This is for Question 5 Part A

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

from google.colab import drive
drive.mount('/content/drive') #Updated the spots where they needed the
locations to change, also downloaded the PennFundan dataset and
tutorial source for the test photo at the end
# Dataset: https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip
# Test Photo:
https://github.com/pytorch/tutorials/blob/d686b662932a380a58b7683425fa
a00c06bcf502/_static/img/tv_tutorial/tv_image05.png
#Source Code:
https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

Mounted at /content/drive
```

TorchVision Object Detection Finetuning Tutorial

.. tip::

```
To get the most of this tutorial, we suggest using this
[Colab
Version](https://colab.research.google.com/github/pytorch/tutorials/
blob/gh-pages/_downloads/
torchvision_finetuning_instance_segmentation.ipynb)_.
This will allow you to experiment with the information presented
below.
```

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.

.. note ::

```
This tutorial works only with torchvision version >=0.16 or nightly.
If you're using torchvision<=0.15, please follow
[this tutorial
instead](https://github.com/pytorch/tutorials/blob/d686b662932a380a58b
7683425faa00c06bcf502/intermediate_source/torchvision_tutorial.rst).
```

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` class, and implement `__len__` and `__getitem__`.

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

- image: :class:`torchvision.tv_tensors.Image` of shape `[3, H, W]`, a pure tensor, or a PIL Image of size `(H, W)`
- target: a dict containing the following fields
  - `boxes`, :class:`torchvision.tv_tensors.BoundingBoxes` 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 :class:`torch.Tensor` 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 :class:`torch.Tensor` 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 :class:`torch.Tensor` of shape `[N]`: instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks`, :class:`torchvision.tv_tensors.Mask` 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`.

.. note :: For Windows, please install `pycocotools` from gautamchitnis_ with command

```
pip install
git+https://github.com/gautamchitnis/cocoapi.git@cocodataset-
master#subdirectory=PythonAPI
```

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. After downloading and extracting the zip file_, 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

So each image has a corresponding segmentation mask, where each color correspond to a different instance. Let's write a :class:`torch.utils.data.Dataset` class for this dataset. In the code below, we are wrapping images, bounding boxes and masks into `torchvision.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 :class:`torchvision.tv_tensors.Image`, bounding boxes into :class:`torchvision.tv_tensors.BoundingBoxes` and masks into :class:`torchvision.tv_tensors.Mask`. As `torchvision.TVTensor` are :class:`torch.Tensor` subclasses, wrapped objects are also tensors and inherit the plain :class:`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.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
        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,
"PNGImages"))))
        self.masks = list(sorted(os.listdir(os.path.join(root,
"PedMasks"))))

    def __getitem__(self, idx):
        # load images and masks
```

```
        img_path = os.path.join(self.root, "PNGImages",
self.imgs[idx])
        mask_path = os.path.join(self.root, "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" to
/root/.cache/torch/hub/checkpoints/fasterrcnn_resnet50_fpn_coco-
258fb6c6.pth
100%|██████████| 160M/160M [00:01<00:00, 141MB/s]
```

2 - Modifying the model to add a different backbone

```python
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" to /root/.cache/torch/hub/checkpoints/mobilenet_v2-
7ebf99e0.pth
100%|██████████| 13.6M/13.6M [00:00<00:00, 32.3MB/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:

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

# Since v0.15.0 torchvision provides `new Transforms API
<https://pytorch.org/vision/stable/transforms.html>`_
# 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('drive/MyDrive/data/PennFudanPed',
get_transform(train=True))
data_loader = torch.utils.data.DataLoader(
```

```
    dataset,
    batch_size=2,
    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:557: UserWarning: This DataLoader will create 4 worker
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

```
{'loss_classifier': tensor(0.1144, grad_fn=<NllLossBackward0>),
'loss_box_reg': tensor(0.0404, grad_fn=<DivBackward0>),
'loss_objectness': tensor(0.0071,
grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_rpn_box_reg':
tensor(0.0054, grad_fn=<DivBackward0>)}
{'boxes': tensor([], size=(0, 4), grad_fn=<StackBackward0>), 'labels':
tensor([], dtype=torch.int64), 'scores': tensor([],
grad_fn=<IndexBackward0>)}
```

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

