Project 3

Object Detection, Semantic Segmentation, and Instance Segmentation

Wenxiang He

301417521

Due date: 23:59 November 8th (2023)

(3 Free late-day used, total 1 days remain)

Undergraduate

Kaggle submission ID: Wenxiang He

Part 1

List of config:

Initial config:

```
'''
# Set the configs for the detection part in here.
# TODO: approx 15 lines
'''
cfg = get_cfg()
cfg.OUTPUT_DIR = "{}/output/".format(BASE_DIR)

# this one with highest box AP
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.y
aml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ("plane_test",)

cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS =
model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml")  # Let
training initialize from model zoo
cfg.SOLVER.IMS_PER_BATCH = 2  # This is the real "batch size" commonly known to deep learning
people
cfg.SOLVER.BASE_LR = 0.00025  # pick a good LR
cfg.SOLVER.MAX_ITER = 300
cfg.SOLVER.STEPS = []  # Decay learning rate at 1000th and 1200th iterations
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128        # The "RoIHead batch size". 128 is
faster, and good enough for this toy dataset (default: 512)
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Second modified config:

```
'''
# Set the configs for the detection part in here.
# TODO: approx 15 lines
'''
cfg = get_cfg()
cfg.OUTPUT_DIR = "{}/output/".format(BASE_DIR)

# this one with highest box AP
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.y
aml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ("plane_test",)

cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS =
model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml")  # Let
training initialize from model zoo
cfg.SOLVER.IMS_PER_BATCH = 2  # This is the real "batch size" commonly known to deep learning
people
cfg.SOLVER.BASE_LR = 0.00025  # pick a good LR
cfg.SOLVER.MAX_ITER = 5000
cfg.SOLVER.STEPS = (3500, 4500)
cfg.SOLVER.STEPS = []  # Decay learning rate at 1000th and 1200th iterations
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128        # The "RoIHead batch size". 128 is
faster, and good enough for this toy dataset (default: 512)
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Final config:

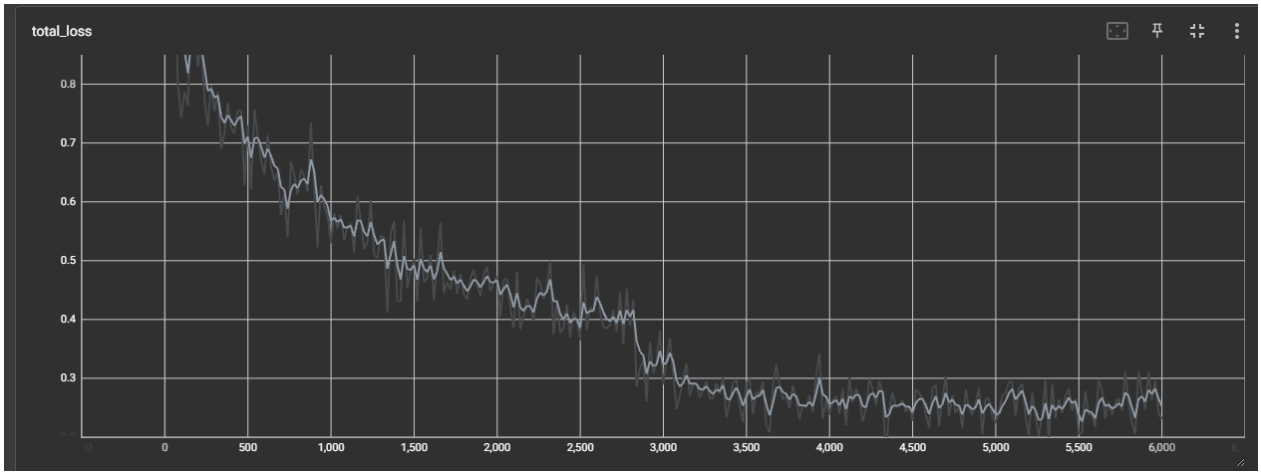
```
cfg = get_cfg()
cfg.OUTPUT_DIR = "{}/output/".format(BASE_DIR)

# Using a less heavy pre-trained model
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.y
aml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ("plane_test",)

# Reducing the number of worker threads to lighten CPU load
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 = 3
cfg.SOLVER.MAX_ITER = 6000

# The learning rate is higher; observe the training stability
cfg.SOLVER.BASE_LR = 0.005  # Adjust the learning rate according to the need

# Consider adding a decay step to reduce the learning rate more rapidly
cfg.SOLVER.STEPS = (2800, 3800, 5000)
cfg.SOLVER.MOMENTUM = 0.9

cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 256
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1

cfg.SOLVER.WARMUP_ITERS = 500  # A warm-up period of 500 iterations
cfg.SOLVER.GRADIENT_CLIP_VALUE = 1.0  # Clip gradients to have a maximum norm of 1.0

# Disable mixed precision training
cfg.SOLVER.FP16_ENABLED = True

# Keep the anchor generator sizes if it's crucial for detecting planes of various sizes
cfg.MODEL.ANCHOR_GENERATOR.SIZES = [[8, 16, 32, 64, 128, 256]]  # Slightly simpler scale
cfg.MODEL.ANCHOR_GENERATOR.ASPECT_RATIOS = [[0.33, 0.5, 1.0, 2.0, 3.0]]

# Reduce the input image size if it is an option for you
cfg.INPUT.MIN_SIZE_TRAIN = (480,)
cfg.INPUT.MAX_SIZE_TRAIN = 1333
cfg.INPUT.MIN_SIZE_TEST = 480
cfg.INPUT.MAX_SIZE_TEST = 1333

std =  [57.375, 57.120, 58.395]
cfg.INPUT.FORMAT = "BGR"
cfg.MODEL.PIXEL_STD = std
```

List of Modifications:

