Project 3 Object Detection, Semantic Segmentation, and Instance Segmentation

Wenbin(Freya) Li
Student ID: 301563887
Kaggle team name: Freya Li
Kaggle name: LIFREYA
Late day: used 4 late days

Part 1

The configuration

```
cfg.merge_from_file(model_zoo.get_config_file("COCO-
Detection/faster_rcnn_R_101_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("data_detection_train",)
cfg.DATASETS.TEST = ("data_detection_val",)

cfg.DATALOADER.NUM_WORKERS = 2
cfg.SOLVER.IMS_PER_BATCH = 4
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 2000
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 700
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Performance Improvement Factors

For the object detection task, I implemented a model using the faster_rcnn_R_101_FPN_3x configuration from the Detectron2 Model Zoo. This choice was motivated by its slightly higher box AP on the COCO dataset, indicating a more refined capability for accurate object localization. The model was fine-tuned on a custom dataset with the following significant settings:

Increased Iterations: The number of training iterations was set to 2000 to allow the model to converge properly. Early experimentation showed that the loss continued to decrease, suggesting that the model benefits from the additional training time to stabilize its loss.

Batch Size Per Image: The BATCH_SIZE_PER_IMAGE was increased to 700, which is above the default values. This allowed the model to have more examples to learn from during each step of the backpropagation, potentially improving the generalization capability on the object detection task.

Learning Rate: A conservative learning rate of 0.00025 was chosen to ensure that the model gradually adapts to the features of the dataset without overshooting the optimal values during training.

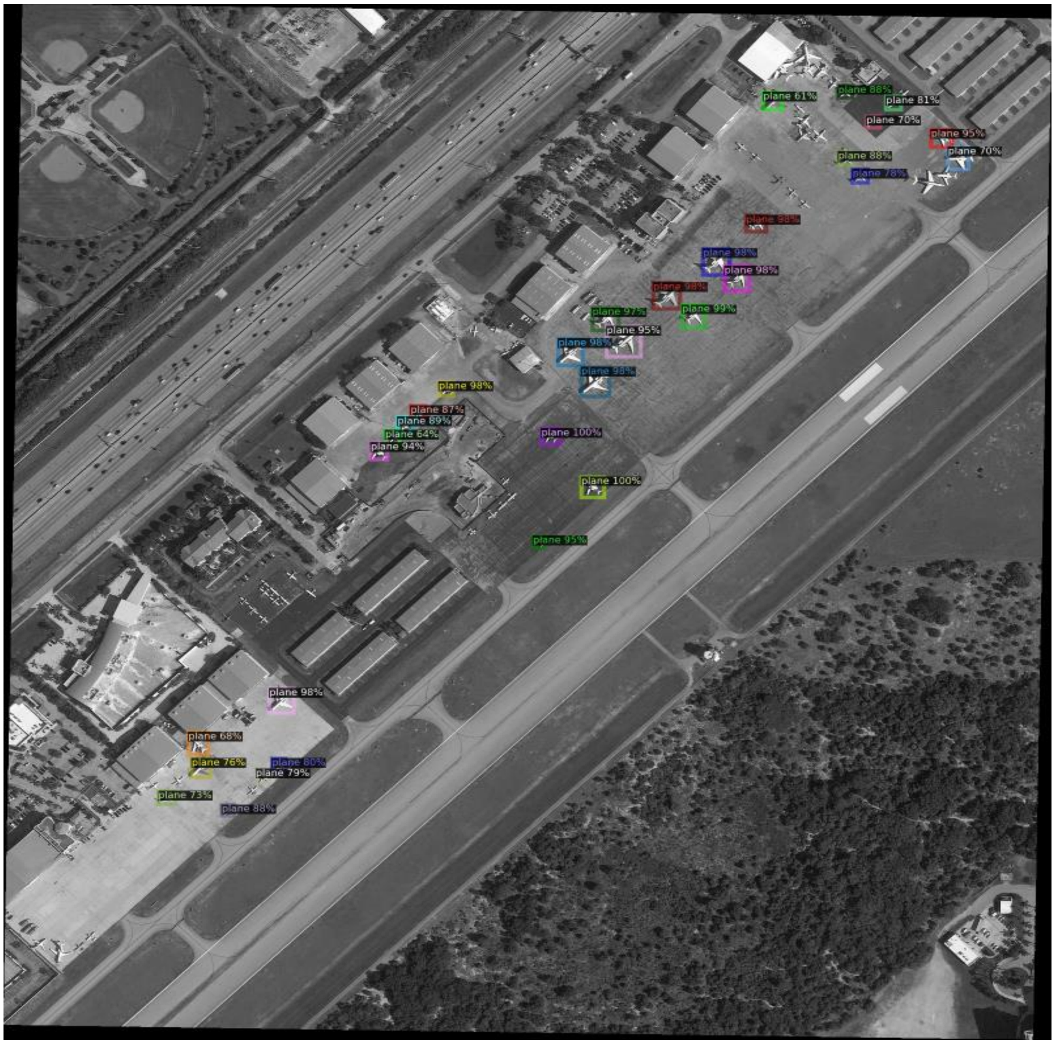
Training loss and accuracy

The training loss of last 300 iterations is shown in the notebook output.

