

```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

## TorchVision Object Detection Finetuning Tutorial

For this tutorial, we will be finetuning a pre-trained [Mask R-CNN](#) model on the [Penn-Fudan Database for Pedestrian Detection and Segmentation](#). It contains 170 images with 345 instances of pedestrians, and we will use it to illustrate how to use the new features in torchvision in order to train an object detection and instance segmentation model on a custom dataset.

### ✓ Defining the Dataset

The reference scripts for training object detection, instance segmentation and person keypoint detection allows for easily supporting adding new custom datasets. The dataset should inherit from the standard `torch.utils.data.Dataset` {interpreted-text role="class"} class, and implement `__len__` and `__getitem__`.

The only specificity that we require is that the dataset `__getitem__` should return a tuple:

- `image`: `torchvision.tv_tensors.Image` {interpreted-text role="class"} of shape `[3, H, W]`, a pure tensor, or a PIL Image of size `(H, W)`
- `target`: a dict containing the following fields
  - `boxes`, `torchvision.tv_tensors.BoundingBoxes` {interpreted-text role="class"} of shape `[N, 4]`: the coordinates of the `N` bounding boxes in `[x0, y0, x1, y1]` format, ranging from 0 to `W` and 0 to `H`
  - `labels`, integer `torch.Tensor` {interpreted-text role="class"} of shape `[N]`: the label for each bounding box. 0 represents always the background class.
  - `image_id`, int: an image identifier. It should be unique between all the images in the dataset, and is used during evaluation
  - `area`, float `torch.Tensor` {interpreted-text role="class"} of shape `[N]`: the area of the bounding box. This is used during evaluation with the COCO metric, to separate the metric scores between small, medium and large boxes.
  - `iscrowd`, uint8 `torch.Tensor` {interpreted-text role="class"} of shape `[N]`: instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks`, `torchvision.tv_tensors.Mask` {interpreted-text role="class"} of shape `[N, H, W]`: the segmentation masks for each one of the objects

If your dataset is compliant with above requirements then it will work for both training and evaluation codes from the reference script.

Evaluation code will use scripts from `pycocotools` which can be installed with `pip install pycocotools`.

One note on the `labels`. The model considers class 0 as background. If your dataset does not contain the background class, you should not have 0 in your `labels`. For example, assuming you have just two classes, *cat* and *dog*, you can define 1 (not 0) to represent *cats* and 2 to represent *dogs*. So, for instance, if one of the images has both classes, your `labels` tensor should look like `[1, 2]`.

Additionally, if you want to use aspect ratio grouping during training (so that each batch only contains images with similar aspect ratios), then it is recommended to also implement a `get_height_and_width` method, which returns the height and the width of the image. If this method is not provided, we query all elements of the dataset via `__getitem__`, which loads the image in memory and is slower than if a custom method is provided.

### Writing a custom dataset for PennFudan

Let's write a dataset for the PennFudan dataset. First, let's download the dataset and extract the [zip file](#):

```
wget https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip -P data
cd data && unzip PennFudanPed.zip
```

We have the following folder structure:

```
PennFudanPed/
  PedMasks/
    FudanPed00001_mask.png
    FudanPed00002_mask.png
```

```

FudanPed00003_mask.png
FudanPed00004_mask.png
...
PNGImages/
FudanPed00001.png
FudanPed00002.png
FudanPed00003.png
FudanPed00004.png

```

Here is one example of a pair of images and segmentation masks

```

from google.colab import drive
drive.mount('/content/drive')

```

Mounted at /content/drive

```

import matplotlib.pyplot as plt
from torchvision.io import read_image

```

```

image = read_image("/content/drive/MyDrive/PennFudanPed/PNGImages/FudanPed00001.png")
mask = read_image("/content/drive/MyDrive/PennFudanPed/PNGImages/FudanPed00001.png")

