Anil Poonai

Github link for files and code: https://github.com/DevonARP/DeepLearning_A2

For question 1 I wrote out all my notes and work before writing them here, I'll add my work/notes at the end of each part for this question

Problem 1: Part A

$L(w_1,w_2) = .5(aw_1^2 + bw_2^2)$

$\nabla L(w_1,w_2) = aw_1 + bw_2$

$\nabla^2 L(w_1,w_2) = a + b$

I'm making a = 1 and b = 2 to keep the double derivate positive and keep the graph concave up so it has a minimum value.

$\nabla^2 L(w_1,w_2) = a + b = 1 + 2 = 3$

Now, we go back to the first derivative

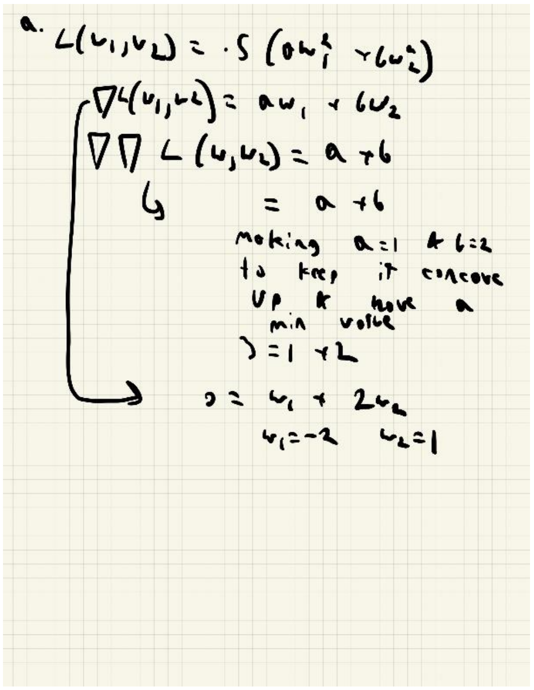
$\nabla L(w_1,w_2) = aw_1 + bw_2 = w_1 + 2w_2$

I can use $w_1 = -2$ and $w_2 = 1$, this gives the minimum value of 0.

References:

https://study.com/learn/lesson/how-to-find-the-maximum-value-of-a-function.html#:~:text=We%20will%20set%20the%20first,will%20be%20a%20minimum%20value

https://www.ocf.berkeley.edu/~reinholz/ed/07fa_m155/lectures/second_derivative.pdf



a. $L(w_1,w_2) = .5\,(aw_1^2 + bw_2^2)$

$\nabla L(w_1,w_2) = aw_1 + bw_2$

$\nabla\nabla\, L(w_1,w_2) = a + b$

$\hookrightarrow \quad = a + b$

making $a=1$ & $b=2$ to keep it concave up & have a min value

$) = 1 + 2$

$0 = w_1 + 2w_2$

$w_1 = -2 \quad w_2 = 1$

Problem 1: Part B

P_i is a dropout layer, I'll be using N for the dropout rate

Have to grab the derivative of L with respect to the corresponding weight

$W_i(t+1) = w_i(t) - N(\partial L/\partial w_i)$ This is being rearranged to follow the dropout rate argument formula

$W_1 = w_1(t) - Naw_1(t) = w_1(t)(1-Na) = p_1w_1(t)$

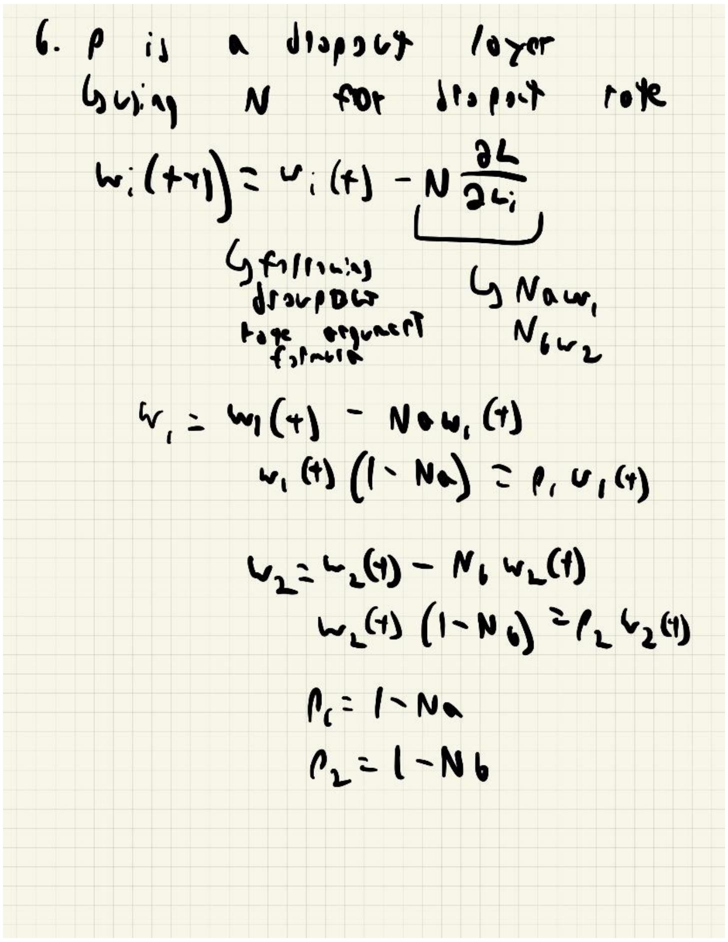
$W_2 = w_2(t) - Nbw_2(t) = w_2(t)(1-Nb) = p_2w_2(t)$

$P_1 = 1 - Na$

$P_2 = 1 - Nb$

References:

https://towardsdatascience.com/simplified-math-behind-dropout-in-deep-learning-6d50f3f47275#:~:text=In%20Keras%2C%20the%20dropout%20rate,can%20adversely%20affect%20the%20training.



6. p is a dropout layer

↳ using N for dropout rate

$$w_i(t+1) = w_i(t) - \underbrace{N\frac{\partial L}{\partial w_i}}$$

↳ following dropout rate argument formula

↳ Naw_1
Nbw_2

$w_1 = w_1(t) - Naw_1(t)$

$w_1(t)(1-Na) = p_1 w_1(t)$

$w_2 = w_2(t) - Nbw_2(t)$

$w_2(t)(1-Nb) = p_2 w_2(t)$

$p_1 = 1 - Na$

$p_2 = 1 - Nb$

Problem 1: Part C

It converges when the gradient of the cost function becomes 0.

In this case both $|p_1|$ and $|p_2|$ need to be less than 1

We can rearrange that to be $|1 – Na|$ and $|1 – Nb|$ need to both be less than 1

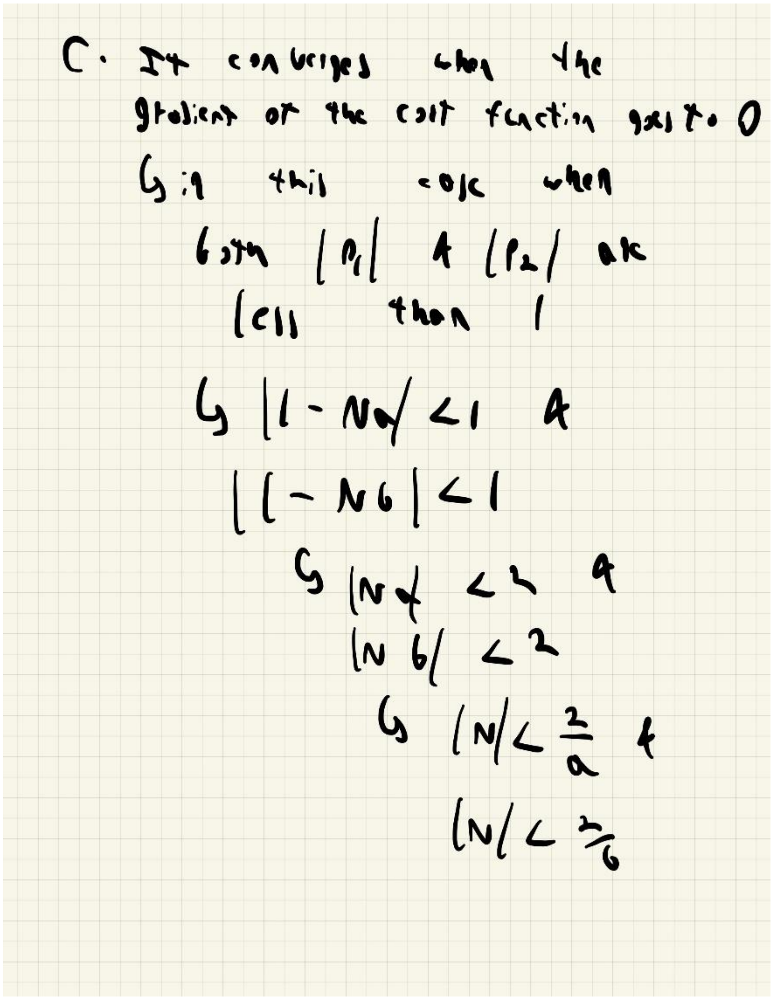
Then we can make it $|Na|$ and $|Nb|$ both need to be less than 2

And we can end off with $|N|$ needs to be less than both 2/a and 2/b.

References:

https://www.cs.umd.edu/~djacobs/CMSC426/GradientDescent.pdf

https://www.cs.ubc.ca/~schmidtm/Courses/540-W18/L4.pdf



C. It converges when the gradient of the cost function goes to 0

↳ in this case when both $|p_1|$ & $|p_2|$ are less than 1

↳ $|1 - Na| < 1$ &

$|1 - Nb| < 1$

↳ $|Na| < 2$ &

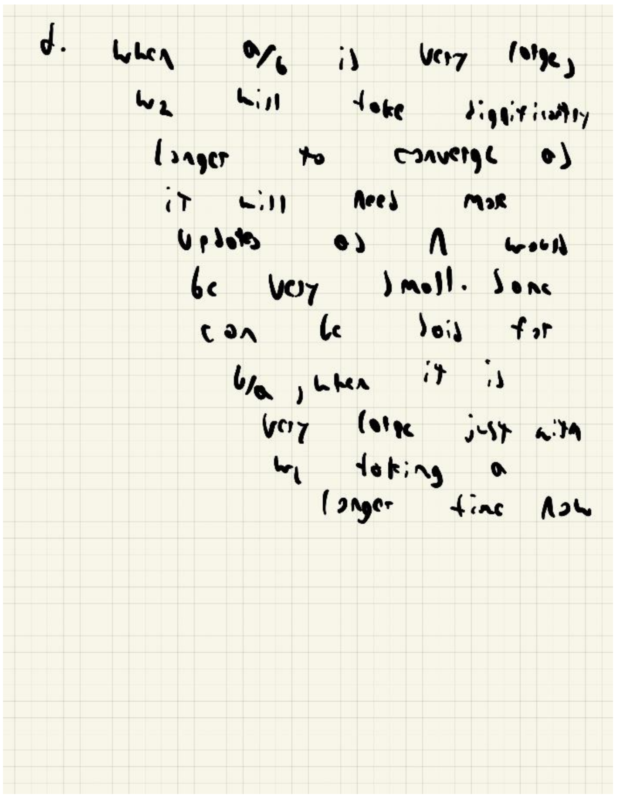
$|Nb| < 2$

↳ $|N| < \frac{2}{a}$ &

$|N| < \frac{2}{b}$

Problem 1: Part D

When either a/b or b/a is very large, in the first case w_2 will take significantly longer to converge as it will need more updates because N would be very small. Same can be said in the second case just with w_1.



d. When a/b is very large, w_2 will take significantly longer to converge as it will need more updates as N would be very small. Same can be said for b/a, when it is very large just with w_1 taking a longer time now

Problem 2: Part A

I would use the Sobel filter, which uses two kernels, one for the horizontal edges and one for the vertical edges.

Horizontal edges

-1	0	1
-2	0	2
1	0	1

Vertical edges

-1	-2	-1
0	0	0
1	2	1

This works by looking for string changes in the image. The higher the sum of the numbers after the convolution, the more likely there is an edge there and the positive or negative sign indicates the direction of the edge. The output form both filters are then combined to see all the edges detected.

References:
https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection

https://automaticaddison.com/how-the-sobel-operator-works/

https://www.cs.auckland.ac.nz/compsci373s1c/PatricesLectures/Edge%20detection-Sobel_2up.pdf

Problem 2: Part B

I'm going to use a Box Blur Kernel because I end up mentioning the Gaussian Blur Kernel in Part D.

1	1	1
1	1	1
1	1	1

This works by giving each pixel the same weight and adding them all up then dividing by 9 in this case at the end, this makes it so that the output is a relative value to the other output points after the kernel as it adds up all of the points in a region and averages it out for every region.

References:

https://medium.com/hackernoon/cv-for-busy-developers-convolutions-5c984f216e8c#:~:text=The%20convolution%20of%20a%20Gaussian,the%20kernel%20values%20is%2016.

Problem 2: Part C

I'm combining some concepts from regular sharpening and edge detection for this.

0	-1	0
0	2	0
0	-1	0

We're focusing on the horizontal sharpening by grabbing the middle row of the region, this focuses on the center primarily and the middle row right after, with no focus on anything else as those values on the filter are zero.

References:

https://medium.com/@boelsmaxence/introduction-to-image-processing-filters-179607f9824a

https://en.wikipedia.org/wiki/Kernel_(image_processing)

Problem 2: Part D

I'm going to use a Gaussian Blur Kernel, this can also be used to blue an image.

1	2	1
2	4	2
1	2	1

This works by giving the pixel near the center of the kernel more weight than the ones on the edges, this helps mute noise as the box blur would treat all the points with the same weight.

References:

https://medium.com/hackernoon/cv-for-busy-developers-convolutions-5c984f216e8c#:~:text=The%20convolution%20of%20a%20Gaussian,the%20kernel%20values%20is%2016.

Problem 3: Part A

Jaccard similarity is supposed to find how similar 2 sets of data are, so it has to range between 0, no correlation at all, and 1, they are the exact same set. It can't be out of that range since the extremes would be either 0 and 1 for no relation and 100% relation, any number between 0 and 1 would represent a partial match with the higher the value being the higher match percentage. This can also be described as the IOU being the division of the overlap of the predicted and ground truth, so it can only be between the ranges 0 and 1.

IOU = |A∩B| / |A∪B| #A is the predicted and B is the ground truth

B counts for all true positives, false negatives, and false positives while A counts for true positives

Problem 3: Part B

IOU isn't differentiable inherently as it ranges from 0 to 1, which means it can't differentiate between distances from how similar sets are as you would need all real numbers available. It also isn't differentiable in the case mentioned because the parameters have no influence on it's gradient, so in the case of the top left to bottom right corners, the gradients are just going to be 0. Also, using the equation mentioned above indicates that the gradients would just be 0 everywhere as well, the parameters don't play a role in IOU calculation, it also is because the IOU metric itself isn't continuous natively.

Problem 4:

Code and answers will be below in an attached pdf to this document

Problem 5: Part A

Code and answers will be below in an attached pdf to this document

Problem 5: Part B

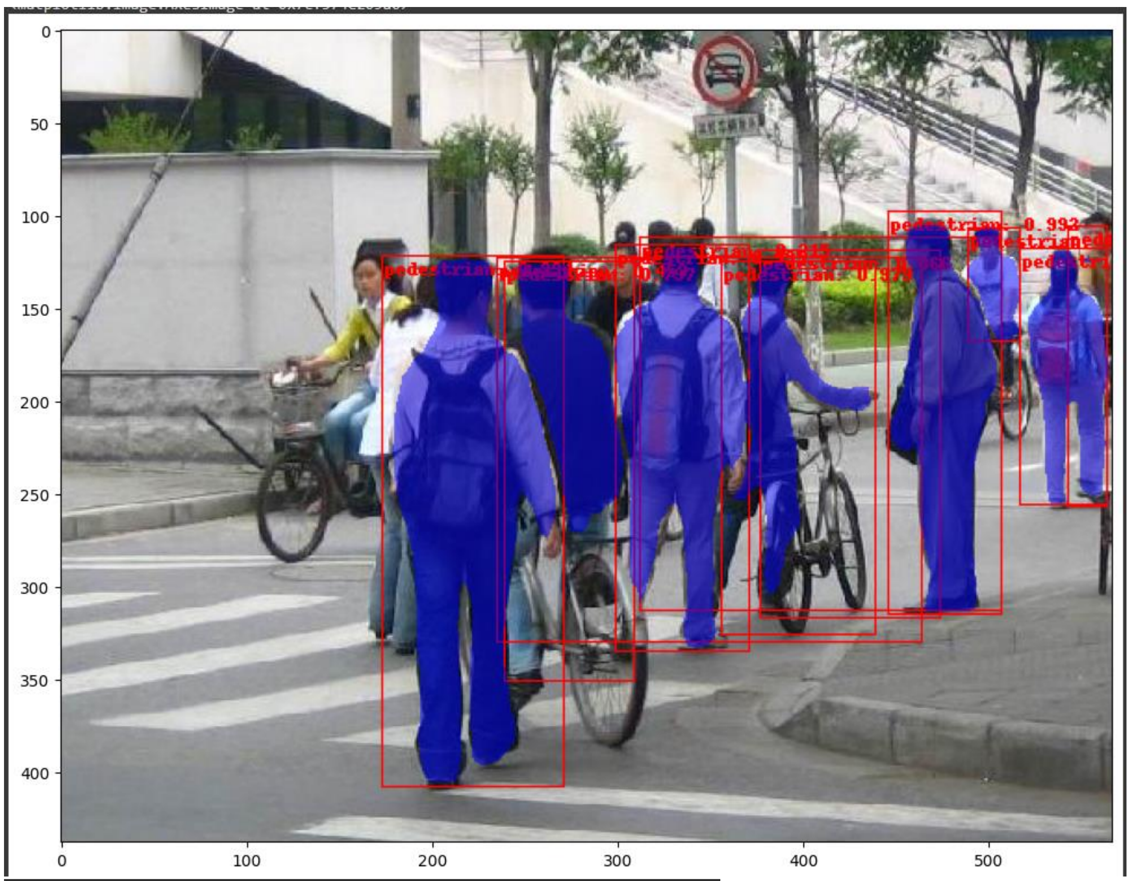
Code will be below in an attached pdf to this document

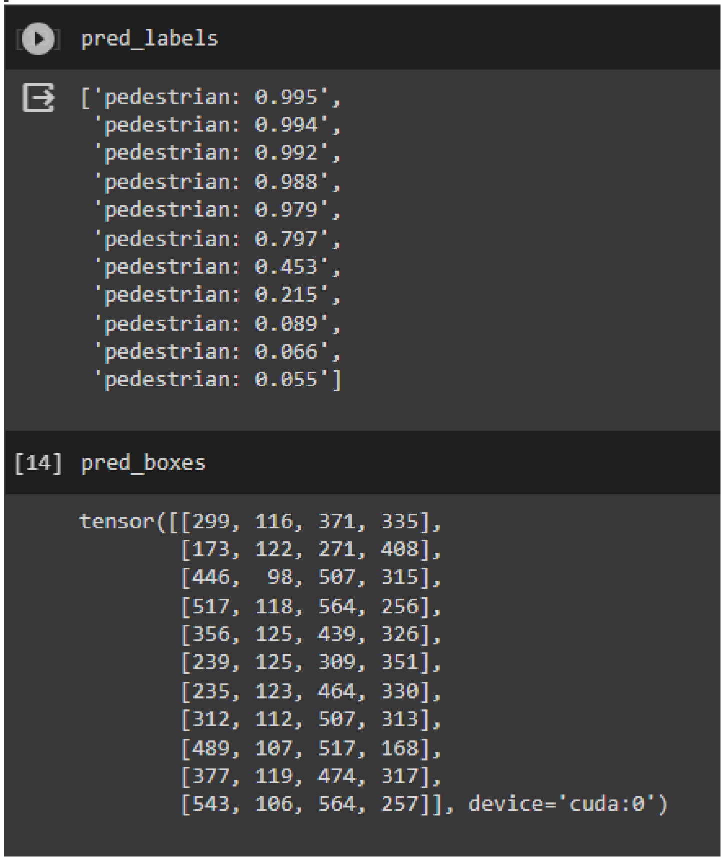
The backbone model is slightly better regarding loss after training as it has a lower loss and loss classifier but when looking at the example image it actually picks up 1 less person than the finetuning model but it is more confident in it's predictions. This also leads to the IOU metric being an indicator, with the higher values being given towards the finetuning model, meaning it actually ends up matching more objects with that model. I'm not saying which is better as that's depends on what outcome is wanted but I do like the results from the finetuning a model a bit more, the IOU metric helps a lot.

