TorchVision Instance Segmentation Finetuning Tutorial

For this tutorial, we will be finetuning a pre-trained Mask R-CNN (https://arxiv.org/abs/1703.06870) model in the *Penn-Fudan Database for Pedestrian Detection and Segmentation* (https://www.cis.upenn.edu/~jshi/ped_html/). 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 instance segmentation model on a custom dataset.

First, we need to install `pycocotools`. This library will be used for computing the evaluation metrics following the COCO metric for intersection over union.

In [1]:
```
%%shell

pip install cython
# Install pycocotools, the version by default in Colab
# has a bug fixed in https://github.com/cocodataset/cocoapi/pull/354
pip install -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) h
ttps://us-python.pkg.dev/colab-wheels/public/simple/ (https://us-python.p
kg.dev/colab-wheels/public/simple/)
Requirement already satisfied: cython in /usr/local/lib/python3.9/dist-pa
ckages (0.29.33)
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) h
ttps://us-python.pkg.dev/colab-wheels/public/simple/ (https://us-python.p
kg.dev/colab-wheels/public/simple/)
Collecting git+https://github.com/cocodataset/cocoapi.git#subdirectory=Py
thonAPI
  Cloning https://github.com/cocodataset/cocoapi.git (https://github.com/
cocodataset/cocoapi.git) to /tmp/pip-req-build-acn71qm_
  Running command git clone --filter=blob:none --quiet https://github.co
m/cocodataset/cocoapi.git (https://github.com/cocodataset/cocoapi.git) /t
mp/pip-req-build-acn71qm_
  Resolved https://github.com/cocodataset/cocoapi.git (https://github.co
m/cocodataset/cocoapi.git) to commit 8c9bcc3cf640524c4c20a9c40e89cb6a2f2f
a0e9
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools>=18.0 in /usr/local/lib/python
3.9/dist-packages (from pycocotools==2.0) (57.4.0)
Requirement already satisfied: cython>=0.27.3 in /usr/local/lib/python3.
9/dist-packages (from pycocotools==2.0) (0.29.33)
Requirement already satisfied: matplotlib>=2.1.0 in /usr/local/lib/python
3.9/dist-packages (from pycocotools==2.0) (3.5.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python
3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.4.4)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.9/
dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (8.4.0)
Requirement already satisfied: pyparsing>=2.2.1 in /usr/local/lib/python
3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (3.0.9)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.9/d
ist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (0.11.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/di
st-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.22.4)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python
3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (4.39.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pyt
hon3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (2.8.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.
9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (23.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-
packages (from python-dateutil>=2.7->matplotlib>=2.1.0->pycocotools==2.0)
(1.15.0)
Building wheels for collected packages: pycocotools
  Building wheel for pycocotools (setup.py) ... done
  Created wheel for pycocotools: filename=pycocotools-2.0-cp39-cp39-linux
_x86_64.whl size=397886 sha256=8f2ae7f7df7be4ae93faf5b14bcd674f002d508758
2e86fc4c4c9fc63d2b91e1
  Stored in directory: /tmp/pip-ephem-wheel-cache-pxm7_weo/wheels/13/c1/d
```

```
6/a321055f7089f1a6af654fbf794536b196999f082a9cb68a37
Successfully built pycocotools
Installing collected packages: pycocotools
  Attempting uninstall: pycocotools
    Found existing installation: pycocotools 2.0.6
    Uninstalling pycocotools-2.0.6:
      Successfully uninstalled pycocotools-2.0.6
Successfully installed pycocotools-2.0
```

Out[1]:

Defining the Dataset

The [torchvision reference scripts for training object detection, instance segmentation and person keypoint detection (https://github.com/pytorch/vision/tree/v0.3.0/references/detection)](https://github.com/pytorch/vision/tree/v0.3.0/references/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:

- image: a PIL Image of size (H, W)
- target: a dict containing the following fields
  - `boxes` (`FloatTensor[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` (`Int64Tensor[N]`): the label for each bounding box
  - `image_id` (`Int64Tensor[1]`): an image identifier. It should be unique between all the images in the dataset, and is used during evaluation
  - `area` (`Tensor[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` (`UInt8Tensor[N]`): instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks` (`UInt8Tensor[N, H, W]`): The segmentation masks for each one of the objects
  - (optionally) `keypoints` (`FloatTensor[N, K, 3]`): For each one of the `N` objects, it contains the `K` keypoints in `[x, y, visibility]` format, defining the object. `visibility=0` means that the keypoint is not visible. Note that for data augmentation, the notion of flipping a keypoint is dependent on the data representation, and you should probably adapt `references/detection/transforms.py` for your new keypoint representation

If your model returns the above methods, they will make it work for both training and evaluation, and will use the evaluation scripts from 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 ratio), 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 Penn-Fudan

Let's write a dataset for the Penn-Fudan dataset.

