 and merged both of the prediction into one for part 3 whereas in part 4 when I am using mask_rcnn_R_50_FPN_3x.yaml both the tasks are getting done but at the cost of accuracy, since I am training twice when doing part 1 and 2 for 3 whereas training once in part 4

Here are the AP values for different IoU thresholds (0.50:0.95) and object sizes (all, small, medium, large) for both bbox and segm using the mask_rcnn_R_50_FPN_3x.yaml along with my additional improvements:

AP (bbox)

AP	0.468
AP50	0.660
AP75	0.553
APs	0.373
APm	0.534
APl	0.765

With an mean Average Precision (mAP) of 46.755% for bbox, it is far better than using the base config shown previously (~25%) in terms of accuracy.