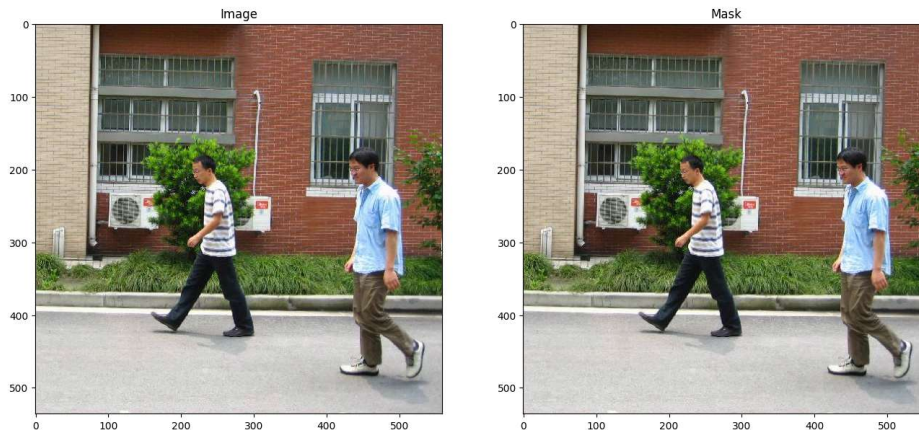
```

```

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))

```

<matplotlib.image.AxesImage at 0x7de775771030>



So each image has a corresponding segmentation mask, where each color correspond to a different instance. Let's write a `torch.utils.data.Dataset` class for this dataset. In the code below, we are wrapping images, bounding boxes and masks into `torchvision.tv_tensors.TVTensor` classes so that we will be able to apply torchvision built-in transformations ([new Transforms API](#)) for the given object detection and segmentation task. Namely, image tensors will be wrapped by `torchvision.tv_tensors.Image`, bounding boxes into `torchvision.tv_tensors.BoundingBoxes` and masks into `torchvision.tv_tensors.Mask`. As `torchvision.tv_tensors.TVTensor` are `torch.Tensor` subclasses, wrapped objects are also tensors and inherit the plain `torch.Tensor` API. For more information about torchvision `tv_tensors` see [this documentation](#).

```

import os
import torch

from torchvision.io import read_image
from torchvision.ops.bboxes 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
        self.transforms = transforms
        # load all image files, sorting them to
        # ensure that they are aligned
        self.imgs = list(sorted(os.listdir(os.path.join(root, "/content/drive/MyDrive/PennFudanPed/PNGImages"))))
        self.masks = list(sorted(os.listdir(os.path.join(root, "/content/drive/MyDrive/PennFudanPed/PedMasks"))))

    def __getitem__(self, idx):
        # load images and masks
        img_path = os.path.join(self.root, "/content/drive/MyDrive/PennFudanPed/PNGImages", self.imgs[idx])
        mask_path = os.path.join(self.root, "/content/drive/MyDrive/PennFudanPed/PedMasks", self.masks[idx])
        img = read_image(img_path)
        mask = read_image(mask_path)
        # instances are encoded as different colors
        obj_ids = torch.unique(mask)
        # first id is the background, so remove it
        obj_ids = obj_ids[1:]
        num_objs = len(obj_ids)

        # split the color-encoded mask into a set
        # of binary masks
        masks = (mask == obj_ids[:, None, None]).to(dtype=torch.uint8)

        # get bounding box coordinates for each mask
        boxes = masks_to_boxes(masks)

        # there is only one class
        labels = torch.ones((num_objs,), dtype=torch.int64)

        image_id = idx
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
        # suppose all instances are not crowd
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        # Wrap sample and targets into torchvision tv_tensors:
        img = tv_tensors.Image(img)

        target = {}
        target["boxes"] = tv_tensors.BoundingBoxes(boxes, format="XYXY", canvas_size=F.get_size(img))
        target["masks"] = tv_tensors.Mask(masks)
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