First, let's download and extract the data, present in a zip file at https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip (https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip)

In [2]:
```
%%shell

# download the Penn-Fudan dataset
wget https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip .
# extract it in the current folder
unzip PennFudanPed.zip
```

```
--2023-03-10 21:45:51--  https://www.cis.upenn.edu/~jshi/ped_html/PennFud
anPed.zip (https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip)
Resolving www.cis.upenn.edu (www.cis.upenn.edu)... 158.130.69.163, 2607:f
470:8:64:5ea5::d
Connecting to www.cis.upenn.edu (www.cis.upenn.edu)|158.130.69.163|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: 53723336 (51M) [application/zip]
Saving to: 'PennFudanPed.zip'

PennFudanPed.zip    100%[===================>]  51.23M  11.4MB/s    in 5.
6s

2023-03-10 21:45:57 (9.21 MB/s) - 'PennFudanPed.zip' saved [53723336/5372
3336]

--2023-03-10 21:45:57--  http://./ (http://./)
Resolving . (.)... failed: No address associated with hostname.
wget: unable to resolve host address '.'
```

Let's have a look at the dataset and how it is layed down.

The data is structured as follows

```
PennFudanPed/
  PedMasks/
    FudanPed00001_mask.png
    FudanPed00002_mask.png
    FudanPed00003_mask.png
    FudanPed00004_mask.png
    ...
  PNGImages/
    FudanPed00001.png
    FudanPed00002.png
    FudanPed00003.png
    FudanPed00004.png
```

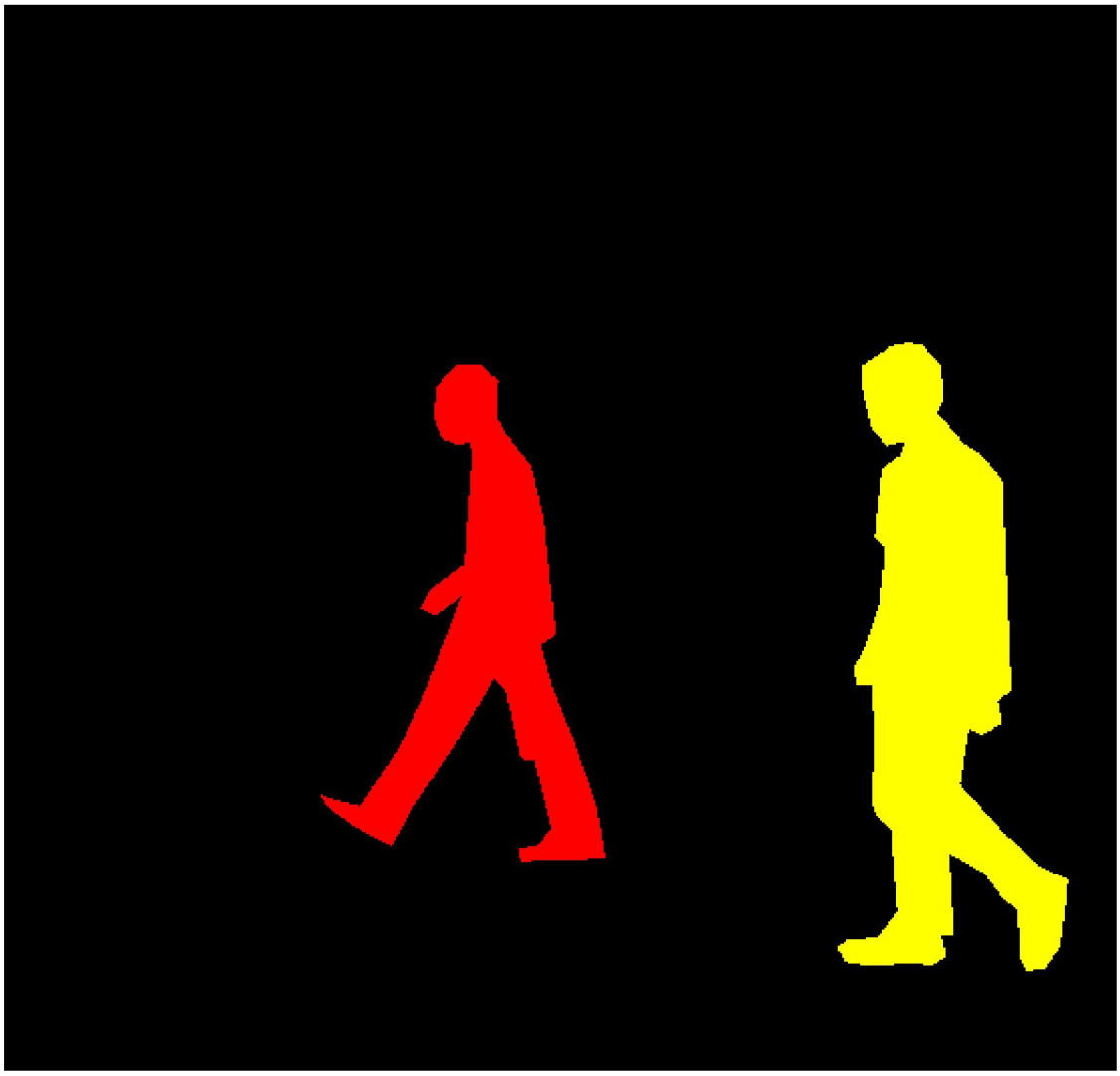
Here is one example of an image in the dataset, with its corresponding instance segmentation mask