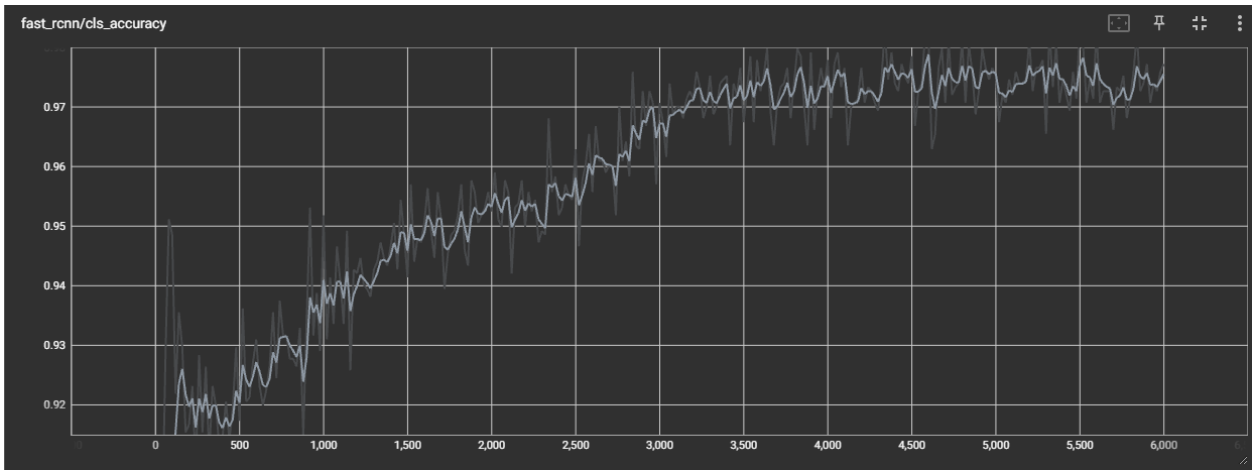
- Increase MAX_ITER from 300 to 5000 to 6000
- Change BASE_LR from 0.00025 to 0.005
- Increase NUM_WORKERS from 2 to 4
- Increase IMS_PER_BATCH from 2 to 3
- Increase BATCH_SIZE_PER_IMAGE from 128 to 256
- Add WARMUP_ITERS = 500
- Set SOLVER.FP16_ENABLED = True
- Add ANCHOR_GENERATOR.SIZES = [[8, 16, 32, 64, 128, 256]]

- Add ANCHOR_GENERATOR.ASPECT_RATIOS = [[0.33, 0.5, 1.0, 2.0, 3.0]]
- Add PIXEL_STD = [57.375, 57.120, 58.395]
- Add MOMENTUM = 0.9
- Add STEPS

Final plot for total training loss and accuracy:



around 0.2



around 0.98

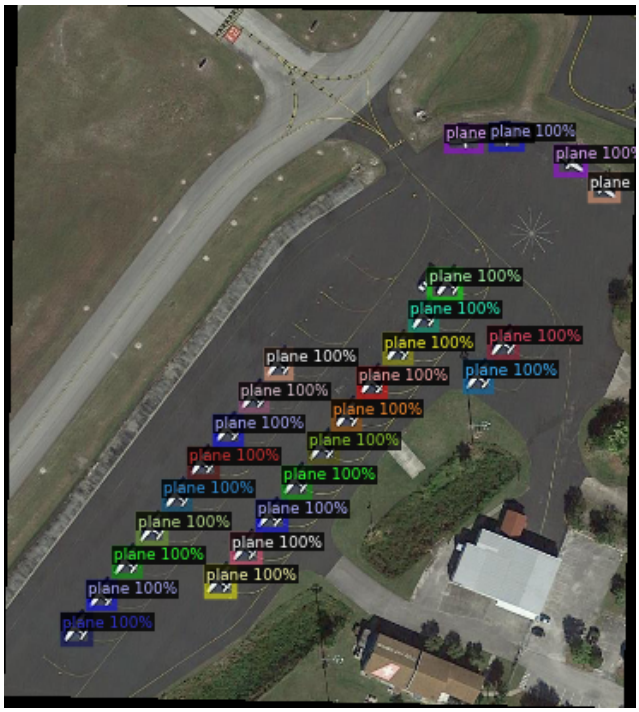
Factors which helped improve the performance:

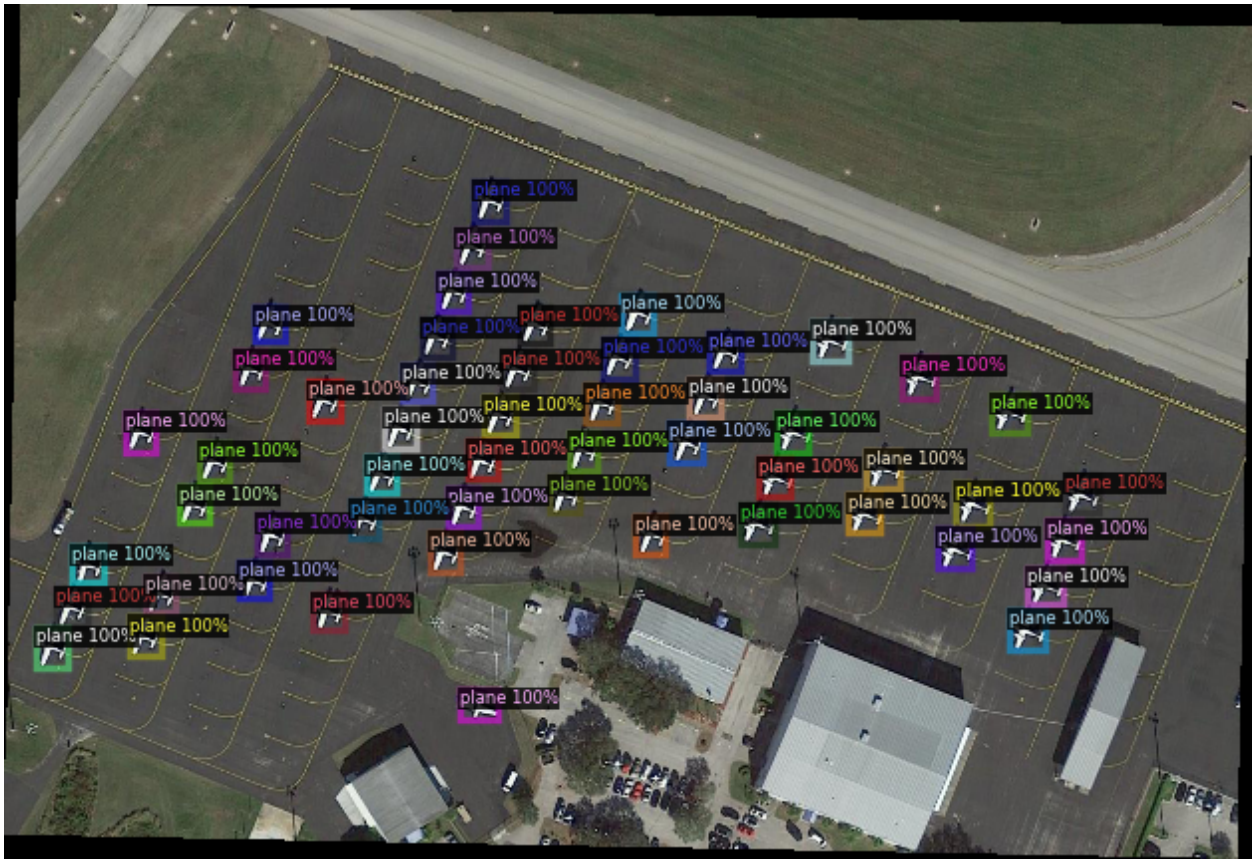
- **MAX_ITER**(maximum number of iterations): Increased from 300 to 5,000 to 6,000, which means that the model iterates more times on the training data and has more opportunities to learn and optimize its weights.
-
- **BASE_LR** (Base learning rate): Increased from 0.00025 to 0.002 to 0.005, a higher learning rate can accelerate learning, but may also cause training instability or model convergence. Try to find the right learning rate to get good training results.
-
- **NUM_WORKERS**(number of worker threads): increased from 2 to 4, which speeds up data loading and also increases memory usage.
-
- **IMS_PER_BATCH**(number of images per batch): Increased from 2 to 3, larger batches can improve training efficiency, but also increase memory requirements.
-
- **BATCH_SIZE_PER_IMAGE**(batch size per image): Increased from 128 to 256, which determines how many area suggestions will be generated per image during training. Larger values increase the stability of the training.
-
- **WARMUP_ITERS**: Set to 500 will slowly increase the learning rate at the beginning of training, helping the model to start the learning process steadily, preventing the initial learning rate from being too high and causing the training to be unstable.
-
- **FP16_ENABLED**(enable floating-point 16-bit training): Set to True to allow the model to be trained using half-precision floating-point numbers, which reduces memory usage and potentially speeds up training.
-
- **ANCHOR_GENERATOR**. **SIZE** and **ANCHOR_GENERATOR**. **ASPECT_RATIO**: These determine the size and aspect ratio of the anchor point used to generate the region proposal. Adjusting these parameters helps the model better fit the size and shape of the target in the data set.
-

- **PIXEL_STD**(pixel standard deviation): Setting this value can change the way the image data is normalized, affecting the distribution of model inputs.
-
- **Momentum**: Set to 0.9, this helps to accelerate gradient descent in the relevant direction while suppressing oscillations, usually leading to faster convergence and improved training stability.
-
- **STEPS** (Learning rate decay step): This parameter defines the number of iterations on which to reduce the learning rate. Proper learning rate attenuation enables rapid learning in the early stages of the model and fine-tuning and stabilization of model performance in the later stages. The learning rate will decrease at 2800, 3800, and 5000 iterations, which will gradually reduce the magnitude of weight updates as the model rapidly approaches the optimal solution, helping fine-tune the model for better performance.

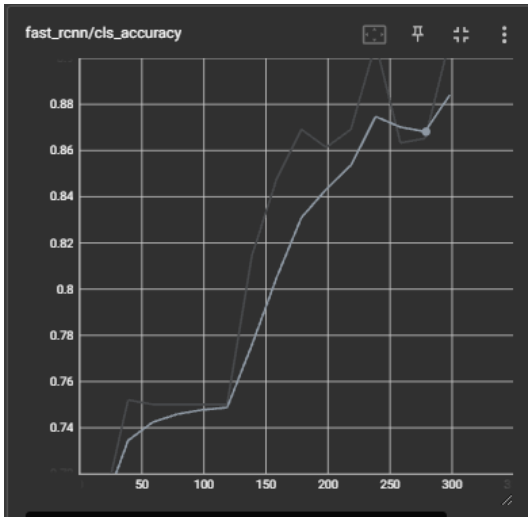
Among them, I mainly modified **MAX_ITER, BASE_LR, Momentum, STEPS** to improve accuracy and reduce loss.

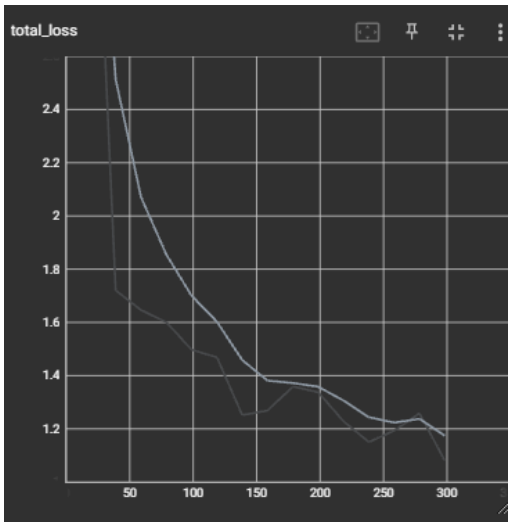
Visualization of 3 samples from the test set with predicted results:





Ablation Study:






```
[11/12 02:32:16 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 19.530 | 38.807 | 17.371 | 13.370 | 25.916 | 47.265 |
```

Initially, I set max iteration time as 300, and base learning rate as 0.00025, get a AP50 of 38.807 and accuracy around 0.88, loss around 1.2.
Here is a visualization of a certain sample from the test set.

Then on the second modified config, I set max iteration time to be 5000, and add a decay step to decrease learning rate at iteration 3500 and 4500, this increases the AP50 to 69.764

```
[11/11 21:22:19 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 53.901 | 69.764 | 61.560 | 44.219 | 61.239 | 82.375 |
```

Here is a visualization of a certain sample from the test set.

As seen above from two same certain samples, the accuracy of plane detection is greatly increased.

