CS-GY 6953 / ECE-GY 7123
Deep Learning

Homework #2

Artem Shlepchenko
as14836

Problem 1

Importing necessary library.

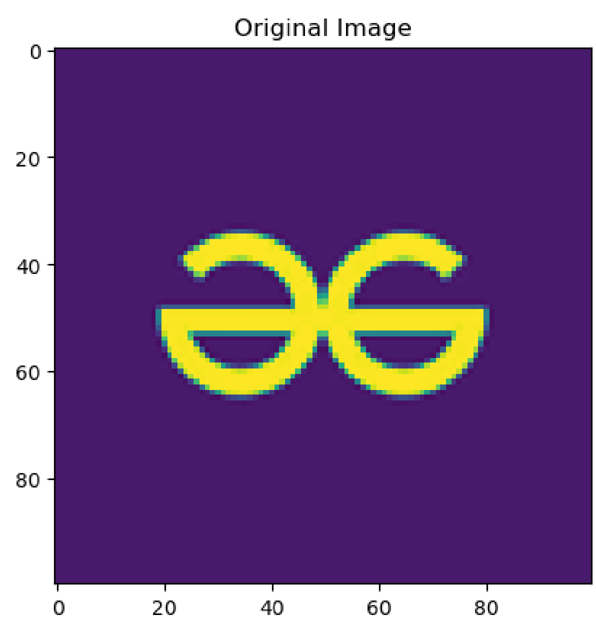
```
from PIL import Image, ImageFilter
import numpy as np
import matplotlib.pyplot as plt
import urllib.request
```

Loading an image(example).

```
urllib.request.urlretrieve(
'https://media.geeksforgeeks.org/wp-content/uploads/20210318103632/gfg-300x300.png',
"gfg.png")
img = np.array(Image.open("gfg.png").convert('L').resize((100, 100)))
# Print the image size
print(img.shape)
```

Displaying the image.

```
plt.figure()
plt.imshow(img)
plt.title("Original Image")
plt.show()
```

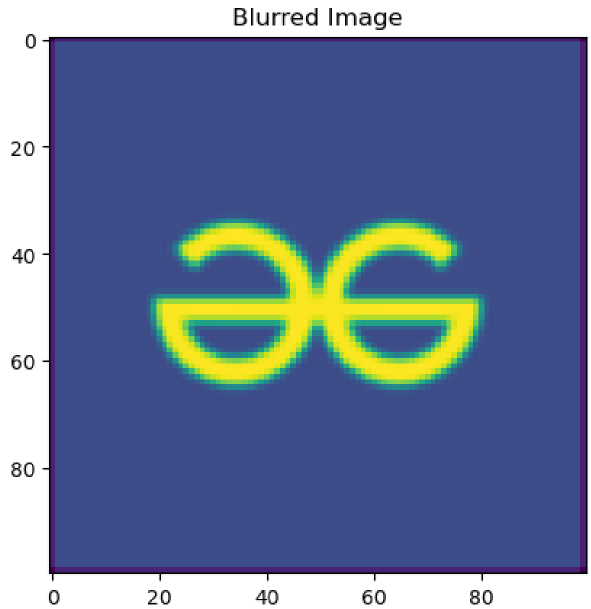


Problem 1 continued on next page. . .

a) Write down the weights of *w* which acts as a blurring filter, i.e., the output is a blurry form in the input.

```
inputSize, padding, filterSize, stride = 100, 1, 3, 1

# This filter works by giving more weight to the central pixel and its immediate
# neighbors, resulting in a stronger blurring effect than the simple averaging
# of a box filter.

# The values in the filter represent the weights or coefficients that are multiplied
# with the pixel values in the corresponding positions to produce the filtered output.
# In this case, the central pixel in the filter has a weight of 4, while the surrounding
# pixels have a weight of 2. This means that the pixel values in the output image are
# more strongly influenced by the nearby pixels than those farther away, resulting in
# a more pronounced blurring effect.
filter = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])

#calculating size of the output image by the convolution formula
targetSize = int(np.floor(((inputSize + 2*padding - filterSize) / stride)) + 1)
output = np.zeros((targetSize, targetSize))

#Padding input image so that size of output image is same as input image.
inputImage = np.pad(img, padding, mode = 'constant')
for i in range(targetSize):
    for j in range(targetSize):
        subset = np.array(inputImage)[i:i+filterSize, j:j+filterSize]
        output[i, j] = np.sum(np.multiply(subset, filter))

plt.figure()
plt.imshow(output)
plt.title("Blurred Image")
plt.show()
```