In [3]:
```
from PIL import Image
Image.open('PennFudanPed/PNGImages/FudanPed00001.png')
```

Out[3]:

In [5]:
```
mask = Image.open('PennFudanPed/PedMasks/FudanPed00001_mask.png')
# each mask instance has a different color, from zero to N, where
# N is the number of instances. In order to make visualization easier,
# let's adda color palette to the mask.
mask = mask.convert('P')
mask.putpalette([
    0, 0, 0, # black background
    255, 0, 0, # index 1 is red
    255, 255, 0, # index 2 is yellow
    255, 153, 0, # index 3 is orange
])
mask
```

Out[5]:



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 [6]:
```
import os
import numpy as np
import torch
import torch.utils.data
from PIL import Image


class PennFudanDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms=None):
        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 ad masks
        img_path = os.path.join(self.root, "PNGImages", self.imgs[idx])
        mask_path = os.path.join(self.root, "PedMasks", self.masks[idx])
        img = Image.open(img_path).convert("RGB")
        # note that we haven't converted the mask to RGB,
        # because each color corresponds to a different instance
        # with 0 being background
        mask = Image.open(mask_path)

        mask = np.array(mask)
        # instances are encoded as different colors
        obj_ids = np.unique(mask)
        # first id is the background, so remove it
        obj_ids = obj_ids[1:]

        # split the color-encoded mask into a set
        # of binary masks
        masks = mask == obj_ids[:, None, None]

        # get bounding box coordinates for each mask
        num_objs = len(obj_ids)
        boxes = []
        for i in range(num_objs):
            pos = np.where(masks[i])
            xmin = np.min(pos[1])
            xmax = np.max(pos[1])
            ymin = np.min(pos[0])
            ymax = np.max(pos[0])
            boxes.append([xmin, ymin, xmax, ymax])

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        # there is only one class
        labels = torch.ones((num_objs,), dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)

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

```
        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        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. Let's see how the outputs are structured for this dataset

In [7]:
```
dataset = PennFudanDataset('PennFudanPed/')
dataset[0]
```

Out[7]:
```
(<PIL.Image.Image image mode=RGB size=559x536 at 0x7F86FC119880>,
 {'boxes': tensor([[159., 181., 301., 430.],
         [419., 170., 534., 485.]]),
  'labels': tensor([1, 1]),
  'masks': tensor([[[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]],

         [[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]]], dtype=torch.uint8),
  'image_id': tensor([0]),
  'area': tensor([35358., 36225.]),
  'iscrowd': tensor([0, 0])})
```

So we can see that by default, the dataset returns a `PIL.Image` and a dictionary containing several fields, including `boxes`, `labels` and `masks`.