```
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('drive/MyDrive/data/PennFudanPed',
```

```python
get_transform(train=True))
dataset_test = PennFudanDataset('drive/MyDrive/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 for 5 epochs
num_epochs = 5
```

```python
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!")
```

```
Downloading:
"https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-
bf2d0c1e.pth" to
/root/.cache/torch/hub/checkpoints/maskrcnn_resnet50_fpn_coco-
bf2d0c1e.pth
100%|██████████| 170M/170M [00:01<00:00, 140MB/s]
```

```
Epoch: [0]  [ 0/60]  eta: 0:08:32  lr: 0.000090  loss: 2.9040 (2.9040)
loss_classifier: 0.8063 (0.8063)  loss_box_reg: 0.2864 (0.2864)
loss_mask: 1.7528 (1.7528)  loss_objectness: 0.0557 (0.0557)
loss_rpn_box_reg: 0.0028 (0.0028)  time: 8.5405  data: 0.7895  max
mem: 2148
Epoch: [0]  [10/60]  eta: 0:01:11  lr: 0.000936  loss: 1.4138 (1.8127)
loss_classifier: 0.5713 (0.5281)  loss_box_reg: 0.2840 (0.2832)
loss_mask: 0.5955 (0.9743)  loss_objectness: 0.0220 (0.0239)
loss_rpn_box_reg: 0.0028 (0.0032)  time: 1.4238  data: 0.0855  max
mem: 3041
Epoch: [0]  [20/60]  eta: 0:00:40  lr: 0.001783  loss: 0.8679 (1.2905)
loss_classifier: 0.2435 (0.3770)  loss_box_reg: 0.2475 (0.2576)
loss_mask: 0.3593 (0.6307)  loss_objectness: 0.0164 (0.0204)
loss_rpn_box_reg: 0.0033 (0.0048)  time: 0.6335  data: 0.0115  max
mem: 3041
Epoch: [0]  [30/60]  eta: 0:00:26  lr: 0.002629  loss: 0.6184 (1.0573)
loss_classifier: 0.1425 (0.2887)  loss_box_reg: 0.2219 (0.2575)
loss_mask: 0.2009 (0.4885)  loss_objectness: 0.0079 (0.0166)
loss_rpn_box_reg: 0.0073 (0.0060)  time: 0.5860  data: 0.0089  max
mem: 3041
Epoch: [0]  [40/60]  eta: 0:00:16  lr: 0.003476  loss: 0.4260 (0.8925)
loss_classifier: 0.0628 (0.2297)  loss_box_reg: 0.1934 (0.2366)
loss_mask: 0.1704 (0.4070)  loss_objectness: 0.0042 (0.0136)
loss_rpn_box_reg: 0.0039 (0.0056)  time: 0.5971  data: 0.0095  max
mem: 3041
Epoch: [0]  [50/60]  eta: 0:00:07  lr: 0.004323  loss: 0.3985 (0.7969)
loss_classifier: 0.0396 (0.1942)  loss_box_reg: 0.1551 (0.2237)
loss_mask: 0.1441 (0.3614)  loss_objectness: 0.0022 (0.0119)
loss_rpn_box_reg: 0.0034 (0.0057)  time: 0.5937  data: 0.0086  max
mem: 3041
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2984 (0.7316)
```

```
loss_classifier: 0.0380 (0.1735)  loss_box_reg: 0.1196 (0.2106)
loss_mask: 0.1490 (0.3309)  loss_objectness: 0.0013 (0.0106)
loss_rpn_box_reg: 0.0044 (0.0059)  time: 0.6078  data: 0.0084  max
mem: 3041
Epoch: [0] Total time: 0:00:44 (0.7489 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:25  model_time: 0.2112 (0.2112)
evaluator_time: 0.0035 (0.0035)  time: 0.5069  data: 0.2905  max mem:
3041
Test:  [49/50]  eta: 0:00:00  model_time: 0.1151 (0.1320)
evaluator_time: 0.0060 (0.0107)  time: 0.1344  data: 0.0054  max mem:
3041
Test: Total time: 0:00:08 (0.1603 s / it)
Averaged stats: model_time: 0.1151 (0.1320)  evaluator_time: 0.0060
(0.0107)
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 |
maxDets=100 ] = 0.712
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.978
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.887
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.418
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.445
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.724
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.321
 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 ] = 0.567
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.800
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.774
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.737
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
```

