TorchVision Object Detection Finetuning Tutorial

1. Defining the Dataset

torch.utils.data.Dataset 사용

이미지: PIL image of size (H, W)

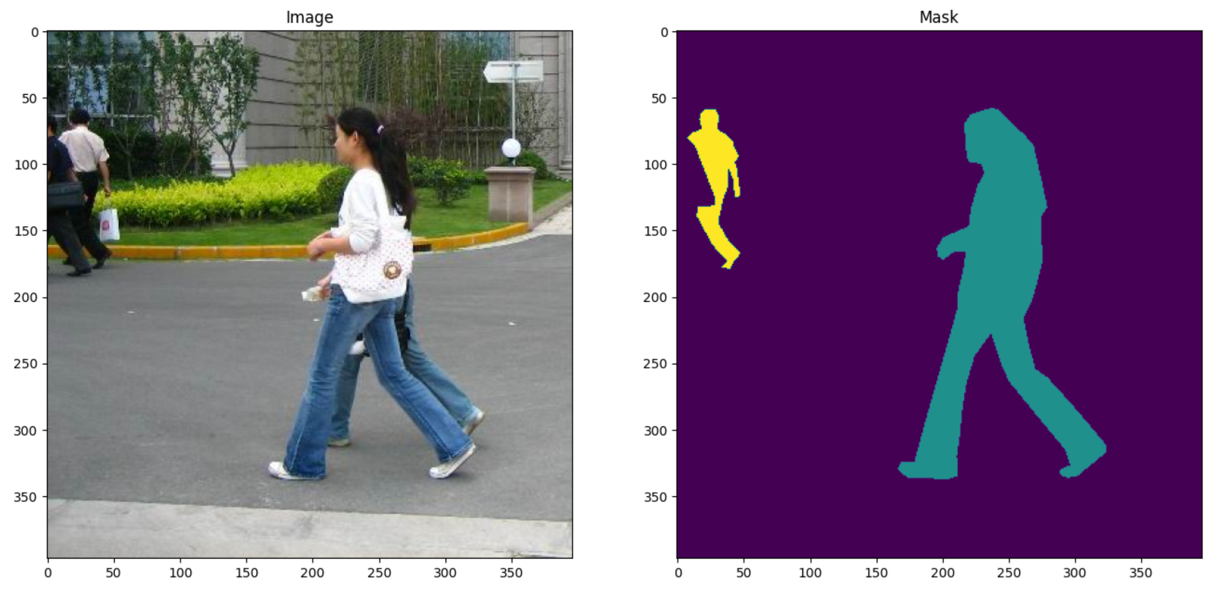
타겟: 아래와 같은 필드를 포함

- boxes [3, H, W]: N개의 bounding boxes의 [x0, y0, x1, y1] 좌표
- labels [N, 4]: 각 bounding box의 label
- image_id [N]: 이미지 식별자
- area [N]: bounding box의 영역
- iscrowd [N]: evaluation에서 iscrowd=True는 제외
- masks [N, H, W]: 각 객체들의 segmentation masks

Writing a custom dataset for PennFudan

우선 PennFudan dataset 중 임의의 데이터의 image와 mask를 확인한다.

```
import matplotlib.pyplot as plt
from torchvision.io import read_image
# FudanPed000004 img, mask 확인
image = read_image("/content/drive/MyDrive/PennFudanPed/PNGIm
mask = read_image("/content/drive/MyDrive/PennFudanPed/PedMas

plt.figure(figsize=(16, 8))
plt.subplot(121)
plt.title("Image")
plt.imshow(image.permute(1, 2, 0))
plt.subplot(122)
plt.title("Mask")
plt.imshow(mask.permute(1, 2, 0))
```



각 이미지는 segmentation mask가 있고, 각 색상은 다른 인스턴스에 대응한다.