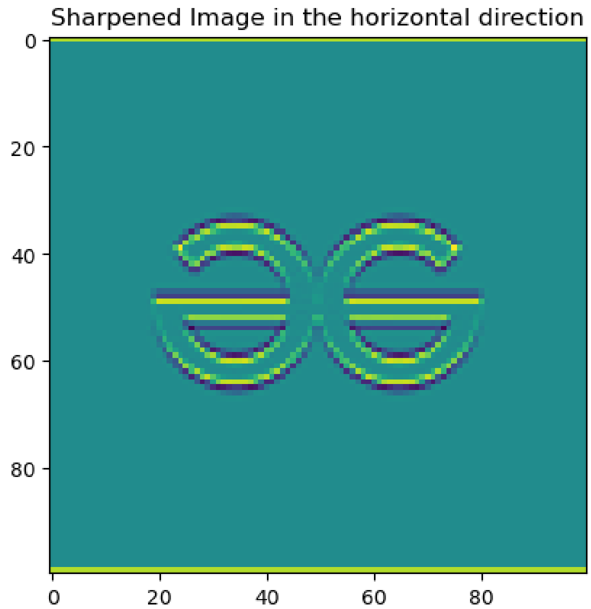
Problem 1 continued on next page. . .



b) Write down the weights of w which acts as a sharpening filter in the horizontal direction.

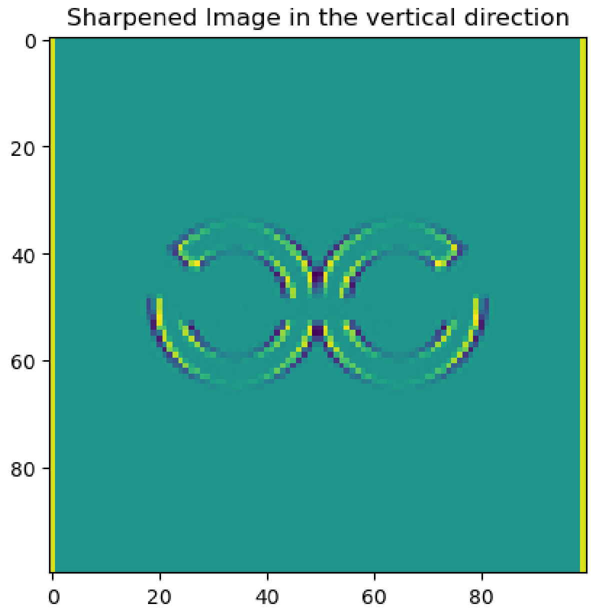
```
inputSize, padding, filterSize, stride = 100, 1, 3, 1

# This filter detects horizontal edges and enhances them, making the image appear sharper.
# The values in the filter represent the weights or coefficients that are multiplied with
# the pixel values in the corresponding positions to produce the filtered output.
# Note that the sum of the values in the filter is 0, which means that the filter is a
# high-pass filter that preserves the overall brightness of the image.
filter = np.array([[0, -1, 0], [0, 2, 0], [0, -1, 0]])

#calculating size of the output image by the convolution formula
targetSize = int(np.floor(((inputSize + 2*padding - filterSize) / stride)) + 1)
output = np.zeros((targetSize, targetSize))

#Padding input image so that size of output image is same as input image.
inputImage = np.pad(img, padding, mode = 'constant')
for i in range(targetSize):
    for j in range(targetSize):
        subset = np.array(inputImage)[i:i+filterSize, j:j+filterSize]
        output[i, j] = np.sum(np.multiply(subset, filter))

plt.figure()
plt.imshow(output)
plt.title("Sharpened Image in the horizontal direction")
plt.show()
```

Problem 1 continued on next page. . .



c) Write down the weights of w which acts as a sharpening filter in the vertical direction.

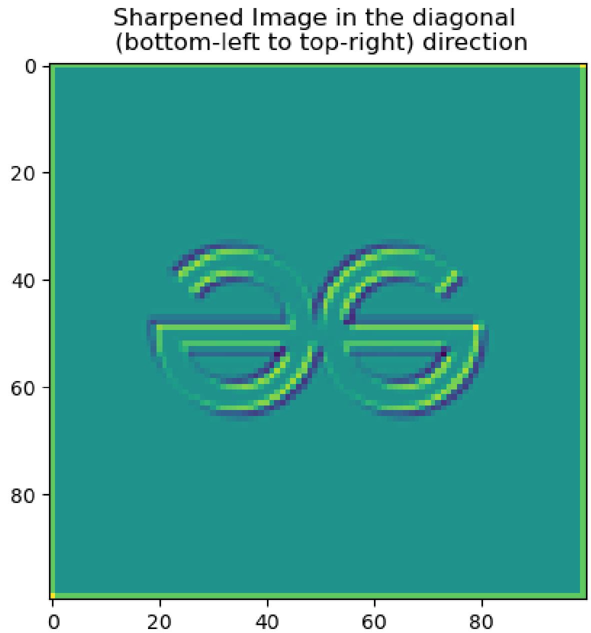
```
inputSize, padding, filterSize, stride = 100, 1, 3, 1

# This filter detects vertical edges and enhances them, making the image appear sharper.
# The values in the filter represent the weights or coefficients that are multiplied
# with the pixel values in the corresponding positions to produce the filtered output.
# Note that the sum of the values in the filter is 0, which means that the filter is
# a high-pass filter that preserves the overall brightness of the image.
filter = np.array([[0, 0, 0], [-1, 2, -1], [0, 0, 0]])

#calculating size of the output image by the convolution formula
targetSize = int(np.floor(((inputSize + 2*padding - filterSize) / stride)) + 1)
output = np.zeros((targetSize, targetSize))

#Padding input image so that size of output image is same as input image.
inputImage = np.pad(img, padding, mode = 'constant')
for i in range(targetSize):
    for j in range(targetSize):
        subset = np.array(inputImage)[i:i+filterSize, j:j+filterSize]
        output[i, j] = np.sum(np.multiply(subset, filter))

plt.figure()
plt.imshow(output)
plt.title("Sharpened Image in the vertical direction")
plt.show()
```