I also increase **BATCH_SIZE_PER_IMAGE, IMS_PER_BATCH, NUM_WORKERS**, and set **cfg.SOLVER.FP16_ENABLE**D = True, to make sufficient use of GPU memory and speed up training

Part 2

Hyperparameter settings:

```
num_epochs = 75
batch_size = 16
learning_rate = 0.001
weight_decay = 1e-5
```

Optimizer:

```
optim = torch.optim.AdamW(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)
```

Learning Rate Scheduler:

```
scheduler = ReduceLROnPlateau(optim, 'min', patience=3, factor=0.1)
```

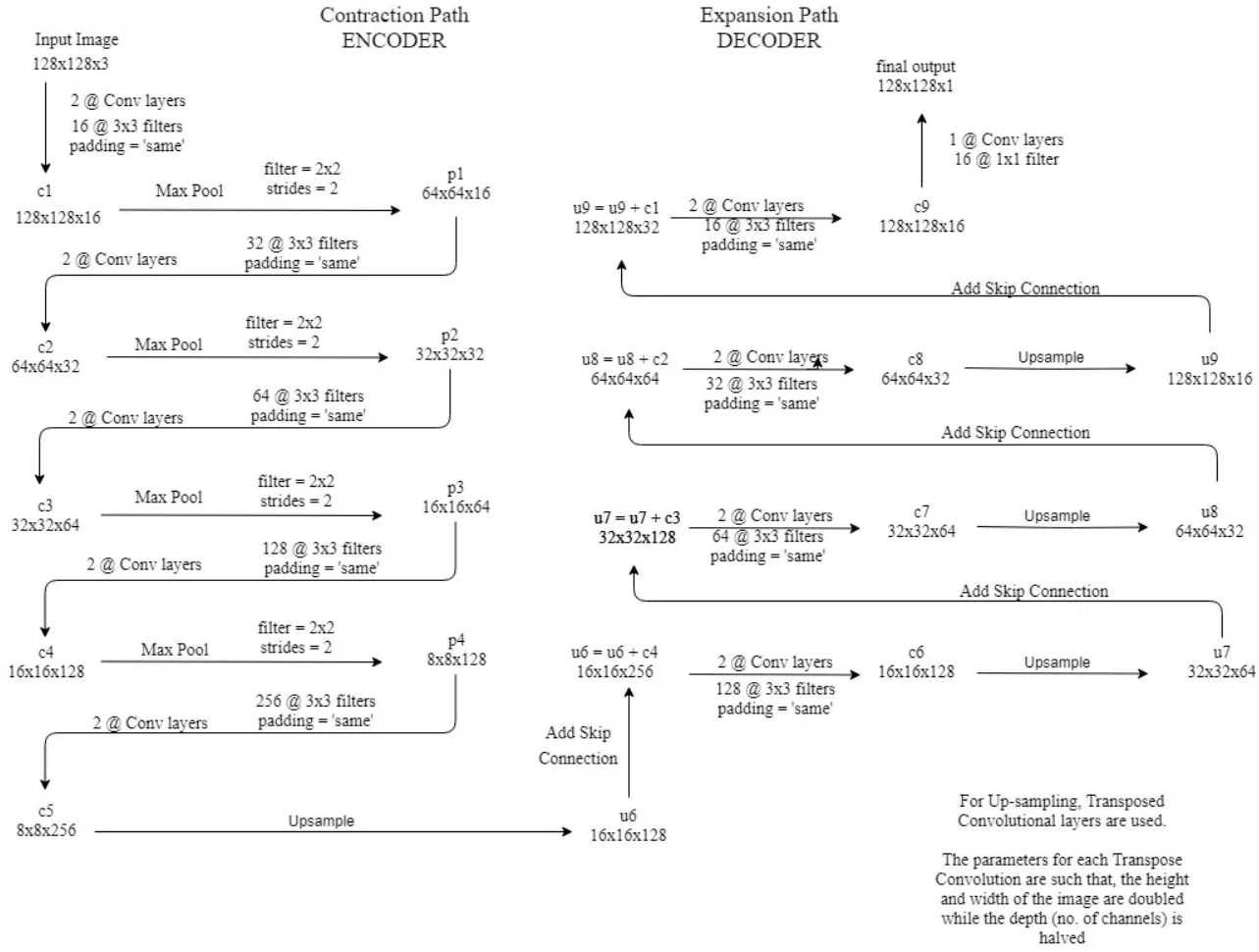
Gradient Scaler:

```
scaler = GradScaler()
```

Final architecture of network:

Architecture adapt from:

[Understanding Semantic Segmentation with UNET | by Harshall Lamba | Towards Data Science]



Modification:

Convolutional Layer:

```
class conv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(conv, self).__init__()
        self.layer = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )
```

```
    def forward(self, x):
        x = self.layer(x)
        return x
```

Change from single Conv2d layer with optional batch normalization (BatchNorm2d) and ReLU activation to two consecutive Conv2d layers, each followed by batch normalization and ReLU activation.

Reason: Increase the network's depth, allowing it to learn more complex features. Using batch normalization and ReLU activation after each convolution helps to accelerate training and improve gradient flow.

Downsampling & upsampling Module:

```
class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.layer = nn.Sequential(
            conv(in_ch, out_ch),
            nn.MaxPool2d(2)
        )

    def forward(self, x):
        x = self.layer(x)
        return x

class up(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=False):
        super(up, self).__init__()
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        else:
            self.up = nn.ConvTranspose2d(in_ch, out_ch, 2, stride=2)
        self.conv = conv(out_ch*2, out_ch)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # Concatenate on the channels axis
        x = torch.cat([x2, x1], dim=1)
        x = self.conv(x)
        return x
```

Downsampling unchanged, upsampling module add skip connections, which pass features from the corresponding layer of the encoder to the decoder

Reason: Helps to retain more spatial information in the output, thus improving segmentation accuracy.