```
maxDets=100 ] = 0.981
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.912
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.362
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.340
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.750
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.319
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.774
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.775
 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 ] = 0.733
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.782
Epoch: [1]  [ 0/60]  eta: 0:00:56  lr: 0.005000  loss: 0.2784 (0.2784)
loss_classifier: 0.0530 (0.0530)  loss_box_reg: 0.0800 (0.0800)
loss_mask: 0.1416 (0.1416)  loss_objectness: 0.0007 (0.0007)
loss_rpn_box_reg: 0.0032 (0.0032)  time: 0.9449  data: 0.3936  max
mem: 3041
Epoch: [1]  [10/60]  eta: 0:00:31  lr: 0.005000  loss: 0.2784 (0.2844)
loss_classifier: 0.0315 (0.0372)  loss_box_reg: 0.0800 (0.0917)
loss_mask: 0.1416 (0.1491)  loss_objectness: 0.0007 (0.0011)
loss_rpn_box_reg: 0.0037 (0.0053)  time: 0.6239  data: 0.0419  max
mem: 3041
Epoch: [1]  [20/60]  eta: 0:00:24  lr: 0.005000  loss: 0.2673 (0.2773)
loss_classifier: 0.0337 (0.0367)  loss_box_reg: 0.0789 (0.0854)
loss_mask: 0.1440 (0.1488)  loss_objectness: 0.0007 (0.0015)
loss_rpn_box_reg: 0.0037 (0.0049)  time: 0.5967  data: 0.0098  max
mem: 3041
Epoch: [1]  [30/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2713 (0.2893)
loss_classifier: 0.0357 (0.0387)  loss_box_reg: 0.0829 (0.0895)
loss_mask: 0.1530 (0.1546)  loss_objectness: 0.0008 (0.0015)
loss_rpn_box_reg: 0.0041 (0.0049)  time: 0.6009  data: 0.0110  max
mem: 3041
Epoch: [1]  [40/60]  eta: 0:00:12  lr: 0.005000  loss: 0.2660 (0.2844)
loss_classifier: 0.0383 (0.0390)  loss_box_reg: 0.0762 (0.0884)
loss_mask: 0.1448 (0.1502)  loss_objectness: 0.0011 (0.0019)
loss_rpn_box_reg: 0.0041 (0.0048)  time: 0.5973  data: 0.0103  max
mem: 3041
Epoch: [1]  [50/60]  eta: 0:00:06  lr: 0.005000  loss: 0.2605 (0.2799)
loss_classifier: 0.0405 (0.0392)  loss_box_reg: 0.0762 (0.0858)
loss_mask: 0.1319 (0.1476)  loss_objectness: 0.0021 (0.0020)
```