d) Write down the weights of w which act as a sharpening filter in a diagonal (bottom-left to top-right) direction.

```
inputSize, padding, filterSize, stride = 100, 1, 3, 1

# This filter detects diagonal edges in the bottom-left to top-right direction and
# enhances them, making the image appear sharper. The values in the filter represent
# the weights or coefficients that are multiplied with the pixel values in the
# corresponding positions to produce the filtered output.

# Note that the sum of the values in the filter is 0, which means that the filter
# is a high-pass filter that preserves the overall brightness of the image.
filter = np.array([[-1, 0, 0], [0, 2, 0], [0, 0, -1]])

#calculating size of the output image by the convolution formula
targetSize = int(np.floor(((inputSize + 2*padding - filterSize) / stride)) + 1)
output = np.zeros((targetSize, targetSize))

#Padding input image so that size of output image is same as input image.
inputImage = np.pad(img, padding, mode = 'constant')
for i in range(targetSize):
    for j in range(targetSize):
        subset = np.array(inputImage)[i:i+filterSize, j:j+filterSize]
        output[i, j] = np.sum(np.multiply(subset, filter))

plt.figure()
plt.imshow(output)
```

```
plt.title("Sharpened Image in the diagonal \n (bottom-left to top-right) direction")
plt.show()
```



e) Give an example of an image operation which cannot be implemented using a 3x3 convolutional filter and briefly explain why.

One example of an image operation that cannot be implemented using a 3x3 convolutional filter is the dilation operation. Dilation is a morphological image processing operation that expands the boundaries of the object in an image.

In dilation, a structuring element is placed on each pixel of the image, and the pixel values are modified based on the presence of neighboring pixels within the structuring element. This operation requires a structuring element larger than 3x3 and hence cannot be implemented using a 3x3 convolutional filter.

The dilation operation can be implemented using other morphological operations, such as erosion and opening, which can be implemented using convolutional filters. But dilation operation cannot be implemented using only a 3x3 convolutional filter.

Problem 2

a)
The ℓ_2 regularised loss would be

$$L_2(w) = L(w) + \lambda(||w||_2)$$

b)

$$\frac{\partial L_2(w)}{\partial w} = \frac{\partial L(w)}{\partial w} + \frac{2\lambda w}{(||w||_2)}$$

$$W = W - \frac{\partial L_2(w)}{\partial w} = w - \left(\frac{\partial L(w)}{\partial w} + \frac{2\lambda w}{(||w||_2)}\right) = w\left(1 - \frac{2\lambda}{(||w||_2)}\right) - \frac{\partial L(w)}{\partial w}$$

c)
The expression above shows that weights are shrunk before applying the descent update.

d)
λ should be higher initially so that the weights converge faster, but once we got close enough, they should change slowly so that the weights stabilise instead of jumping and diverging.

So, we should choose λ such that it should adapt to the distance from the final solution.

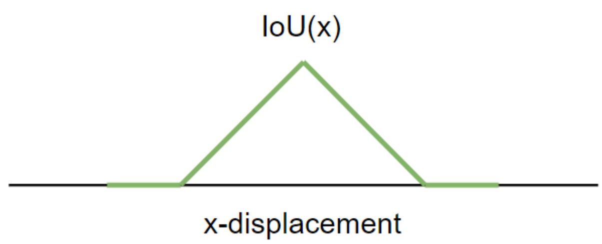
Problem 3

a)
The definition IoU for any two bounding boxes is given by:

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Since the right hand side is non-negative, this number has to be bigger (or equal to) 0.
Moreover, $A \cap B \subseteq A \cup B$, and hence the numerator has to be no bigger than the denominator. Therefore, the IoU metric is bounded between 0 and 1 (inclusive).

b)
Let's take two square boxes A and B with identical sizes, both aligned at the same horizontal level. Then, fix B and imagine "sliding" A from left to right. As A moves, the IoU will start from zero (no overlap), increase (until there is perfect overlap), and then decrease (until there is no overlap again). So, if we plot IoU as a function of horizontal displacement, we should get a curve like this:

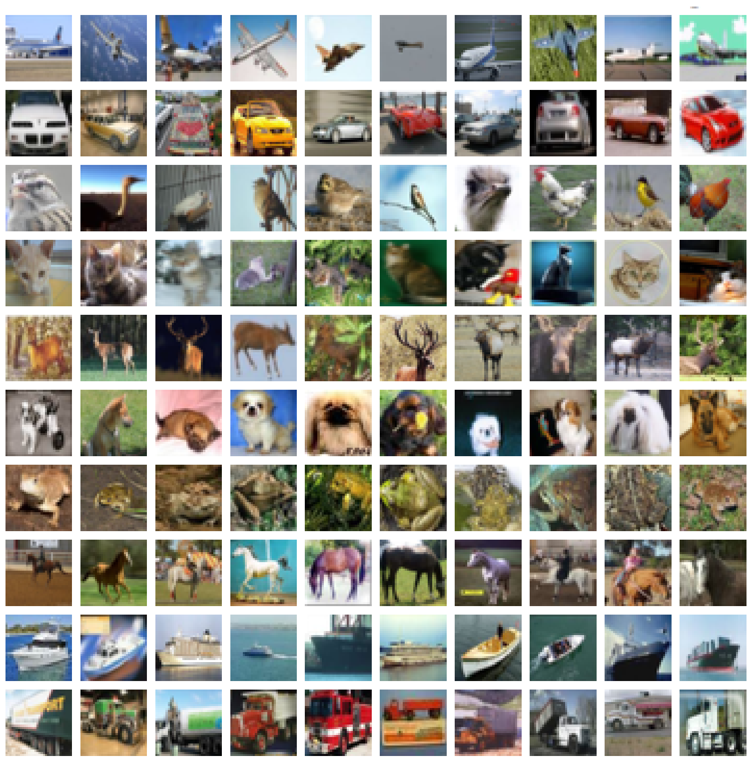


which has 3 kinks and hence is non-differentiable.

Problem 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.

```
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)
```

Problem 4 continued on next page. . .

```
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.

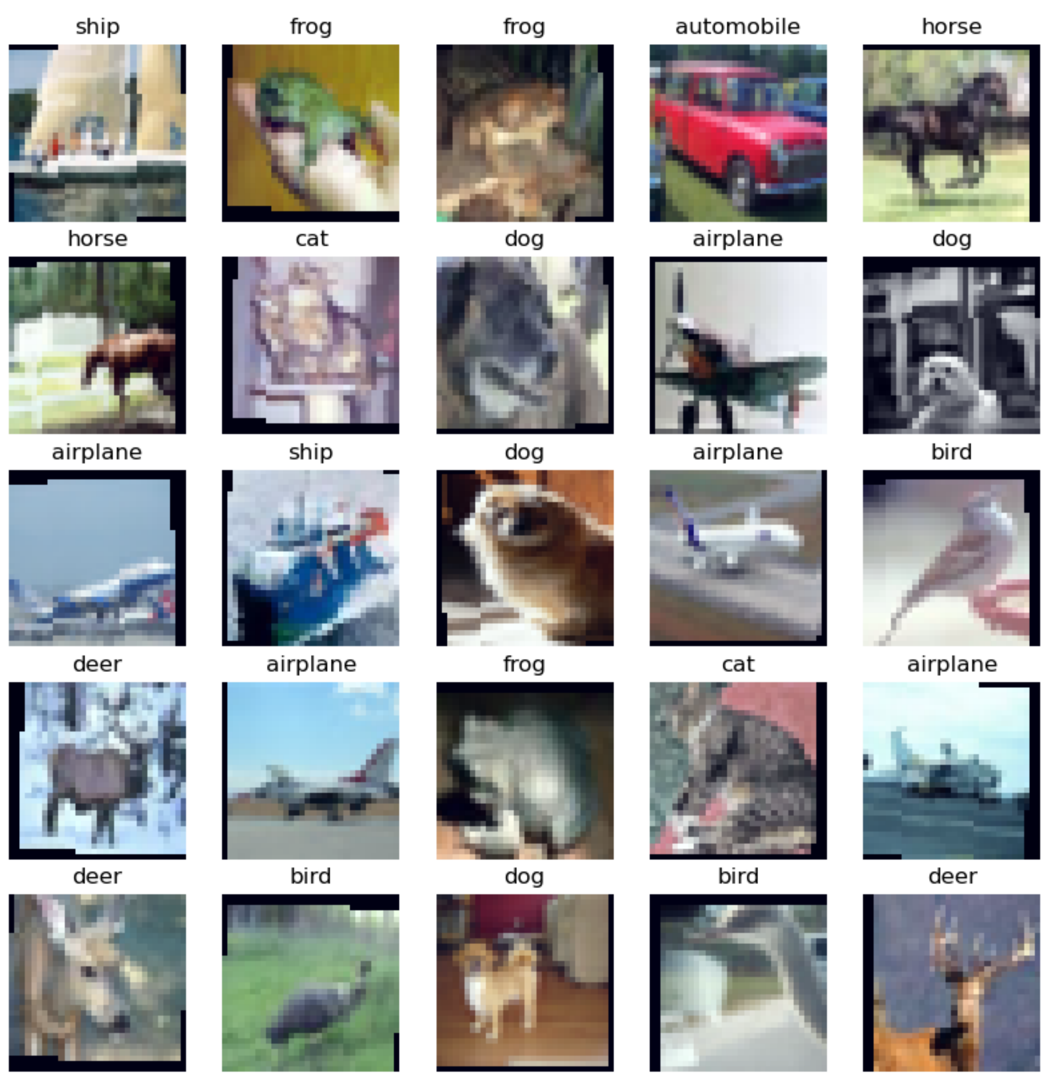
Note that by default PyTorch handles images that are arranged `[channel, height, width]`, but matplotlib expects images to be `[height, width, channel]`, hence we need to permute the dimensions of our images before plotting them.