PennFudan 데이터셋을 전처리 후 읽어들인다.

```
import os
import torch
from torchvision.io import read_image
from torchvision.ops.boxes import masks_to_boxes
from torchvision import tv_tensors
from torchvision.transforms.v2 import functional as F

class PennFudanDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms):
        self.root = root  # 루트 디렉토리
        self.transforms = transforms # 변환 함수
        # 이미지와 마스크를 정렬
        self.imgs = list(sorted(os.listdir(os.path.join(root,
        self.masks = list(sorted(os.listdir(os.path.join(root

    def __getitem__(self, idx):
        # 이미지와 마스크 파일을 읽어들임
        img_path = os.path.join(self.root, "PNGImages", self.
        mask_path = os.path.join(self.root, "PedMasks", self.
        img = read_image(img_path)
        mask = read_image(mask_path)
        # 인스턴스들은 다른 색으로 인코딩
        obj_ids = torch.unique(mask)
        # 첫 번째 ID는 배경이므로 제거
        obj_ids = obj_ids[1:]
        num_objs = len(obj_ids)  # 객체 수

        # color-encoded mask를 binary mask set으로 분리
        masks = (mask == obj_ids[:, None, None]).to(dtype=tor

        # 각 마스크에 대한 bounding box 좌표 계산
        boxes = masks_to_boxes(masks)

        # 모든 객체가 동일한 클래스로 레이블
        labels = torch.ones((num_objs,), dtype=torch.int64)

        image_id = idx  # 이미지 ID는 인덱스로 설정
        # 각 박스의 면적 계산
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - bo
        # 모든 인스턴스가 crowd가 아니라고 가정
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        # 이미지와 타겟을 torchvision tv_tensors로 wrap
        img = tv_tensors.Image(img)

        target = {}  # 타겟
        target["boxes"] = tv_tensors.BoundingBoxes(boxes, for
        target["masks"] = tv_tensors.Mask(masks)  # 마스크
        target["labels"] = labels  # 레이블
        target["image_id"] = image_id  # 이미지 ID
        target["area"] = area  # 면적
        target["iscrowd"] = iscrowd  # iscrowd 플래그

        if self.transforms is not None:
            # 변환 함수가 있으면 이미지와 타겟에 적용
            img, target = self.transforms(img, target)

        return img, target
```

```
    def __len__(self):
        return len(self.imgs)  # 데이터셋의 길이
```

2. Defining model

Mask R-CNN 모델 활용한다. (Faster R-CNN에 각각의 인스턴스의 segmentation mask를 예측하는 mask branch를 더한 모델)

TorchVision Model Zoo에서 모델을 수정하는 2가지 경우는 아래와 같다.

1 - Finetuning from a pretrained model

pre-trained 모델을 사용하여 마지막 레이어만 finetuning 하는 경우

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNN

# COCO 데이터셋으로 pre-trained 모델 로드
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(

# 사용자 정의 클래스 수로 새로운 classifier 교체
num_classes = 2  # 1 클래스 (사람) + 배경
# 분류기의 입력 피처 수
in_features = model.roi_heads.box_predictor.cls_score.in_feat
# pre-trained head를 새로운 head로 교체
model.roi_heads.box_predictor = FastRCNNPredictor(in_features
```

2- Modifying the model to add a different backbone

다른 backbone을 추가하도록 모델을 수정하는 경우

```
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator

# 분류를 위한 pre-trained 모델 로드 및 특징만 반환
backbone = torchvision.models.mobilenet_v2(weights="DEFAULT")
# backbone mobilenet_v2의 출력 채널 수 = 1280
```

```
backbone.out_channels = 1280

# RPN이 각 공간 위치에서 5 x 3 앵커를 생성하도록 설정
# 5가지 size와 3가지 aspect ratio
anchor_generator = AnchorGenerator(
    sizes=((32, 64, 128, 256, 512),),
    aspect_ratios=((0.5, 1.0, 2.0),)
)

# RoI cropping을 수행할 피처 맵과 crop 크기 정의
# backbone이 텐서를 반환하면 featmap_names = [0]
roi_pooler = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0'],
    output_size=7,
    sampling_ratio=2
)

# Faster-RCNN 모델 구성
model = FasterRCNN(
    backbone,
    num_classes=2,
    rpn_anchor_generator=anchor_generator,
    box_roi_pool=roi_pooler
)
```

Object detection and instance segmentation model for PennFudan Dataset

1번 방법을 사용하여 finetuning 진행한다.

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNN
from torchvision.models.detection.mask_rcnn import MaskRCNNPr



def get_model_instance_segmentation(num_classes):
    # COCO 데이터셋으로 pre-trained instance segmentation 로드
    model = torchvision.models.detection.maskrcnn_resnet50_fp
    # classifier를 위한 입력 피처 개수
    in_features = model.roi_heads.box_predictor.cls_score.in_
    # pre-trained head를 새로운 head로 교체
    model.roi_heads.box_predictor = FastRCNNPredictor(in_feat

    # mask classifier를 위한 입력 피처 개수
    in_features_mask = model.roi_heads.mask_predictor.conv5_m
    hidden_layer = 256
    # mask predictor를 새로운 predictor로 교체
    model.roi_heads.mask_predictor = MaskRCNNPredictor(
        in_features_mask,
        hidden_layer,
        num_classes
    )

    return model
```