Finetuning model

```
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.1793 (0.1891)  loss_classifier: 0.0253 (0.0269)
loss_box_reg: 0.0378 (0.0417)  loss_mask: 0.1127 (0.1169)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0029 (0.0030)  time: 0.5977  data: 0.0096  max mem: 3778
```

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.839
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.931
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.355
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.659
 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 ] = 0.424
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.877
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.877
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.894
```





```
pred_labels
```

```
['pedestrian: 0.995',
 'pedestrian: 0.994',
 'pedestrian: 0.992',
 'pedestrian: 0.988',
 'pedestrian: 0.979',
 'pedestrian: 0.797',
 'pedestrian: 0.453',
 'pedestrian: 0.215',
 'pedestrian: 0.089',
 'pedestrian: 0.066',
 'pedestrian: 0.055']
```

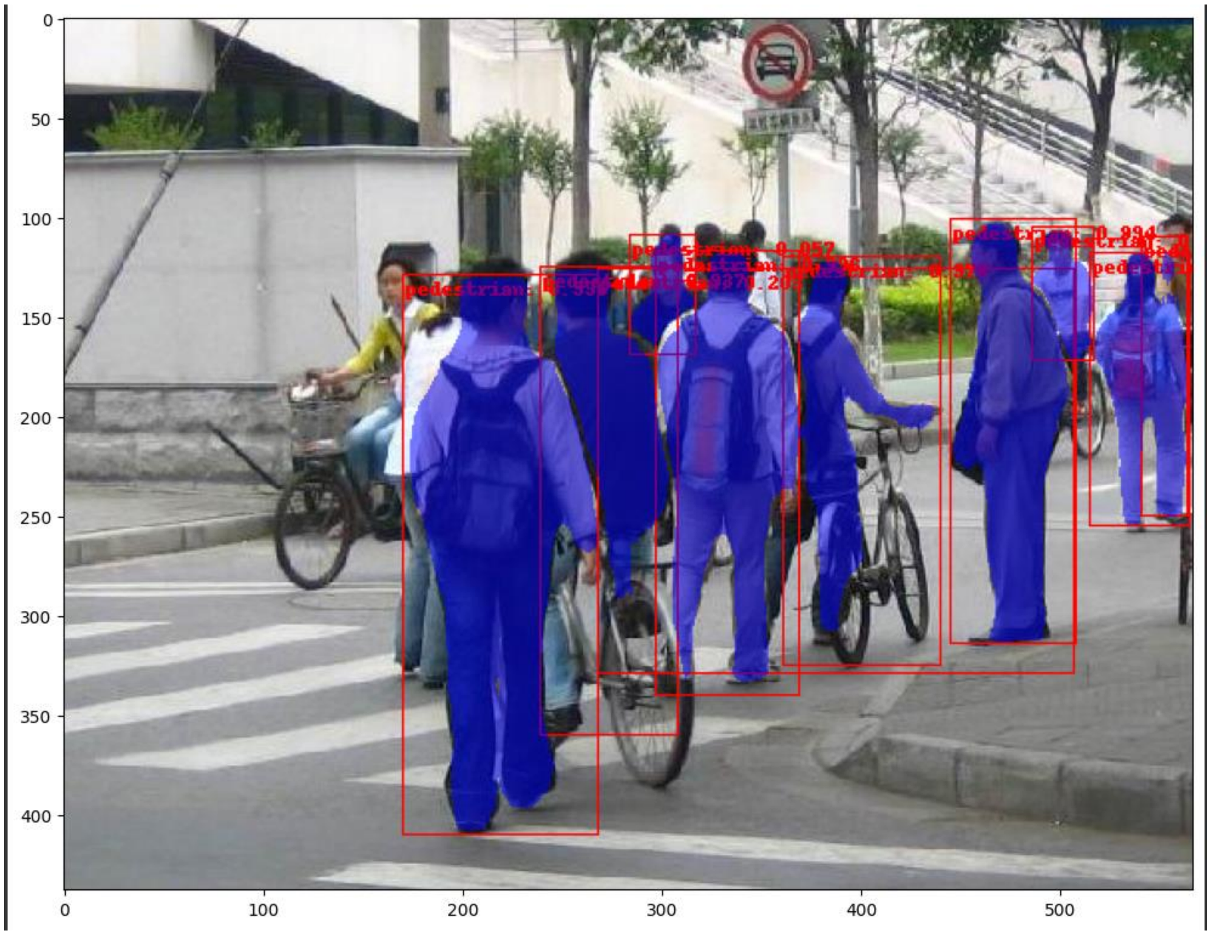
```
[14] pred_boxes
```

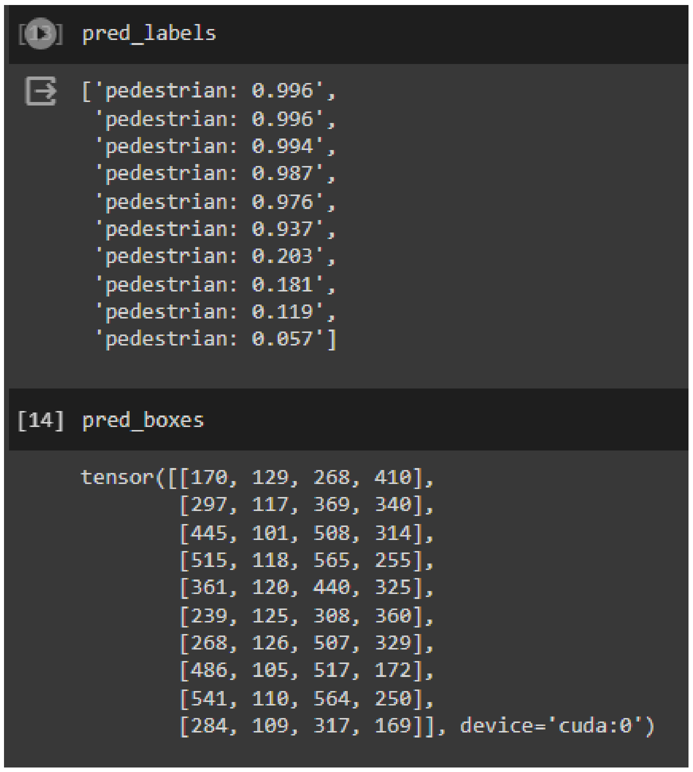
```
tensor([[299, 116, 371, 335],
        [173, 122, 271, 408],
        [446,  98, 507, 315],
        [517, 118, 564, 256],
        [356, 125, 439, 326],
        [239, 125, 309, 351],
        [235, 123, 464, 330],
        [312, 112, 507, 313],
        [489, 107, 517, 168],
        [377, 119, 474, 317],
        [543, 106, 564, 257]], device='cuda:0')
```

Backbone Model

```
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.1621 (0.1808)  loss_classifier: 0.0219 (0.0247)
loss_box_reg: 0.0326 (0.0377)  loss_mask: 0.1089 (0.1143)  loss_objectness: 0.0002 (0.0011)
loss_rpn_box_reg: 0.0033 (0.0029)  time: 0.5792  data: 0.0088  max mem: 3780
```

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.806
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.700
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.354
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.853
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.853
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.872
```





```
[ ] pred_labels

['pedestrian: 0.996',
 'pedestrian: 0.996',
 'pedestrian: 0.994',
 'pedestrian: 0.987',
 'pedestrian: 0.976',
 'pedestrian: 0.937',
 'pedestrian: 0.203',
 'pedestrian: 0.181',
 'pedestrian: 0.119',
 'pedestrian: 0.057']
```

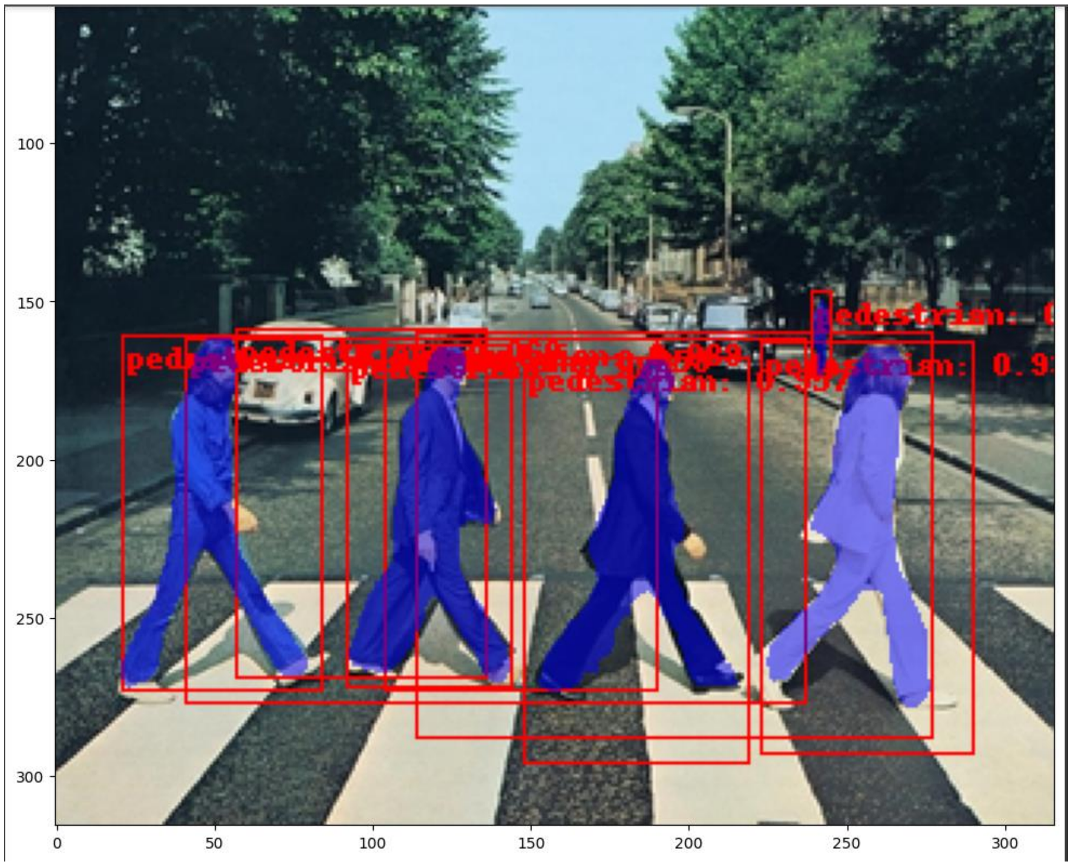
```
[14] pred_boxes

tensor([[170, 129, 268, 410],
        [297, 117, 369, 340],
        [445, 101, 508, 314],
        [515, 118, 565, 255],
        [361, 120, 440, 325],
        [239, 125, 308, 360],
        [268, 126, 507, 329],
        [486, 105, 517, 172],
        [541, 110, 564, 250],
        [284, 109, 317, 169]], device='cuda:0')
```

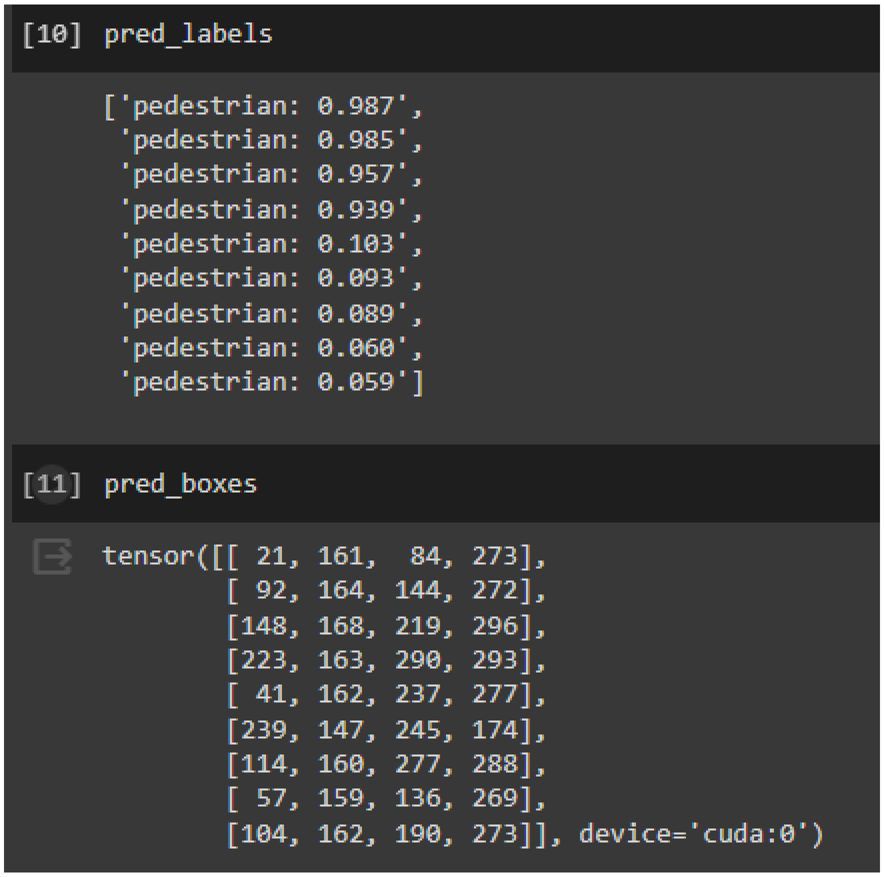
Problem 5: Part C

Code will be below in an attached pdf to this document

There are 5 people in the photograph, 4 are in the center and easy to spot but there's a fifth on the right side a bit in the back, the finetuning model picked up on that and with low confidence but the backbone model only recognized 4 people, being the 4 Beatles members in the middle of the photograph. The finetuning model also picked up some errors, it had bounding boxes around multiple Beatles members as possible objects but gave it a really low confidence score. So the finetuning model performed better here as it caught the 4 members with high confidence, a person in the back with fairly low confidence and had 4 other bounding boxes with multiple people in them but gave those boxes a really low confidence score, I would calculate those as False Positives. The backbone model performed worst not only because it missed a person entirely but because the confidence it had about the 4 Beatles members in the center of the image was low, it gave George Harrison a confidence score of less than .5 but the rest as above .9. All that being said is it important to point out that the finetuning model does have a higher error rate. Again, I'm not saying which is better but I still like the finetuning model a bit more, it might have more errors but it is more confident bout detecting objects and can pick up on hard to see objects.

Finetuning Model

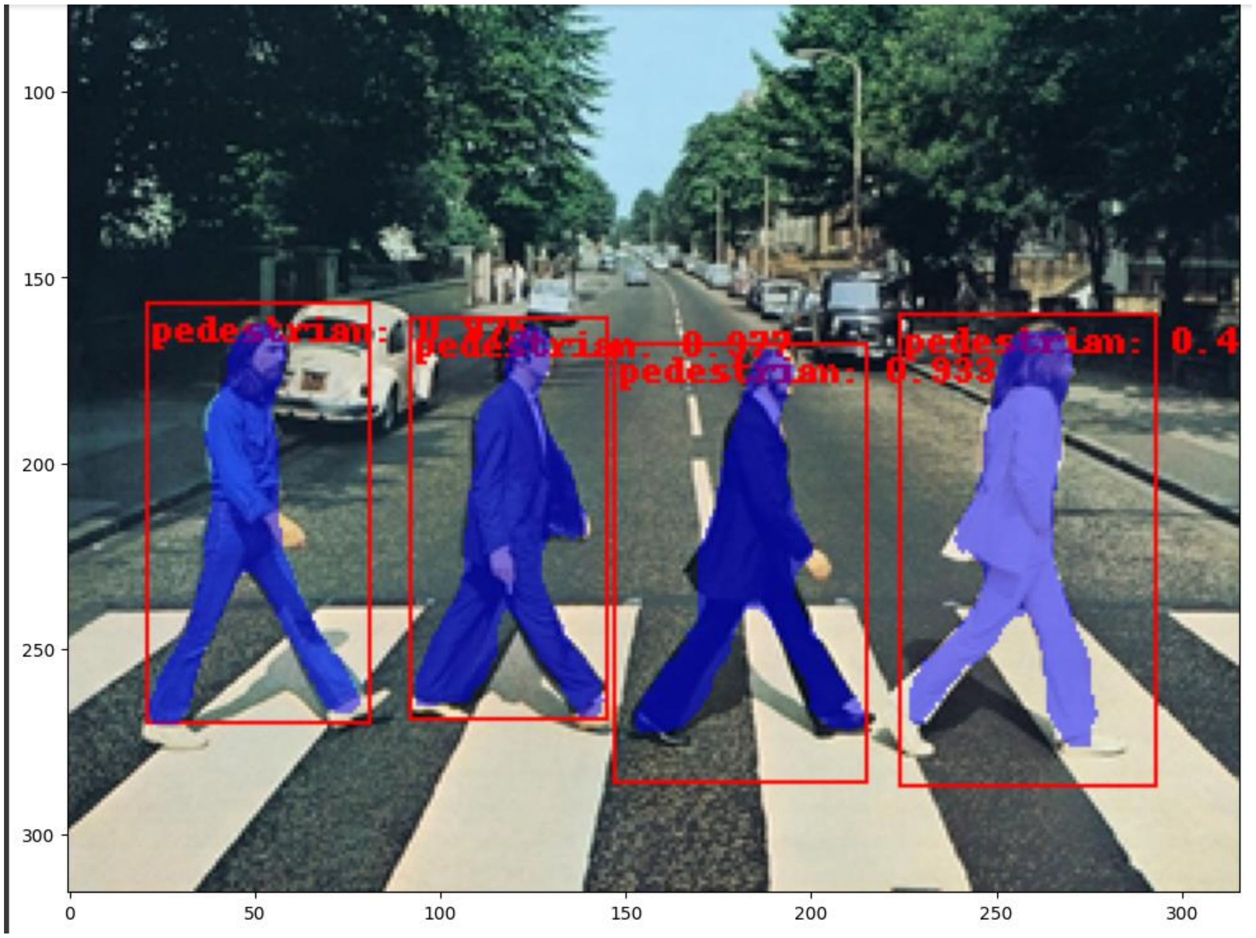


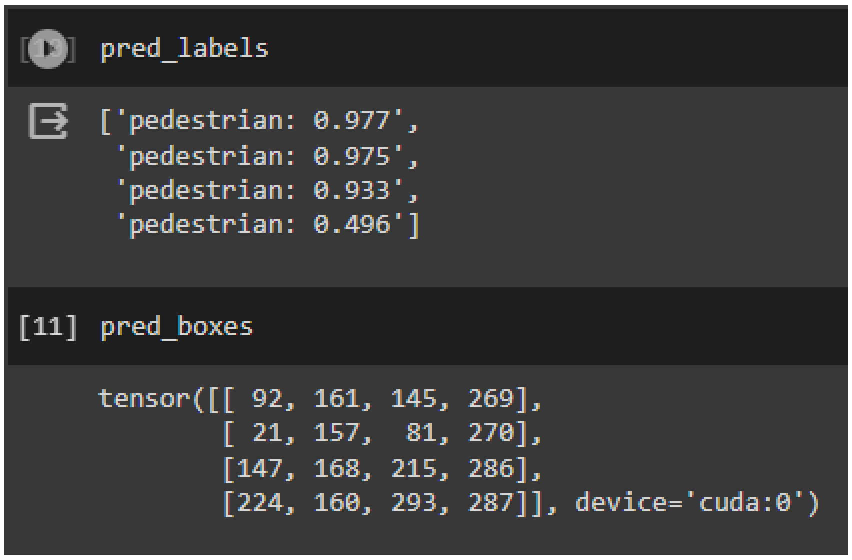


```
[10] pred_labels

     ['pedestrian: 0.987',
      'pedestrian: 0.985',
      'pedestrian: 0.957',
      'pedestrian: 0.939',
      'pedestrian: 0.103',
      'pedestrian: 0.093',
      'pedestrian: 0.089',
      'pedestrian: 0.060',
      'pedestrian: 0.059']
```

```
[11] pred_boxes

     tensor([[ 21, 161,  84, 273],
             [ 92, 164, 144, 272],
             [148, 168, 219, 296],
             [223, 163, 290, 293],
             [ 41, 162, 237, 277],
             [239, 147, 245, 174],
             [114, 160, 277, 288],
             [ 57, 159, 136, 269],
             [104, 162, 190, 273]], device='cuda:0')
```

Backbone Model





```
pred_labels

['pedestrian: 0.977',
 'pedestrian: 0.975',
 'pedestrian: 0.933',
 'pedestrian: 0.496']

[11] pred_boxes

tensor([[ 92, 161, 145, 269],
        [ 21, 157,  81, 270],
        [147, 168, 215, 286],
        [224, 160, 293, 287]], device='cuda:0')
```

Question 4

AlexNet

In this problem, you are asked to train a deep convolutional neural network to perform image classification. In fact, this is a slight variation of a network called *AlexNet*. This is a landmark model in deep learning, and arguably kickstarted the current (and ongoing, and massive) wave of innovation in modern AI when its results were first presented in 2012. AlexNet was the first real-world demonstration of a *deep* classifier that was trained end-to-end on data and that outperformed all other ML models thus far.