```
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 = data.DataLoader(train_data, shuffle=True, batch_size=BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data, batch_size=BATCH_SIZE)

test_iterator = data.DataLoader(test_data, batch_size=BATCH_SIZE)
```

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→64→192→384→256→256

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

  1024→4096→4096→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
            # Conv Layer #1
            nn.Conv2d(in_channels=3,out_channels=64,kernel_size=3,stride=2,padding=1),
            nn.MaxPool2d(kernel_size=2),
            nn.ReLU(inplace=True),
            # Conv Layer #2
            nn.Conv2d(in_channels=64,out_channels=192,kernel_size=3,stride=1,padding=1),
            nn.MaxPool2d(kernel_size=2),
            nn.ReLU(inplace=True),
            # Conv Layer #3
            nn.Conv2d(in_channels=192,out_channels=384,kernel_size=3,stride=1,padding=1),
            nn.ReLU(inplace=True),
            # Conv Layer #4
            nn.Conv2d(in_channels=384,out_channels=256,kernel_size=3,stride=1,padding=1),
            nn.ReLU(inplace=True),
            # Conv Layer #5
            nn.Conv2d(in_channels=256,out_channels=256,kernel_size=3,stride=1,padding=1),
            nn.MaxPool2d(kernel_size=2),
            nn.ReLU(inplace=True)
        )

        self.classifier = nn.Sequential(
            # define according to the steps described above
            # Linear Layer #1
            nn.Dropout(p=0.5),
            nn.Linear(in_features=1024, out_features=4096),
            nn.ReLU(inplace=True),
```

Problem 4 continued on next page. . .

```
            # Linear Layer #2
            nn.Dropout(p=0.5),
            nn.Linear(in_features=4096, out_features=4096),
            nn.ReLU(inplace=True),
            # Linear Layer #3
            nn.Linear(in_features=4096, out_features=output_dim)
        )

    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)
```

Problem 4 continued on next page. . .

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
                   ceil_mode=False)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1),
             padding=(1, 1))
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
                   ceil_mode=False)
    (5): ReLU(inplace=True)
    (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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
                    ceil_mode=False)
    (12): ReLU(inplace=True)
  )
  (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()
```

Problem 4 continued on next page. . .

```
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()`.

Problem 4 continued on next page. . .

```
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.
```

```python
EPOCHS = 25

# Fill training code here
best_val_loss = float('inf')

for epoch in range(EPOCHS):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, train_iterator, optimizer,criterion, device)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion, device)

    if valid_loss < best_val_loss:
        best_val_loss = valid_loss
        torch.save(model.state_dict(), 'best_model.pt')

    end_time = time.monotonic()

    epoch_mins, epoch_sec = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_sec}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 1m 45s
	Train Loss: 2.384 | Train Acc: 21.65%
	 Val. Loss: 1.618 |  Val. Acc: 38.25%
Epoch: 02 | Epoch Time: 1m 50s
	Train Loss: 1.541 | Train Acc: 43.03%
	 Val. Loss: 1.374 |  Val. Acc: 49.61%
Epoch: 03 | Epoch Time: 1m 51s
	Train Loss: 1.353 | Train Acc: 50.88%
	 Val. Loss: 1.197 |  Val. Acc: 56.91%
Epoch: 04 | Epoch Time: 1m 49s
	Train Loss: 1.263 | Train Acc: 54.77%
	 Val. Loss: 1.144 |  Val. Acc: 59.15%
Epoch: 05 | Epoch Time: 1m 51s
	Train Loss: 1.180 | Train Acc: 58.10%
	 Val. Loss: 1.094 |  Val. Acc: 60.06%
Epoch: 06 | Epoch Time: 1m 53s
	Train Loss: 1.117 | Train Acc: 60.31%
	 Val. Loss: 1.059 |  Val. Acc: 62.04%
```