3. Putting everything together

```
from torchvision.transforms import v2 as T
import torch

def get_transform(train):
    transforms = []
    if train:
        transforms.append(T.RandomHorizontalFlip(0.5))  # tra
    transforms.append(T.ToDtype(torch.float, scale=True))  #
    transforms.append(T.ToPureTensor())  # 텐서로 변환
    return T.Compose(transforms)
```

4. Testing forward() method

데이터셋을 반복하기 전, 샘플 데이터로 모델이 예상대로 동작하는지 살펴본다.

```
import utils
# COCO dataset으로 pre-trained Faster R-CNN 모델 로드
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(
# PennFudan dataset 로드
dataset = PennFudanDataset('/content/drive/MyDrive/PennFudanP
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True, # 데이터 셔플링
    collate_fn=utils.collate_fn
)

# 훈련
images, targets = next(iter(data_loader)) # 배치 가져오기
images = list(image for image in images) # 이미지 리스트로 변환
targets = [{k: v for k, v in t.items()} for t in targets] # 타
output = model(images, targets)  # loss 및 detection 결과 빈환
print(output)

# 추론
model.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)] # 임의의
predictions = model(x)  # 예측 결과 반환
print(predictions[0])
```

```
{'loss_classifier': tensor(0.2434, grad_fn=<NllLossBackward0>
{'boxes': tensor([], size=(0, 4), grad_fn=<StackBackward0>),
```

모델이 예상대로 작동한다.