```
: Attempting to copy inputs of <function pairwise_iou at 0x7f5caf0cf5b0> to CPU due to CUDA OOM
:  eta: 0:24:59  iter: 1699  total_loss: 0.8225  loss_cls: 0.1976  loss_box_reg: 0.3877  loss_rpn_cls: 0.06287  loss_rpn_loc: 0.188
:  eta: 0:23:17  iter: 1719  total_loss: 0.7927  loss_cls: 0.1884  loss_box_reg: 0.3798  loss_rpn_cls: 0.05591  loss_rpn_loc: 0.1899
:  eta: 0:21:32  iter: 1739  total_loss: 0.8075  loss_cls: 0.1803  loss_box_reg: 0.3502  loss_rpn_cls: 0.06243  loss_rpn_loc: 0.1776
:  eta: 0:20:01  iter: 1759  total_loss: 0.7377  loss_cls: 0.189  loss_box_reg: 0.3175  loss_rpn_cls: 0.0523  loss_rpn_loc: 0.1708
: Attempting to copy inputs of <function pairwise_iou at 0x7f5caf0cf5b0> to CPU due to CUDA OOM
:  eta: 0:18:22  iter: 1779  total_loss: 0.831  loss_cls: 0.1972  loss_box_reg: 0.3845  loss_rpn_cls: 0.05959  loss_rpn_loc: 0.1952
:  eta: 0:16:41  iter: 1799  total_loss: 0.7402  loss_cls: 0.1674  loss_box_reg: 0.3282  loss_rpn_cls: 0.0535  loss_rpn_loc: 0.1815
: Attempting to copy inputs of <function pairwise_iou at 0x7f5caf0cf5b0> to CPU due to CUDA OOM
:  eta: 0:15:04  iter: 1819  total_loss: 0.7456  loss_cls: 0.2005  loss_box_reg: 0.3827  loss_rpn_cls: 0.05302  loss_rpn_loc: 0.1754
:  eta: 0:13:21  iter: 1839  total_loss: 0.7998  loss_cls: 0.1864  loss_box_reg: 0.3451  loss_rpn_cls: 0.05987  loss_rpn_loc: 0.1942
:  eta: 0:11:40  iter: 1859  total_loss: 0.7665  loss_cls: 0.1869  loss_box_reg: 0.3541  loss_rpn_cls: 0.05981  loss_rpn_loc: 0.2071
:  eta: 0:10:00  iter: 1879  total_loss: 0.7953  loss_cls: 0.1989  loss_box_reg: 0.3471  loss_rpn_cls: 0.05042  loss_rpn_loc: 0.1896
:  eta: 0:08:20  iter: 1899  total_loss: 0.7723  loss_cls: 0.1733  loss_box_reg: 0.3261  loss_rpn_cls: 0.05728  loss_rpn_loc: 0.1874
: Attempting to copy inputs of <function pairwise_iou at 0x7f5caf0cf5b0> to CPU due to CUDA OOM
:  eta: 0:06:40  iter: 1919  total_loss: 0.8513  loss_cls: 0.1809  loss_box_reg: 0.3472  loss_rpn_cls: 0.07915  loss_rpn_loc: 0.1801
:  eta: 0:05:00  iter: 1939  total_loss: 0.7047  loss_cls: 0.1572  loss_box_reg: 0.3188  loss_rpn_cls: 0.04716  loss_rpn_loc: 0.1567
: Attempting to copy inputs of <function pairwise_iou at 0x7f5caf0cf5b0> to CPU due to CUDA OOM
:  eta: 0:03:20  iter: 1959  total_loss: 0.7939  loss_cls: 0.1841  loss_box_reg: 0.3648  loss_rpn_cls: 0.05749  loss_rpn_loc: 0.2224
:  eta: 0:01:39  iter: 1979  total_loss: 0.6926  loss_cls: 0.1756  loss_box_reg: 0.3156  loss_rpn_cls: 0.05292  loss_rpn_loc: 0.1841
:  eta: 0:00:00  iter: 1999  total_loss: 0.6992  loss_cls: 0.1791  loss_box_reg: 0.3269  loss_rpn_cls: 0.05216  loss_rpn_loc: 0.1666
: Overall training speed: 1998 iterations in 3:14:45 (5.8484 s / it)
```

```
Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 33.779 | 69.041 | 27.542 | 31.052 | 46.485 | 55.583 |
OrderedDict([('bbox', {'AP': 33.778596732062425, 'AP50': 69.0412020483395, 'AP75': 27.541609340337477, 'APs': 31.051925836967197, 'APm': 46.4849106334286, 'APl': 55.58307881760774})])
```

The visualization from test and predicted results:



Ablation Study

- Model configurations

I trained on two models for better performance.
When using ‘faster_rcnn_X_101_32x8d_FPN_3x’,

```
Evaluation results for bbox:
|  AP   |  AP50  |  AP75  |  APs  |  APm   |  APl  |
|:-----:|:------:|:------:|:-----:|:------:|:-----:|
| 8.903 | 28.347 | 2.089  | 9.739 | 14.117 | 5.227 |
OrderedDict([('bbox', {'AP': 8.90325322069778, 'AP50': 28.346611318078573,
'AP75': 2.088809884669159, 'APs': 9.738772069763893, 'APm':
14.11725261573524, 'APl': 5.2272683057082725})])
```

When using ‘faster_rcnn_R_101_FPN_3x.yaml’