Defining your model

In this tutorial, we will be using Mask R-CNN (https://arxiv.org/abs/1703.06870), which is based on top of Faster R-CNN (https://arxiv.org/abs/1506.01497). 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 modelzoo. 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 FastRCNNPredic
tor

# load a model pre-trained pre-trained on COCO
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretra
ined=True)

# 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)
```

2 - Modifying the model to add a different backbone

Another common situation arises when the user wants to replace the backbone of a detection model with a different one. For example, the current default backbone (ResNet-50) might be too big for some applications, and smaller models might be necessary.

Here is how we would go into leveraging the functions provided by torchvision to modify a 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(pretrained=True).feature
s
# 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, 51
2),),
                                   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 FasterRCNN model
model = FasterRCNN(backbone,
                   num_classes=2,
                   rpn_anchor_generator=anchor_generator,
                   box_roi_pool=roi_pooler)
```

An Instance segmentation model for PennFudan Dataset

In our case, we want to fine-tune 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:

In [48]:
```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor


def get_instance_segmentation_model(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=T

    # get the 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_clas

    # now get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channel
    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 our custom dataset.

Training and evaluation functions

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`, `references/detection/utils.py` and `references/detection/transforms.py`.

Let's copy those files (and their dependencies) in here so that they are available in the notebook

In [9]:
```
%%shell

# Download TorchVision repo to use some files from
# references/detection
git clone https://github.com/pytorch/vision.git
cd vision
git checkout v0.8.2

cp references/detection/utils.py ../
cp references/detection/transforms.py ../
cp references/detection/coco_eval.py ../
cp references/detection/engine.py ../
cp references/detection/coco_utils.py ../
```

```
Cloning into 'vision'...
remote: Enumerating objects: 316973, done.
remote: Total 316973 (delta 0), reused 0 (delta 0), pack-reused 316973
Receiving objects: 100% (316973/316973), 644.02 MiB | 18.29 MiB/s, done.
Resolving deltas: 100% (291349/291349), done.
Note: switching to 'v0.8.2'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to fa
lse

HEAD is now at 2f40a483d7 [v0.8.X] .circleci: Add Python 3.9 to CI (#306
3)
```

Out[9]:

Let's write some helper functions for data augmentation / transformation, which leverages the functions in `refereces/detection` that we have just copied:

In [10]:
```
from engine import train_one_epoch, evaluate
import utils
import transforms as T


def get_transform(train):
    transforms = []
    # converts the image, a PIL image, into a PyTorch Tensor
    transforms.append(T.ToTensor())
    if train:
        # during training, randomly flip the training images
        # and ground-truth for data augmentation
        transforms.append(T.RandomHorizontalFlip(0.5))
    return T.Compose(transforms)
```

Testing forward() method

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

In [49]:
```
model_pre = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained
dataset = PennFudanDataset('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_pre(images,targets)   # Returns losses and detections
# For inference
model_pre.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
predictions = model_pre(x)           # Returns predictions
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may
be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' a
re deprecated since 0.13 and may be removed in the future. The current be
havior is equivalent to passing `weights=FasterRCNN_ResNet50_FPN_Weights.
COCO_V1`. You can also use `weights=FasterRCNN_ResNet50_FPN_Weights.DEFAU
LT` to get the most up-to-date weights.
  warnings.warn(msg)
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:55
4: UserWarning: This DataLoader will create 4 worker processes in total.
Our suggested max number of worker in current system is 2, which is small
er than what this DataLoader is going to create. Please be aware that exc
essive 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(
```

Note that we do not need to add a mean/std normalization nor image rescaling in the data transforms, as those are handled internally by the Mask R-CNN model.

Putting everything together

We now have the dataset class, the models and the data transforms. Let's instantiate them

In [50]:
```
# use our dataset and defined transformations
dataset = PennFudanDataset('PennFudanPed', get_transform(train=True))
dataset_test = PennFudanDataset('PennFudanPed', get_transform(train=False))

# split the dataset in train and test set
torch.manual_seed(1)
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)
```

Now let's instantiate the model and the optimizer

In [51]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.devic

# our dataset has two classes only - background and person
num_classes = 2

# get the model using our helper function
model_pre = get_instance_segmentation_model(num_classes)
# move model to the right device
model_pre.to(device)

# construct an optimizer
params = [p for p in model_pre.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 which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' a
re deprecated since 0.13 and may be removed in the future. The current be
havior is equivalent to passing `weights=MaskRCNN_ResNet50_FPN_Weights.CO
CO_V1`. You can also use `weights=MaskRCNN_ResNet50_FPN_Weights.DEFAULT`
to get the most up-to-date weights.
  warnings.warn(msg)
```

And now let's train the model for 10 epochs, evaluating at the end of every epoch.