Problem 4 continued on next page. . .

```
Epoch: 07 | Epoch Time: 1m 52s
     Train Loss: 1.073 | Train Acc: 62.12%
      Val. Loss: 1.050 |  Val. Acc: 64.64%
Epoch: 08 | Epoch Time: 1m 56s
     Train Loss: 1.021 | Train Acc: 64.08%
      Val. Loss: 0.970 |  Val. Acc: 65.37%
Epoch: 09 | Epoch Time: 1m 57s
     Train Loss: 0.978 | Train Acc: 65.95%
      Val. Loss: 0.908 |  Val. Acc: 67.94%
Epoch: 10 | Epoch Time: 2m 0s
     Train Loss: 0.938 | Train Acc: 67.19%
      Val. Loss: 0.918 |  Val. Acc: 68.45%
Epoch: 11 | Epoch Time: 1m 59s
     Train Loss: 0.900 | Train Acc: 68.30%
      Val. Loss: 0.882 |  Val. Acc: 69.49%
Epoch: 12 | Epoch Time: 1m 58s
     Train Loss: 0.890 | Train Acc: 68.96%
      Val. Loss: 0.868 |  Val. Acc: 70.55%
Epoch: 13 | Epoch Time: 2m 2s
     Train Loss: 0.847 | Train Acc: 70.55%
      Val. Loss: 0.866 |  Val. Acc: 70.22%
Epoch: 14 | Epoch Time: 2m 0s
     Train Loss: 0.826 | Train Acc: 71.15%
      Val. Loss: 0.816 |  Val. Acc: 71.85%
Epoch: 15 | Epoch Time: 1m 59s
     Train Loss: 0.802 | Train Acc: 71.91%
      Val. Loss: 0.801 |  Val. Acc: 72.59%
Epoch: 16 | Epoch Time: 2m 0s
     Train Loss: 0.774 | Train Acc: 73.09%
      Val. Loss: 0.819 |  Val. Acc: 71.74%
Epoch: 17 | Epoch Time: 2m 2s
     Train Loss: 0.765 | Train Acc: 73.60%
      Val. Loss: 0.791 |  Val. Acc: 73.27%
Epoch: 18 | Epoch Time: 2m 2s
     Train Loss: 0.756 | Train Acc: 74.04%
      Val. Loss: 0.782 |  Val. Acc: 73.12%
Epoch: 19 | Epoch Time: 2m 3s
     Train Loss: 0.726 | Train Acc: 74.90%
      Val. Loss: 0.778 |  Val. Acc: 73.49%
Epoch: 20 | Epoch Time: 2m 2s
     Train Loss: 0.710 | Train Acc: 75.32%
```

Problem 4 continued on next page. . .

```
      Val. Loss: 0.737 |  Val. Acc: 74.64%
Epoch: 21 | Epoch Time: 2m 3s
     Train Loss: 0.693 | Train Acc: 76.35%
      Val. Loss: 0.750 |  Val. Acc: 75.41%
Epoch: 22 | Epoch Time: 2m 4s
     Train Loss: 0.683 | Train Acc: 76.37%
      Val. Loss: 0.736 |  Val. Acc: 75.45%
Epoch: 23 | Epoch Time: 2m 1s
     Train Loss: 0.674 | Train Acc: 76.82%
      Val. Loss: 0.726 |  Val. Acc: 75.70%
Epoch: 24 | Epoch Time: 2m 0s
     Train Loss: 0.658 | Train Acc: 77.40%
      Val. Loss: 0.752 |  Val. Acc: 74.87%
Epoch: 25 | Epoch Time: 2m 3s
     Train Loss: 0.651 | Train Acc: 77.71%
      Val. Loss: 0.779 |  Val. Acc: 73.01%
```

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.

model.load_state_dict(torch.load('best_model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion, device)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
# Also, print out the confusion matrox.
```

```
Test Loss: 0.719 | Test Acc: 75.04%
```

```
def get_predictions(model, iterator, device):

    model.eval()

    labels = []
    probs = []

    # Q4: Fill code here.
    with torch.no_grad():

        for (x, y) in iterator:
            x = x.to(device)
            y_pred, _ = model(x)
            y_prob = F.softmax(y_pred, dim=1)

            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    labels = torch.cat(labels, dim = 0)
    probs = torch.cat(probs, dim = 0)