```
Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl  |
|:------:|:------:|:------:|:------:|:------:|:-----:|
| 10.971 | 32.980 | 3.523  | 11.878 | 16.836 | 2.486 |
OrderedDict([('bbox', {'AP': 10.970560376877158, 'AP50': 32.97972097011354,
'AP75': 3.5233010035527257, 'APs': 11.878112980779447, 'APm':
16.835641593976817, 'APl': 2.4858007983235493})])
```

Performance comparison:

The Average Precision (AP) metric is commonly used to evaluate object detection models. AP computes the average precision value for recall value over 0 to 1. The higher the AP, the better the model is performing. AP can be computed at different Intersection over Union (IoU) thresholds, commonly at 0.5 (AP50) and 0.75 (AP75), and for different object sizes (small, medium, large - APs, APm, APl).

Here are the AP results for the two models:

For the first model (faster_rcnn_R_101_FPN_3x.yaml):

AP: 10.971
AP50: 32.980
AP75: 3.523
APs: 11.878
APm: 16.836
APl: 2.486
For the second model (faster_rcnn_X_101_32x8d_FPN_3x.yaml):

AP: 8.903
AP50: 28.347
AP75: 2.089
APs: 9.739
APm: 14.117
APl: 5.227

Comparing the overall AP values, the first model has a higher AP (10.971) compared to the second model (8.903), indicating that the first model generally performs better across all IoU thresholds.

Moreover, the first model also has higher AP50 and AP75 scores, suggesting it is better at detecting objects with both loose and strict overlap criteria.

However, if you consider object sizes, the first model performs better on small and medium objects (APs and APm), while the second model performs slightly better on large objects (APl).

In conclusion, based on the AP metrics, the first model (faster_rcnn_R_101_FPN_3x.yaml) is better overall. But if your use case prioritizes detecting larger objects, the second model might be preferable despite its lower overall AP.

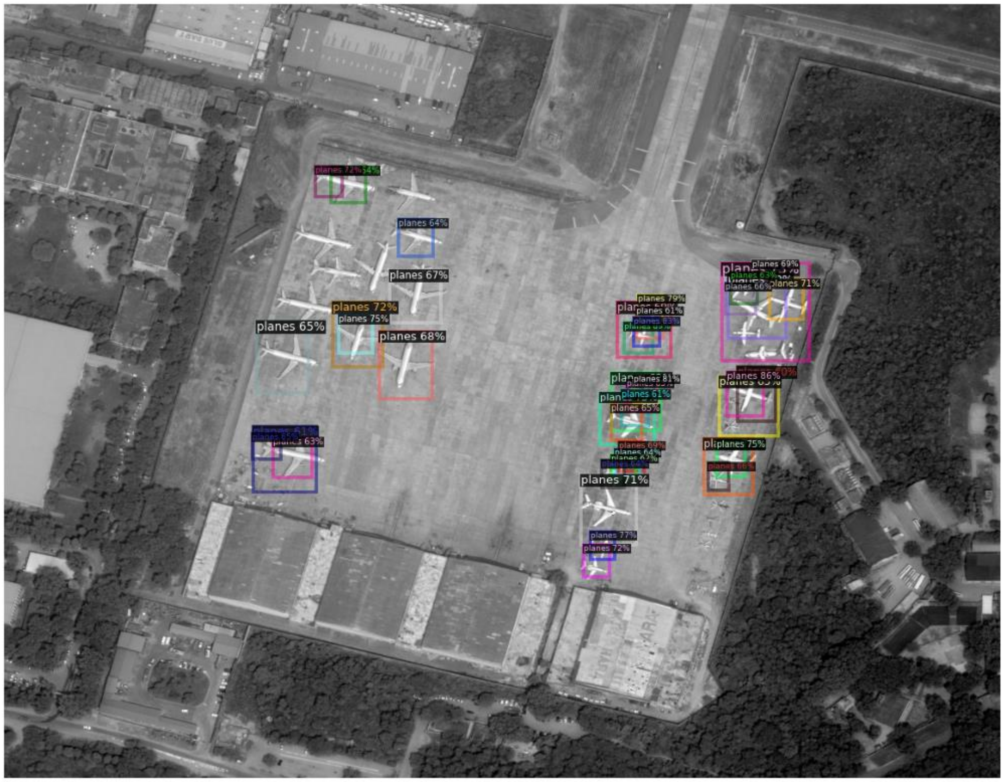
- Maximum Iterations
  Allowing the model to train for more iterations gave it sufficient time to fine-tune the weights on the new dataset. The model accuracy continued to increase as I set the max iterations to 300, 500, 1000, 2000, separately.

<u>Visualization on 300 max iterations, model ‘faster_rcnn_X_101_32x8d_FPN_3x’</u>

Configurations:

```
cfg.merge_from_file(model_zoo.get_config_file("COCO-
Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 300
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Visualization on 300 max iterations, model ‘faster_rcnn_R_101_FPN_3x’



Configurations:

```
cfg.merge_from_file(model_zoo.get_config_file("COCO-
Detection/faster_rcnn_R_101_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 300
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Visualization on 500 max iterations, model ‘faster_rcnn_ X_101_32x8d_FPN_3x’

Configurations:

```
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 500
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Part 2

Parameters:

```
num_epochs = 1000
```

```
batch_size = 4
learning_rate = 1e-2
weight_decay = 1e-5
```

Final Network Architecture:

Layer/Component	Description	Kernel size	Stride	Input Channels	Layer/Component
Input Conv	Conv+Bn+ReLU	3*3	1	3	Input Conv
Down1	Conv+Bn+ReLU + MaxPool	-	-	64	Down1
Conv1	Conv+Bn+ReLU	3*3	1	128	Conv1
Down2	Conv+Bn+ReLU + MaxPool	-	-	128	Down2
Conv2	Conv+Bn+ReLU	3*3	1	256	Conv2
Down3	Conv+Bn+ReLU + MaxPool	-	-	256	Down3
Conv3	Conv+Bn+ReLU	3*3	1	512	Conv3
Down4	Conv+Bn+ReLU + MaxPool	-	-	512	Down4
Up1	Upsample + Conv+Bn+ReLU	-	-	1024 (from Down4 and skip connection)	Up1
Conv4	Conv+Bn+ReLU	3*3	1	256	Conv4
Up2	Upsample + Conv+Bn+ReLU	-	-	512 (from Conv4 and skip connection)	Up2
Conv5	Conv+Bn+ReLU	3*3	1	128	Conv5
Up3	Upsample + Conv+Bn+ReLU	-	-	256 (from Conv5 and skip connection)	Up3
Conv6	Conv+Bn+ReLU	3*3	1	64	Conv6
Up4	Upsample + Conv+Bn+ReLU	-	-	128 (from Conv6 and skip connection)	Up4
Output Conv	Conv	3*3	1	64	Output Conv

Encoder:

Input Convolution (input_conv): 3 input channels to 64 output channels.
Downsampling Blocks (down1, down2, down3, down4): Sequential layers with doubling channel dimensions (64 to 512). Each downsampling block is followed by an additional convolutional layer, maintaining the same number of channels. These additional convolutions (conv1, conv2, conv3) are intended to refine the feature maps after each downsampling, potentially capturing more complex patterns before further reduction in spatial dimensions.