We will train AlexNet using the CIFAR10 dataset, which consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

A lot of the code you will need is already provided in this notebook; all you need to do is to fill in the missing pieces, and interpret your results.

Warning : AlexNet takes a good amount of time to train (~1 minute per epoch on Google Colab). So please budget enough time to do this homework.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import _LRScheduler
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Loading and Preparing the Data

Our dataset is made up of color images but three color channels (red, green and blue), compared to MNIST's black and white images with a single color channel. To normalize our data we need to calculate the means and standard deviations for each of the color channels independently, and normalize them.

```
ROOT = '.data'
train_data = datasets.CIFAR10(root = ROOT,
                              train = True,
                              download = True)

Files already downloaded and verified

# Compute means and standard deviations along the R,G,B channel

means = train_data.data.mean(axis = (0,1,2)) / 255
stds = train_data.data.std(axis = (0,1,2)) / 255
```

Next, we will do data augmentation. For each training image we will randomly rotate it (by up to 5 degrees), flip/mirror with probability 0.5, shift by +/-1 pixel. Finally we will normalize each color channel using the means/stds we calculated above.

```
train_transforms = transforms.Compose([
                           transforms.RandomRotation(5),
                           transforms.RandomHorizontalFlip(0.5),
                           transforms.RandomCrop(32, padding = 2),
                           transforms.ToTensor(),
                           transforms.Normalize(mean = means,
                                                std = stds)
```

```
                        ])

test_transforms = transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize(mean = means,
                                                std = stds)
                        ])
```

Next, we'll load the dataset along with the transforms defined above.

We will also create a validation set with 10% of the training samples. The validation set will be used to monitor loss along different epochs, and we will pick the model along the optimization path that performed the best, and report final test accuracy numbers using this model.

```
train_data = datasets.CIFAR10(ROOT,
                              train = True,
                              download = True,
                              transform = train_transforms)

test_data = datasets.CIFAR10(ROOT,
                             train = False,
                             download = True,
                             transform = test_transforms)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
VALID_RATIO = 0.9

n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples

train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples,
n_valid_examples])

valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

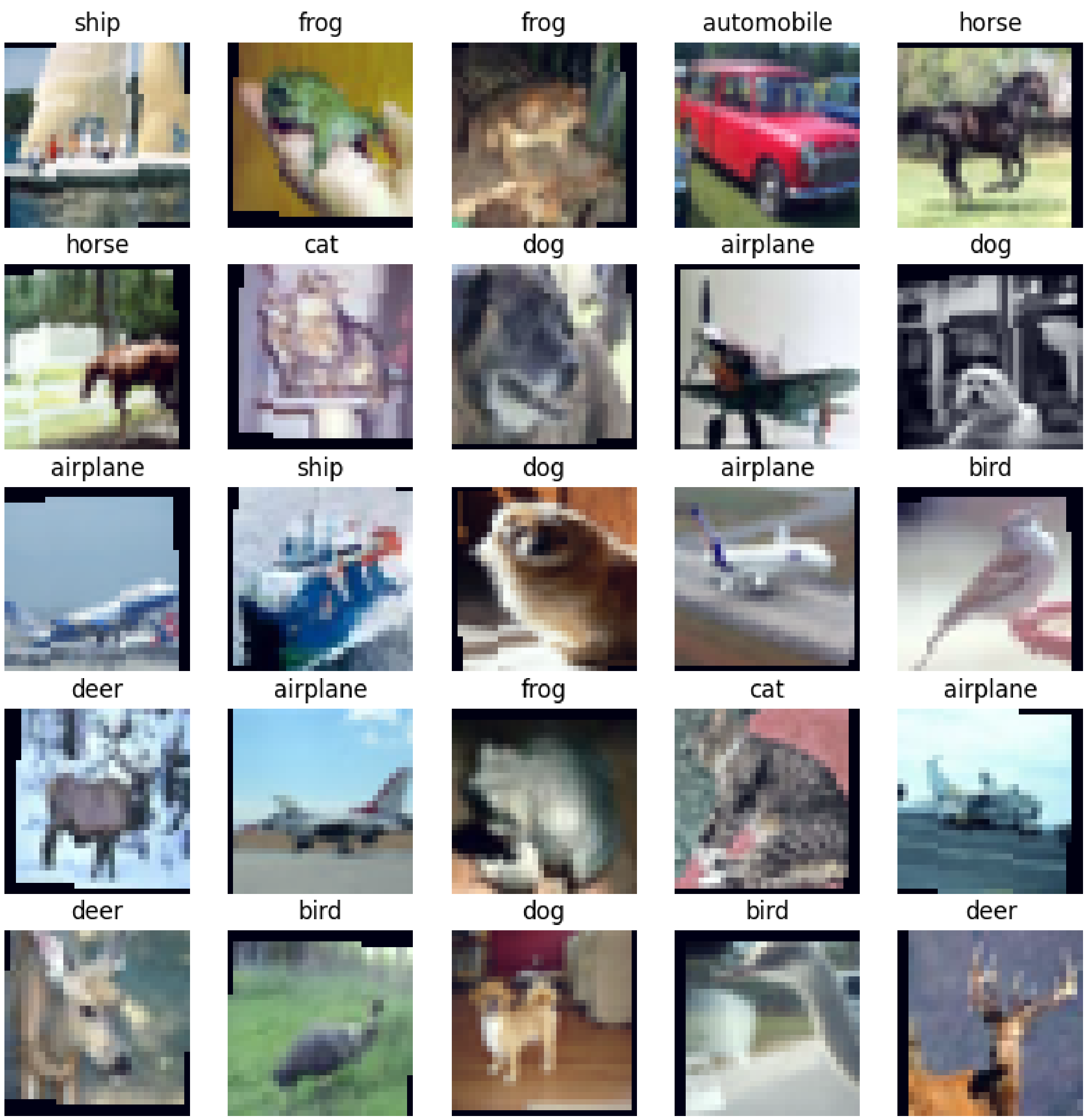
Now, we'll create a function to plot some of the images in our dataset to see what they actually look like.

Note that by default PyTorch handles images that are arranged `[channel, height, width]`, but `matplotlib` expects images to be `[height, width, channel]`, hence we need to permute the dimensions of our images before plotting them.

```
def plot_images(images, labels, classes, normalize = False):

    n_images = len(images)
    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize = (10, 10))

    for i in range(rows*cols):

        ax = fig.add_subplot(rows, cols, i+1)

        image = images[i]

        if normalize:
            image_min = image.min()
            image_max = image.max()
            image.clamp_(min = image_min, max = image_max)
            image.add_(-image_min).div_(image_max - image_min + 1e-5)

        ax.imshow(image.permute(1, 2, 0).cpu().numpy())
        ax.set_title(classes[labels[i]])
        ax.axis('off')
```

One point here: `matplotlib` is expecting the values of every pixel to be between $[0,1]$, however our normalization will cause them to be outside this range. By default `matplotlib` will then clip these values into the $[0,1]$ range. This clipping causes all of the images to look a bit weird - all of the colors are oversaturated. The solution is to normalize each image between [0,1].

```
N_IMAGES = 25

images, labels = zip(*[(image, label) for image, label in
                           [train_data[i] for i in range(N_IMAGES)]])

classes = test_data.classes

plot_images(images, labels, classes, normalize = True)
```



We'll be normalizing our images by default from now on, so we'll write a function that does it for us which we can use whenever we need to renormalize an image.

```
def normalize_image(image):
    image_min = image.min()
    image_max = image.max()
    image.clamp_(min = image_min, max = image_max)
    image.add_(-image_min).div_(image_max - image_min + 1e-5)
    return image
```

The final bit of the data processing is creating the iterators. We will use a large. Generally, a larger batch size means that our model trains faster but is a bit more susceptible to overfitting.

```
# Q1: Create data loaders for train_data, valid_data, test_data
# Use batch size 256


BATCH_SIZE = 256

train_iterator = torch.utils.data.DataLoader(train_data, 
batch_size=BATCH_SIZE, shuffle=True)#Added the iterable wraps for the 
datasets

valid_iterator = torch.utils.data.DataLoader(valid_data, 
batch_size=BATCH_SIZE, shuffle=True)

test_iterator = torch.utils.data.DataLoader(test_data, 
batch_size=BATCH_SIZE, shuffle=True)
```

Defining the Model

Next up is defining the model.

AlexNet will have the following architecture:

- There are 5 2D convolutional layers (which serve as *feature extractors*), followed by 3 linear layers (which serve as the *classifier*).
- All layers (except the last one) have `ReLU` activations. (Use `inplace=True` while defining your ReLUs.)
- All convolutional filter sizes have kernel size 3 x 3 and padding 1.
- Convolutional layer 1 has stride 2. All others have the default stride (1).
- Convolutional layers 1,2, and 5 are followed by a 2D maxpool of size 2.
- Linear layers 1 and 2 are preceded by Dropouts with Bernoulli parameter 0.5.
- For the convolutional layers, the number of channels is set as follows. We start with 3 channels and then proceed like this:
  - $3 \rightarrow 64 \rightarrow 192 \rightarrow 384 \rightarrow 256 \rightarrow 256$

  In the end, if everything is correct you should get a feature map of size $ 2\times2 \times 256 = 1024$.
- For the linear layers, the feature sizes are as follows:
  - $1024 \rightarrow 4096 \rightarrow 4096 \rightarrow 10$.

  (The 10, of course, is because 10 is the number of classes in CIFAR-10).

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

        self.features = nn.Sequential(
            # Define according to the steps described above
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
#Added layers for AlexNet
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )

        self.classifier = nn.Sequential(
            # define according to the steps described above
            nn.Dropout(p=0.5),
            nn.Linear(1024, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        h = x.view(x.shape[0], -1)
        x = self.classifier(h)
        return x, h
```

We'll create an instance of our model with the desired amount of classes.

```
OUTPUT_DIM = 10
model = AlexNet(OUTPUT_DIM)
```

Training the Model

We first initialize parameters in PyTorch by creating a function that takes in a PyTorch module, checking what type of module it is, and then using the `nn.init` methods to actually initialize the parameters.

For convolutional layers we will initialize using the *Kaiming Normal* scheme, also known as *He Normal*. For the linear layers we initialize using the *Xavier Normal* scheme, also known as *Glorot Normal*. For both types of layer we initialize the bias terms to zeros.

```
def initialize_parameters(m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight.data, nonlinearity = 'relu')
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight.data, gain =
nn.init.calculate_gain('relu'))
        nn.init.constant_(m.bias.data, 0)
```

We apply the initialization by using the model's `apply` method. If your definitions above are correct you should get the printed output as below.

```
model.apply(initialize_parameters)

AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=1024, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

We then define the loss function we want to use, the device we'll use and place our model and criterion on to our device.

```
optimizer = optim.Adam(model.parameters(), lr = 1e-3)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
criterion = nn.CrossEntropyLoss()

model = model.to(device)
criterion = criterion.to(device)

# This is formatted as code
```

We define a function to calculate accuracy...

```
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim = True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

As we are using dropout we need to make sure to "turn it on" when training by using `model.train()`.

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in iterator:

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

We also define an evaluation loop, making sure to "turn off" dropout with `model.eval()`.

```
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)
            y = y.to(device)

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Next, we define a function to tell us how long an epoch takes.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Then, finally, we train our model.

Train it for 25 epochs (using the train dataset). At the end of each epoch, compute the validation loss and keep track of the best model. You might find the command `torch.save` helpful.

At the end you should expect to see validation losses of ~76% accuracy.

```
# Q3: train your model here for 25 epochs.
# Print out training and validation loss/accuracy of the model after
each epoch
# Keep track of the model that achieved best validation loss thus far.
import time

EPOCHS = 25
model_copy = copy.deepcopy(model)
accuracy_ref = -1 #Lowest the accuracy can go is 0, so this lets the
accuracy reference and new model be updated on the first epoch
# Fill training code here
for epoch in range(EPOCHS):
    start_time = time.time()
    loss, accuracy =train(model, train_iterator, optimizer, criterion,
device) #Training data
    print(f'Epoch {epoch}, Train Loss {loss}, Train Accuracy
{accuracy}')
    valid_loss, valid_accuracy =evaluate(model, valid_iterator,
criterion, device) #Validation data
    print(f'Epoch {epoch}, Valid Loss {valid_loss}, Valid Accuracy
{valid_accuracy}')
    if valid_accuracy > accuracy_ref: #Copies the new model if the
validation accuracy is better than the previous copy
        accuracy_ref = valid_accuracy
        model_copy = copy.deepcopy(model)
        print(f'Model Copied')
    end_time = time.time()
    min, sec = epoch_time(start_time, end_time)
    print(f'Epoch took {min} minutes and {sec} seconds')
```

```
Epoch 0, Train Loss 2.3859293684363365, Train Accuracy
0.21515447443181818
Epoch 0, Valid Loss 1.6082023859024048, Valid Accuracy
0.40342371314764025
Model Copied
Epoch took 2 minutes and 32 seconds
Epoch 1, Train Loss 1.5127338414842433, Train Accuracy
0.43745649859986524
Epoch 1, Valid Loss 1.3785551190376282, Valid Accuracy
0.48405330926179885
Model Copied
Epoch took 2 minutes and 30 seconds
Epoch 2, Train Loss 1.3497779613191432, Train Accuracy
0.5120134942910888
Epoch 2, Valid Loss 1.2114853024482728, Valid Accuracy
0.5644990801811218
Model Copied
Epoch took 2 minutes and 20 seconds
Epoch 3, Train Loss 1.252753977071155, Train Accuracy
0.5508149858902801
```

```
Epoch 3, Valid Loss 1.1492135107517243, Valid Accuracy
0.5816061586141587
Model Copied
Epoch took 2 minutes and 21 seconds
Epoch 4, Train Loss 1.1764032918621192, Train Accuracy
0.5811496803706343
Epoch 4, Valid Loss 1.1288484692573548, Valid Accuracy
0.6064223349094391
Model Copied
Epoch took 2 minutes and 27 seconds
Epoch 5, Train Loss 1.1137279759753833, Train Accuracy
0.607915483076464
Epoch 5, Valid Loss 1.041766142845154, Valid Accuracy
0.6327435672283173
Model Copied
Epoch took 2 minutes and 25 seconds
Epoch 6, Train Loss 1.0533894764428788, Train Accuracy
0.6306143467399207
Epoch 6, Valid Loss 1.0123582810163498, Valid Accuracy
0.6456916362047196
Model Copied
Epoch took 2 minutes and 34 seconds
Epoch 7, Train Loss 1.016191769729961, Train Accuracy
0.6427503549917177
Epoch 7, Valid Loss 0.9771516680717468, Valid Accuracy
0.656031709909439
Model Copied
Epoch took 2 minutes and 51 seconds
Epoch 8, Train Loss 0.9675288193605163, Train Accuracy
0.6617116477679122
Epoch 8, Valid Loss 0.9208255469799042, Valid Accuracy
0.6822150737047196
Model Copied
Epoch took 3 minutes and 7 seconds
Epoch 9, Train Loss 0.9362695372917436, Train Accuracy
0.673189808360555
Epoch 9, Valid Loss 0.9574713259935379, Valid Accuracy 0.6697265625
Epoch took 2 minutes and 47 seconds
Epoch 10, Train Loss 0.915364300662821, Train Accuracy
0.67897372150963
Epoch 10, Valid Loss 0.8721305072307587, Valid Accuracy
0.6986787676811218
Model Copied
Epoch took 2 minutes and 43 seconds
Epoch 11, Train Loss 0.8793735033409162, Train Accuracy
0.6945791904899207
Epoch 11, Valid Loss 0.8667667210102081, Valid Accuracy
0.6991842836141586
Model Copied
```

```
Epoch took 2 minutes and 40 seconds
Epoch 12, Train Loss 0.8457419658926401, Train Accuracy
0.7068705609576269
Epoch 12, Valid Loss 0.8426105201244354, Valid Accuracy
0.7082375913858414
Model Copied
Epoch took 2 minutes and 43 seconds
Epoch 13, Train Loss 0.8229072757742621, Train Accuracy
0.7161807529628277
Epoch 13, Valid Loss 0.8680290371179581, Valid Accuracy
0.7025850176811218
Epoch took 2 minutes and 53 seconds
Epoch 14, Train Loss 0.8056459250775251, Train Accuracy
0.7222514203326269
Epoch 14, Valid Loss 0.7988647848367691, Valid Accuracy
0.7249540448188782
Model Copied
Epoch took 2 minutes and 49 seconds
Epoch 15, Train Loss 0.7857820635492151, Train Accuracy
0.7266770242290064
Epoch 15, Valid Loss 0.7764606267213822, Valid Accuracy
0.7301700353622437
Model Copied
Epoch took 2 minutes and 44 seconds
Epoch 16, Train Loss 0.7603419257158582, Train Accuracy
0.73777432536537
Epoch 16, Valid Loss 0.7669012516736984, Valid Accuracy
0.7389131426811218
Model Copied
Epoch took 2 minutes and 50 seconds
Epoch 17, Train Loss 0.7526960955424742, Train Accuracy
0.74021573161537
Epoch 17, Valid Loss 0.7977280527353287, Valid Accuracy
0.7365923702716828
Epoch took 2 minutes and 35 seconds
Epoch 18, Train Loss 0.7299478788944808, Train Accuracy
0.74827237224037
Epoch 18, Valid Loss 0.7806057006120681, Valid Accuracy
0.7405560672283172
Model Copied
Epoch took 2 minutes and 42 seconds
Epoch 19, Train Loss 0.7180966392836787, Train Accuracy
0.7520321377299048
Epoch 19, Valid Loss 0.7532464444637299, Valid Accuracy
0.7397977948188782
Epoch took 2 minutes and 41 seconds
Epoch 20, Train Loss 0.695985344323245, Train Accuracy
0.7610040838745508
Epoch 20, Valid Loss 0.7738403081893921, Valid Accuracy
```

```
0.7301470577716828
Epoch took 2 minutes and 45 seconds
Epoch 21, Train Loss 0.687265569852157, Train Accuracy
0.7641637074676427
Epoch 21, Valid Loss 0.7499783217906952, Valid Accuracy
0.7522977948188782
Model Copied
Epoch took 2 minutes and 44 seconds
Epoch 22, Train Loss 0.6752578406171366, Train Accuracy
0.7682998935607347
Epoch 22, Valid Loss 0.7266848772764206, Valid Accuracy
0.7558938413858414
Model Copied
Epoch took 2 minutes and 48 seconds
Epoch 23, Train Loss 0.6537438587031581, Train Accuracy
0.7748322087255392
Epoch 23, Valid Loss 0.7215611875057221, Valid Accuracy
0.7567210465669632
Model Copied
Epoch took 2 minutes and 48 seconds
Epoch 24, Train Loss 0.6516660617833788, Train Accuracy
0.7765660512853753
Epoch 24, Valid Loss 0.6996587306261063, Valid Accuracy
0.7605583637952804
Model Copied
Epoch took 2 minutes and 49 seconds
```

Evaluating the model

We then load the parameters of our model that achieved the best validation loss. You should expect to see ~75% accuracy of this model on the test dataset.

Finally, plot the confusion matrix of this model and comment on any interesting patterns you can observe there. For example, which two classes are confused the most?

```python
# Q4: Load the best performing model, evaluate it on the test dataset,
and print test accuracy.