    return labels, probs
```

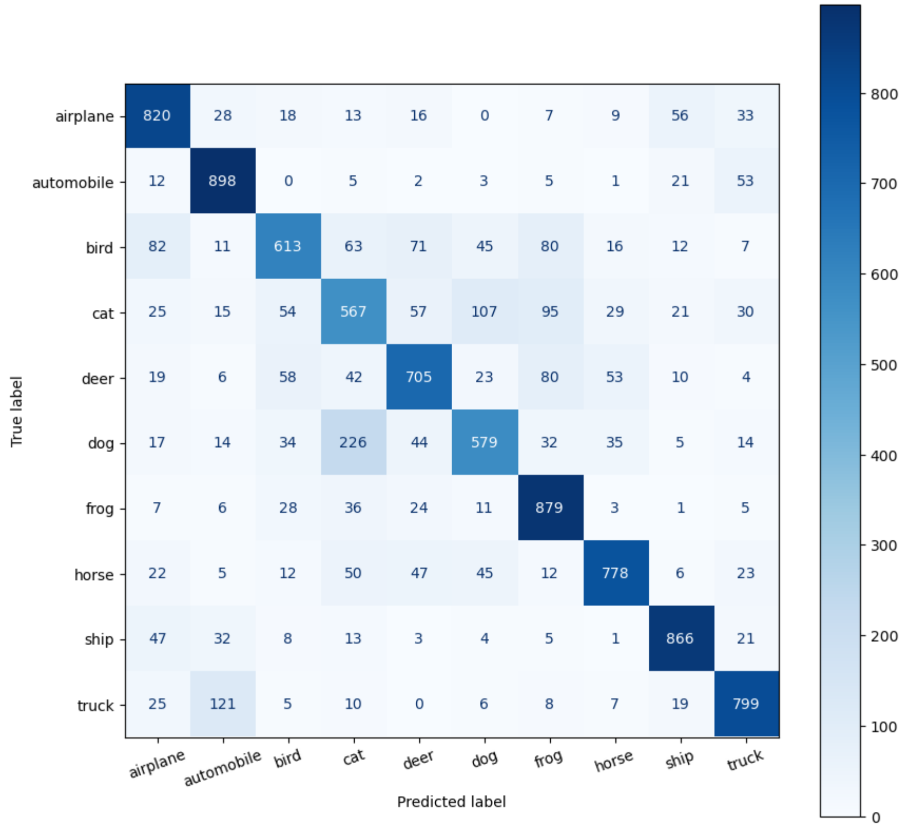
```
labels, probs = get_predictions(model, test_iterator, device)
```

```
pred_labels = torch.argmax(probs, 1)
```

```
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)
```

Problem 4 continued on next page. . .



From the figure above we can see that the most confused classes are cat and dog.

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.)

Problem 5

A lot of code is taken and modified from (`https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html`) as asked in the question.

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

```
%%shell

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

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

```
%%shell

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

Defining the Dataset

The dataset should inherit from the standard `torch.utils.data.Dataset` class, and implement `__len__` and `__getitem__`.
The only specificity that we require is that the dataset `__getitem__` should return:

- image: a PIL Image of size (H, W)
- target: a dict containing the following fields
  - `boxes` (`FloatTensor[N, 4]`): the coordinates of the `N` bounding boxes in `[x0, y0, x1, y1]` format, ranging from `0` to `W` and `0` to `H`
  - `labels` (`Int64Tensor[N]`): the label for each bounding box
  - `image_id` (`Int64Tensor[1]`): an image identifier. It should be unique between all the images in the dataset, and is used during evaluation

- `area` (`Tensor[N]`): The area of the bounding box. This is used during evaluation with the COCO metric, to separate the metric scores between small, medium and large boxes.
- `iscrowd` (`UInt8Tensor[N]`): instances with `iscrowd=True` will be ignored during evaluation.
- (optionally) `masks` (`UInt8Tensor[N, H, W]`): The segmentation masks for each one of the objects.
- (optionally) `keypoints` (`FloatTensor[N, K, 3]`): For each one of the `N` objects, it contains the `K` keypoints in `[x, y, visibility]` format, defining the object. `visibility=0` means that the keypoint is not visible. Note that for data augmentation, the notion of flipping a keypoint is dependent on the data representation, and you should probably adapt `references/detection/transforms.py` for your new keypoint representation

Writing a custom dataset for Penn-Fudan

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

```
import os
import numpy as np
import torch
import torch.utils.data
from PIL import Image

class PennFudanDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms=None):
        self.root = root
        self.transforms = transforms
        # load all image files, sorting them to
        # ensure that they are aligned
        self.imgs = list(sorted(os.listdir(os.path.join(root, "PNGImages"))))
        self.masks = list(sorted(os.listdir(os.path.join(root, "PedMasks"))))

    def __getitem__(self, idx):
        # load images ad masks
        img_path = os.path.join(self.root, "PNGImages", self.imgs[idx])
        mask_path = os.path.join(self.root, "PedMasks", self.masks[idx])
        img = Image.open(img_path).convert("RGB")
        # note that we haven't converted the mask to RGB,
        # because each color corresponds to a different instance
        # with 0 being background
        mask = Image.open(mask_path)

        mask = np.array(mask)
        # instances are encoded as different colors
```

Problem 5 continued on next page. . .

```
        obj_ids = np.unique(mask)
        # first id is the background, so remove it
        obj_ids = obj_ids[1:]

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

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

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

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

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

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

        return img, target

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

Problem 5 continued on next page. . .