```
loss_rpn_box_reg: 0.0040 (0.0053)  time: 0.5930  data: 0.0102  max
mem: 3132
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2641 (0.2789)
loss_classifier: 0.0406 (0.0397)  loss_box_reg: 0.0738 (0.0848)
loss_mask: 0.1278 (0.1472)  loss_objectness: 0.0005 (0.0019)
loss_rpn_box_reg: 0.0037 (0.0054)  time: 0.5962  data: 0.0086  max
mem: 3132
Epoch: [1] Total time: 0:00:36 (0.6078 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:25  model_time: 0.1575 (0.1575)
evaluator_time: 0.0035 (0.0035)  time: 0.5199  data: 0.3574  max mem:
3132
Test:  [49/50]  eta: 0:00:00  model_time: 0.1032 (0.1119)
evaluator_time: 0.0036 (0.0054)  time: 0.1195  data: 0.0038  max mem:
3132
Test: Total time: 0:00:06 (0.1344 s / it)
Averaged stats: model_time: 0.1032 (0.1119)  evaluator_time: 0.0036
(0.0054)
Accumulating evaluation results...
DONE (t=0.02s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.816
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.987
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.960
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.440
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.489
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.829
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.362
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.852
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.852
 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 ] = 0.867
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.860
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 ] = 0.989
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.921
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.429
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.421
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.766
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.331
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.790
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.790
 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 ] = 0.733
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.797
Epoch: [2]  [ 0/60]  eta: 0:01:04  lr: 0.005000  loss: 0.2961 (0.2961)
loss_classifier: 0.0745 (0.0745)  loss_box_reg: 0.0721 (0.0721)
loss_mask: 0.1432 (0.1432)  loss_objectness: 0.0002 (0.0002)
loss_rpn_box_reg: 0.0062 (0.0062)  time: 1.0802  data: 0.4941  max
mem: 3132
Epoch: [2]  [10/60]  eta: 0:00:32  lr: 0.005000  loss: 0.2604 (0.2647)
loss_classifier: 0.0426 (0.0437)  loss_box_reg: 0.0721 (0.0735)
loss_mask: 0.1367 (0.1407)  loss_objectness: 0.0006 (0.0016)
loss_rpn_box_reg: 0.0046 (0.0051)  time: 0.6582  data: 0.0519  max
mem: 3132
Epoch: [2]  [20/60]  eta: 0:00:25  lr: 0.005000  loss: 0.2255 (0.2370)
loss_classifier: 0.0294 (0.0360)  loss_box_reg: 0.0615 (0.0648)
loss_mask: 0.1238 (0.1304)  loss_objectness: 0.0006 (0.0013)
loss_rpn_box_reg: 0.0029 (0.0044)  time: 0.6032  data: 0.0079  max
mem: 3132
Epoch: [2]  [30/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2249 (0.2365)
loss_classifier: 0.0272 (0.0351)  loss_box_reg: 0.0517 (0.0633)
loss_mask: 0.1255 (0.1322)  loss_objectness: 0.0008 (0.0014)
loss_rpn_box_reg: 0.0029 (0.0044)  time: 0.5744  data: 0.0084  max
mem: 3132
Epoch: [2]  [40/60]  eta: 0:00:12  lr: 0.005000  loss: 0.2193 (0.2306)
loss_classifier: 0.0292 (0.0344)  loss_box_reg: 0.0504 (0.0618)
loss_mask: 0.1233 (0.1292)  loss_objectness: 0.0006 (0.0012)
loss_rpn_box_reg: 0.0031 (0.0040)  time: 0.5783  data: 0.0091  max
mem: 3132
```