MyModel Module:

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()

        # Encoder
        self.input_conv = conv(3, 16)
        self.down1 = down(16, 32)
        self.down2 = down(32, 64)
        self.down3 = down(64, 128)
        self.down4 = down(128, 256)

        # Decoder
        self.up1 = up(256, 128)
        self.up2 = up(128, 64)
        self.up3 = up(64, 32)
        self.up4 = up(32, 16)

        # Final 1x1 convolution
        self.output_conv = nn.Conv2d(16, 1, 1)

    def forward(self, x):
        c1 = self.input_conv(x)
        c2 = self.down1(c1)
        c3 = self.down2(c2)
        c4 = self.down3(c3)
        c5 = self.down4(c4)

        u6 = self.up1(c5, c4)
        u7 = self.up2(u6, c3)
        u8 = self.up3(u7, c2)
        u9 = self.up4(u8, c1)

        output = self.output_conv(u9)
        return output
```

Encoder & Decoder:

The encoder part of the network consists of a series of convolutional and downsampling layers that encode the input image into a set of feature maps at various resolutions.

Start with an initial convolutional layer that expands the number of channels from 3 (RGB image) to 16. Then apply a sequence of down modules, each of which applies a convolution followed by max pooling. With each down module, double the number of feature channels (16 -> 32 -> 64 -> 128 -> 256), while the spatial resolution is halved due to pooling.

The decoder part consists of a series of upsampling layers that progressively increase the resolution of the feature maps.

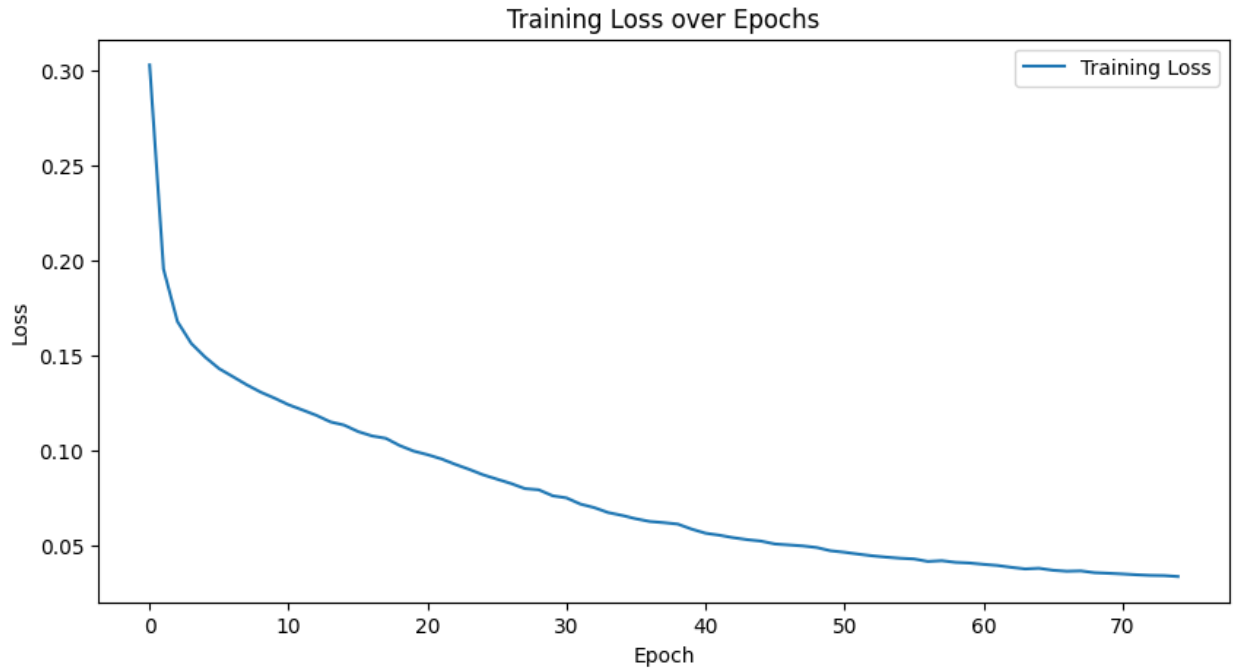
Each up module uses either a transposed convolution or bilinear upsampling (depending on whether bilinear is True or False) to upsample the feature map, followed by a convolutional block to refine the features. Included skip connections, which concatenate the upsampled

feature maps with the feature maps from the corresponding encoder layer (e.g., u6 = self.up1(c5, c4)). It helps the network to recover spatial information that might be lost during downsampling.

Reason: As the diagram shown above, the original structure is optimized to achieve the U-Net structure, increase the network depth and add skip connections, and the network can better learn and retain spatial messages of multiple scales.

Loss functions used:

```
crit = nn.BCEWithLogitsLoss()
loss = crit(pred, mask)
```



Final mean IoU:

```
#images: 7980, Mean IoU: 0.9529
```

Visualize 3 images from the test set:

From left to right (test image->GT mask->predict mask)

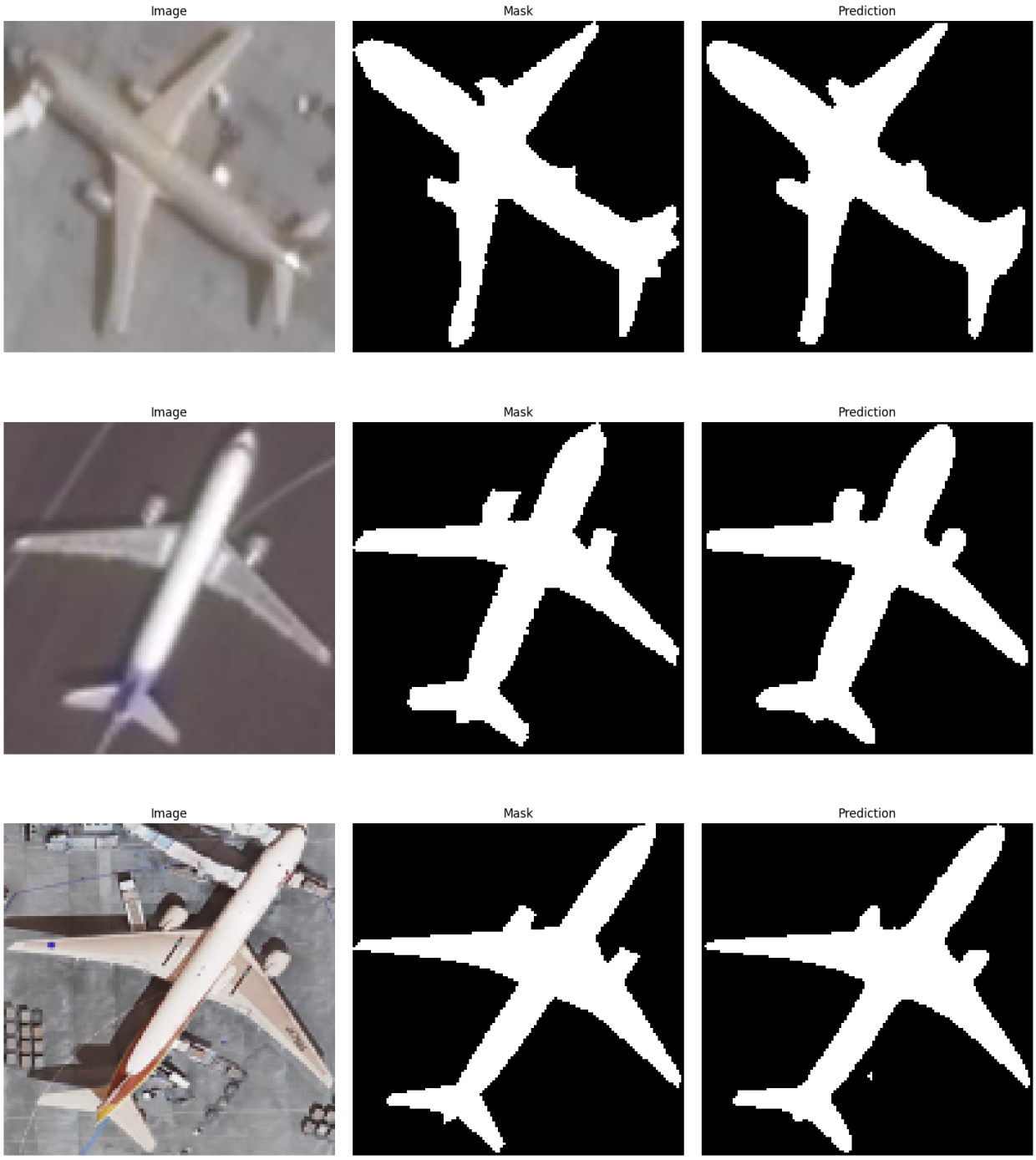


Image
Mask
Prediction
Image
Mask
Prediction
Image
Mask
Prediction

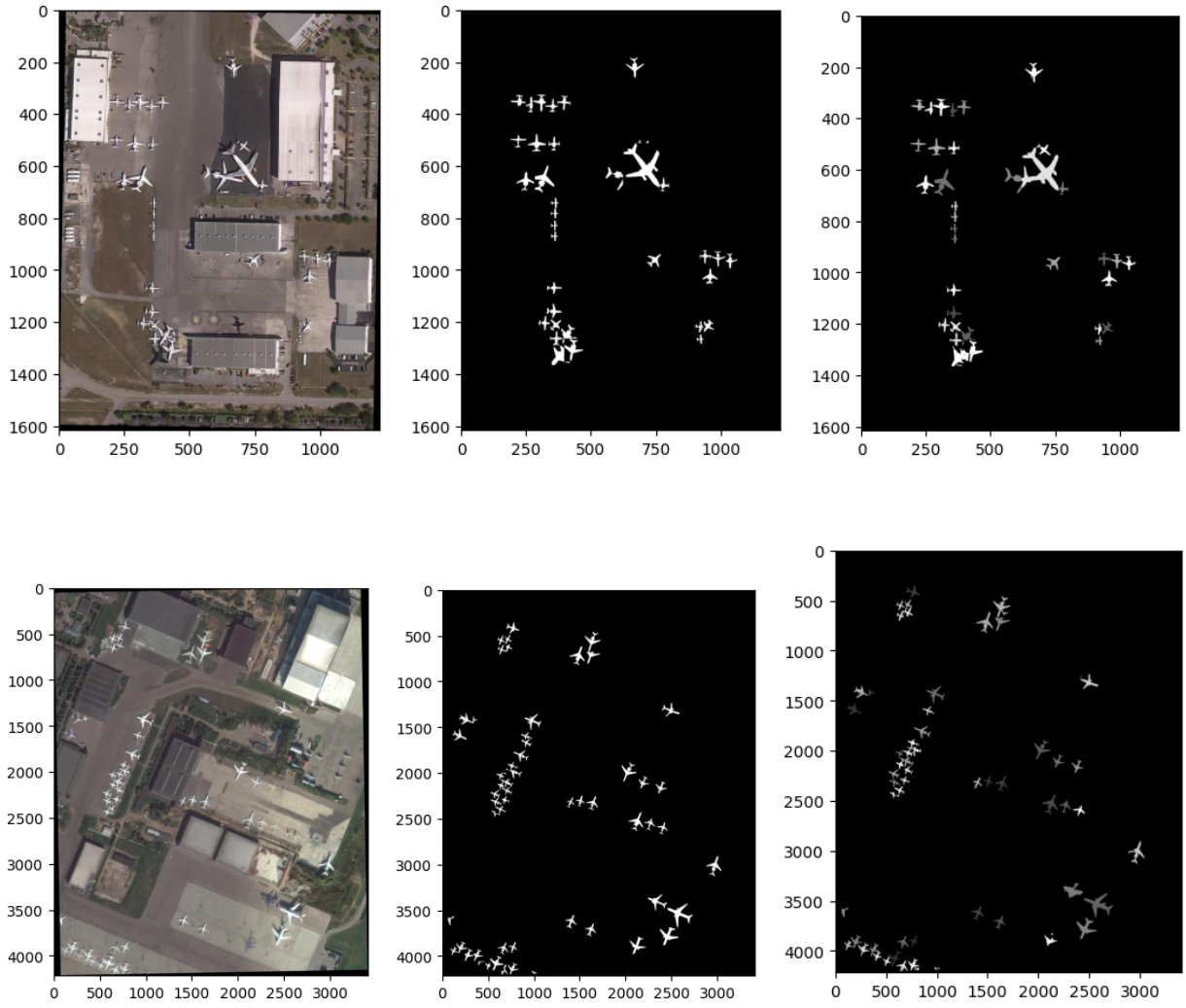
Part 3

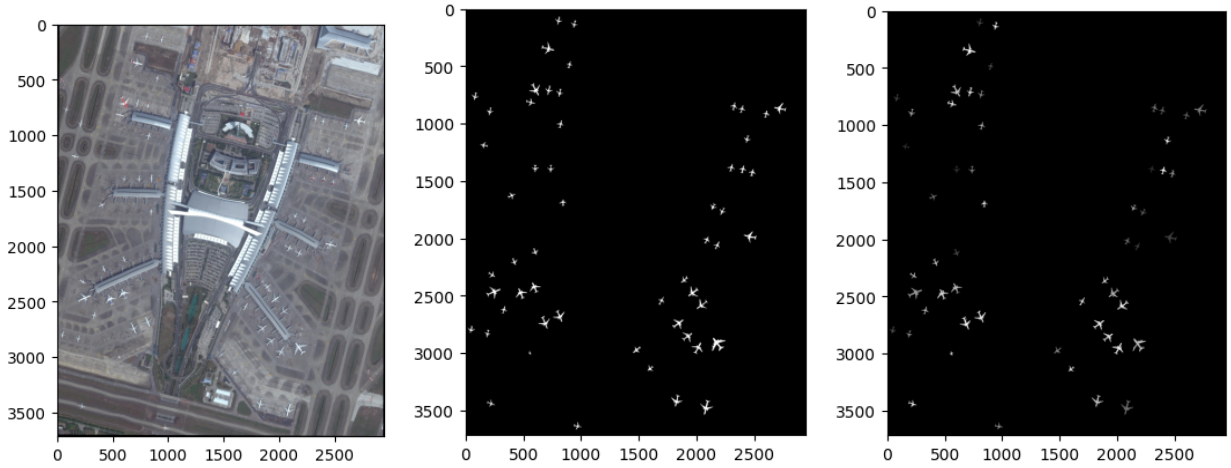
Kaggle info:

- Name: Wenxiang He
- Best Score: 0.54012

Visualization of results:

From left to right (test image->GT mask->predict mask)




0
500
1000
1500
2000
2500
3000
3500
0
500
1000
1500
2000
2500

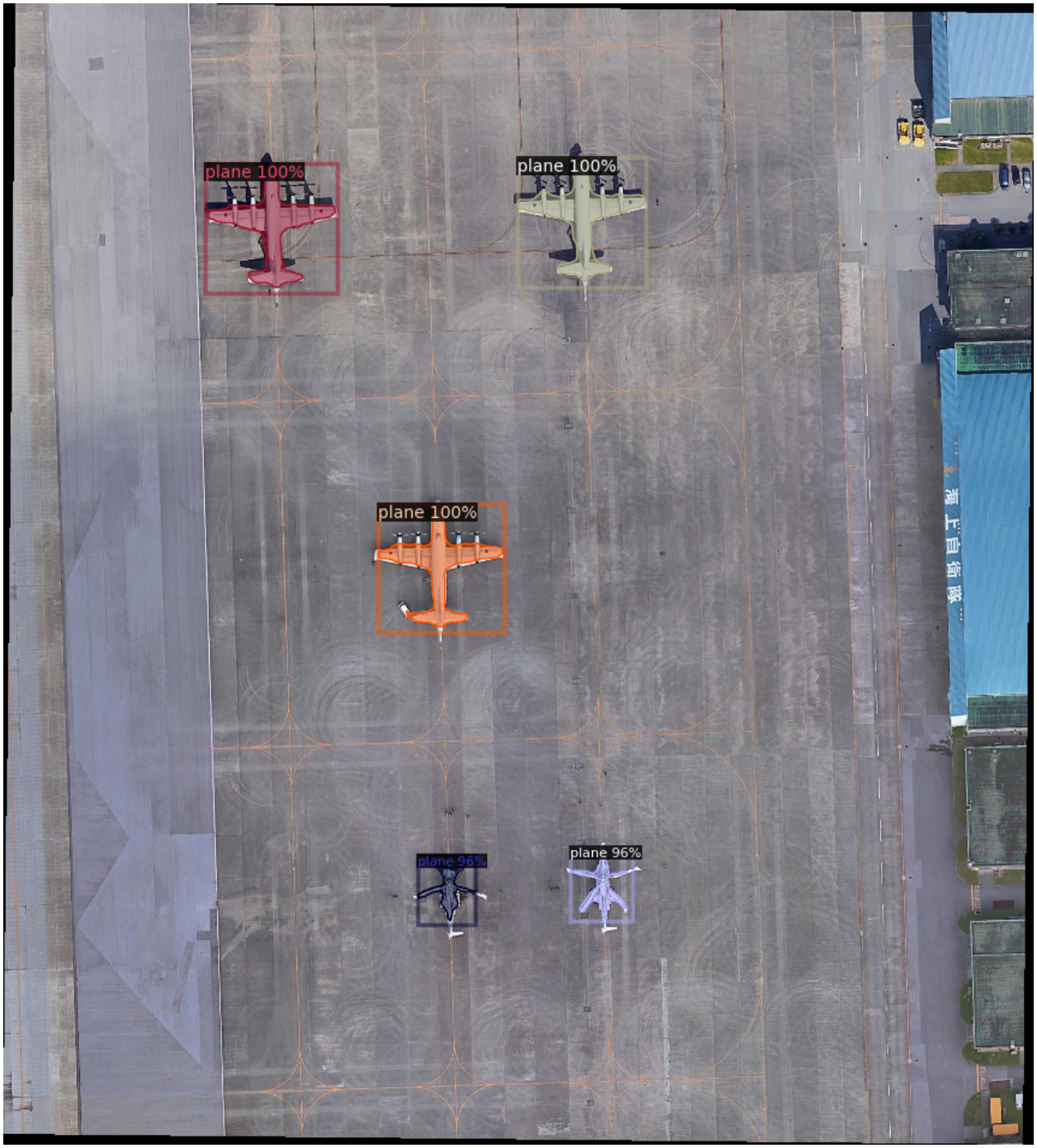
Part 4

Config: same as final config in part 1, modify from Faster R-CNN to Mask R-CNN, and max iteration = 2000