Decoder:

Upsampling Blocks (up1, up2, up3, up4): These layers upsample the feature maps and concatenate them with the corresponding feature maps from the encoder, following the U-Net architecture pattern. The channel dimensions are halved sequentially (1024 to 64). After each upsampling and concatenation, additional convolutional layers (conv4, conv5, conv6) are included to further process the combined feature maps. The rationale for these layers is similar to the encoder: to refine features at each scale of the decoder before the final output layer.
Output Convolution (outc): The final convolutional layer that maps the 64 feature channels to the single output channel for mask prediction. This layer remains unchanged, as its purpose is to generate the final prediction mask.

Modifications added apart from upsampling and downsampling:

Enhanced Feature Processing: The intermediate convolutions added after each downsampling and before each upsampling are designed to enhance the processing of features. This allows the network to potentially learn more intricate details at each level of the feature hierarchy.

Increased Model Capacity: More convolutional layers introduce additional parameters, allowing the network to represent more complex functions. This can improve the model's ability to learn from the data, provided there is enough data to prevent overfitting.

The modifications have led to an improved prediction score from 0.45544 to 0.55123 on Kaggle, suggesting that the additional layers are beneficial in capturing the nuances of the segmentation task more effectively.

Training Loss

Loss function:
nn.BCEWithLogitsLoss()

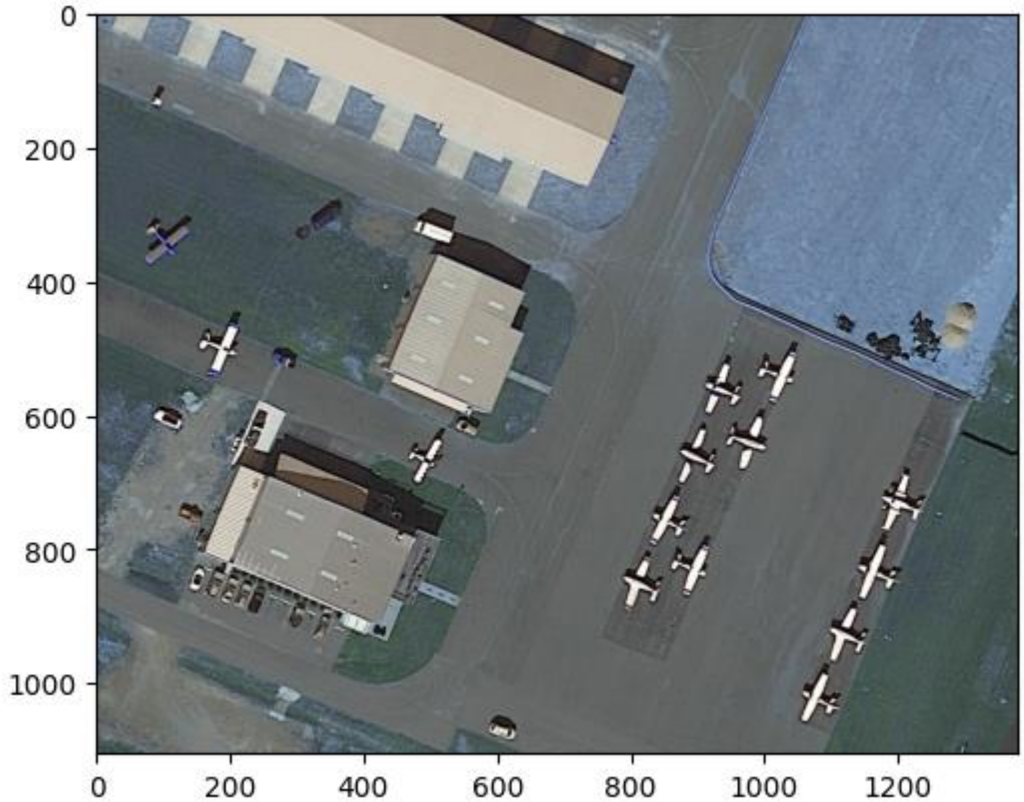
Followed are screenshots of training loss for the first eight and the last eight epochs.