```
Epoch: [2]  [50/60]  eta: 0:00:06  lr: 0.005000  loss: 0.2193 (0.2335)
loss_classifier: 0.0298 (0.0348)  loss_box_reg: 0.0609 (0.0630)
loss_mask: 0.1224 (0.1302)  loss_objectness: 0.0005 (0.0013)
loss_rpn_box_reg: 0.0031 (0.0042)  time: 0.5997  data: 0.0085  max
mem: 3132
Epoch: [2]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2122 (0.2300)
loss_classifier: 0.0267 (0.0341)  loss_box_reg: 0.0508 (0.0611)
loss_mask: 0.1272 (0.1294)  loss_objectness: 0.0003 (0.0012)
loss_rpn_box_reg: 0.0033 (0.0041)  time: 0.5890  data: 0.0075  max
mem: 3132
Epoch: [2] Total time: 0:00:36 (0.6033 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:25  model_time: 0.1435 (0.1435)
evaluator_time: 0.0036 (0.0036)  time: 0.5156  data: 0.3669  max mem:
3132
Test:  [49/50]  eta: 0:00:00  model_time: 0.1125 (0.1128)
evaluator_time: 0.0052 (0.0057)  time: 0.1273  data: 0.0051  max mem:
3132
Test: Total time: 0:00:06 (0.1388 s / it)
Averaged stats: model_time: 0.1125 (0.1128)  evaluator_time: 0.0052
(0.0057)
Accumulating evaluation results...
DONE (t=0.03s).
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.801
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.985
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.942
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.465
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.614
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.813
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.353
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.834
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.834
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.467
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.867
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.842
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.771
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.993
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.946
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.534
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.335
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.785
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.335
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.803
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.803
 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 ] = 0.633
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.814
Epoch: [3]  [ 0/60]  eta: 0:01:26  lr: 0.000500  loss: 0.1888 (0.1888)
loss_classifier: 0.0211 (0.0211)  loss_box_reg: 0.0529 (0.0529)
loss_mask: 0.1116 (0.1116)  loss_objectness: 0.0003 (0.0003)
loss_rpn_box_reg: 0.0029 (0.0029)  time: 1.4483  data: 0.7804  max
mem: 3132
Epoch: [3]  [10/60]  eta: 0:00:34  lr: 0.000500  loss: 0.1989 (0.2184)
loss_classifier: 0.0280 (0.0328)  loss_box_reg: 0.0579 (0.0566)
loss_mask: 0.1126 (0.1228)  loss_objectness: 0.0004 (0.0013)
loss_rpn_box_reg: 0.0040 (0.0050)  time: 0.6991  data: 0.0769  max
mem: 3409
Epoch: [3]  [20/60]  eta: 0:00:25  lr: 0.000500  loss: 0.1989 (0.2140)
loss_classifier: 0.0270 (0.0307)  loss_box_reg: 0.0579 (0.0554)
loss_mask: 0.1194 (0.1229)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0027 (0.0040)  time: 0.6098  data: 0.0080  max
mem: 3409
Epoch: [3]  [30/60]  eta: 0:00:18  lr: 0.000500  loss: 0.1789 (0.2006)
loss_classifier: 0.0226 (0.0274)  loss_box_reg: 0.0390 (0.0468)
loss_mask: 0.1171 (0.1218)  loss_objectness: 0.0002 (0.0010)
loss_rpn_box_reg: 0.0023 (0.0035)  time: 0.5718  data: 0.0096  max
mem: 3409
Epoch: [3]  [40/60]  eta: 0:00:12  lr: 0.000500  loss: 0.1722 (0.1987)
loss_classifier: 0.0218 (0.0269)  loss_box_reg: 0.0320 (0.0467)
```

```
loss_mask: 0.1109 (0.1207)  loss_objectness: 0.0002 (0.0010)
loss_rpn_box_reg: 0.0024 (0.0033)  time: 0.5643  data: 0.0095  max
mem: 3409
Epoch: [3]  [50/60]  eta: 0:00:06  lr: 0.000500  loss: 0.1764 (0.1961)
loss_classifier: 0.0250 (0.0269)  loss_box_reg: 0.0332 (0.0465)
loss_mask: 0.1058 (0.1185)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0018 (0.0031)  time: 0.5900  data: 0.0113  max
mem: 3409
Epoch: [3]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1764 (0.1959)
loss_classifier: 0.0283 (0.0270)  loss_box_reg: 0.0430 (0.0465)
loss_mask: 0.1093 (0.1183)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0017 (0.0031)  time: 0.5978  data: 0.0102  max
mem: 3409
Epoch: [3] Total time: 0:00:36 (0.6087 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:37  model_time: 0.2348 (0.2348)
evaluator_time: 0.0051 (0.0051)  time: 0.7442  data: 0.5027  max mem:
3409
Test:  [49/50]  eta: 0:00:00  model_time: 0.1037 (0.1161)
evaluator_time: 0.0034 (0.0056)  time: 0.1183  data: 0.0035  max mem:
3409
Test: Total time: 0:00:07 (0.1430 s / it)
Averaged stats: model_time: 0.1037 (0.1161)  evaluator_time: 0.0034
(0.0056)
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 ] = 0.829
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.993
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.955
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.499
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.549
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.842
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.365
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.860
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.860
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
```