# Also, print out the confusion matrox.

def get_predictions(model, iterator, device):

    model.eval()
    with torch.no_grad():
        labels = []
        probs = []

    # Q4: Fill code here.
```

```
        for (x, y) in iterator:
            x, y = x.to(device), y.to(device)
            predicted_output = model(x) #Prediction on test data
            labels.append(y)
            probs.append(predicted_output[0]) #Ignore the second
tensor



        labels = torch.cat(labels, dim = 0) #Converts all of the
tensors into 1 big tensor
        probs = torch.cat(probs, dim = 0)

    return labels, probs

labels, probs = get_predictions(model_copy, test_iterator, device)
#New model is used

pred_labels = torch.argmax(probs, 1) #Gets index or largest
possibility

print(torch.eq(labels, pred_labels)) #Matches the prediction and label
tensor values
```

```
tensor([False,  True,  True,  ...,  True,  True, False])
```

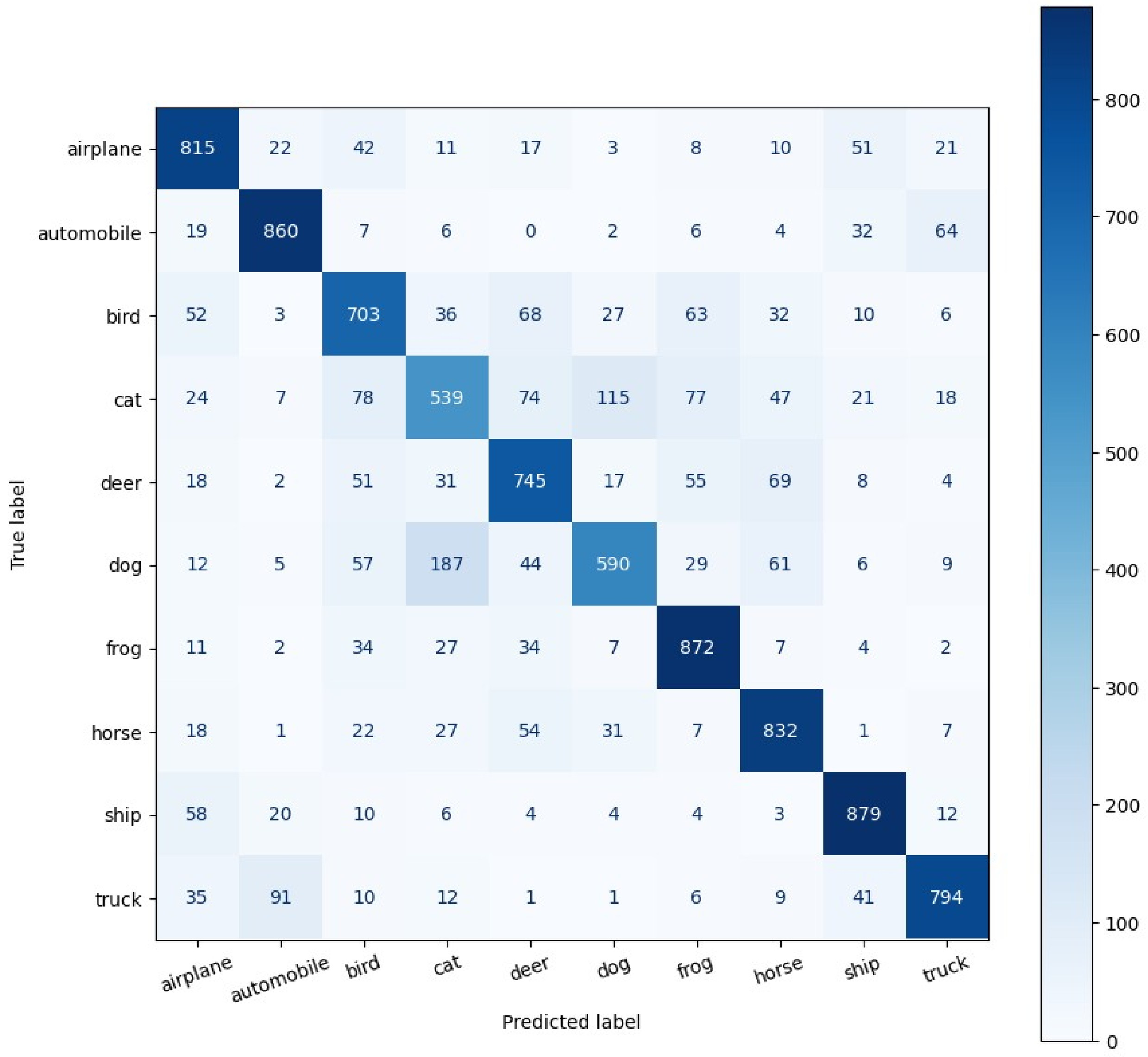
```
torch.eq(labels, pred_labels).sum() #Since it's boolean, I can just
sum it all up to get the amount of matches
```

```
tensor(7629)
```

```
torch.eq(labels, pred_labels).sum()/len(torch.eq(labels, pred_labels))
#Percentage that was predicted correctly at 76%
```

```
tensor(0.7629)
```

```
def plot_confusion_matrix(labels, pred_labels, classes):

    fig = plt.figure(figsize = (10, 10));
    ax = fig.add_subplot(1, 1, 1);
    cm = confusion_matrix(labels, pred_labels);
    cm = ConfusionMatrixDisplay(cm, display_labels = classes);
    cm.plot(values_format = 'd', cmap = 'Blues', ax = ax)
    plt.xticks(rotation = 20)

plot_confusion_matrix(labels, pred_labels, classes)
```



Conclusion

That's it! As a side project (this is not for credit and won't be graded), feel free to play around with different design choices that you made while building this network.

- Whether or not to normalize the color channels in the input.
- The learning rate parameter in Adam.
- The batch size.
- The number of training epochs.
- (and if you are feeling brave -- the AlexNet architecture itself.)

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.

```python
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',
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
```

```
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!
```

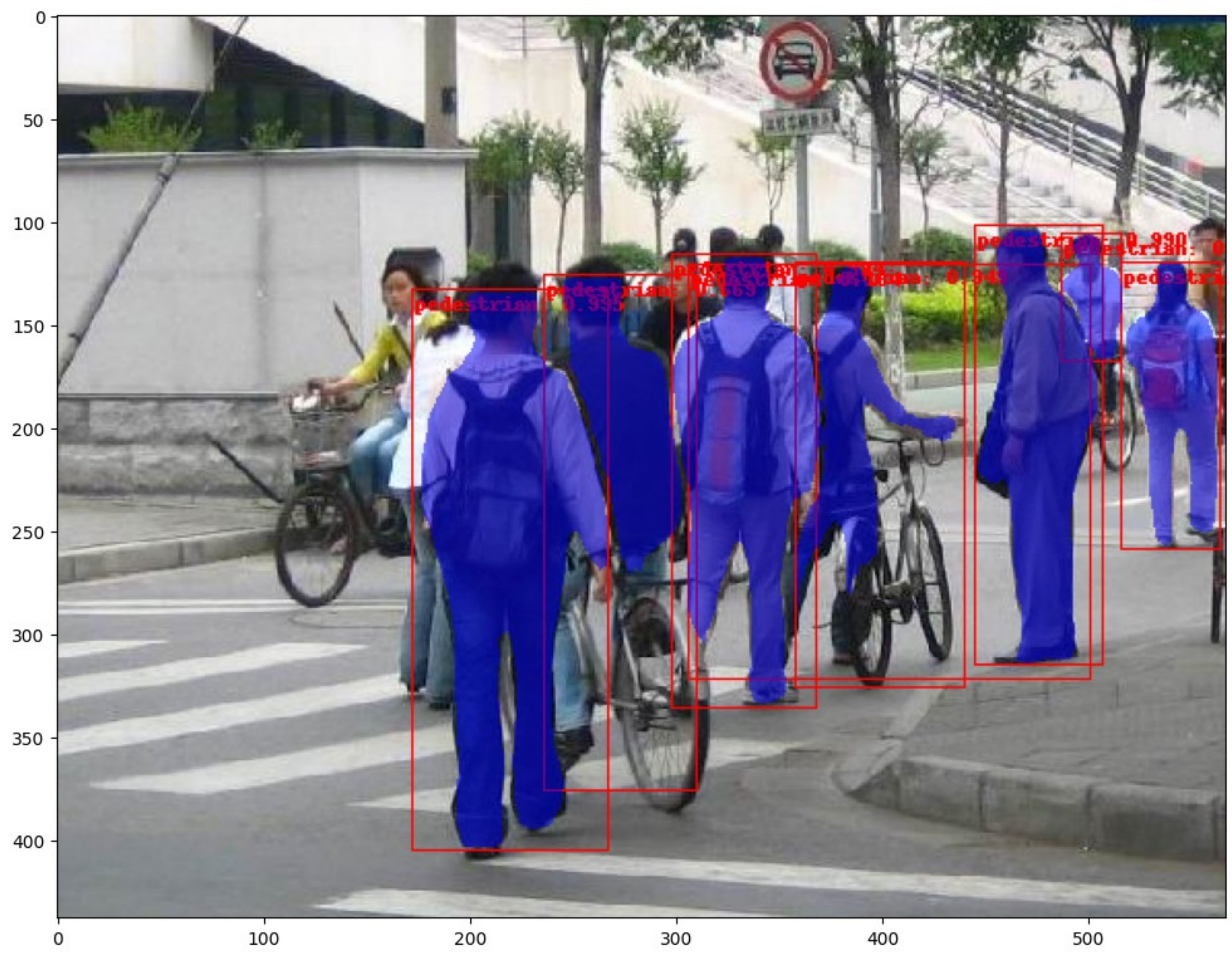
So after one epoch of training, we obtain a COCO-style mAP > 50, and a mask mAP of 65.

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_.

This is for Question 5 Part B and C using the finetuning option

```
# 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, 157MB/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(0.0989, grad_fn=<NllLossBackward0>),
'loss_box_reg': tensor(0.0605, grad_fn=<DivBackward0>),
'loss_objectness': tensor(0.0056,
grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_rpn_box_reg':
tensor(0.0073, 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',
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 = 10
```

```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, 92.5MB/s]

Epoch: [0]  [ 0/60]  eta: 0:09:22  lr: 0.000090  loss: 4.0232 (4.0232)
loss_classifier: 0.6608 (0.6608)  loss_box_reg: 0.1815 (0.1815)
loss_mask: 3.1586 (3.1586)  loss_objectness: 0.0194 (0.0194)
loss_rpn_box_reg: 0.0029 (0.0029)  time: 9.3771  data: 1.6565  max
mem: 2596
Epoch: [0]  [10/60]  eta: 0:01:07  lr: 0.000936  loss: 1.7460 (2.3481)
loss_classifier: 0.4496 (0.4475)  loss_box_reg: 0.3494 (0.3553)
loss_mask: 1.0930 (1.5102)  loss_objectness: 0.0304 (0.0277)
loss_rpn_box_reg: 0.0061 (0.0074)  time: 1.3599  data: 0.1567  max
mem: 2978
Epoch: [0]  [20/60]  eta: 0:00:39  lr: 0.001783  loss: 1.0728 (1.5656)
loss_classifier: 0.2308 (0.3128)  loss_box_reg: 0.2986 (0.3006)
loss_mask: 0.4108 (0.9232)  loss_objectness: 0.0153 (0.0225)
loss_rpn_box_reg: 0.0058 (0.0066)  time: 0.5580  data: 0.0099  max
mem: 2978
Epoch: [0]  [30/60]  eta: 0:00:25  lr: 0.002629  loss: 0.5781 (1.2421)
loss_classifier: 0.1208 (0.2424)  loss_box_reg: 0.1701 (0.2828)
loss_mask: 0.2195 (0.6921)  loss_objectness: 0.0083 (0.0175)
loss_rpn_box_reg: 0.0041 (0.0072)  time: 0.5665  data: 0.0111  max
mem: 3066
Epoch: [0]  [40/60]  eta: 0:00:15  lr: 0.003476  loss: 0.5432 (1.0732)
loss_classifier: 0.0734 (0.2017)  loss_box_reg: 0.1987 (0.2805)
loss_mask: 0.2021 (0.5695)  loss_objectness: 0.0044 (0.0144)
loss_rpn_box_reg: 0.0059 (0.0072)  time: 0.5562  data: 0.0098  max
mem: 3066
Epoch: [0]  [50/60]  eta: 0:00:07  lr: 0.004323  loss: 0.4499 (0.9406)
loss_classifier: 0.0452 (0.1706)  loss_box_reg: 0.1744 (0.2575)
loss_mask: 0.1770 (0.4932)  loss_objectness: 0.0026 (0.0123)
loss_rpn_box_reg: 0.0052 (0.0070)  time: 0.5357  data: 0.0101  max
mem: 3066
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.3485 (0.8478)
```

```
loss_classifier: 0.0368 (0.1514)  loss_box_reg: 0.1203 (0.2363)
loss_mask: 0.1702 (0.4425)  loss_objectness: 0.0015 (0.0108)
loss_rpn_box_reg: 0.0044 (0.0069)  time: 0.5412  data: 0.0090  max
mem: 3066
Epoch: [0] Total time: 0:00:42 (0.7033 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:27  model_time: 0.2204 (0.2204)
evaluator_time: 0.0211 (0.0211)  time: 0.5590  data: 0.3109  max mem:
3066
Test:  [49/50]  eta: 0:00:00  model_time: 0.1112 (0.1196)
evaluator_time: 0.0061 (0.0107)  time: 0.1321  data: 0.0052  max mem:
3066
Test: Total time: 0:00:07 (0.1491 s / it)
Averaged stats: model_time: 0.1112 (0.1196)  evaluator_time: 0.0061
(0.0107)
Accumulating evaluation results...
DONE (t=0.03s).
Accumulating evaluation results...
DONE (t=0.03s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.662
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.849
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.357
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.559
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.680
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.338
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.727
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.727
 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 ] = 0.713
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.736
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.693
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.960
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.868
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.369
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.459
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.713
 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 ] = 0.750
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.751
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.754
Epoch: [1]  [ 0/60]  eta: 0:01:02  lr: 0.005000  loss: 0.3659 (0.3659)
loss_classifier: 0.0570 (0.0570)  loss_box_reg: 0.1407 (0.1407)
loss_mask: 0.1630 (0.1630)  loss_objectness: 0.0016 (0.0016)
loss_rpn_box_reg: 0.0036 (0.0036)  time: 1.0337  data: 0.4756  max
mem: 3066
Epoch: [1]  [10/60]  eta: 0:00:30  lr: 0.005000  loss: 0.3129 (0.3190)
loss_classifier: 0.0385 (0.0399)  loss_box_reg: 0.1143 (0.1172)
loss_mask: 0.1380 (0.1558)  loss_objectness: 0.0005 (0.0009)
loss_rpn_box_reg: 0.0049 (0.0052)  time: 0.6191  data: 0.0496  max
mem: 3066
Epoch: [1]  [20/60]  eta: 0:00:24  lr: 0.005000  loss: 0.2991 (0.3156)
loss_classifier: 0.0334 (0.0396)  loss_box_reg: 0.0931 (0.1061)
loss_mask: 0.1543 (0.1634)  loss_objectness: 0.0007 (0.0016)
loss_rpn_box_reg: 0.0048 (0.0050)  time: 0.5884  data: 0.0081  max
mem: 3066
Epoch: [1]  [30/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2847 (0.3090)
loss_classifier: 0.0421 (0.0408)  loss_box_reg: 0.0901 (0.1063)
loss_mask: 0.1517 (0.1552)  loss_objectness: 0.0015 (0.0017)
loss_rpn_box_reg: 0.0043 (0.0049)  time: 0.5998  data: 0.0097  max
mem: 3464
Epoch: [1]  [40/60]  eta: 0:00:12  lr: 0.005000  loss: 0.2822 (0.3030)
loss_classifier: 0.0421 (0.0400)  loss_box_reg: 0.0887 (0.1028)
loss_mask: 0.1373 (0.1540)  loss_objectness: 0.0007 (0.0014)
loss_rpn_box_reg: 0.0038 (0.0048)  time: 0.6153  data: 0.0090  max
mem: 3464
Epoch: [1]  [50/60]  eta: 0:00:06  lr: 0.005000  loss: 0.2812 (0.3023)
loss_classifier: 0.0341 (0.0400)  loss_box_reg: 0.0887 (0.1020)
loss_mask: 0.1392 (0.1538)  loss_objectness: 0.0005 (0.0014)
```

```
loss_rpn_box_reg: 0.0038 (0.0051)  time: 0.6257  data: 0.0098  max
mem: 3464
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2764 (0.2969)
loss_classifier: 0.0402 (0.0396)  loss_box_reg: 0.0861 (0.0994)
loss_mask: 0.1343 (0.1512)  loss_objectness: 0.0006 (0.0014)
loss_rpn_box_reg: 0.0057 (0.0052)  time: 0.6134  data: 0.0094  max
mem: 3464
Epoch: [1] Total time: 0:00:36 (0.6161 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:39  model_time: 0.2075 (0.2075)
evaluator_time: 0.0281 (0.0281)  time: 0.7880  data: 0.5507  max mem:
3464
Test:  [49/50]  eta: 0:00:00  model_time: 0.0969 (0.1079)
evaluator_time: 0.0032 (0.0060)  time: 0.1103  data: 0.0038  max mem:
3464
Test: Total time: 0:00:06 (0.1350 s / it)
Averaged stats: model_time: 0.0969 (0.1079)  evaluator_time: 0.0032
(0.0060)
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 ] = 0.777
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.966
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.916
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.367
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.581
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.801
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.396
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.827
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.827
 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 ] = 0.725
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.846
IoU metric: segm
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.743
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.971
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.921
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.354
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.532
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.761
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.377
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.787
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.788
 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 ] = 0.750
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.798
Epoch: [2]  [ 0/60]  eta: 0:00:57  lr: 0.005000  loss: 0.2624 (0.2624)
loss_classifier: 0.0328 (0.0328)  loss_box_reg: 0.0768 (0.0768)
loss_mask: 0.1485 (0.1485)  loss_objectness: 0.0001 (0.0001)
loss_rpn_box_reg: 0.0043 (0.0043)  time: 0.9614  data: 0.3912  max
mem: 3464
Epoch: [2]  [10/60]  eta: 0:00:29  lr: 0.005000  loss: 0.2395 (0.2361)
loss_classifier: 0.0326 (0.0314)  loss_box_reg: 0.0622 (0.0653)
loss_mask: 0.1212 (0.1340)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0043 (0.0047)  time: 0.5913  data: 0.0433  max
mem: 3464
Epoch: [2]  [20/60]  eta: 0:00:23  lr: 0.005000  loss: 0.2164 (0.2381)
loss_classifier: 0.0306 (0.0326)  loss_box_reg: 0.0592 (0.0659)
loss_mask: 0.1212 (0.1343)  loss_objectness: 0.0005 (0.0006)
loss_rpn_box_reg: 0.0042 (0.0047)  time: 0.5783  data: 0.0097  max
mem: 3464
Epoch: [2]  [30/60]  eta: 0:00:17  lr: 0.005000  loss: 0.2076 (0.2267)
loss_classifier: 0.0270 (0.0301)  loss_box_reg: 0.0574 (0.0631)
loss_mask: 0.1179 (0.1286)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0031 (0.0042)  time: 0.6047  data: 0.0093  max
mem: 3464
Epoch: [2]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.2118 (0.2347)
loss_classifier: 0.0270 (0.0311)  loss_box_reg: 0.0595 (0.0677)
loss_mask: 0.1246 (0.1308)  loss_objectness: 0.0002 (0.0005)
loss_rpn_box_reg: 0.0037 (0.0046)  time: 0.5985  data: 0.0094  max
mem: 3464
```