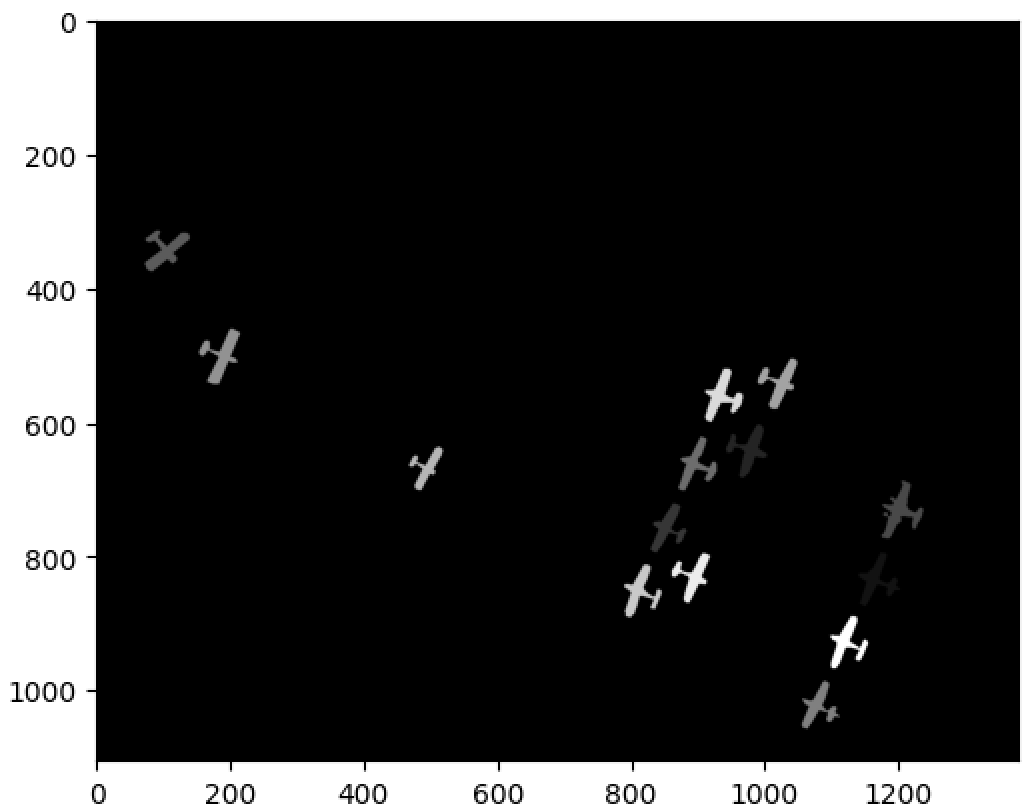
```
99% 988/1000 [10:58:27<08:10, 40.88s/it]
100% 1995/1995 [00:39<00:00, 51.42it/s]
<ipython-input-18-0718f663f380>:27: UserWarning: To copy construct from a tensor
  mask = torch.tensor(mask, device=torch.device('cuda'), requires_grad = True)
Epoch: 0, Loss: 0.2715514302253723
100% 1995/1995 [00:39<00:00, 52.11it/s]
Epoch: 1, Loss: 0.2338429093360901
100% 1995/1995 [00:39<00:00, 51.94it/s]
Epoch: 2, Loss: 0.22646574676036835
100% 1995/1995 [00:39<00:00, 50.48it/s]
Epoch: 3, Loss: 0.2256052941083908
100% 1995/1995 [00:40<00:00, 51.18it/s]
Epoch: 4, Loss: 0.2179722934961319
100% 1995/1995 [00:39<00:00, 51.04it/s]
Epoch: 5, Loss: 0.2159695029258728
100% 1995/1995 [00:39<00:00, 51.65it/s]
Epoch: 6, Loss: 0.23614364862442017
100% 1995/1995 [00:39<00:00, 52.39it/s]
Epoch: 7, Loss: 0.22967232763767242
100% 1995/1995 [00:39<00:00, 51.98it/s]
Epoch: 8, Loss: 0.22536417841911316
```

```
Epoch: 977, Loss: [illegible]
100% 1995/1995 [00:40<00:00, 50.43it/s]
Epoch: 978, Loss: 0.10472418367862701
100% 1995/1995 [00:40<00:00, 50.66it/s]
Epoch: 979, Loss: 0.10485922545194626
100% 1995/1995 [00:40<00:00, 50.67it/s]
Epoch: 980, Loss: 0.10514947026968002
100% 1995/1995 [00:39<00:00, 50.80it/s]
Epoch: 981, Loss: 0.10510682314634323
100% 1995/1995 [00:39<00:00, 51.62it/s]
Epoch: 982, Loss: 0.10442555695772171
100% 1995/1995 [00:39<00:00, 50.23it/s]
Epoch: 983, Loss: 0.1047881543636322
100% 1995/1995 [00:40<00:00, 49.85it/s]
Epoch: 984, Loss: 0.10461023449897766
100% 1995/1995 [00:41<00:00, 50.07it/s]
Epoch: 985, Loss: 0.10441111773252487
100% 1995/1995 [00:41<00:00, 48.03it/s]
Epoch: 986, Loss: 0.10428806394338608
100% 1995/1995 [00:41<00:00, 48.25it/s]
Epoch: 987, Loss: 0.10402864962816238
100% 1995/1995 [00:41<00:00, 47.72it/s]
```

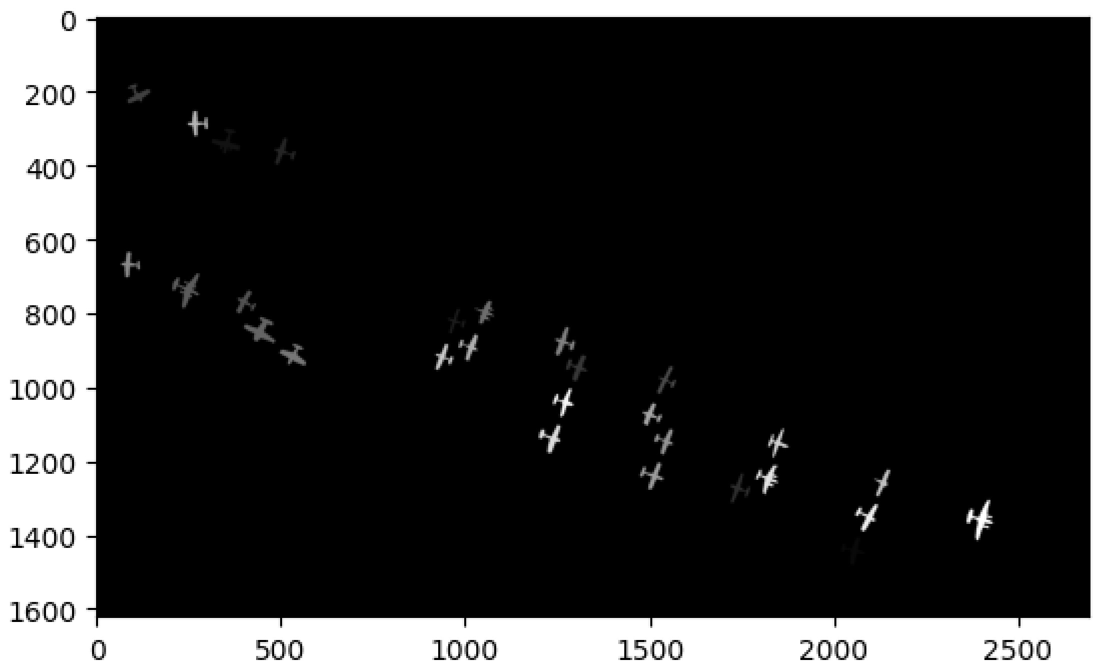
Accuracy:
The final IoU is: `Mean IoU: 0.8550710928283288`