training과 validation은 아래와 같다.

```
from engine import train_one_epoch, evaluate

# GPU, CPU(GPU가 안될 경우)에서 훈련
device = torch.device('cuda') if torch.cuda.is_available() el

# 데이터셋의 클래스 수 (배경, 사람)
num_classes = 2
# 데이터셋 준비 및 변환
dataset = PennFudanDataset('/content/drive/MyDrive/PennFudanP
dataset_test = PennFudanDataset('/content/drive/MyDrive/PennF

# train set / test set 분할
indices = torch.randperm(len(dataset)).tolist()
dataset = torch.utils.data.Subset(dataset, indices[:-50])
dataset_test = torch.utils.data.Subset(dataset_test, indices[

# train / validation 데이터 로더
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True, # 데이터 셔플링
    collate_fn=utils.collate_fn
)

data_loader_test = torch.utils.data.DataLoader(
    dataset_test,
    batch_size=1,
    shuffle=False, # 데이터 셔플링 x
    collate_fn=utils.collate_fn
)

# get_model_instance_segmentation으로 fine-tuning 된 모델 가져오
model = get_model_instance_segmentation(num_classes)

# 디바이스로 모델 이동
model.to(device)

# 옵티마이저 생성
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(
    params,
    lr=0.005,
    momentum=0.9,
    weight_decay=0.0005
)

# learning rate scheduler 생성
```

```
lr_scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.1
)

# epoch을 2로 훈련
num_epochs = 2

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model, optimizer, data_loader, device, ep
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)

print("That's it!")
```

```
Epoch: [0]  [ 0/60]  eta: 1:15:46  lr: 0.000090  loss: 3.66
91 (3.6691)  loss_classifier: 0.9223 (0.9223)  loss_box_re
g: 0.5541 (0.5541)  loss_mask: 2.1704 (2.1704)  loss_object
ness: 0.0156 (0.0156)  loss_rpn_box_reg: 0.0066 (0.0066)  t
ime: 75.7814  data: 1.0771
Epoch: [0]  [10/60]  eta: 0:49:24  lr: 0.000936  loss: 1.75
47 (2.1734)  loss_classifier: 0.5625 (0.5984)  loss_box_re
g: 0.3073 (0.3562)  loss_mask: 0.8230 (1.1936)  loss_object
ness: 0.0166 (0.0201)  loss_rpn_box_reg: 0.0024 (0.0052)  t
ime: 59.2864  data: 1.2222
Epoch: [0]  [20/60]  eta: 0:39:12  lr: 0.001783  loss: 1.00
39 (1.4476)  loss_classifier: 0.2009 (0.3986)  loss_box_re
g: 0.2206 (0.2929)  loss_mask: 0.4393 (0.7346)  loss_object
ness: 0.0163 (0.0163)  loss_rpn_box_reg: 0.0040 (0.0051)  t
ime: 57.9699  data: 1.1896
Epoch: [0]  [30/60]  eta: 0:29:12  lr: 0.002629  loss: 0.54
16 (1.1456)  loss_classifier: 0.1065 (0.2953)  loss_box_re
g: 0.1913 (0.2759)  loss_mask: 0.1855 (0.5559)  loss_object
ness: 0.0060 (0.0134)  loss_rpn_box_reg: 0.0056 (0.0050)  t
```

```
ime: 57.9566  data: 1.1238
Epoch: [0]  [40/60]  eta: 0:19:22  lr: 0.003476  loss: 0.42
84 (0.9703)  loss_classifier: 0.0519 (0.2361)  loss_box_re
g: 0.1790 (0.2554)  loss_mask: 0.1724 (0.4616)  loss_object
ness: 0.0049 (0.0121)  loss_rpn_box_reg: 0.0047 (0.0052)  t
ime: 57.4342  data: 1.1695
Epoch: [0]  [50/60]  eta: 0:09:41  lr: 0.004323  loss: 0.41
37 (0.8550)  loss_classifier: 0.0526 (0.2006)  loss_box_re
g: 0.1431 (0.2318)  loss_mask: 0.1645 (0.4058)  loss_object
ness: 0.0036 (0.0111)  loss_rpn_box_reg: 0.0053 (0.0056)  t
ime: 57.6848  data: 1.2272
Epoch: [0]  [59/60]  eta: 0:00:57  lr: 0.005000  loss: 0.32
44 (0.7720)  loss_classifier: 0.0429 (0.1755)  loss_box_re
g: 0.1136 (0.2098)  loss_mask: 0.1484 (0.3714)  loss_object
ness: 0.0016 (0.0098)  loss_rpn_box_reg: 0.0047 (0.0055)  t
ime: 56.4504  data: 1.1882
Epoch: [0] Total time: 0:57:37 (57.6317 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:08:10  model_time: 9.7520 (9.7520)
evaluator_time: 0.0256 (0.0256)  time: 9.8009  data: 0.0232
Test:  [49/50]  eta: 0:00:07  model_time: 7.1923 (7.6235)
evaluator_time: 0.0062 (0.0111)  time: 7.3481  data: 0.0222
Test: Total time: 0:06:22 (7.6592 s / it)
Averaged stats: model_time: 7.1923 (7.6235)  evaluator_tim
e: 0.0062 (0.0111)
Accumulating evaluation results...
DONE (t=0.02s).
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | m
axDets=100 ] = 0.704
 Average Precision  (AP) @[ IoU=0.50      | area=   all | m
axDets=100 ] = 0.977
 Average Precision  (AP) @[ IoU=0.75      | area=   all | m
axDets=100 ] = 0.845
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | m
axDets=100 ] = 0.417
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | m
axDets=100 ] = 0.604
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | m
axDets=100 ] = 0.715
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets=  1 ] = 0.331
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets= 10 ] = 0.769
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets=100 ] = 0.769
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | m
axDets=100 ] = 0.650
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | m
axDets=100 ] = 0.733
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | m
axDets=100 ] = 0.775
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | m
axDets=100 ] = 0.684
 Average Precision  (AP) @[ IoU=0.50      | area=   all | m
axDets=100 ] = 0.963
 Average Precision  (AP) @[ IoU=0.75      | area=   all | m
axDets=100 ] = 0.853
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | m
axDets=100 ] = 0.284
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | m
axDets=100 ] = 0.336
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | m
axDets=100 ] = 0.705
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets=  1 ] = 0.314
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets= 10 ] = 0.736
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | m
axDets=100 ] = 0.739
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | m
axDets=100 ] = 0.600
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | m
axDets=100 ] = 0.722
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | m
axDets=100 ] = 0.743
```