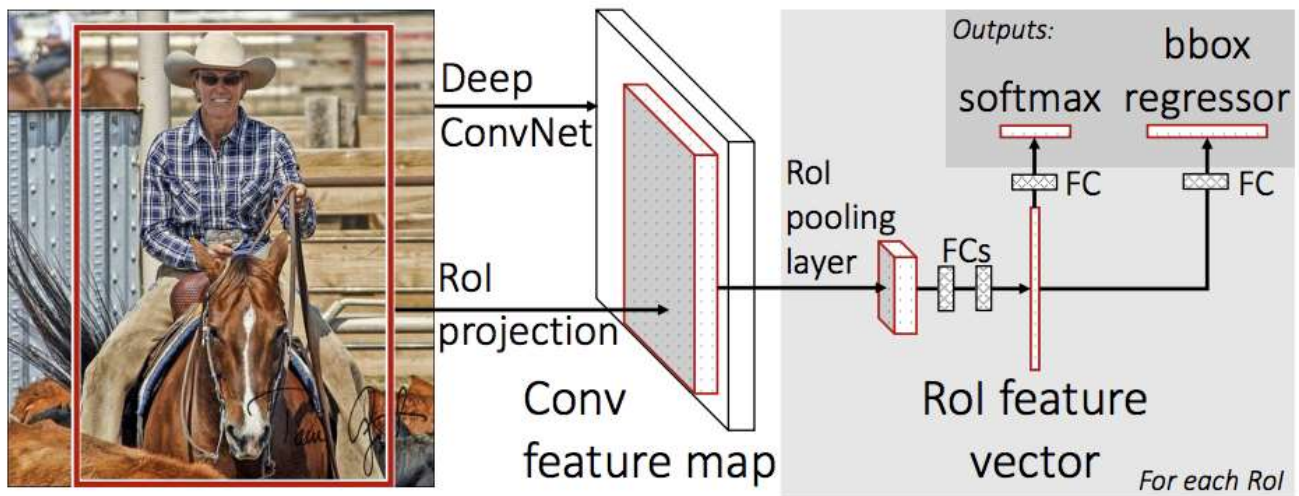
    def __len__(self):
        return len(self.imgs)

```

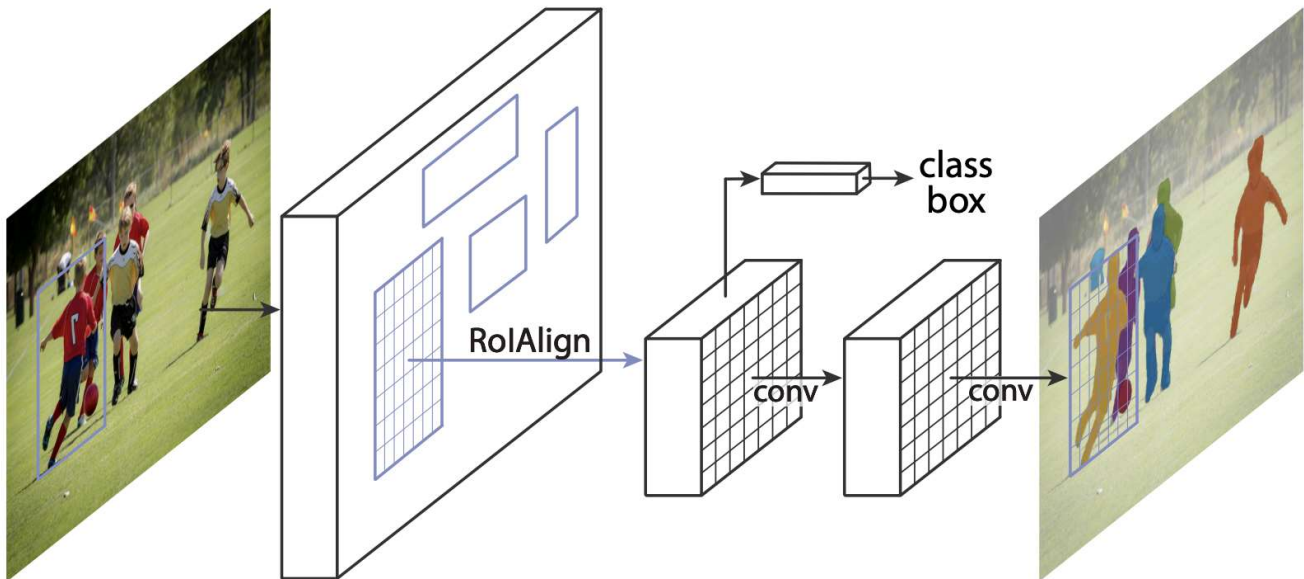
That's all for the dataset. Now let's define a model that can perform predictions on this dataset.

## ✓ Defining your model

In this tutorial, we will be using [Mask R-CNN](#), which is based on top of [Faster R-CNN](#). Faster R-CNN is a model that predicts both bounding boxes and class scores for potential objects in the image.



Mask R-CNN adds an extra branch into Faster R-CNN, which also predicts segmentation masks for each instance.



There are two common situations where one might want to modify one of the available models in TorchVision Model Zoo. The first is when we want to start from a pre-trained model, and just finetune the last layer. The other is when we want to replace the backbone of the model with a different one (for faster predictions, for example).

Let's go see how we would do one or another in the following sections.

## 1 - Finetuning from a pretrained model

Let's suppose that you want to start from a model pre-trained on COCO and want to finetune it for your particular classes. Here is a possible way of doing it:

```

import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

# load a model pre-trained on COCO
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT")

# replace the classifier with a new one, that has
# num_classes which is user-defined
num_classes = 2 # 1 class (person) + background
# get number of input features for the classifier
in_features = model.roi_heads.box_predictor.cls_score.in_features
# replace the pre-trained head with a new one
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

```

📄 Downloading: "[https://download.pytorch.org/models/fasterrcnn\\_resnet50\\_fpn\\_coco-258fb6c6.pth](https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth)" to /root/.cache/torch/hub/checkpoint100%|██████████| 160M/160M [00:01<00:00, 95.7MB/s]



## ✓ 2 - Modifying the model to add a different backbone

```

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

# load a pre-trained model for classification and return
# only the features
backbone = torchvision.models.mobilenet_v2(weights="DEFAULT").features
# ``FasterRCNN`` needs to know the number of
# output channels in a backbone. For mobilenet_v2, it's 1280
# so we need to add it here
backbone.out_channels = 1280

# let's make the RPN generate 5 x 3 anchors per spatial
# location, with 5 different sizes and 3 different aspect
# ratios. We have a Tuple[Tuple[int]] because each feature
# map could potentially have different sizes and
# aspect ratios
anchor_generator = AnchorGenerator(
    sizes=((32, 64, 128, 256, 512),),
    aspect_ratios=((0.5, 1.0, 2.0),)
)

# let's define what are the feature maps that we will
# use to perform the region of interest cropping, as well as
# the size of the crop after rescaling.
# if your backbone returns a Tensor, featmap_names is expected to
# be [0]. More generally, the backbone should return an
# ``OrderedDict[Tensor]``, and in ``featmap_names`` you can choose which
# feature maps to use.
roi_pooler = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0'],
    output_size=7,
    sampling_ratio=2
)

# put the pieces together inside a Faster-RCNN model
model = FasterRCNN(
    backbone,
    num_classes=2,
    rpn_anchor_generator=anchor_generator,
    box_roi_pool=roi_pooler
)

```

📄 Downloading: "[https://download.pytorch.org/models/mobilenet\\_v2-7ebf99e0.pth](https://download.pytorch.org/models/mobilenet_v2-7ebf99e0.pth)" to /root/.cache/torch/hub/checkpoints/mobilenet\_v2-7100%|██████████| 13.6M/13.6M [00:00<00:00, 49.8MB/s]



## ✓ Object detection and instance segmentation model for PennFudan Dataset

In our case, we want to finetune from a pre-trained model, given that our dataset is very small, so we will be following approach number 1.

Here we want to also compute the instance segmentation masks, so we will be using Mask R-CNN:

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor

def get_model_instance_segmentation(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(weights="DEFAULT")

    # get number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    # now get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(
        in_features_mask,
        hidden_layer,
        num_classes
    )

    return model
```

That's it, this will make `model` be ready to be trained and evaluated on your custom dataset.

## ✓ Putting everything together

In `references/detection/`, we have a number of helper functions to simplify training and evaluating detection models. Here, we will use `references/detection/engine.py` and `references/detection/utils.py`. Just download everything under `references/detection` to your folder and use them here. On Linux if you have `wget`, you can download them using below commands:

Since v0.15.0 torchvision provides [new Transforms API](#) to easily write data augmentation pipelines for Object Detection and Segmentation tasks.

Let's write some helper functions for data augmentation / transformation:

```
from torchvision.transforms import v2 as T

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

## ✓ Testing forward() method (Optional)

Before iterating over the dataset, it's good to see what the model expects during training and inference time on sample data.

```
import utils

model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT")
dataset = PennFudanDataset('data/PennFudanPed', get_transform(train=True))
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2
```




```

        batch_size=4,
        shuffle=True,
        num_workers=4,
        collate_fn=utils.collate_fn
    )

# For Training
images, targets = next(iter(data_loader))
images = list(image for image in images)
targets = [{k: v for k, v in t.items()} for t in targets]
output = model(images, targets) # Returns losses and detections
print(output)

# For inference
model.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
predictions = model(x) # Returns predictions
print(predictions[0])

```

 /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:558: UserWarning  
warnings.warn(\_create\_warning\_msg(

```

-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-15-2dfb92c3e36f> in <cell line: 18>()
    16 images = list(image for image in images)
    17 targets = [{k: v for k, v in t.items()} for t in targets]
--> 18 output = model(images, targets) # Returns losses and detections
    19 print(output)
    20

```

```

----- 12 frames -----
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/conv.py in
_conv_forward(self, input, weight, bias)
    454         weight, bias, self.stride,
    455         _pair(0), self.dilation, self.groups)
--> 456         return F.conv2d(input, weight, bias, self.stride,
    457                         self.padding, self.dilation, self.groups)
    458

```

KeyboardInterrupt:

Let's now write the main function which performs the training and the validation:

```

os.system("wget https://raw.githubusercontent.com/pytorch/vision/main/references/detection/engine.py")
os.system("wget https://raw.githubusercontent.com/pytorch/vision/main/references/detection/utils.py")
os.system("wget https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_utils.py")
os.system("wget https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_eval.py")
os.system("wget https://raw.githubusercontent.com/pytorch/vision/main/references/detection/transforms.py")

```

 0

```

import utils
from engine import train_one_epoch, evaluate

# train on the GPU or on the CPU, if a GPU is not available
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

# our dataset has two classes only - background and person
num_classes = 2
# use our dataset and defined transformations
dataset = PennFudanDataset('data/PennFudanPed', get_transform(train=True))
dataset_test = PennFudanDataset('data/PennFudanPed', get_transform(train=False))

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

# define training and validation data loaders
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True,
    num_workers=4,
    collate_fn=utils.collate_fn
)

data_loader_test = torch.utils.data.DataLoader(
    dataset_test,
    batch_size=1,
    shuffle=False,
    num_workers=4,
    collate_fn=utils.collate_fn
)

# get the model using our helper function
model = get_model_instance_segmentation(num_classes)

# move model to the right device
model.to(device)

# construct an optimizer
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
)

# and a learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.1
)

# let's train it just for 2 epochs
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, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)

print("That's it!")

```





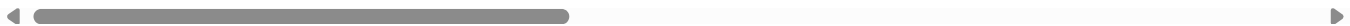
```

Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.756
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.325
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.769
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.769
Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.533
Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.700
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.780
Epoch: [1] [ 0/60] eta: 0:00:53 lr: 0.005000 loss: 0.1850 (0.1850) loss_classifier: 0.0179 (0.0179) loss_box_reg: 0.0499
Epoch: [1] [10/60] eta: 0:00:29 lr: 0.005000 loss: 0.2655 (0.2674) loss_classifier: 0.0277 (0.0317) loss_box_reg: 0.0836
Epoch: [1] [20/60] eta: 0:00:23 lr: 0.005000 loss: 0.3031 (0.3044) loss_classifier: 0.0407 (0.0408) loss_box_reg: 0.1063
Epoch: [1] [30/60] eta: 0:00:16 lr: 0.005000 loss: 0.2756 (0.2986) loss_classifier: 0.0411 (0.0395) loss_box_reg: 0.1002
Epoch: [1] [40/60] eta: 0:00:11 lr: 0.005000 loss: 0.2470 (0.2902) loss_classifier: 0.0260 (0.0387) loss_box_reg: 0.0766
Epoch: [1] [50/60] eta: 0:00:05 lr: 0.005000 loss: 0.2441 (0.2811) loss_classifier: 0.0260 (0.0377) loss_box_reg: 0.0619
Epoch: [1] [59/60] eta: 0:00:00 lr: 0.005000 loss: 0.2312 (0.2743) loss_classifier: 0.0340 (0.0366) loss_box_reg: 0.0569
Epoch: [1] Total time: 0:00:33 (0.5609 s / it)
creating index...
index created!
Test: [ 0/50] eta: 0:00:18 model_time: 0.1436 (0.1436) evaluator_time: 0.0034 (0.0034) time: 0.3761 data: 0.2275 max men
Test: [49/50] eta: 0:00:00 model_time: 0.0970 (0.1023) evaluator_time: 0.0036 (0.0048) time: 0.1078 data: 0.0041 max men
Test: Total time: 0:00:05 (0.1187 s / it)
Averaged stats: model_time: 0.0970 (0.1023) evaluator_time: 0.0036 (0.0048)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.777
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.988
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.936
Average Precision (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.373
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.611
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.793
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.349
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.820
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.820
Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.533
Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.787
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.829
IoU metric: segm
Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.755
Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.988
Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.902
Average Precision (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.365
Average Precision (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.615
Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.771
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.337
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.792
Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.792
Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.567
Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.762
Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.800
That's it!

```

So after one epoch of training, we obtain a COCO-style mAP > 50, and a mask mAP of 65.

But what do the predictions look like? Let's take one image in the dataset and verify



```

import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes, draw_segmentation_masks

image = read_image("/content/drive/MyDrive/PennFudanPed/PNGImages/FudanPed00046.png")
eval_transform = get_transform(train=False)


model.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model([x, ])
    pred = predictions[0]

image = (255.0 * (image - image.min()) / (image.max() - image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels, colors="red")

masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5, colors="blue")

plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))

```

 <matplotlib.image.AxesImage at 0x7de7731e4280>