```
Epoch: [2]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2745 (0.2435)
loss_classifier: 0.0311 (0.0325)  loss_box_reg: 0.0837 (0.0720)
loss_mask: 0.1364 (0.1328)  loss_objectness: 0.0005 (0.0007)
loss_rpn_box_reg: 0.0054 (0.0054)  time: 0.5857  data: 0.0099  max
mem: 3464
Epoch: [2]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2684 (0.2462)
loss_classifier: 0.0331 (0.0332)  loss_box_reg: 0.0837 (0.0742)
loss_mask: 0.1263 (0.1328)  loss_objectness: 0.0006 (0.0008)
loss_rpn_box_reg: 0.0035 (0.0052)  time: 0.5700  data: 0.0085  max
mem: 3464
Epoch: [2] Total time: 0:00:35 (0.5971 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:27  model_time: 0.1593 (0.1593)
evaluator_time: 0.0136 (0.0136)  time: 0.5565  data: 0.3816  max mem:
3464
Test:  [49/50]  eta: 0:00:00  model_time: 0.0989 (0.1055)
evaluator_time: 0.0030 (0.0047)  time: 0.1097  data: 0.0037  max mem:
3464
Test: Total time: 0:00:06 (0.1273 s / it)
Averaged stats: model_time: 0.0989 (0.1055)  evaluator_time: 0.0030
(0.0047)
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 ] = 0.746
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.981
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.352
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.579
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.760
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.385
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.788
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.788
 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 ] = 0.775
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.798
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.739
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.972
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.890
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.394
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.495
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.751
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.376
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.781
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.783
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.725
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.792
Epoch: [3]  [ 0/60]  eta: 0:00:59  lr: 0.000500  loss: 0.2168 (0.2168)
loss_classifier: 0.0278 (0.0278)  loss_box_reg: 0.0707 (0.0707)
loss_mask: 0.1120 (0.1120)  loss_objectness: 0.0021 (0.0021)
loss_rpn_box_reg: 0.0043 (0.0043)  time: 0.9965  data: 0.3444  max
mem: 3464
Epoch: [3]  [10/60]  eta: 0:00:30  lr: 0.000500  loss: 0.2418 (0.2369)
loss_classifier: 0.0279 (0.0308)  loss_box_reg: 0.0656 (0.0665)
loss_mask: 0.1386 (0.1345)  loss_objectness: 0.0007 (0.0010)
loss_rpn_box_reg: 0.0036 (0.0041)  time: 0.6190  data: 0.0402  max
mem: 3464
Epoch: [3]  [20/60]  eta: 0:00:25  lr: 0.000500  loss: 0.2288 (0.2314)
loss_classifier: 0.0299 (0.0303)  loss_box_reg: 0.0602 (0.0632)
loss_mask: 0.1326 (0.1333)  loss_objectness: 0.0005 (0.0009)
loss_rpn_box_reg: 0.0032 (0.0037)  time: 0.6124  data: 0.0094  max
mem: 3464
Epoch: [3]  [30/60]  eta: 0:00:18  lr: 0.000500  loss: 0.2084 (0.2180)
loss_classifier: 0.0284 (0.0304)  loss_box_reg: 0.0457 (0.0558)
loss_mask: 0.1106 (0.1270)  loss_objectness: 0.0004 (0.0009)
loss_rpn_box_reg: 0.0030 (0.0038)  time: 0.6178  data: 0.0095  max
mem: 3464
Epoch: [3]  [40/60]  eta: 0:00:12  lr: 0.000500  loss: 0.1870 (0.2101)
loss_classifier: 0.0254 (0.0296)  loss_box_reg: 0.0372 (0.0535)
```

```
loss_mask: 0.1054 (0.1226)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0022 (0.0035)  time: 0.5934  data: 0.0098  max
mem: 3464
Epoch: [3]  [50/60]  eta: 0:00:06  lr: 0.000500  loss: 0.1886 (0.2074)
loss_classifier: 0.0208 (0.0285)  loss_box_reg: 0.0435 (0.0524)
loss_mask: 0.1120 (0.1220)  loss_objectness: 0.0004 (0.0010)
loss_rpn_box_reg: 0.0020 (0.0035)  time: 0.6067  data: 0.0106  max
mem: 3464
Epoch: [3]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1886 (0.2092)
loss_classifier: 0.0252 (0.0288)  loss_box_reg: 0.0435 (0.0524)
loss_mask: 0.1151 (0.1234)  loss_objectness: 0.0004 (0.0009)
loss_rpn_box_reg: 0.0031 (0.0036)  time: 0.6042  data: 0.0099  max
mem: 3464
Epoch: [3] Total time: 0:00:36 (0.6153 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:28  model_time: 0.1925 (0.1925)
evaluator_time: 0.0142 (0.0142)  time: 0.5605  data: 0.3520  max mem:
3464
Test:  [49/50]  eta: 0:00:00  model_time: 0.1048 (0.1132)
evaluator_time: 0.0047 (0.0065)  time: 0.1237  data: 0.0090  max mem:
3464
Test: Total time: 0:00:07 (0.1403 s / it)
Averaged stats: model_time: 0.1048 (0.1132)  evaluator_time: 0.0047
(0.0065)
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 ] = 0.828
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.981
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.950
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.368
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.650
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.846
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.419
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.866
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.866
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.880
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.758
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.378
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.517
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.772
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.382
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.797
 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 ] = 0.567
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.812
Epoch: [4]  [ 0/60]  eta: 0:01:03  lr: 0.000500  loss: 0.1764 (0.1764)
loss_classifier: 0.0218 (0.0218)  loss_box_reg: 0.0406 (0.0406)
loss_mask: 0.1091 (0.1091)  loss_objectness: 0.0004 (0.0004)
loss_rpn_box_reg: 0.0045 (0.0045)  time: 1.0649  data: 0.3697  max
mem: 3464
Epoch: [4]  [10/60]  eta: 0:00:31  lr: 0.000500  loss: 0.2057 (0.2016)
loss_classifier: 0.0229 (0.0288)  loss_box_reg: 0.0416 (0.0444)
loss_mask: 0.1205 (0.1236)  loss_objectness: 0.0006 (0.0006)
loss_rpn_box_reg: 0.0044 (0.0043)  time: 0.6200  data: 0.0414  max
mem: 3464
Epoch: [4]  [20/60]  eta: 0:00:25  lr: 0.000500  loss: 0.2057 (0.2106)
loss_classifier: 0.0295 (0.0301)  loss_box_reg: 0.0496 (0.0495)
loss_mask: 0.1205 (0.1261)  loss_objectness: 0.0007 (0.0008)
loss_rpn_box_reg: 0.0037 (0.0041)  time: 0.6055  data: 0.0097  max
mem: 3700
Epoch: [4]  [30/60]  eta: 0:00:18  lr: 0.000500  loss: 0.1907 (0.2052)
loss_classifier: 0.0274 (0.0292)  loss_box_reg: 0.0480 (0.0478)
loss_mask: 0.1135 (0.1238)  loss_objectness: 0.0005 (0.0007)
loss_rpn_box_reg: 0.0029 (0.0036)  time: 0.6023  data: 0.0097  max
```

```
mem: 3700
Epoch: [4]  [40/60]  eta: 0:00:12  lr: 0.000500  loss: 0.1797 (0.2047)
loss_classifier: 0.0246 (0.0302)  loss_box_reg: 0.0437 (0.0480)
loss_mask: 0.1135 (0.1223)  loss_objectness: 0.0005 (0.0007)
loss_rpn_box_reg: 0.0027 (0.0034)  time: 0.5845  data: 0.0109  max
mem: 3700
Epoch: [4]  [50/60]  eta: 0:00:06  lr: 0.000500  loss: 0.1739 (0.1996)
loss_classifier: 0.0253 (0.0290)  loss_box_reg: 0.0385 (0.0468)
loss_mask: 0.1050 (0.1199)  loss_objectness: 0.0005 (0.0008)
loss_rpn_box_reg: 0.0023 (0.0032)  time: 0.5931  data: 0.0120  max
mem: 3700
Epoch: [4]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1693 (0.1997)
loss_classifier: 0.0275 (0.0292)  loss_box_reg: 0.0385 (0.0467)
loss_mask: 0.1055 (0.1197)  loss_objectness: 0.0005 (0.0009)
loss_rpn_box_reg: 0.0023 (0.0032)  time: 0.6095  data: 0.0095  max
mem: 3700
Epoch: [4] Total time: 0:00:36 (0.6121 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:26  model_time: 0.1775 (0.1775)
evaluator_time: 0.0206 (0.0206)  time: 0.5346  data: 0.3344  max mem:
3700
Test:  [49/50]  eta: 0:00:00  model_time: 0.0987 (0.1066)
evaluator_time: 0.0026 (0.0048)  time: 0.1097  data: 0.0037  max mem:
3700
Test: Total time: 0:00:06 (0.1277 s / it)
Averaged stats: model_time: 0.0987 (0.1066)  evaluator_time: 0.0026
(0.0048)
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 ] = 0.840
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.981
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.950
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.368
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.674
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.859
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.426
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.881
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.881
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.897
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.757
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.913
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.378
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.530
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.770
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.385
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.799
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.801
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.810
Epoch: [5]  [ 0/60]  eta: 0:00:59  lr: 0.000500  loss: 0.2363 (0.2363)
loss_classifier: 0.0346 (0.0346)  loss_box_reg: 0.0638 (0.0638)
loss_mask: 0.1342 (0.1342)  loss_objectness: 0.0002 (0.0002)
loss_rpn_box_reg: 0.0035 (0.0035)  time: 0.9918  data: 0.4285  max
mem: 3700
Epoch: [5]  [10/60]  eta: 0:00:30  lr: 0.000500  loss: 0.1874 (0.1939)
loss_classifier: 0.0242 (0.0280)  loss_box_reg: 0.0449 (0.0424)
loss_mask: 0.1151 (0.1199)  loss_objectness: 0.0004 (0.0011)
loss_rpn_box_reg: 0.0021 (0.0025)  time: 0.6078  data: 0.0489  max
mem: 3700
Epoch: [5]  [20/60]  eta: 0:00:24  lr: 0.000500  loss: 0.1931 (0.2105)
loss_classifier: 0.0356 (0.0326)  loss_box_reg: 0.0477 (0.0501)
loss_mask: 0.1145 (0.1234)  loss_objectness: 0.0004 (0.0014)
loss_rpn_box_reg: 0.0021 (0.0030)  time: 0.5918  data: 0.0103  max
mem: 3700
Epoch: [5]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1931 (0.2008)
loss_classifier: 0.0284 (0.0289)  loss_box_reg: 0.0477 (0.0472)
```

```
loss_mask: 0.1142 (0.1206)  loss_objectness: 0.0003 (0.0012)
loss_rpn_box_reg: 0.0028 (0.0030)  time: 0.5921  data: 0.0098  max
mem: 3700
Epoch: [5]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1741 (0.1982)
loss_classifier: 0.0214 (0.0286)  loss_box_reg: 0.0373 (0.0459)
loss_mask: 0.1122 (0.1197)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0026 (0.0030)  time: 0.5814  data: 0.0102  max
mem: 3700
Epoch: [5]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1935 (0.1971)
loss_classifier: 0.0248 (0.0283)  loss_box_reg: 0.0424 (0.0465)
loss_mask: 0.1114 (0.1183)  loss_objectness: 0.0004 (0.0009)
loss_rpn_box_reg: 0.0034 (0.0031)  time: 0.6012  data: 0.0099  max
mem: 3700
Epoch: [5]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1741 (0.1930)
loss_classifier: 0.0218 (0.0271)  loss_box_reg: 0.0325 (0.0444)
loss_mask: 0.1061 (0.1175)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0026 (0.0031)  time: 0.6125  data: 0.0085  max
mem: 3700
Epoch: [5] Total time: 0:00:36 (0.6051 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:28  model_time: 0.1591 (0.1591)
evaluator_time: 0.0116 (0.0116)  time: 0.5638  data: 0.3914  max mem:
3700
Test:  [49/50]  eta: 0:00:00  model_time: 0.1025 (0.1087)
evaluator_time: 0.0039 (0.0052)  time: 0.1167  data: 0.0041  max mem:
3700
Test: Total time: 0:00:06 (0.1351 s / it)
Averaged stats: model_time: 0.1025 (0.1087)  evaluator_time: 0.0039
(0.0052)
Accumulating evaluation results...
DONE (t=0.03s).
Accumulating evaluation results...
DONE (t=0.03s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.837
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.930
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.388
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.629
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.858
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.423
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.875
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.875
 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 ] = 0.812
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.892
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.762
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.969
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.932
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.403
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.544
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.385
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.804
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.806
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.633
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.813
Epoch: [6]  [ 0/60]  eta: 0:01:24  lr: 0.000050  loss: 0.2115 (0.2115)
loss_classifier: 0.0332 (0.0332)  loss_box_reg: 0.0507 (0.0507)
loss_mask: 0.1236 (0.1236)  loss_objectness: 0.0005 (0.0005)
loss_rpn_box_reg: 0.0036 (0.0036)  time: 1.4092  data: 0.6883  max
mem: 3700
Epoch: [6]  [10/60]  eta: 0:00:33  lr: 0.000050  loss: 0.2112 (0.1921)
loss_classifier: 0.0329 (0.0284)  loss_box_reg: 0.0381 (0.0406)
loss_mask: 0.1210 (0.1200)  loss_objectness: 0.0002 (0.0004)
loss_rpn_box_reg: 0.0027 (0.0027)  time: 0.6647  data: 0.0694  max
mem: 3700
Epoch: [6]  [20/60]  eta: 0:00:25  lr: 0.000050  loss: 0.1861 (0.1938)
loss_classifier: 0.0253 (0.0281)  loss_box_reg: 0.0354 (0.0426)
loss_mask: 0.1169 (0.1195)  loss_objectness: 0.0003 (0.0005)
loss_rpn_box_reg: 0.0027 (0.0032)  time: 0.6113  data: 0.0086  max
```

```
mem: 3700
Epoch: [6]  [30/60]  eta: 0:00:18  lr: 0.000050  loss: 0.1861 (0.1884)
loss_classifier: 0.0226 (0.0266)  loss_box_reg: 0.0354 (0.0416)
loss_mask: 0.1085 (0.1164)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0024 (0.0031)  time: 0.6113  data: 0.0095  max
mem: 3700
Epoch: [6]  [40/60]  eta: 0:00:12  lr: 0.000050  loss: 0.1644 (0.1829)
loss_classifier: 0.0184 (0.0256)  loss_box_reg: 0.0317 (0.0396)
loss_mask: 0.1084 (0.1143)  loss_objectness: 0.0002 (0.0005)
loss_rpn_box_reg: 0.0022 (0.0029)  time: 0.5842  data: 0.0091  max
mem: 3700
Epoch: [6]  [50/60]  eta: 0:00:06  lr: 0.000050  loss: 0.1649 (0.1882)
loss_classifier: 0.0205 (0.0266)  loss_box_reg: 0.0322 (0.0410)
loss_mask: 0.1118 (0.1170)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0018 (0.0029)  time: 0.5804  data: 0.0091  max
mem: 3700
Epoch: [6]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1811 (0.1904)
loss_classifier: 0.0275 (0.0268)  loss_box_reg: 0.0370 (0.0415)
loss_mask: 0.1207 (0.1183)  loss_objectness: 0.0004 (0.0008)
loss_rpn_box_reg: 0.0027 (0.0030)  time: 0.5921  data: 0.0083  max
mem: 3700
Epoch: [6] Total time: 0:00:36 (0.6128 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:40  model_time: 0.1714 (0.1714)
evaluator_time: 0.0194 (0.0194)  time: 0.8045  data: 0.6114  max mem:
3700
Test:  [49/50]  eta: 0:00:00  model_time: 0.1001 (0.1105)
evaluator_time: 0.0025 (0.0057)  time: 0.1101  data: 0.0037  max mem:
3700
Test: Total time: 0:00:07 (0.1411 s / it)
Averaged stats: model_time: 0.1001 (0.1105)  evaluator_time: 0.0025
(0.0057)
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 ] = 0.836
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.941
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.355
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.655
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.858
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.420
 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 ] = 0.467
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.812
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.892
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.763
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.917
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.403
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.540
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.386
 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 ] = 0.805
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.633
 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 ] = 0.811
Epoch: [7]  [ 0/60]  eta: 0:00:58  lr: 0.000050  loss: 0.1546 (0.1546)
loss_classifier: 0.0144 (0.0144)  loss_box_reg: 0.0233 (0.0233)
loss_mask: 0.1152 (0.1152)  loss_objectness: 0.0002 (0.0002)
loss_rpn_box_reg: 0.0015 (0.0015)  time: 0.9699  data: 0.3689  max
mem: 3700
Epoch: [7]  [10/60]  eta: 0:00:31  lr: 0.000050  loss: 0.1793 (0.1782)
loss_classifier: 0.0260 (0.0260)  loss_box_reg: 0.0386 (0.0390)
loss_mask: 0.1070 (0.1103)  loss_objectness: 0.0002 (0.0003)
loss_rpn_box_reg: 0.0030 (0.0026)  time: 0.6248  data: 0.0409  max
mem: 3700
Epoch: [7]  [20/60]  eta: 0:00:24  lr: 0.000050  loss: 0.1793 (0.1823)
```

```
loss_classifier: 0.0254 (0.0261)  loss_box_reg: 0.0397 (0.0411)
loss_mask: 0.1049 (0.1116)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0030 (0.0029)  time: 0.6018  data: 0.0095  max
mem: 3700
Epoch: [7]  [30/60]  eta: 0:00:17  lr: 0.000050  loss: 0.1716 (0.1806)
loss_classifier: 0.0230 (0.0257)  loss_box_reg: 0.0397 (0.0396)
loss_mask: 0.1068 (0.1118)  loss_objectness: 0.0003 (0.0005)
loss_rpn_box_reg: 0.0027 (0.0029)  time: 0.5863  data: 0.0099  max
mem: 3700
Epoch: [7]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1733 (0.1817)
loss_classifier: 0.0228 (0.0256)  loss_box_reg: 0.0364 (0.0394)
loss_mask: 0.1077 (0.1133)  loss_objectness: 0.0003 (0.0005)
loss_rpn_box_reg: 0.0028 (0.0030)  time: 0.5703  data: 0.0087  max
mem: 3700
Epoch: [7]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1733 (0.1853)
loss_classifier: 0.0205 (0.0257)  loss_box_reg: 0.0364 (0.0405)
loss_mask: 0.1171 (0.1155)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0031 (0.0030)  time: 0.5830  data: 0.0088  max
mem: 3700
Epoch: [7]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1799 (0.1869)
loss_classifier: 0.0235 (0.0261)  loss_box_reg: 0.0418 (0.0411)
loss_mask: 0.1171 (0.1161)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0028 (0.0030)  time: 0.5975  data: 0.0084  max
mem: 3700
Epoch: [7] Total time: 0:00:36 (0.6016 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:27  model_time: 0.1995 (0.1995)
evaluator_time: 0.0102 (0.0102)  time: 0.5437  data: 0.3323  max mem:
3700
Test:  [49/50]  eta: 0:00:00  model_time: 0.0981 (0.1083)
evaluator_time: 0.0026 (0.0047)  time: 0.1100  data: 0.0036  max mem:
3700
Test: Total time: 0:00:06 (0.1294 s / it)
Averaged stats: model_time: 0.0981 (0.1083)  evaluator_time: 0.0026
(0.0047)
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 ] = 0.839
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.941
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.355
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.659
 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 ] = 0.424
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.879
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.879
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.896
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.763
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.918
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.403
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.534
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.386
 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 ] = 0.805
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.633
 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 ] = 0.811
Epoch: [8]  [ 0/60]  eta: 0:01:05  lr: 0.000050  loss: 0.1549 (0.1549)
loss_classifier: 0.0242 (0.0242)  loss_box_reg: 0.0308 (0.0308)
loss_mask: 0.0977 (0.0977)  loss_objectness: 0.0003 (0.0003)
loss_rpn_box_reg: 0.0020 (0.0020)  time: 1.0998  data: 0.4827  max
mem: 3700
Epoch: [8]  [10/60]  eta: 0:00:29  lr: 0.000050  loss: 0.1596 (0.1569)
loss_classifier: 0.0237 (0.0219)  loss_box_reg: 0.0263 (0.0260)
loss_mask: 0.1095 (0.1062)  loss_objectness: 0.0003 (0.0005)
```