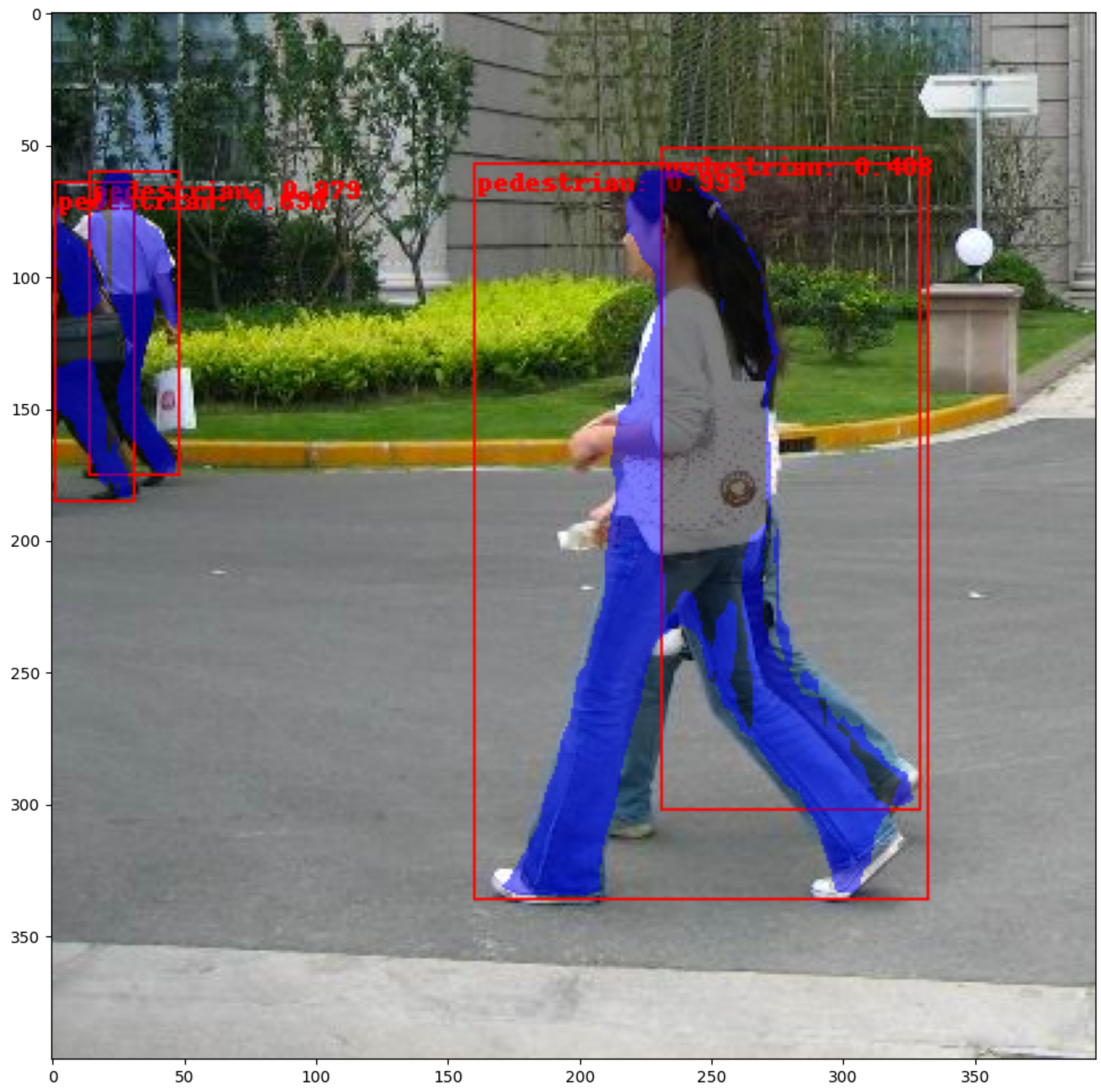
1 epoch으로 COCO-style mAP는 70.4, COCO mask mAP는 68.4를 얻었다.

```
 IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | max
 Average Precision  (AP) @[ IoU=0.50      | area=   all | max
 Average Precision  (AP) @[ IoU=0.75      | area=   all | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | max
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | max
 Average Precision  (AP) @[ IoU=0.50      | area=   all | max
 Average Precision  (AP) @[ IoU=0.75      | area=   all | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | max
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | max
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | max
That's it!
```

모든 epoch을 마치면 결과는 위와 같다. COOC style mAP는 76.7 COCO mask mAP는 74이다.

위 모델로 예측한 이미지 FudanPed00004의 instance segmentation 결과는 아래와 같다.

```
import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes, draw_segme

# 이미지 로드
image = read_image("/content/drive/MyDrive/PennFudanPed/PNGIm
eval_transform = get_transform(train=False)
# 모델 평가 모드
model.eval()
with torch.no_grad():
    x = eval_transform(image)
    # RGBA -> RGP 변환 후 디바이스로 이동
    x = x[:3, ...].to(device)
    predictions = model([x, ])
    pred = predictions[0]

# 이미지 정규화 및 변환
image = (255.0 * (image - image.min()) / (image.max() - image
image = image[:3, ...]
# 예측 레이블
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
pred_boxes = pred["boxes"].long()
# bounding box 그리기
output_image = draw_bounding_boxes(image, pred_boxes, pred_la

masks = (pred["masks"] > 0.7).squeeze(1)
# segmentation mask 그리기
output_image = draw_segmentation_masks(output_image, masks, a

# 예측 결과 시각화
plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
```


pedestrian: 0.879
pedestrian: 0.690
pedestrian: 0.993
pedestrian: 0.408
0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
350

결과는 위와 같다.