```
maxDets=100 ] = 0.500
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.867
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.869
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.780
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.993
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.947
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.490
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.330
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.791
 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 ] = 0.810
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.810
 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 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.820
Epoch: [4]  [ 0/60]  eta: 0:01:05  lr: 0.000500  loss: 0.1847 (0.1847)
loss_classifier: 0.0303 (0.0303)  loss_box_reg: 0.0470 (0.0470)
loss_mask: 0.1053 (0.1053)  loss_objectness: 0.0003 (0.0003)
loss_rpn_box_reg: 0.0018 (0.0018)  time: 1.0919  data: 0.4482  max
mem: 3409
Epoch: [4]  [10/60]  eta: 0:00:31  lr: 0.000500  loss: 0.1847 (0.1761)
loss_classifier: 0.0267 (0.0258)  loss_box_reg: 0.0378 (0.0385)
loss_mask: 0.1053 (0.1082)  loss_objectness: 0.0006 (0.0013)
loss_rpn_box_reg: 0.0023 (0.0023)  time: 0.6351  data: 0.0488  max
mem: 3409
Epoch: [4]  [20/60]  eta: 0:00:23  lr: 0.000500  loss: 0.1661 (0.1715)
loss_classifier: 0.0193 (0.0226)  loss_box_reg: 0.0287 (0.0339)
loss_mask: 0.1105 (0.1118)  loss_objectness: 0.0006 (0.0013)
loss_rpn_box_reg: 0.0015 (0.0020)  time: 0.5681  data: 0.0099  max
mem: 3409
Epoch: [4]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1761 (0.1765)
loss_classifier: 0.0203 (0.0227)  loss_box_reg: 0.0295 (0.0343)
loss_mask: 0.1157 (0.1161)  loss_objectness: 0.0005 (0.0010)
loss_rpn_box_reg: 0.0015 (0.0024)  time: 0.5754  data: 0.0097  max
```