```
loss_rpn_box_reg: 0.0020 (0.0022)  time: 0.5801  data: 0.0547  max
mem: 3700
Epoch: [8]  [20/60]  eta: 0:00:23  lr: 0.000050  loss: 0.1797 (0.1906)
loss_classifier: 0.0262 (0.0277)  loss_box_reg: 0.0315 (0.0408)
loss_mask: 0.1151 (0.1184)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.5693  data: 0.0113  max
mem: 3700
Epoch: [8]  [30/60]  eta: 0:00:18  lr: 0.000050  loss: 0.1824 (0.1826)
loss_classifier: 0.0262 (0.0254)  loss_box_reg: 0.0394 (0.0391)
loss_mask: 0.1151 (0.1145)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0028 (0.0029)  time: 0.6157  data: 0.0108  max
mem: 3700
Epoch: [8]  [40/60]  eta: 0:00:12  lr: 0.000050  loss: 0.1793 (0.1847)
loss_classifier: 0.0236 (0.0260)  loss_box_reg: 0.0394 (0.0401)
loss_mask: 0.1075 (0.1150)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.6154  data: 0.0103  max
mem: 3700
Epoch: [8]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1794 (0.1860)
loss_classifier: 0.0266 (0.0254)  loss_box_reg: 0.0393 (0.0395)
loss_mask: 0.1112 (0.1176)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.5890  data: 0.0097  max
mem: 3700
Epoch: [8]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1794 (0.1867)
loss_classifier: 0.0276 (0.0260)  loss_box_reg: 0.0365 (0.0408)
loss_mask: 0.1053 (0.1166)  loss_objectness: 0.0002 (0.0006)
loss_rpn_box_reg: 0.0026 (0.0028)  time: 0.5954  data: 0.0087  max
mem: 3700
Epoch: [8] Total time: 0:00:36 (0.6055 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:28  model_time: 0.1800 (0.1800)
evaluator_time: 0.0126 (0.0126)  time: 0.5767  data: 0.3823  max mem:
3700
Test:  [49/50]  eta: 0:00:00  model_time: 0.1195 (0.1192)
evaluator_time: 0.0047 (0.0068)  time: 0.1389  data: 0.0058  max mem:
3700
Test: Total time: 0:00:07 (0.1499 s / it)
Averaged stats: model_time: 0.1195 (0.1192)  evaluator_time: 0.0047
(0.0068)
Accumulating evaluation results...
DONE (t=0.03s).
Accumulating evaluation results...
DONE (t=0.03s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.840
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.931
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.355
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.659
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.861
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.425
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.878
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.878
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.895
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.765
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.918
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.403
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.535
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.777
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.385
 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 ] = 0.805
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.633
 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 ] = 0.811
Epoch: [9]  [ 0/60]  eta: 0:01:36  lr: 0.000005  loss: 0.1179 (0.1179)
loss_classifier: 0.0076 (0.0076)  loss_box_reg: 0.0123 (0.0123)
loss_mask: 0.0972 (0.0972)  loss_objectness: 0.0002 (0.0002)
loss_rpn_box_reg: 0.0006 (0.0006)  time: 1.6021  data: 0.8491  max
mem: 3700
```

```
Epoch: [9]  [10/60]  eta: 0:00:34  lr: 0.000005  loss: 0.1728 (0.1847)
loss_classifier: 0.0242 (0.0255)  loss_box_reg: 0.0357 (0.0386)
loss_mask: 0.1076 (0.1176)  loss_objectness: 0.0002 (0.0007)
loss_rpn_box_reg: 0.0020 (0.0023)  time: 0.6896  data: 0.0844  max
mem: 3700
Epoch: [9]  [20/60]  eta: 0:00:26  lr: 0.000005  loss: 0.1840 (0.1879)
loss_classifier: 0.0292 (0.0284)  loss_box_reg: 0.0365 (0.0414)
loss_mask: 0.1125 (0.1150)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0026 (0.0025)  time: 0.6100  data: 0.0099  max
mem: 3700
Epoch: [9]  [30/60]  eta: 0:00:19  lr: 0.000005  loss: 0.1834 (0.1829)
loss_classifier: 0.0286 (0.0269)  loss_box_reg: 0.0356 (0.0394)
loss_mask: 0.1061 (0.1132)  loss_objectness: 0.0003 (0.0008)
loss_rpn_box_reg: 0.0031 (0.0026)  time: 0.6119  data: 0.0100  max
mem: 3778
Epoch: [9]  [40/60]  eta: 0:00:12  lr: 0.000005  loss: 0.1670 (0.1854)
loss_classifier: 0.0234 (0.0268)  loss_box_reg: 0.0321 (0.0401)
loss_mask: 0.1090 (0.1150)  loss_objectness: 0.0002 (0.0006)
loss_rpn_box_reg: 0.0032 (0.0028)  time: 0.5948  data: 0.0085  max
mem: 3778
Epoch: [9]  [50/60]  eta: 0:00:06  lr: 0.000005  loss: 0.1898 (0.1883)
loss_classifier: 0.0244 (0.0267)  loss_box_reg: 0.0388 (0.0407)
loss_mask: 0.1180 (0.1174)  loss_objectness: 0.0002 (0.0006)
loss_rpn_box_reg: 0.0032 (0.0029)  time: 0.5872  data: 0.0101  max
mem: 3778
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.1793 (0.1891)
loss_classifier: 0.0253 (0.0269)  loss_box_reg: 0.0378 (0.0417)
loss_mask: 0.1127 (0.1169)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0029 (0.0030)  time: 0.5977  data: 0.0096  max
mem: 3778
Epoch: [9] Total time: 0:00:37 (0.6213 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:42  model_time: 0.2411 (0.2411)
evaluator_time: 0.0180 (0.0180)  time: 0.8438  data: 0.5829  max mem:
3778
Test:  [49/50]  eta: 0:00:00  model_time: 0.0998 (0.1116)
evaluator_time: 0.0029 (0.0051)  time: 0.1125  data: 0.0038  max mem:
3778
Test: Total time: 0:00:06 (0.1391 s / it)
Averaged stats: model_time: 0.0998 (0.1116)  evaluator_time: 0.0029
(0.0051)
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 ] = 0.839
```

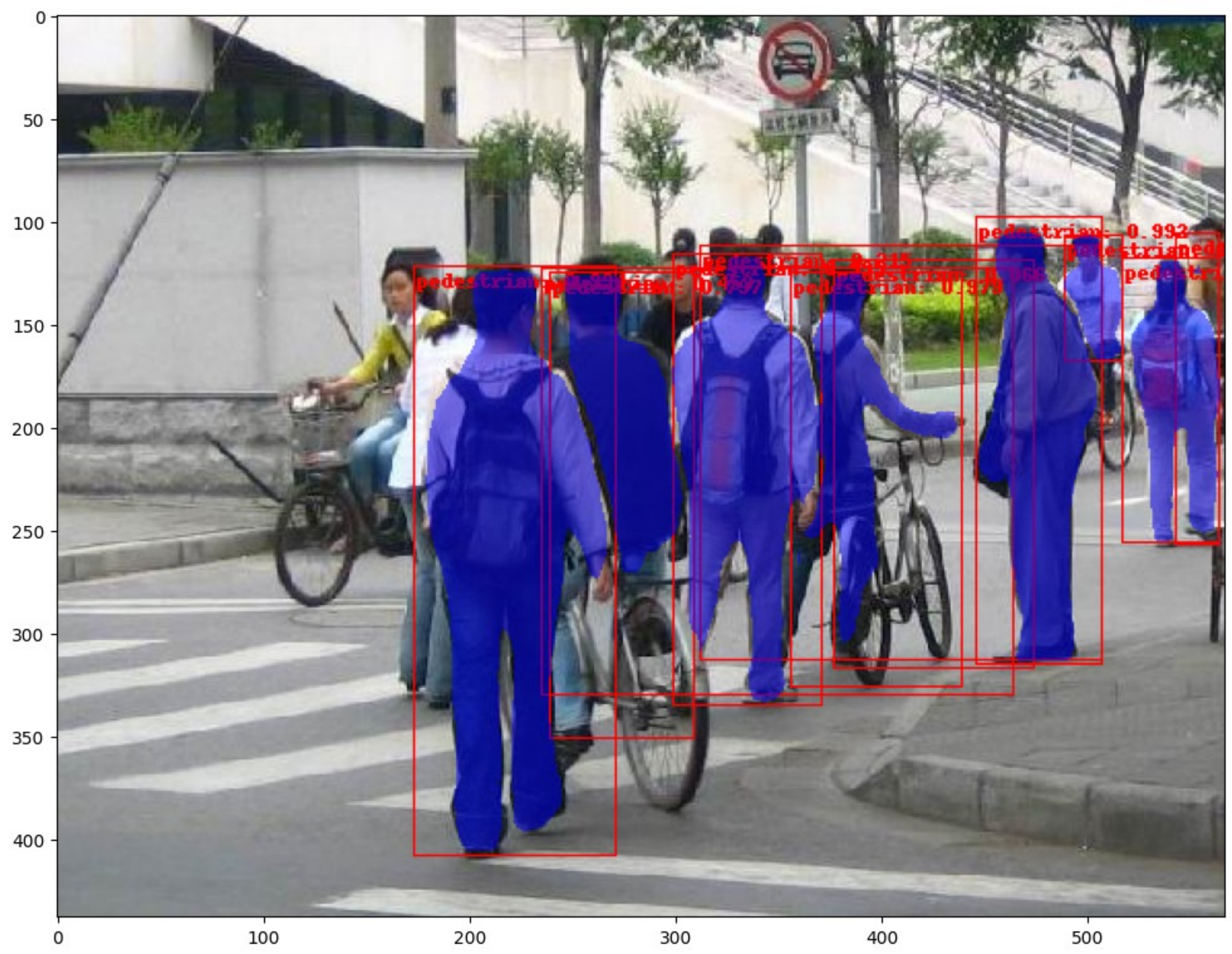
```
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.979
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.931
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.355
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.659
 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 ] = 0.424
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.877
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.877
 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 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.894
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.767
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.918
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.403
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.535
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.779
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.386
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.804
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.806
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.633
 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 ] = 0.812
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 0x7ef574e209a0>
```



```
pred_labels
```

```
['pedestrian: 0.995',
 'pedestrian: 0.994',
 'pedestrian: 0.992',
 'pedestrian: 0.988',
 'pedestrian: 0.979',
 'pedestrian: 0.797',
 'pedestrian: 0.453',
 'pedestrian: 0.215',
 'pedestrian: 0.089',
 'pedestrian: 0.066',
 'pedestrian: 0.055']
```

```
pred_boxes
```

```
tensor([[299, 116, 371, 335],
        [173, 122, 271, 408],
        [446,  98, 507, 315],
        [517, 118, 564, 256],
        [356, 125, 439, 326],
        [239, 125, 309, 351],
        [235, 123, 464, 330],
        [312, 112, 507, 313],
        [489, 107, 517, 168],
        [377, 119, 474, 317],
        [543, 106, 564, 257]], device='cuda:0')
```

The results look good!

Wrapping up

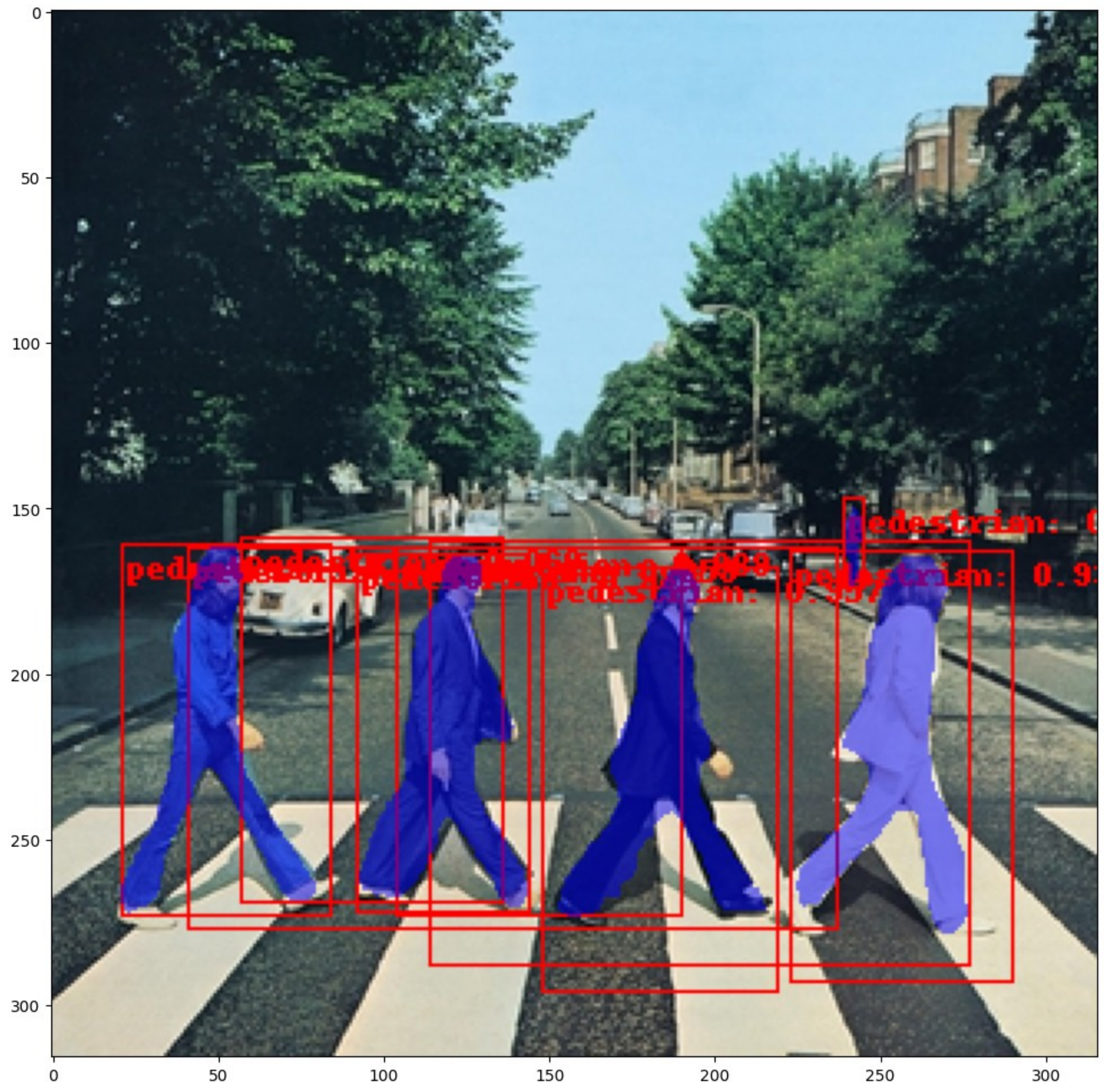
In this tutorial, you have learned how to create your own training pipeline for object detection models on a custom dataset. For that, you wrote a `torch.utils.data.Dataset` class that returns the images and the ground truth boxes and segmentation masks. You also leveraged a Mask R-CNN model pre-trained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

```
image = read_image("drive/MyDrive/Beatles_-_Abbey_Road.jpg")
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 0x7ef4a2a48cd0>
```



```
pred_labels
```

```
['pedestrian: 0.987',
 'pedestrian: 0.985',
 'pedestrian: 0.957',
 'pedestrian: 0.939',
 'pedestrian: 0.103',
 'pedestrian: 0.093',
 'pedestrian: 0.089',
 'pedestrian: 0.060',
 'pedestrian: 0.059']
```

```
pred_boxes
```

```
tensor([[ 21, 161,  84, 273],
        [ 92, 164, 144, 272],
        [148, 168, 219, 296],
        [223, 163, 290, 293],
        [ 41, 162, 237, 277],
        [239, 147, 245, 174],
        [114, 160, 277, 288],
        [ 57, 159, 136, 269],
        [104, 162, 190, 273]], device='cuda:0')
```

This is for Question 5 Part B and C using the backbone option

```
# 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.

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" to /root/.cache/torch/hub/checkpoints/mobilenet_v2-
7ebf99e0.pth
100%|██████████| 13.6M/13.6M [00:00<00:00, 80.2MB/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:
```

```python
        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])
```

```
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 [00:00<00:00, 171MB/s]
/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(0.2291, grad_fn=<NllLossBackward0>),
'loss_box_reg': tensor(0.1100, grad_fn=<DivBackward0>),
'loss_objectness': tensor(0.0078,
grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_rpn_box_reg':
tensor(0.0105, 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',
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 = 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_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, 154MB/s]

Epoch: [0]  [ 0/60]  eta: 0:09:12  lr: 0.000090  loss: 5.0986 (5.0986)
loss_classifier: 0.5976 (0.5976)  loss_box_reg: 0.1450 (0.1450)
loss_mask: 4.3445 (4.3445)  loss_objectness: 0.0105 (0.0105)
loss_rpn_box_reg: 0.0011 (0.0011)  time: 9.2085  data: 1.7603  max
mem: 2065
```

```
Epoch: [0]  [10/60]  eta: 0:01:05  lr: 0.000936  loss: 1.7777 (2.5739)
loss_classifier: 0.4427 (0.4398)  loss_box_reg: 0.2589 (0.2873)
loss_mask: 1.1316 (1.8147)  loss_objectness: 0.0235 (0.0256)
loss_rpn_box_reg: 0.0042 (0.0064)  time: 1.3122  data: 0.1694  max
mem: 2764
Epoch: [0]  [20/60]  eta: 0:00:37  lr: 0.001783  loss: 0.9253 (1.6445)
loss_classifier: 0.1694 (0.2943)  loss_box_reg: 0.2363 (0.2486)
loss_mask: 0.4028 (1.0708)  loss_objectness: 0.0174 (0.0243)
loss_rpn_box_reg: 0.0060 (0.0064)  time: 0.5337  data: 0.0097  max
mem: 3211
Epoch: [0]  [30/60]  eta: 0:00:24  lr: 0.002629  loss: 0.5503 (1.2777)
loss_classifier: 0.0942 (0.2279)  loss_box_reg: 0.2240 (0.2401)
loss_mask: 0.1915 (0.7836)  loss_objectness: 0.0093 (0.0189)
loss_rpn_box_reg: 0.0062 (0.0071)  time: 0.5428  data: 0.0086  max
mem: 3211
Epoch: [0]  [40/60]  eta: 0:00:15  lr: 0.003476  loss: 0.4641 (1.0915)
loss_classifier: 0.0733 (0.1892)  loss_box_reg: 0.2205 (0.2392)
loss_mask: 0.1874 (0.6412)  loss_objectness: 0.0037 (0.0150)
loss_rpn_box_reg: 0.0075 (0.0069)  time: 0.5498  data: 0.0110  max
mem: 3211
Epoch: [0]  [50/60]  eta: 0:00:07  lr: 0.004323  loss: 0.4495 (0.9692)
loss_classifier: 0.0539 (0.1633)  loss_box_reg: 0.2102 (0.2342)
loss_mask: 0.1874 (0.5518)  loss_objectness: 0.0021 (0.0130)
loss_rpn_box_reg: 0.0058 (0.0069)  time: 0.5793  data: 0.0148  max
mem: 3222
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.3703 (0.8745)
loss_classifier: 0.0415 (0.1442)  loss_box_reg: 0.1577 (0.2203)
loss_mask: 0.1652 (0.4921)  loss_objectness: 0.0017 (0.0112)
loss_rpn_box_reg: 0.0058 (0.0067)  time: 0.5651  data: 0.0123  max
mem: 3222
Epoch: [0] Total time: 0:00:41 (0.6973 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:28  model_time: 0.2362 (0.2362)
evaluator_time: 0.0045 (0.0045)  time: 0.5697  data: 0.3278  max mem:
3222
Test:  [49/50]  eta: 0:00:00  model_time: 0.0982 (0.1290)
evaluator_time: 0.0053 (0.0121)  time: 0.1231  data: 0.0049  max mem:
3222
Test: Total time: 0:00:07 (0.1594 s / it)
Averaged stats: model_time: 0.0982 (0.1290)  evaluator_time: 0.0053
(0.0121)
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 ] = 0.584
```

```
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.676
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.267
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.596
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.596
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.266
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.400
 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 ] = 0.668
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.664
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.961
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.798
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.294
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.432
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.688
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.300
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.722
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.723
 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 ] = 0.692
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.732
Epoch: [1]  [ 0/60]  eta: 0:00:52  lr: 0.005000  loss: 0.2154 (0.2154)
loss_classifier: 0.0134 (0.0134)  loss_box_reg: 0.0351 (0.0351)
```