```
cfg = get_cfg()
cfg.OUTPUT_DIR = "{}/output/".format(BASE_DIR)

# Using a less heavy pre-trained model
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_r
cnn_X_101_32x8d_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("plane_train",)
cfg.DATASETS.TEST = ("plane_test",)

# Reducing the number of worker threads to lighten CPU load
cfg.DATALOADER.NUM_WORKERS = 4
cfg.MODEL.WEIGHTS =
model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_F
PN_3x.yaml")
cfg.SOLVER.IMS_PER_BATCH = 3
cfg.SOLVER.MAX_ITER = 2000

# The learning rate is higher; observe the training stability
cfg.SOLVER.BASE_LR = 0.005  # Adjust the learning rate according to the need

# Consider adding a decay step to reduce the learning rate more rapidly
cfg.SOLVER.STEPS = (1200, 1700)
cfg.SOLVER.MOMENTUM = 0.9

cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 256
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1

cfg.SOLVER.WARMUP_ITERS = 500  # A warm-up period of 500 iterations
cfg.SOLVER.GRADIENT_CLIP_VALUE = 1.0  # Clip gradients to have a maximum norm
of 1.0

# Disable mixed precision training
cfg.SOLVER.FP16_ENABLED = True


# Keep the anchor generator sizes if it's crucial for detecting planes of
various sizes
cfg.MODEL.ANCHOR_GENERATOR.SIZES = [[8, 16, 32, 64, 128, 256]]  # Slightly
simpler scale
cfg.MODEL.ANCHOR_GENERATOR.ASPECT_RATIOS = [[0.33, 0.5, 1.0, 2.0, 3.0]]

# Reduce the input image size if it is an option for you
cfg.INPUT.MIN_SIZE_TRAIN = (480,)
cfg.INPUT.MAX_SIZE_TRAIN = 1333
cfg.INPUT.MIN_SIZE_TEST = 480
cfg.INPUT.MAX_SIZE_TEST = 1333

std =  [57.375, 57.120, 58.395]
cfg.INPUT.FORMAT = "BGR"
cfg.MODEL.PIXEL_STD = std
```

Visualization of 3 samples from the test set:


plane 100%
plane 100%
plane 100%
plane 99%
plane 100%
plane 97%
plane 99%
plane 100%
plane 100%
plane 100%
plane 100%
plane 98%
plane 98%
plane 100%
plane 100%
plane 87%
plane 97%
plane 99%
plane 100%
plane 100%
plane 100%
plane 100%
plane 99%
plane 100%



Evaluation:

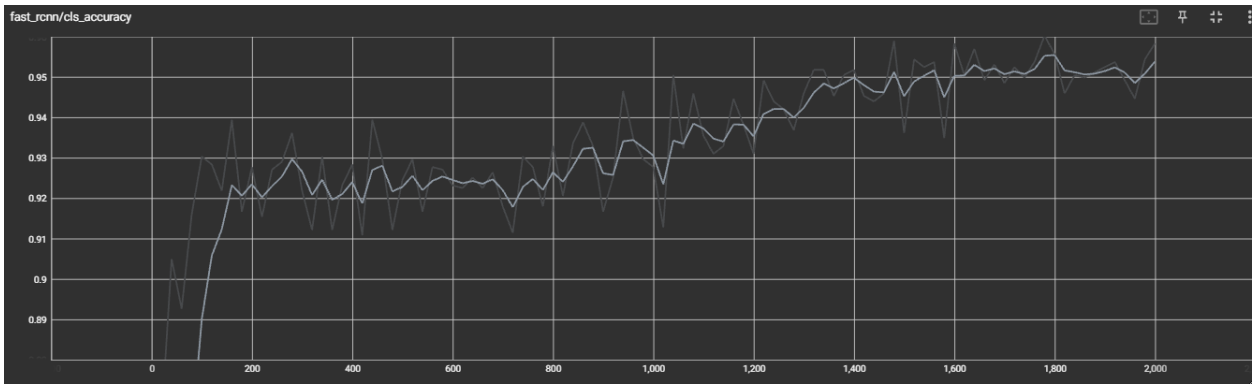
BBox AP:

AP	AP50	AP75	APs	APm	APl
42.926	64.890	47.779	32.887	49.815	78.293

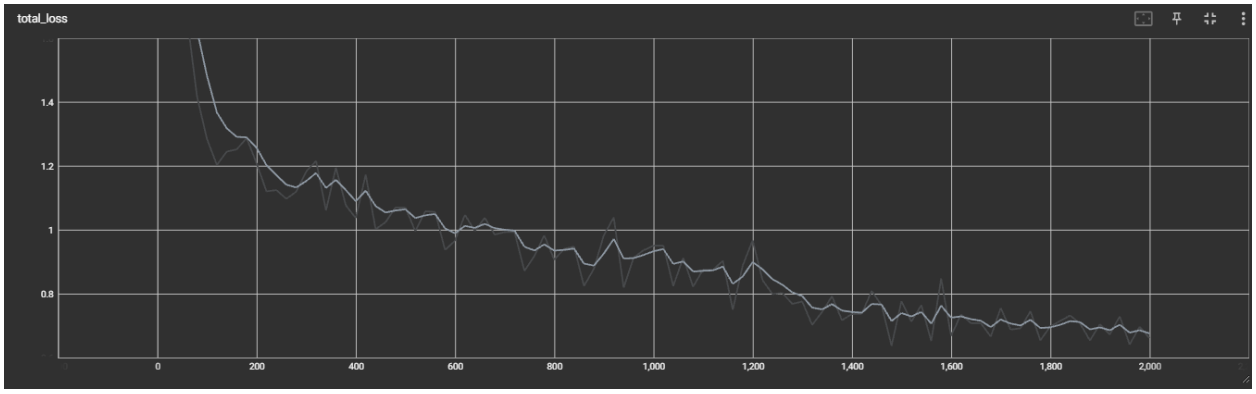
Segmentation AP:

AP	AP50	AP75	APs	APm	APl
5.862	20.038	2.300	1.359	6.378	42.025

Accuracy and Loss



Around 0.96



Around 0.4

Differences with Part 3:

Detection: Both have good detection accuracy, mask r-cnn can get similar AP50 in only ⅓ of iterations.

Segmentation: part 3’s segmentation has a more stable shape of mask, while some of Mask R-CNN's masks do not appear to cover the entire plane.

From my point of view, As seen from the above image, Mask R-CNN seems to have similar object detection and segmentation results. If with the same iteration, Mask R-CNN should perform better in object detection and instance segmentation, but it is more competitively expensive and requires more computational resources, and it takes nearly 2 hours for only 2000 iterations.