In [52]:
```
# let's train it for 10 epochs
from torch.optim.lr_scheduler import StepLR
num_epochs = 10

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

```
Epoch: [0]  [ 0/60]  eta: 0:01:24  lr: 0.000090  loss: 2.7899 (2.7899)  l
oss_classifier: 0.7401 (0.7401)  loss_box_reg: 0.3405 (0.3405)  loss_mas
k: 1.6637 (1.6637)  loss_objectness: 0.0430 (0.0430)  loss_rpn_box_reg:
0.0025 (0.0025)  time: 1.4015  data: 0.7582  max mem: 13537
Epoch: [0]  [10/60]  eta: 0:00:33  lr: 0.000936  loss: 1.3931 (1.7265)  l
oss_classifier: 0.5150 (0.4821)  loss_box_reg: 0.2960 (0.2981)  loss_mas
k: 0.7157 (0.9198)  loss_objectness: 0.0169 (0.0217)  loss_rpn_box_reg:
0.0045 (0.0048)  time: 0.6735  data: 0.0770  max mem: 13537
Epoch: [0]  [20/60]  eta: 0:00:24  lr: 0.001783  loss: 1.0069 (1.2306)  l
oss_classifier: 0.2235 (0.3354)  loss_box_reg: 0.2910 (0.2865)  loss_mas
k: 0.3239 (0.5872)  loss_objectness: 0.0109 (0.0171)  loss_rpn_box_reg:
0.0042 (0.0045)  time: 0.5759  data: 0.0090  max mem: 13537
Epoch: [0]  [30/60]  eta: 0:00:18  lr: 0.002629  loss: 0.5559 (1.0164)  l
oss_classifier: 0.0983 (0.2557)  loss_box_reg: 0.2704 (0.2874)  loss_mas
k: 0.1833 (0.4536)  loss_objectness: 0.0099 (0.0147)  loss_rpn_box_reg:
0.0045 (0.0050)  time: 0.5847  data: 0.0098  max mem: 13537
Epoch: [0]  [40/60]  eta: 0:00:12  lr: 0.003476  loss: 0.4528 (0.8841)  l
oss_classifier: 0.0629 (0.2076)  loss_box_reg: 0.2131 (0.2693)  loss_mas
k: 0.1685 (0.3901)  loss_objectness: 0.0034 (0.0118)  loss_rpn_box_reg:
```

Now that training has finished, let's have a look at what it actually predicts in a test image

In [73]:
```
# pick one image from the test set
img, _ = dataset_test[0]
# put the model in evaluation mode
model_pre.eval()
with torch.no_grad():
    prediction = model_pre([img.to(device)])
```

Printing the prediction shows that we have a list of dictionaries. Each element of the list corresponds to a different image. As we have a single image, there is a single dictionary in the list. The dictionary contains the predictions for the image we passed. In this case, we can see that it contains `boxes`, `labels`, `masks` and `scores` as fields.

In [74]:
```
prediction
```

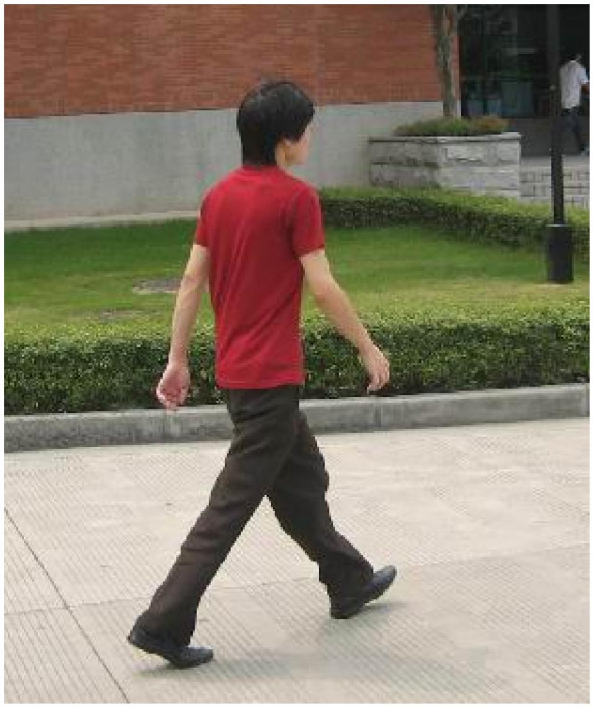
Out[74]:
```
[{'boxes': tensor([[ 64.1500,  35.4738, 199.0857, 328.9819],
          [276.6224,  23.7726, 290.9286,  73.4785],
          [ 82.5368,  38.9507, 194.0530, 214.4267]], device='cuda:0'),
  'labels': tensor([1, 1, 1], device='cuda:0'),
  'scores': tensor([0.9987, 0.5238, 0.1695], device='cuda:0'),
  'masks': tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            ...,
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.]]],