```
loss_mask: 0.1655 (0.1655)  loss_objectness: 0.0000 (0.0000)
loss_rpn_box_reg: 0.0014 (0.0014)  time: 0.8740  data: 0.3722  max
mem: 3222
Epoch: [1]  [10/60]  eta: 0:00:29  lr: 0.005000  loss: 0.2632 (0.3034)
loss_classifier: 0.0393 (0.0371)  loss_box_reg: 0.0935 (0.1025)
loss_mask: 0.1525 (0.1554)  loss_objectness: 0.0010 (0.0015)
loss_rpn_box_reg: 0.0036 (0.0068)  time: 0.5839  data: 0.0398  max
mem: 3222
Epoch: [1]  [20/60]  eta: 0:00:23  lr: 0.005000  loss: 0.2812 (0.3210)
loss_classifier: 0.0393 (0.0425)  loss_box_reg: 0.1028 (0.1159)
loss_mask: 0.1403 (0.1537)  loss_objectness: 0.0010 (0.0017)
loss_rpn_box_reg: 0.0061 (0.0072)  time: 0.5770  data: 0.0092  max
mem: 3222
Epoch: [1]  [30/60]  eta: 0:00:17  lr: 0.005000  loss: 0.2619 (0.2951)
loss_classifier: 0.0371 (0.0390)  loss_box_reg: 0.1001 (0.1037)
loss_mask: 0.1330 (0.1445)  loss_objectness: 0.0008 (0.0017)
loss_rpn_box_reg: 0.0057 (0.0061)  time: 0.5975  data: 0.0097  max
mem: 3499
Epoch: [1]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.2293 (0.2814)
loss_classifier: 0.0254 (0.0362)  loss_box_reg: 0.0697 (0.0951)
loss_mask: 0.1265 (0.1430)  loss_objectness: 0.0004 (0.0014)
loss_rpn_box_reg: 0.0037 (0.0057)  time: 0.5854  data: 0.0087  max
mem: 3499
Epoch: [1]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2533 (0.2769)
loss_classifier: 0.0271 (0.0350)  loss_box_reg: 0.0671 (0.0901)
loss_mask: 0.1385 (0.1447)  loss_objectness: 0.0004 (0.0017)
loss_rpn_box_reg: 0.0040 (0.0055)  time: 0.5632  data: 0.0091  max
mem: 3499
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2533 (0.2748)
loss_classifier: 0.0292 (0.0352)  loss_box_reg: 0.0677 (0.0869)
loss_mask: 0.1429 (0.1455)  loss_objectness: 0.0004 (0.0015)
loss_rpn_box_reg: 0.0044 (0.0058)  time: 0.5583  data: 0.0081  max
mem: 3499
Epoch: [1] Total time: 0:00:35 (0.5834 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:36  model_time: 0.2662 (0.2662)
evaluator_time: 0.0041 (0.0041)  time: 0.7247  data: 0.4532  max mem:
3499
Test:  [49/50]  eta: 0:00:00  model_time: 0.0967 (0.1095)
evaluator_time: 0.0038 (0.0059)  time: 0.1158  data: 0.0042  max mem:
3499
Test: Total time: 0:00:06 (0.1344 s / it)
Averaged stats: model_time: 0.0967 (0.1095)  evaluator_time: 0.0038
(0.0059)
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 ] = 0.738
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.969
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.932
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.381
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.688
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.752
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.330
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.796
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.796
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.806
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.692
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.974
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.826
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.329
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.619
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.706
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.311
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.748
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.748
 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 ] = 0.750
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.753
Epoch: [2]  [ 0/60]  eta: 0:00:53  lr: 0.005000  loss: 0.1752 (0.1752)
loss_classifier: 0.0203 (0.0203)  loss_box_reg: 0.0485 (0.0485)
loss_mask: 0.1022 (0.1022)  loss_objectness: 0.0004 (0.0004)
loss_rpn_box_reg: 0.0038 (0.0038)  time: 0.8881  data: 0.3680  max
mem: 3499
Epoch: [2]  [10/60]  eta: 0:00:29  lr: 0.005000  loss: 0.2351 (0.2578)
loss_classifier: 0.0322 (0.0362)  loss_box_reg: 0.0631 (0.0793)
loss_mask: 0.1351 (0.1367)  loss_objectness: 0.0005 (0.0007)
loss_rpn_box_reg: 0.0038 (0.0048)  time: 0.5899  data: 0.0413  max
mem: 3499
Epoch: [2]  [20/60]  eta: 0:00:23  lr: 0.005000  loss: 0.2351 (0.2564)
loss_classifier: 0.0303 (0.0325)  loss_box_reg: 0.0725 (0.0793)
loss_mask: 0.1351 (0.1394)  loss_objectness: 0.0005 (0.0007)
loss_rpn_box_reg: 0.0036 (0.0046)  time: 0.5669  data: 0.0089  max
mem: 3499
Epoch: [2]  [30/60]  eta: 0:00:17  lr: 0.005000  loss: 0.2122 (0.2387)
loss_classifier: 0.0246 (0.0306)  loss_box_reg: 0.0517 (0.0711)
loss_mask: 0.1209 (0.1320)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0031 (0.0043)  time: 0.5666  data: 0.0091  max
mem: 3499
Epoch: [2]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.2139 (0.2401)
loss_classifier: 0.0334 (0.0315)  loss_box_reg: 0.0517 (0.0713)
loss_mask: 0.1146 (0.1319)  loss_objectness: 0.0004 (0.0007)
loss_rpn_box_reg: 0.0031 (0.0046)  time: 0.5879  data: 0.0099  max
mem: 3506
Epoch: [2]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2241 (0.2373)
loss_classifier: 0.0339 (0.0311)  loss_box_reg: 0.0579 (0.0691)
loss_mask: 0.1267 (0.1320)  loss_objectness: 0.0007 (0.0007)
loss_rpn_box_reg: 0.0034 (0.0044)  time: 0.5892  data: 0.0091  max
mem: 3506
Epoch: [2]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2103 (0.2388)
loss_classifier: 0.0257 (0.0310)  loss_box_reg: 0.0546 (0.0685)
loss_mask: 0.1331 (0.1340)  loss_objectness: 0.0005 (0.0008)
loss_rpn_box_reg: 0.0029 (0.0045)  time: 0.5629  data: 0.0074  max
mem: 3506
Epoch: [2] Total time: 0:00:34 (0.5829 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:27  model_time: 0.1925 (0.1925)
evaluator_time: 0.0037 (0.0037)  time: 0.5428  data: 0.3454  max mem:
3506
Test:  [49/50]  eta: 0:00:00  model_time: 0.0969 (0.1068)
evaluator_time: 0.0032 (0.0049)  time: 0.1137  data: 0.0037  max mem:
3506
Test: Total time: 0:00:06 (0.1284 s / it)
Averaged stats: model_time: 0.0969 (0.1068)  evaluator_time: 0.0032
(0.0049)
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 ] = 0.764
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.970
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.905
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.292
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.696
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.786
 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 ] = 0.820
 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 ] = 0.433
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.758
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.837
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.748
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.973
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.938
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.315
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.567
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.769
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.326
 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 ] = 0.433
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.758
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.805
Epoch: [3]  [ 0/60]  eta: 0:00:55  lr: 0.000500  loss: 0.1499 (0.1499)
loss_classifier: 0.0188 (0.0188)  loss_box_reg: 0.0275 (0.0275)
loss_mask: 0.0995 (0.0995)  loss_objectness: 0.0013 (0.0013)
loss_rpn_box_reg: 0.0028 (0.0028)  time: 0.9178  data: 0.3975  max
mem: 3506
Epoch: [3]  [10/60]  eta: 0:00:30  lr: 0.000500  loss: 0.1933 (0.1984)
loss_classifier: 0.0250 (0.0240)  loss_box_reg: 0.0381 (0.0457)
loss_mask: 0.1201 (0.1236)  loss_objectness: 0.0010 (0.0020)
loss_rpn_box_reg: 0.0023 (0.0030)  time: 0.6011  data: 0.0473  max
mem: 3506
Epoch: [3]  [20/60]  eta: 0:00:23  lr: 0.000500  loss: 0.1933 (0.1941)
loss_classifier: 0.0217 (0.0244)  loss_box_reg: 0.0381 (0.0418)
loss_mask: 0.1204 (0.1229)  loss_objectness: 0.0007 (0.0018)
loss_rpn_box_reg: 0.0023 (0.0033)  time: 0.5586  data: 0.0100  max
mem: 3506
Epoch: [3]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1934 (0.1968)
loss_classifier: 0.0248 (0.0255)  loss_box_reg: 0.0394 (0.0434)
loss_mask: 0.1204 (0.1232)  loss_objectness: 0.0004 (0.0016)
loss_rpn_box_reg: 0.0032 (0.0032)  time: 0.5505  data: 0.0101  max
mem: 3506
Epoch: [3]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1934 (0.1993)
loss_classifier: 0.0257 (0.0261)  loss_box_reg: 0.0436 (0.0445)
loss_mask: 0.1213 (0.1236)  loss_objectness: 0.0006 (0.0017)
loss_rpn_box_reg: 0.0034 (0.0034)  time: 0.5638  data: 0.0108  max
mem: 3506
Epoch: [3]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1763 (0.1969)
loss_classifier: 0.0241 (0.0258)  loss_box_reg: 0.0361 (0.0436)
loss_mask: 0.1137 (0.1223)  loss_objectness: 0.0007 (0.0019)
loss_rpn_box_reg: 0.0025 (0.0032)  time: 0.5697  data: 0.0088  max
mem: 3506
Epoch: [3]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1844 (0.1977)
loss_classifier: 0.0217 (0.0261)  loss_box_reg: 0.0374 (0.0444)
loss_mask: 0.1123 (0.1220)  loss_objectness: 0.0004 (0.0017)
loss_rpn_box_reg: 0.0027 (0.0034)  time: 0.5720  data: 0.0080  max
mem: 3506
Epoch: [3] Total time: 0:00:34 (0.5757 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:26  model_time: 0.2132 (0.2132)
evaluator_time: 0.0038 (0.0038)  time: 0.5280  data: 0.3096  max mem:
3506
Test:  [49/50]  eta: 0:00:00  model_time: 0.1036 (0.1098)
evaluator_time: 0.0050 (0.0060)  time: 0.1220  data: 0.0047  max mem:
3506
Test: Total time: 0:00:06 (0.1367 s / it)
```

```
Averaged stats: model_time: 0.1036 (0.1098)  evaluator_time: 0.0050
(0.0060)
Accumulating evaluation results...
DONE (t=0.03s).
Accumulating evaluation results...
DONE (t=0.03s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.791
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.977
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.923
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.372
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.703
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.812
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.348
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.850
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.850
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.866
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.754
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.977
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.937
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.368
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.523
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.773
 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 ] = 0.803
 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 ] = 0.600
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.812
Epoch: [4]  [ 0/60]  eta: 0:01:20  lr: 0.000500  loss: 0.2263 (0.2263)
loss_classifier: 0.0418 (0.0418)  loss_box_reg: 0.0667 (0.0667)
loss_mask: 0.1081 (0.1081)  loss_objectness: 0.0019 (0.0019)
loss_rpn_box_reg: 0.0078 (0.0078)  time: 1.3454  data: 0.6900  max
mem: 3506
Epoch: [4]  [10/60]  eta: 0:00:31  lr: 0.000500  loss: 0.1923 (0.1878)
loss_classifier: 0.0294 (0.0274)  loss_box_reg: 0.0394 (0.0393)
loss_mask: 0.1081 (0.1172)  loss_objectness: 0.0003 (0.0008)
loss_rpn_box_reg: 0.0016 (0.0031)  time: 0.6336  data: 0.0698  max
mem: 3506
Epoch: [4]  [20/60]  eta: 0:00:24  lr: 0.000500  loss: 0.1878 (0.1919)
loss_classifier: 0.0271 (0.0274)  loss_box_reg: 0.0394 (0.0438)
loss_mask: 0.1124 (0.1166)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0022 (0.0034)  time: 0.5651  data: 0.0086  max
mem: 3506
Epoch: [4]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1777 (0.1855)
loss_classifier: 0.0248 (0.0265)  loss_box_reg: 0.0356 (0.0394)
loss_mask: 0.1126 (0.1156)  loss_objectness: 0.0002 (0.0009)
loss_rpn_box_reg: 0.0017 (0.0030)  time: 0.5693  data: 0.0113  max
mem: 3506
Epoch: [4]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1566 (0.1846)
loss_classifier: 0.0217 (0.0255)  loss_box_reg: 0.0259 (0.0393)
loss_mask: 0.1126 (0.1161)  loss_objectness: 0.0003 (0.0008)
loss_rpn_box_reg: 0.0014 (0.0028)  time: 0.5630  data: 0.0128  max
mem: 3506
Epoch: [4]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1653 (0.1845)
loss_classifier: 0.0224 (0.0254)  loss_box_reg: 0.0278 (0.0396)
loss_mask: 0.1163 (0.1159)  loss_objectness: 0.0004 (0.0008)
loss_rpn_box_reg: 0.0020 (0.0028)  time: 0.5632  data: 0.0133  max
mem: 3506
Epoch: [4]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1707 (0.1862)
loss_classifier: 0.0241 (0.0255)  loss_box_reg: 0.0333 (0.0407)
loss_mask: 0.1126 (0.1164)  loss_objectness: 0.0004 (0.0008)
loss_rpn_box_reg: 0.0029 (0.0030)  time: 0.5827  data: 0.0109  max
mem: 3506
Epoch: [4] Total time: 0:00:35 (0.5868 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:26  model_time: 0.1612 (0.1612)
evaluator_time: 0.0042 (0.0042)  time: 0.5203  data: 0.3537  max mem:
3506
Test:  [49/50]  eta: 0:00:00  model_time: 0.0973 (0.1117)
```

```
evaluator_time: 0.0033 (0.0072)  time: 0.1171  data: 0.0053  max mem:
3506
Test: Total time: 0:00:06 (0.1387 s / it)
Averaged stats: model_time: 0.0973 (0.1117)  evaluator_time: 0.0033
(0.0072)
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 ] = 0.802
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.913
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.335
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.700
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.354
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.849
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.849
 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 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.868
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.758
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.945
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.313
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.527
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.780
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.334
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.802
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.802
 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 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.815
Epoch: [5]  [ 0/60]  eta: 0:01:01  lr: 0.000500  loss: 0.1311 (0.1311)
loss_classifier: 0.0145 (0.0145)  loss_box_reg: 0.0197 (0.0197)
loss_mask: 0.0942 (0.0942)  loss_objectness: 0.0002 (0.0002)
loss_rpn_box_reg: 0.0024 (0.0024)  time: 1.0227  data: 0.4592  max
mem: 3506
Epoch: [5]  [10/60]  eta: 0:00:27  lr: 0.000500  loss: 0.1704 (0.1697)
loss_classifier: 0.0172 (0.0197)  loss_box_reg: 0.0356 (0.0336)
loss_mask: 0.1118 (0.1133)  loss_objectness: 0.0006 (0.0010)
loss_rpn_box_reg: 0.0024 (0.0021)  time: 0.5558  data: 0.0470  max
mem: 3506
Epoch: [5]  [20/60]  eta: 0:00:22  lr: 0.000500  loss: 0.1704 (0.1813)
loss_classifier: 0.0216 (0.0224)  loss_box_reg: 0.0365 (0.0369)
loss_mask: 0.1150 (0.1184)  loss_objectness: 0.0004 (0.0009)
loss_rpn_box_reg: 0.0026 (0.0028)  time: 0.5431  data: 0.0072  max
mem: 3506
Epoch: [5]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1792 (0.1873)
loss_classifier: 0.0254 (0.0234)  loss_box_reg: 0.0371 (0.0392)
loss_mask: 0.1242 (0.1211)  loss_objectness: 0.0003 (0.0008)
loss_rpn_box_reg: 0.0027 (0.0028)  time: 0.5843  data: 0.0089  max
mem: 3506
Epoch: [5]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1743 (0.1817)
loss_classifier: 0.0224 (0.0225)  loss_box_reg: 0.0323 (0.0373)
loss_mask: 0.1098 (0.1183)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0017 (0.0027)  time: 0.5693  data: 0.0092  max
mem: 3506
Epoch: [5]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1669 (0.1854)
loss_classifier: 0.0195 (0.0239)  loss_box_reg: 0.0334 (0.0392)
loss_mask: 0.1054 (0.1183)  loss_objectness: 0.0006 (0.0011)
loss_rpn_box_reg: 0.0024 (0.0029)  time: 0.5706  data: 0.0094  max
mem: 3506
Epoch: [5]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1669 (0.1838)
loss_classifier: 0.0234 (0.0236)  loss_box_reg: 0.0390 (0.0390)
loss_mask: 0.1064 (0.1172)  loss_objectness: 0.0003 (0.0011)
loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.5905  data: 0.0086  max
mem: 3506
Epoch: [5] Total time: 0:00:34 (0.5787 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:32  model_time: 0.1621 (0.1621)
evaluator_time: 0.0040 (0.0040)  time: 0.6579  data: 0.4907  max mem:
```

```
3506
Test:  [49/50]  eta: 0:00:00  model_time: 0.0967 (0.1065)
evaluator_time: 0.0041 (0.0050)  time: 0.1136  data: 0.0037  max mem:
3506
Test: Total time: 0:00:06 (0.1310 s / it)
Averaged stats: model_time: 0.0967 (0.1065)  evaluator_time: 0.0041
(0.0050)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.809
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.973
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.913
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.706
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.831
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.356
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.854
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.854
 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 ] = 0.783
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.872
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.760
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.313
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.562
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.781
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.334
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.802
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.802
 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 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.815
Epoch: [6]  [ 0/60]  eta: 0:00:59  lr: 0.000050  loss: 0.1643 (0.1643)
loss_classifier: 0.0163 (0.0163)  loss_box_reg: 0.0271 (0.0271)
loss_mask: 0.1181 (0.1181)  loss_objectness: 0.0001 (0.0001)
loss_rpn_box_reg: 0.0027 (0.0027)  time: 0.9919  data: 0.4327  max
mem: 3506
Epoch: [6]  [10/60]  eta: 0:00:29  lr: 0.000050  loss: 0.1693 (0.1709)
loss_classifier: 0.0187 (0.0200)  loss_box_reg: 0.0339 (0.0332)
loss_mask: 0.1108 (0.1150)  loss_objectness: 0.0003 (0.0004)
loss_rpn_box_reg: 0.0023 (0.0024)  time: 0.5922  data: 0.0479  max
mem: 3506
Epoch: [6]  [20/60]  eta: 0:00:23  lr: 0.000050  loss: 0.1793 (0.1807)
loss_classifier: 0.0242 (0.0241)  loss_box_reg: 0.0339 (0.0363)
loss_mask: 0.1108 (0.1167)  loss_objectness: 0.0004 (0.0007)
loss_rpn_box_reg: 0.0023 (0.0028)  time: 0.5668  data: 0.0090  max
mem: 3506
Epoch: [6]  [30/60]  eta: 0:00:17  lr: 0.000050  loss: 0.1793 (0.1808)
loss_classifier: 0.0242 (0.0245)  loss_box_reg: 0.0327 (0.0368)
loss_mask: 0.1136 (0.1161)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0025 (0.0028)  time: 0.5922  data: 0.0084  max
mem: 3506
Epoch: [6]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1654 (0.1841)
loss_classifier: 0.0206 (0.0245)  loss_box_reg: 0.0300 (0.0384)
loss_mask: 0.1133 (0.1173)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0025 (0.0030)  time: 0.5777  data: 0.0092  max
mem: 3506
Epoch: [6]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1654 (0.1815)
loss_classifier: 0.0191 (0.0243)  loss_box_reg: 0.0300 (0.0377)
loss_mask: 0.1091 (0.1157)  loss_objectness: 0.0003 (0.0010)
loss_rpn_box_reg: 0.0022 (0.0029)  time: 0.5662  data: 0.0090  max
mem: 3780
Epoch: [6]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1691 (0.1797)
loss_classifier: 0.0206 (0.0238)  loss_box_reg: 0.0285 (0.0370)
loss_mask: 0.1148 (0.1152)  loss_objectness: 0.0004 (0.0009)
loss_rpn_box_reg: 0.0020 (0.0028)  time: 0.5693  data: 0.0076  max
mem: 3780
Epoch: [6] Total time: 0:00:35 (0.5842 s / it)
creating index...
```