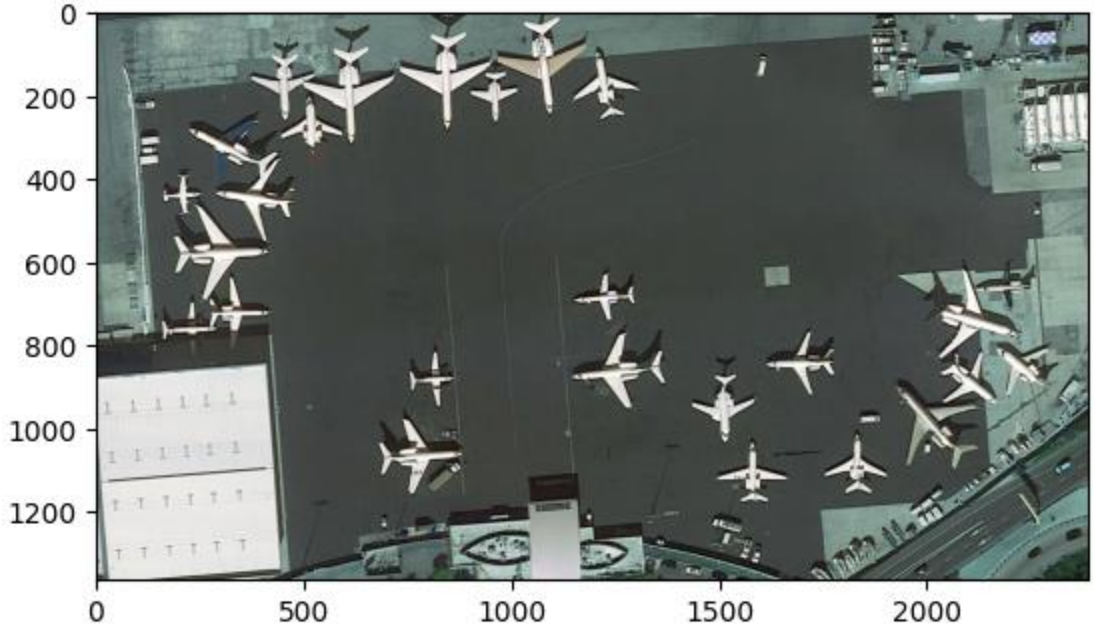
Visualization of three random images

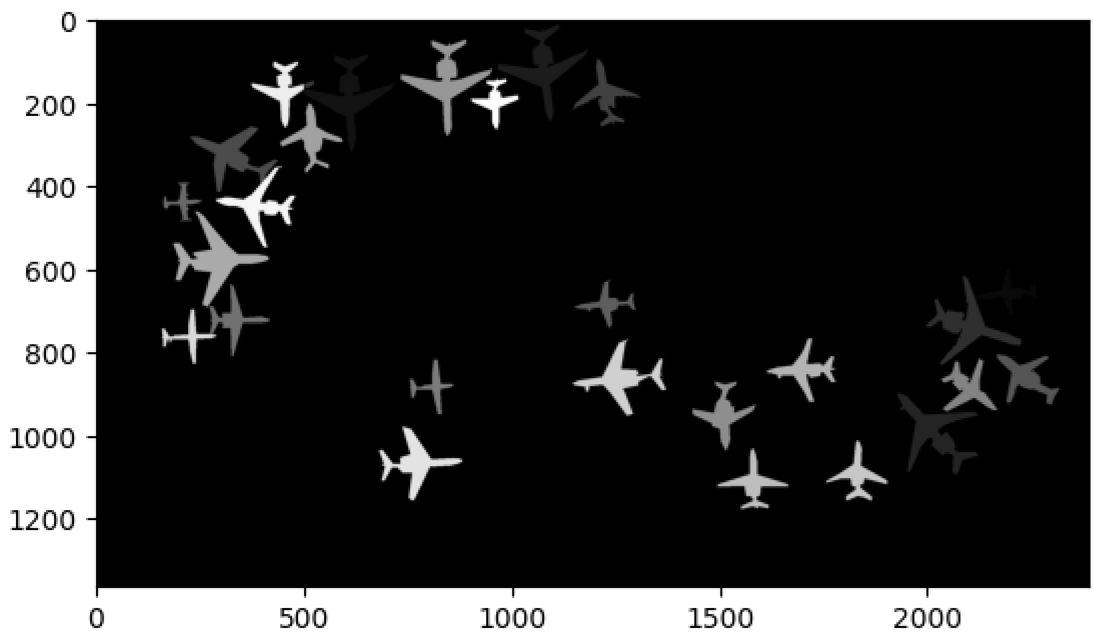





0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
200
400
600
800
1000
1200
0
500
1000
1500
2000