          [[[0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            ...,
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.]]],


          [[[0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            ...,
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.]]]], device='cuda:0')}]
```

Let's inspect the image and the predicted segmentation masks.

For that, we need to convert the image, which has been rescaled to 0-1 and had the channels flipped so that we have it in `[C, H, W]` format.

In [75]:
```
Image.fromarray(img.mul(255).permute(1, 2, 0).byte().numpy())
```

Out[75]:



And let's now visualize the top predicted segmentation mask. The masks are predicted as `[N, 1, H, W]`, where `N` is the number of predictions, and are probability maps between 0-1.

In [76]:
```
Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
```

Out[76]:

In [39]:
```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator

def get_instance_segmentation_model_backbone(num_classes):
    # load an instance segmentation model pre-trained on COCO
    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 FasterRCNN model
    model = FasterRCNN(backbone,
                       num_classes=2,
                       rpn_anchor_generator=anchor_generator,
                       box_roi_pool=roi_pooler)


    return model
```

In [43]:
```
import torch
import torchvision
from torchvision.models.detection import MaskRCNN
from torchvision.models.detection.backbone_utils import resnet_fpn_backbone


def get_instance_segmentation_model_backbone(num_classes):

    backbone = resnet_fpn_backbone('resnet50', pretrained=True)

    # Create the Mask R-CNN model with the pre-trained backbone
    model = MaskRCNN(backbone, num_classes=num_classes)


    # Replace the pre-trained head with a new one
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_clas
    return model
```

In [44]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.devic

# our dataset has two classes only - background and person
num_classes = 2

# get the model using our helper function
model = get_instance_segmentation_model_backbone(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 which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:135:
UserWarning: Using 'backbone_name' as positional parameter(s) is deprecat
ed since 0.13 and may be removed in the future. Please use keyword parame
ter(s) instead.
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may
be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' a
re deprecated since 0.13 and may be removed in the future. The current be
havior is equivalent to passing `weights=ResNet50_Weights.IMAGENET1K_V1`.
You can also use `weights=ResNet50_Weights.DEFAULT` to get the most up-to
-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth"
to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth

  0%|          | 0.00/97.8M [00:00<?, ?B/s]
```

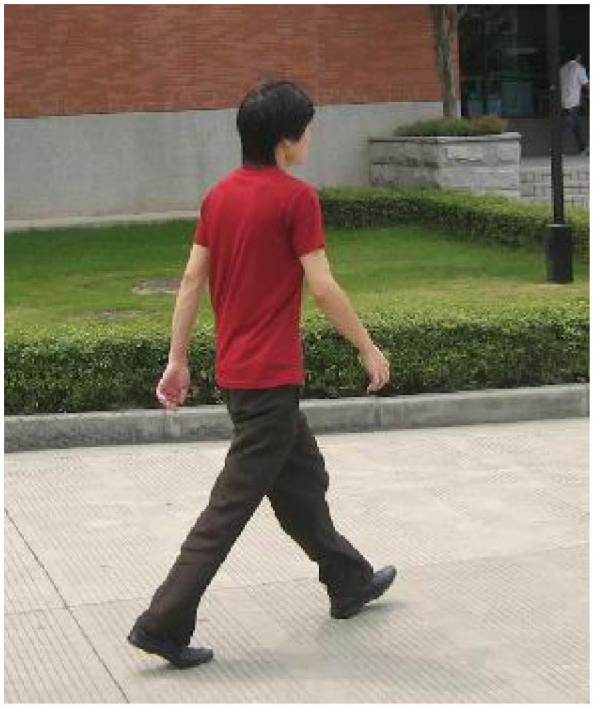
In [45]:
```
# let's train it for 10 epochs
import torch
torch.cuda.empty_cache()
import gc
gc.collect()
from torch.optim.lr_scheduler import StepLR
num_epochs = 10

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