```
index created!
Test:  [ 0/50]  eta: 0:00:28  model_time: 0.1869 (0.1869)
evaluator_time: 0.0060 (0.0060)  time: 0.5776  data: 0.3833  max mem:
3780
Test:  [49/50]  eta: 0:00:00  model_time: 0.0973 (0.1077)
evaluator_time: 0.0033 (0.0052)  time: 0.1150  data: 0.0036  max mem:
3780
Test: Total time: 0:00:06 (0.1305 s / it)
Averaged stats: model_time: 0.0973 (0.1077)  evaluator_time: 0.0033
(0.0052)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.808
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.918
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.704
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.831
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.355
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.855
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.855
 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 ] = 0.783
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.873
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.759
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.318
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.528
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.780
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.334
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.802
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.802
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.814
Epoch: [7]  [ 0/60]  eta: 0:01:28  lr: 0.000050  loss: 0.1616 (0.1616)
loss_classifier: 0.0205 (0.0205)  loss_box_reg: 0.0288 (0.0288)
loss_mask: 0.1102 (0.1102)  loss_objectness: 0.0001 (0.0001)
loss_rpn_box_reg: 0.0021 (0.0021)  time: 1.4667  data: 0.6278  max
mem: 3780
Epoch: [7]  [10/60]  eta: 0:00:32  lr: 0.000050  loss: 0.1826 (0.1937)
loss_classifier: 0.0241 (0.0265)  loss_box_reg: 0.0363 (0.0411)
loss_mask: 0.1177 (0.1229)  loss_objectness: 0.0004 (0.0007)
loss_rpn_box_reg: 0.0021 (0.0025)  time: 0.6578  data: 0.0641  max
mem: 3780
Epoch: [7]  [20/60]  eta: 0:00:24  lr: 0.000050  loss: 0.1834 (0.1907)
loss_classifier: 0.0241 (0.0258)  loss_box_reg: 0.0402 (0.0413)
loss_mask: 0.1172 (0.1201)  loss_objectness: 0.0004 (0.0007)
loss_rpn_box_reg: 0.0026 (0.0029)  time: 0.5638  data: 0.0084  max
mem: 3780
Epoch: [7]  [30/60]  eta: 0:00:17  lr: 0.000050  loss: 0.1867 (0.1909)
loss_classifier: 0.0240 (0.0260)  loss_box_reg: 0.0443 (0.0416)
loss_mask: 0.1058 (0.1197)  loss_objectness: 0.0004 (0.0006)
loss_rpn_box_reg: 0.0026 (0.0030)  time: 0.5671  data: 0.0102  max
mem: 3780
Epoch: [7]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1575 (0.1866)
loss_classifier: 0.0200 (0.0247)  loss_box_reg: 0.0340 (0.0401)
loss_mask: 0.1044 (0.1184)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0020 (0.0028)  time: 0.5652  data: 0.0095  max
mem: 3780
Epoch: [7]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1498 (0.1816)
loss_classifier: 0.0164 (0.0240)  loss_box_reg: 0.0250 (0.0378)
loss_mask: 0.1044 (0.1164)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0017 (0.0027)  time: 0.5425  data: 0.0088  max
mem: 3780
Epoch: [7]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1522 (0.1790)
loss_classifier: 0.0212 (0.0239)  loss_box_reg: 0.0261 (0.0367)
loss_mask: 0.1036 (0.1150)  loss_objectness: 0.0002 (0.0006)
loss_rpn_box_reg: 0.0024 (0.0028)  time: 0.5749  data: 0.0091  max
```

```
mem: 3780
Epoch: [7] Total time: 0:00:35 (0.5869 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:25  model_time: 0.1866 (0.1866)
evaluator_time: 0.0039 (0.0039)  time: 0.5075  data: 0.3155  max mem:
3780
Test:  [49/50]  eta: 0:00:00  model_time: 0.1186 (0.1148)
evaluator_time: 0.0055 (0.0068)  time: 0.1294  data: 0.0048  max mem:
3780
Test: Total time: 0:00:06 (0.1398 s / it)
Averaged stats: model_time: 0.1186 (0.1148)  evaluator_time: 0.0055
(0.0068)
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 ] = 0.810
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.700
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.834
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.356
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.856
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.856
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.876
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.954
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.318
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.531
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.782
 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 ] = 0.802
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.802
 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 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.815
Epoch: [8]  [ 0/60]  eta: 0:01:06  lr: 0.000050  loss: 0.1370 (0.1370)
loss_classifier: 0.0163 (0.0163)  loss_box_reg: 0.0187 (0.0187)
loss_mask: 0.0959 (0.0959)  loss_objectness: 0.0004 (0.0004)
loss_rpn_box_reg: 0.0057 (0.0057)  time: 1.1106  data: 0.4501  max
mem: 3780
Epoch: [8]  [10/60]  eta: 0:00:29  lr: 0.000050  loss: 0.1865 (0.2064)
loss_classifier: 0.0221 (0.0261)  loss_box_reg: 0.0353 (0.0489)
loss_mask: 0.1295 (0.1263)  loss_objectness: 0.0002 (0.0008)
loss_rpn_box_reg: 0.0027 (0.0044)  time: 0.5999  data: 0.0469  max
mem: 3780
Epoch: [8]  [20/60]  eta: 0:00:23  lr: 0.000050  loss: 0.1865 (0.1967)
loss_classifier: 0.0244 (0.0263)  loss_box_reg: 0.0408 (0.0450)
loss_mask: 0.1164 (0.1205)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0030 (0.0043)  time: 0.5597  data: 0.0091  max
mem: 3780
Epoch: [8]  [30/60]  eta: 0:00:17  lr: 0.000050  loss: 0.1610 (0.1818)
loss_classifier: 0.0210 (0.0243)  loss_box_reg: 0.0292 (0.0389)
loss_mask: 0.0997 (0.1145)  loss_objectness: 0.0003 (0.0005)
loss_rpn_box_reg: 0.0026 (0.0036)  time: 0.5602  data: 0.0101  max
mem: 3780
Epoch: [8]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1455 (0.1775)
loss_classifier: 0.0183 (0.0239)  loss_box_reg: 0.0238 (0.0373)
loss_mask: 0.0973 (0.1124)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0021 (0.0032)  time: 0.5544  data: 0.0081  max
mem: 3780
Epoch: [8]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1653 (0.1799)
loss_classifier: 0.0229 (0.0248)  loss_box_reg: 0.0294 (0.0371)
loss_mask: 0.1098 (0.1143)  loss_objectness: 0.0003 (0.0007)
loss_rpn_box_reg: 0.0018 (0.0030)  time: 0.5777  data: 0.0088  max
mem: 3780
Epoch: [8]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1663 (0.1786)
```

```
loss_classifier: 0.0229 (0.0245)  loss_box_reg: 0.0291 (0.0367)
loss_mask: 0.1133 (0.1139)  loss_objectness: 0.0002 (0.0007)
loss_rpn_box_reg: 0.0018 (0.0029)  time: 0.5682  data: 0.0086  max
mem: 3780
Epoch: [8] Total time: 0:00:34 (0.5768 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:42  model_time: 0.3319 (0.3319)
evaluator_time: 0.0050 (0.0050)  time: 0.8421  data: 0.5039  max mem:
3780
Test:  [49/50]  eta: 0:00:00  model_time: 0.0978 (0.1148)
evaluator_time: 0.0033 (0.0058)  time: 0.1142  data: 0.0036  max mem:
3780
Test: Total time: 0:00:07 (0.1426 s / it)
Averaged stats: model_time: 0.0978 (0.1148)  evaluator_time: 0.0033
(0.0058)
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 ] = 0.806
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.700
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.354
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.853
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.853
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.872
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.318
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.529
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.782
 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 ] = 0.803
 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 ] = 0.467
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.816
Epoch: [9]  [ 0/60]  eta: 0:00:53  lr: 0.000005  loss: 0.1116 (0.1116)
loss_classifier: 0.0113 (0.0113)  loss_box_reg: 0.0150 (0.0150)
loss_mask: 0.0852 (0.0852)  loss_objectness: 0.0000 (0.0000)
loss_rpn_box_reg: 0.0002 (0.0002)  time: 0.8908  data: 0.3519  max
mem: 3780
Epoch: [9]  [10/60]  eta: 0:00:29  lr: 0.000005  loss: 0.1478 (0.1777)
loss_classifier: 0.0200 (0.0243)  loss_box_reg: 0.0245 (0.0380)
loss_mask: 0.0980 (0.1113)  loss_objectness: 0.0003 (0.0013)
loss_rpn_box_reg: 0.0020 (0.0028)  time: 0.5810  data: 0.0399  max
mem: 3780
Epoch: [9]  [20/60]  eta: 0:00:23  lr: 0.000005  loss: 0.1664 (0.1761)
loss_classifier: 0.0231 (0.0243)  loss_box_reg: 0.0284 (0.0357)
loss_mask: 0.1047 (0.1119)  loss_objectness: 0.0003 (0.0017)
loss_rpn_box_reg: 0.0020 (0.0025)  time: 0.5605  data: 0.0095  max
mem: 3780
Epoch: [9]  [30/60]  eta: 0:00:17  lr: 0.000005  loss: 0.1671 (0.1812)
loss_classifier: 0.0244 (0.0245)  loss_box_reg: 0.0301 (0.0375)
loss_mask: 0.1119 (0.1150)  loss_objectness: 0.0005 (0.0014)
loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.5652  data: 0.0098  max
mem: 3780
Epoch: [9]  [40/60]  eta: 0:00:11  lr: 0.000005  loss: 0.1601 (0.1795)
loss_classifier: 0.0228 (0.0243)  loss_box_reg: 0.0293 (0.0375)
loss_mask: 0.1107 (0.1135)  loss_objectness: 0.0005 (0.0013)
loss_rpn_box_reg: 0.0029 (0.0029)  time: 0.5756  data: 0.0094  max
mem: 3780
Epoch: [9]  [50/60]  eta: 0:00:05  lr: 0.000005  loss: 0.1601 (0.1808)
loss_classifier: 0.0220 (0.0245)  loss_box_reg: 0.0309 (0.0381)
loss_mask: 0.1089 (0.1141)  loss_objectness: 0.0003 (0.0011)
```

```
loss_rpn_box_reg: 0.0029 (0.0029)  time: 0.5913  data: 0.0095  max
mem: 3780
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.1621 (0.1808)
loss_classifier: 0.0219 (0.0247)  loss_box_reg: 0.0326 (0.0377)
loss_mask: 0.1089 (0.1143)  loss_objectness: 0.0002 (0.0011)
loss_rpn_box_reg: 0.0033 (0.0029)  time: 0.5792  data: 0.0088  max
mem: 3780
Epoch: [9] Total time: 0:00:34 (0.5811 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:40  model_time: 0.3076 (0.3076)
evaluator_time: 0.0055 (0.0055)  time: 0.8024  data: 0.4880  max mem:
3780
Test:  [49/50]  eta: 0:00:00  model_time: 0.1000 (0.1120)
evaluator_time: 0.0034 (0.0052)  time: 0.1157  data: 0.0037  max mem:
3780
Test: Total time: 0:00:06 (0.1375 s / it)
Averaged stats: model_time: 0.1000 (0.1120)  evaluator_time: 0.0034
(0.0052)
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 ] = 0.806
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.919
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.348
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.700
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.354
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.853
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.853
 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 ] = 0.775
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.872
IoU metric: segm
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.761
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.313
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.529
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.782
 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 ] = 0.803
 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 ] = 0.467
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.816
That's it!
```

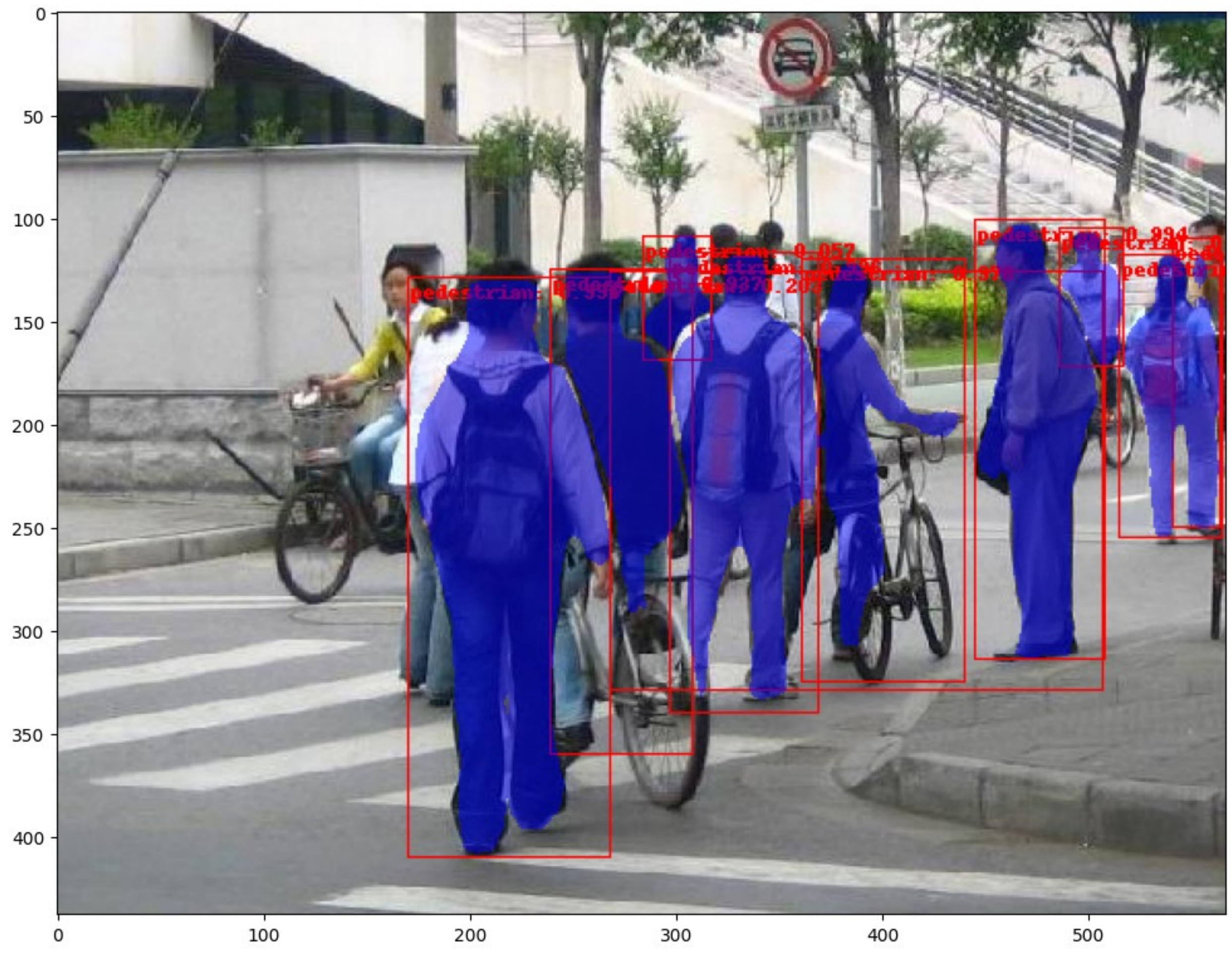
So after one epoch of training, we obtain a COCO-style mAP > 50, and a mask mAP of 65.

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 0x7c62d689c3d0>
```



```
pred_labels
```

```
['pedestrian: 0.996',
 'pedestrian: 0.996',
 'pedestrian: 0.994',
 'pedestrian: 0.987',
 'pedestrian: 0.976',
 'pedestrian: 0.937',
 'pedestrian: 0.203',
 'pedestrian: 0.181',
 'pedestrian: 0.119',
 'pedestrian: 0.057']
```

```
pred_boxes
```

```
tensor([[170, 129, 268, 410],
        [297, 117, 369, 340],
        [445, 101, 508, 314],
        [515, 118, 565, 255],
        [361, 120, 440, 325],
        [239, 125, 308, 360],
        [268, 126, 507, 329],
        [486, 105, 517, 172],
        [541, 110, 564, 250],
        [284, 109, 317, 169]], device='cuda:0')
```

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.

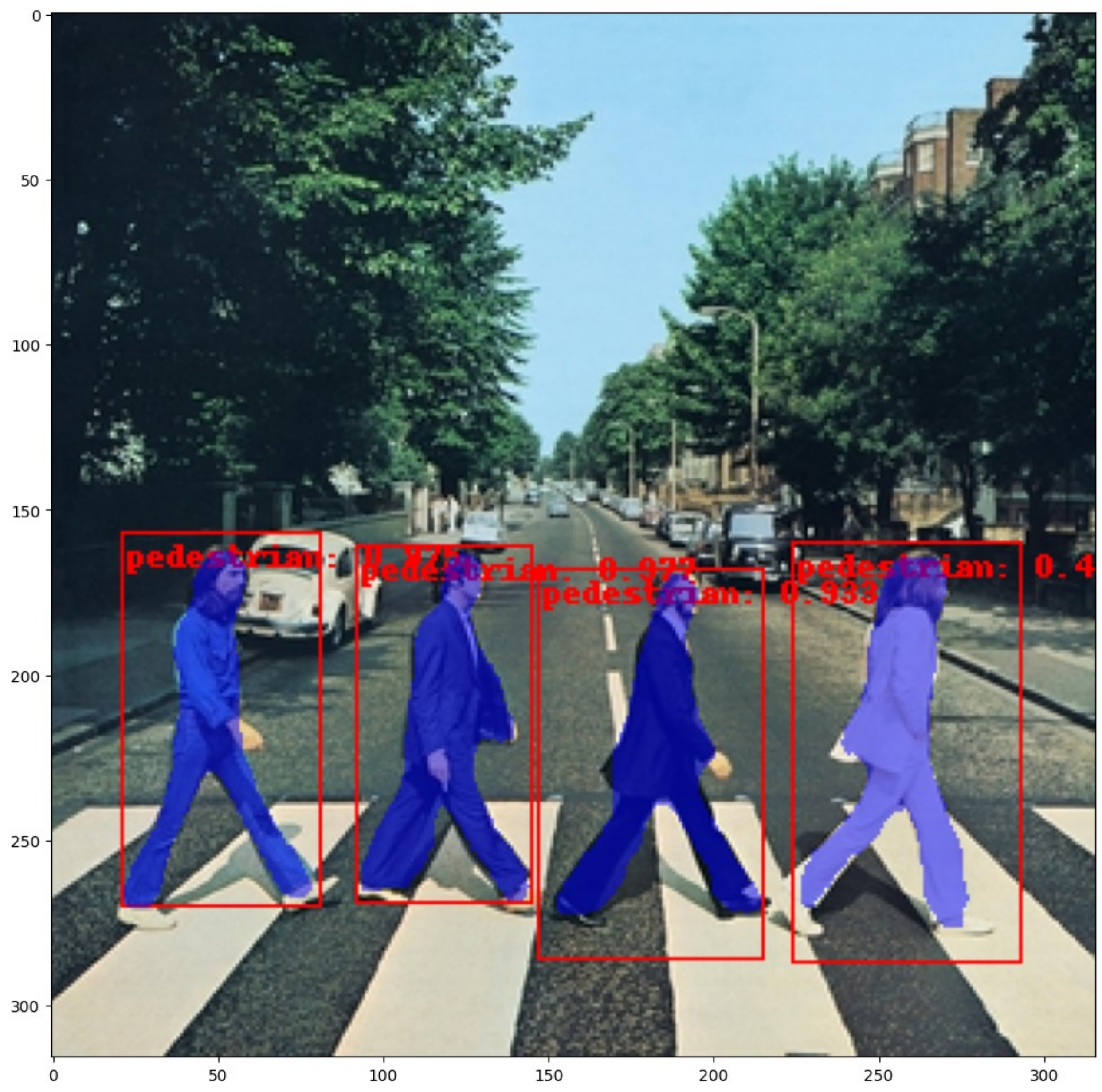
For a more complete example, which includes multi-machine / multi-GPU training, check `references/detection/train.py`, which is present in the torchvision repository.

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

```
image = read_image("drive/MyDrive/Beatles_-_Abbey_Road.jpg")
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", font_size=2)

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))
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/utils.py:223:
UserWarning: Argument 'font_size' will be ignored since 'font' is not
set.
  warnings.warn("Argument 'font_size' will be ignored since 'font' is
not set.")
```

```
<matplotlib.image.AxesImage at 0x7c62d68430a0>
```



```
pred_labels
```

```
['pedestrian: 0.977',
 'pedestrian: 0.975',
 'pedestrian: 0.933',
 'pedestrian: 0.496']
```

```
pred_boxes
```

```
tensor([[ 92, 161, 145, 269],
        [ 21, 157,  81, 270],
        [147, 168, 215, 286],
        [224, 160, 293, 287]], device='cuda:0')
```