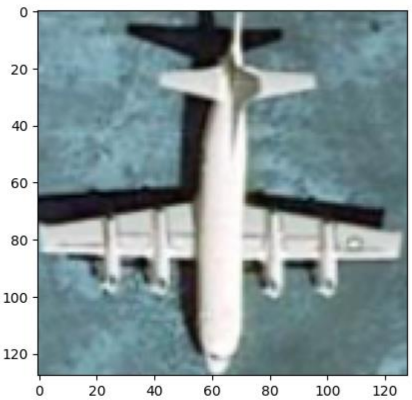

0
200
400
600
800
1000
1200
0
500
1000
1500
2000

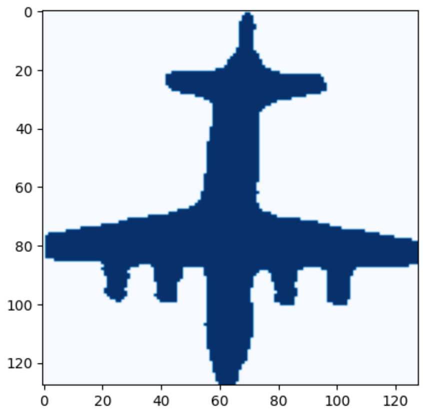
Part3

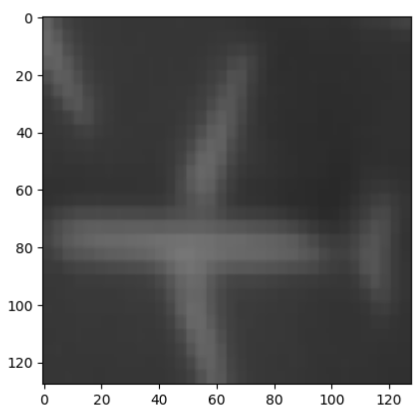
Kaggle Team name: Freya Li

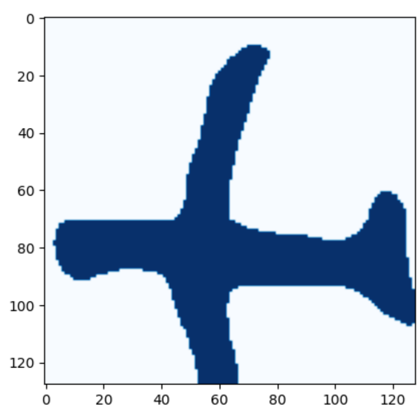
Member: Wenbin(Freya) Li

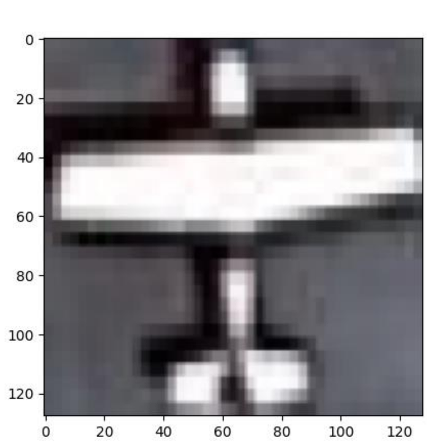
Best score: 0.55123

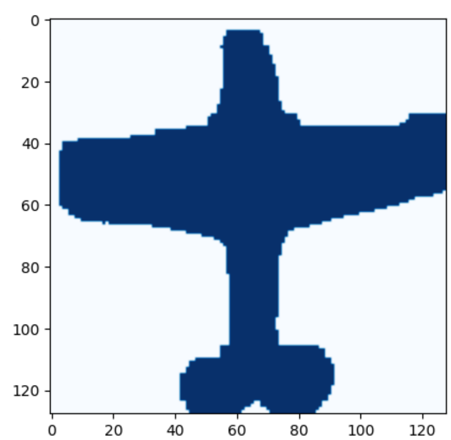






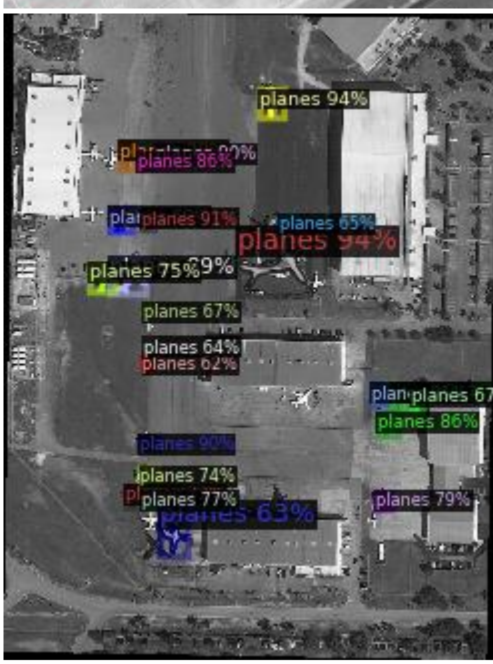


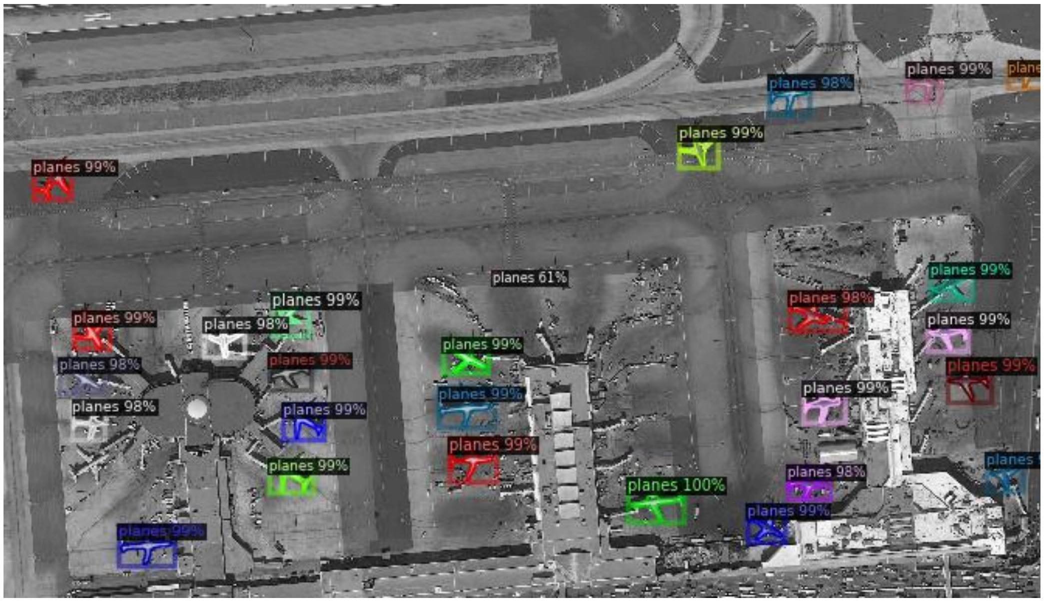




Part 4

Visualization results for Mask R-CNN





Training Loss:

```
:  eta: 0:05:20  iter: 79  total_loss: 1.113  loss_cls: 0.1683  loss_box_reg: 0.349  loss_mask: 0.2719  loss_rpn_cls: 0.1134  loss_rpn_loc: 0.1972
:  eta: 0:04:55  iter: 99  total_loss: 0.9795  loss_cls: 0.1657  loss_box_reg: 0.2803  loss_mask: 0.2641  loss_rpn_cls: 0.09097  loss_rpn_loc: 0.1815
:  eta: 0:04:33  iter: 119  total_loss: 0.8704  loss_cls: 0.1343  loss_box_reg: 0.2676  loss_mask: 0.2423  loss_rpn_cls: 0.06327  loss_rpn_loc: 0.1466
:  eta: 0:04:22  iter: 139  total_loss: 0.9897  loss_cls: 0.1729  loss_box_reg: 0.26  loss_mask: 0.2414  loss_rpn_cls: 0.07126  loss_rpn_loc: 0.1908
:  eta: 0:04:12  iter: 159  total_loss: 1.041  loss_cls: 0.1589  loss_box_reg: 0.3335  loss_mask: 0.2614  loss_rpn_cls: 0.07407  loss_rpn_loc: 0.1849
:  eta: 0:04:04  iter: 179  total_loss: 1.166  loss_cls: 0.1625  loss_box_reg: 0.2848  loss_mask: 0.3364  loss_rpn_cls: 0.1228  loss_rpn_loc: 0.2244
:  eta: 0:03:52  iter: 199  total_loss: 0.9232  loss_cls: 0.1313  loss_box_reg: 0.2631  loss_mask: 0.2495  loss_rpn_cls: 0.09088  loss_rpn_loc: 0.2
:  eta: 0:03:33  iter: 219  total_loss: 0.9039  loss_cls: 0.1165  loss_box_reg: 0.2152  loss_mask: 0.2416  loss_rpn_cls: 0.05581  loss_rpn_loc: 0.1465
:  eta: 0:03:18  iter: 239  total_loss: 0.9836  loss_cls: 0.1503  loss_box_reg: 0.2912  loss_mask: 0.2808  loss_rpn_cls: 0.0718  loss_rpn_loc: 0.2294
:  eta: 0:03:03  iter: 259  total_loss: 1.048  loss_cls: 0.1286  loss_box_reg: 0.3128  loss_mask: 0.2781  loss_rpn_cls: 0.0758  loss_rpn_loc: 0.2198
:  eta: 0:02:47  iter: 279  total_loss: 0.8634  loss_cls: 0.1211  loss_box_reg: 0.2372  loss_mask: 0.2637  loss_rpn_cls: 0.06665  loss_rpn_loc: 0.1558
:  eta: 0:02:30  iter: 299  total_loss: 0.9177  loss_cls: 0.1311  loss_box_reg: 0.2791  loss_mask: 0.2583  loss_rpn_cls: 0.05918  loss_rpn_loc: 0.1913
:  eta: 0:02:14  iter: 319  total_loss: 1.083  loss_cls: 0.1408  loss_box_reg: 0.2647  loss_mask: 0.2806  loss_rpn_cls: 0.08283  loss_rpn_loc: 0.171
:  eta: 0:01:57  iter: 339  total_loss: 1.079  loss_cls: 0.1521  loss_box_reg: 0.2996  loss_mask: 0.2671  loss_rpn_cls: 0.06762  loss_rpn_loc: 0.1886