```
mem: 3409
Epoch: [4]  [40/60]  eta: 0:00:12  lr: 0.000500  loss: 0.1798 (0.1763)
loss_classifier: 0.0241 (0.0237)  loss_box_reg: 0.0340 (0.0355)
loss_mask: 0.1105 (0.1136)  loss_objectness: 0.0004 (0.0010)
loss_rpn_box_reg: 0.0018 (0.0024)  time: 0.6095  data: 0.0096  max
mem: 3409
Epoch: [4]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1686 (0.1776)
loss_classifier: 0.0243 (0.0249)  loss_box_reg: 0.0307 (0.0363)
loss_mask: 0.1059 (0.1129)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0021 (0.0026)  time: 0.5992  data: 0.0092  max
mem: 3409
Epoch: [4]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1707 (0.1810)
loss_classifier: 0.0243 (0.0256)  loss_box_reg: 0.0365 (0.0379)
loss_mask: 0.1098 (0.1140)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0020 (0.0026)  time: 0.5792  data: 0.0075  max
mem: 3409
Epoch: [4] Total time: 0:00:36 (0.6021 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:26  model_time: 0.1503 (0.1503)
evaluator_time: 0.0039 (0.0039)  time: 0.5395  data: 0.3835  max mem:
3409
Test:  [49/50]  eta: 0:00:00  model_time: 0.1037 (0.1115)
evaluator_time: 0.0036 (0.0049)  time: 0.1197  data: 0.0040  max mem:
3409
Test: Total time: 0:00:06 (0.1344 s / it)
Averaged stats: model_time: 0.1037 (0.1115)  evaluator_time: 0.0036
(0.0049)
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 ] = 0.848
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.993
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.955
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.533
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.549
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.860
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.373
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.874
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.874
 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 ] = 0.867
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.883
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.783
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.993
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.947
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.512
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.353
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.796
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.339
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.815
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.815
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.600
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.824
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("drive/MyDrive/_static/img/tv_tutorial/tv_image05.png")
eval_transform = get_transform(train=False)
```

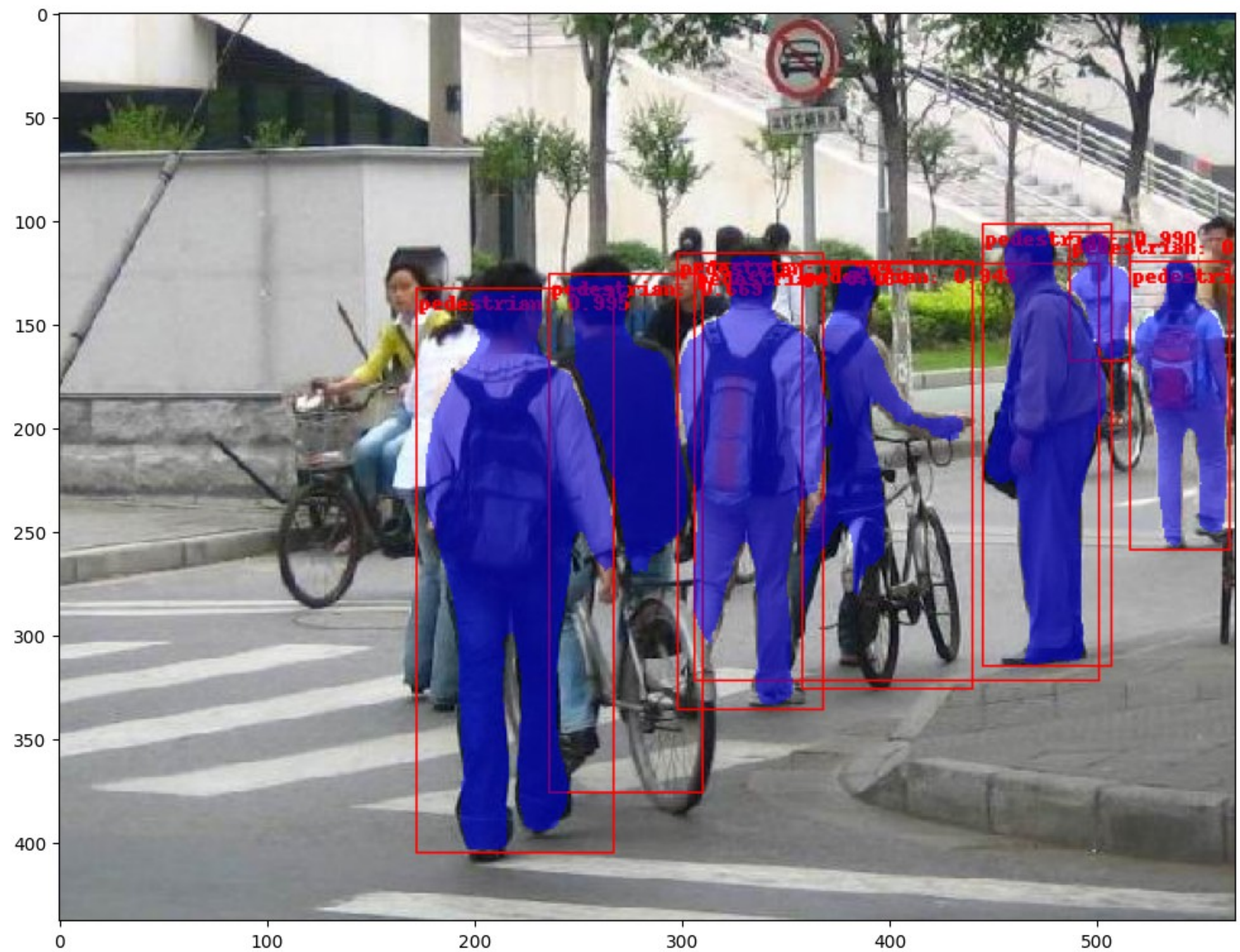
```
model.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model([x, ])
    pred = predictions[0]


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 0x7fe26fd7dcf0>
```



The results look good!

Wrapping up

In this tutorial, you have learned how to create your own training pipeline for object detection models on a custom dataset. For that, you wrote a `torch.utils.data.Dataset` class that returns the images and the ground truth boxes and segmentation masks. You also leveraged a Mask R-CNN model pre-trained on COCO train2017 in order to perform transfer learning on this new dataset.

For a more complete example, which includes multi-machine / multi-GPU training, check `references/detection/train.py`, which is present in the torchvision repository.

You can download a full source file for this tutorial here_.