```
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:55
4: UserWarning: This DataLoader will create 4 worker processes in total.
Our suggested max number of worker in current system is 2, which is small
er than what this DataLoader is going to create. Please be aware that exc
essive 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(
```

In [66]:
```
# pick one image from the test set
img, _ = dataset_test[0]
# put the model in evaluation mode
model.eval()
with torch.no_grad():
    prediction_bb = model([img.to(device)])
```

In [70]:
```
Image.fromarray(img.mul(255).permute(1, 2, 0).byte().numpy())
```

Out[70]:

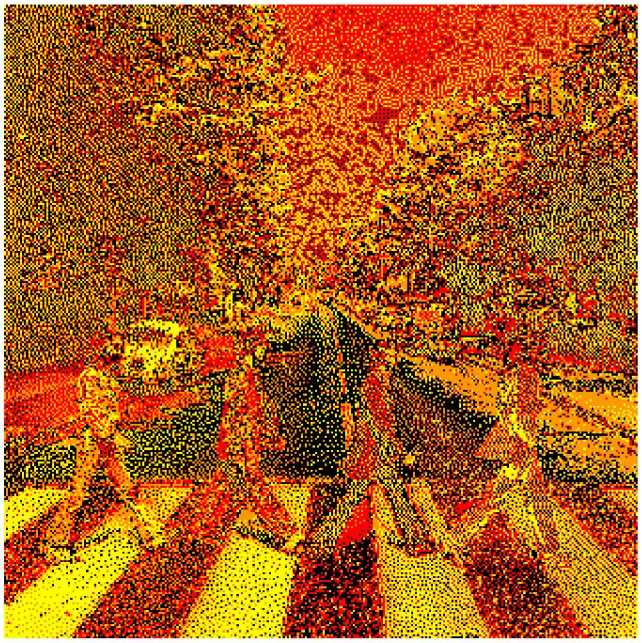


In [69]:
```
Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
```

Out[69]:

In [3]:
```
# pick one image from the test set
import torch
from torchvision import transforms
import torchvision.transforms.functional as F
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Define the number of classes (including the background) in your dataset
num_classes = 2  # assuming binary segmentation (foreground/background)

# Load the input image
input_image = Image.open('Beatles.jpeg').convert('RGB')

# Preprocess the input image
preprocess = torchvision.transforms.Compose([
    torchvision.transforms.Resize((800, 800)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0)  # create a mini-batch as expected


# Set the models to evaluation mode
model.eval()  #backbone model
model_pre.eval() #predefined model

# Run inference on the input image using the two models
with torch.no_grad():
    # Model 1
    output1 = model(input_batch)
    mask1 = output1[0]['masks'][0, 0].cpu().numpy()  # extract the binary m
    mask1 = (mask1 > 0.5).astype(np.uint8)  # threshold the mask to obtain
    mask1 = Image.fromarray(mask1 * 255)  # convert the mask to a PIL image

    # Model 2
    output2 = model_pre(input_batch)
    mask2 = output2[0]['masks'][0, 0].cpu().numpy()  # extract the binary m
    mask2 = (mask2 > 0.5).astype(np.uint8)  # threshold the mask to obtain
    mask2 = Image.fromarray(mask2 * 255)  # convert the mask to a PIL image

# Visualize the input image and the predicted segmentation masks
fig, ax = plt.subplots(1, 3, figsize=(10, 10))
ax[0].imshow(input_image)
ax[0].set_title('Input Image')
ax[1].imshow(mask1, cmap='gray')
ax[1].set_title('Model 1 Segmentation Mask')
ax[2].imshow(mask2, cmap='gray')
ax[2].set_title('Model 2 Segmentation Mask')
plt.show()
```

Out[3]:



Looks pretty good!

Wrapping up

In this tutorial, you have learned how to create your own training pipeline for instance segmentation 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 GitHub repo (https://github.com/pytorch/vision/tree/v0.8.2/references/detection)](https://github.com/pytorch/vision/tree/v0.8.2/references/detection).

1 References

https://www.codingame.com/playgrounds/2524/basic-image-manipulation/filtering
https://datacarpentry.org/image-processing/06-blurring/
https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

5 5 **3 / 3**

✓ **- 0 pts** *Correct*

- 0.5 pts Click here to replace this description.

- 1 pts Click here to replace this description.

- 3 pts Missing question