:  eta: 0:01:43  iter: 359  total_loss: 0.9803  loss_cls: 0.1257  loss_box_reg: 0.2915  loss_mask: 0.2824  loss_rpn_cls: 0.102  loss_rpn_loc: 0.1902
:  eta: 0:01:31  iter: 379  total_loss: 1.135  loss_cls: 0.1798  loss_box_reg: 0.2782  loss_mask: 0.3277  loss_rpn_cls: 0.09089  loss_rpn_loc: 0.2425
:  eta: 0:01:17  iter: 399  total_loss: 1.041  loss_cls: 0.1418  loss_box_reg: 0.3073  loss_mask: 0.2905  loss_rpn_cls: 0.07736  loss_rpn_loc: 0.2165
:  eta: 0:01:02  iter: 419  total_loss: 1.04  loss_cls: 0.1438  loss_box_reg: 0.3251  loss_mask: 0.2826  loss_rpn_cls: 0.06128  loss_rpn_loc: 0.1874
:  eta: 0:00:45  iter: 439  total_loss: 0.8089  loss_cls: 0.1271  loss_box_reg: 0.2386  loss_mask: 0.2402  loss_rpn_cls: 0.05927  loss_rpn_loc: 0.1541
:  eta: 0:00:30  iter: 459  total_loss: 0.8565  loss_cls: 0.1178  loss_box_reg: 0.2636  loss_mask: 0.2438  loss_rpn_cls: 0.05404  loss_rpn_loc: 0.2111
:  eta: 0:00:15  iter: 479  total_loss: 0.8743  loss_cls: 0.1416  loss_box_reg: 0.2685  loss_mask: 0.2467  loss_rpn_cls: 0.06156  loss_rpn_loc: 0.1998
:  eta: 0:00:00  iter: 499  total_loss: 1.056  loss_cls: 0.147  loss_box_reg: 0.2621  loss_mask: 0.2744  loss_rpn_cls: 0.06179  loss_rpn_loc: 0.1834
: Overall training speed: 498 iterations in 0:08:17 (0.9994 s / it)
: Total training time: 0:08:19 (0:00:01 on hooks)
tection_checkpoint]: [DetectionCheckpointer] Loading from /content/drive/My Drive/CMPT_CV_lab3/output-mask/model_final.pth ...
```

Evaluation:

Evaluation results for bbox:

AP	AP50	AP75	APs	APm	APl
10.867	32.925	4.379	6.473	15.235	32.782

Custom Segmentation Model (Part 3):
Pros: Potentially higher accuracy in segmentation as indicated by the IoU.
Cons: Longer training times and possibly higher computational and memory requirements.

Mask R-CNN (Part 4):
Pros: Streamlined training process with pre-trained models and potentially lower computational complexity.

Cons: May offer less accuracy for the segmentation task as indicated by the lower AP50 metric, compared to the custom segmentation model's IoU score.

Overall Recommendation:
If the primary requirement is high-precision segmentation, the custom model from Part 3 seems preferable due to its high IoU score.
If the goal is to have a balanced approach between detection and segmentation with a streamlined training process, Mask R-CNN would be a suitable choice.
The choice between the two approaches ultimately depends on the specific application requirements, the available computational resources, and the desired balance between accuracy and efficiency.