Instance Segmentation Model with ResNet Backbone (Option 1)

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor
import torchvision
from torchvision.models.detection import FasterRCNN, MaskRCNN
from torchvision.models.detection.rpn import AnchorGenerator
import torch.nn as nn

def get_instance_segmentation_model_1(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)

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

Instance Segmentation Model with MobileNet Backbone (Option 2)

```
def get_instance_segmentation_model_2(num_classes):


    backbone = torchvision.models.mobilenet_v2(pretrained=True).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)

    mask_roi_pooler = torchvision.ops.MultiScaleRoIAlign(
        featmap_names=['0'], output_size=14, sampling_ratio=2)

    # put the pieces together inside a FasterRCNN model
    model = MaskRCNN(backbone,
                     num_classes=num_classes,
                     rpn_anchor_generator=anchor_generator,
                     box_roi_pool=roi_pooler,
                    mask_roi_pooler=mask_roi_pooler)

    return model
```

Training and evaluation functions

In `references/detection/`, we have a number of helper functions to simplify training and evaluating detection models. Here, we will use `references/detection/engine.py`, `references/detection/utils.py` and `references/detection/transforms.py`.

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

```
%%shell

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

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

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

```python
from engine import train_one_epoch, evaluate
import utils
import transforms as T


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

Putting everything together

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

```python
# use our dataset and defined transformations
dataset = PennFudanDataset('PennFudanPed', get_transform(train=True))
dataset_test = PennFudanDataset('PennFudanPed', get_transform(train=False))

# split the dataset in train and test set
torch.manual_seed(1)
indices = torch.randperm(len(dataset)).tolist()
dataset = torch.utils.data.Subset(dataset, indices[:-50])
dataset_test = torch.utils.data.Subset(dataset_test, indices[-50:])

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

Problem 5 continued on next page. . .

```
    collate_fn=utils.collate_fn)

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

Now let's instantiate the model (Option 1: ResNet) and the optimizer

```
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

# get the model using our helper function
model_1 = get_instance_segmentation_model_1(num_classes)

# move model to the right device
model_1.to(device)


# construct an optimizer
params = [p for p in model_1.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005,
                            momentum=0.9, weight_decay=0.0005)

# and a learning rate scheduler which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

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

```
# let's train it for 10 epochs
from torch.optim.lr_scheduler import StepLR
num_epochs = 10

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model_1, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
```

```
Epoch: [0]  [ 0/60]  eta: 0:07:17  lr: 0.000090  loss: 2.7899 (2.7899)  loss_classifier: 0.7401 (0.7401)
    loss_box_reg: 0.3405 (0.3405)  loss_mask: 1.6637 (1.6637)  loss_objectness: 0.0430 (0.0430)
    loss_rpn_box_reg: 0.0025 (0.0025)  time: 7.2903  data: 0.3273  max mem: 2162
Epoch: [0]  [10/60]  eta: 0:00:59  lr: 0.000936  loss: 1.3949 (1.7270)  loss_classifier: 0.5158 (0.4826)
    loss_box_reg: 0.2960 (0.2981)  loss_mask: 0.7157 (0.9198)  loss_objectness: 0.0169 (0.0217)
    loss_rpn_box_reg: 0.0045 (0.0048)  time: 1.1989  data: 0.0434  max mem: 3320
Epoch: [0]  [20/60]  eta: 0:00:35  lr: 0.001783  loss: 1.0076 (1.2311)  loss_classifier: 0.2244 (0.3358)
    loss_box_reg: 0.2910 (0.2865)  loss_mask: 0.3239 (0.5872)  loss_objectness: 0.0109 (0.0171)
    loss_rpn_box_reg: 0.0042 (0.0045)  time: 0.5689  data: 0.0180  max mem: 3320
Epoch: [0]  [30/60]  eta: 0:00:23  lr: 0.002629  loss: 0.5587 (1.0162)  loss_classifier: 0.0983 (0.2554)
    loss_box_reg: 0.2665 (0.2870)  loss_mask: 0.1833 (0.4537)  loss_objectness: 0.0102 (0.0151)
    loss_rpn_box_reg: 0.0045 (0.0050)  time: 0.5637  data: 0.0178  max mem: 3320
Epoch: [0]  [40/60]  eta: 0:00:14  lr: 0.003476  loss: 0.4513 (0.8823)  loss_classifier: 0.0606 (0.2065)
    loss_box_reg: 0.2153 (0.2675)  loss_mask: 0.1689 (0.3901)  loss_objectness: 0.0083 (0.0128)
    loss_rpn_box_reg: 0.0056 (0.0053)  time: 0.5675  data: 0.0150  max mem: 3320
Epoch: [0]  [50/60]  eta: 0:00:06  lr: 0.004323  loss: 0.3895 (0.7843)  loss_classifier: 0.0418 (0.1747)
    loss_box_reg: 0.1625 (0.2453)  loss_mask: 0.1720 (0.3475)  loss_objectness: 0.0035 (0.0109)
    loss_rpn_box_reg: 0.0049 (0.0058)  time: 0.5375  data: 0.0131  max mem: 3320
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.3133 (0.7138)  loss_classifier: 0.0366 (0.1543)
    loss_box_reg: 0.1154 (0.2265)  loss_mask: 0.1490 (0.3173)  loss_objectness: 0.0013 (0.0097)
    loss_rpn_box_reg: 0.0059 (0.0060)  time: 0.5332  data: 0.0106  max mem: 3320
Epoch: [0] Total time: 0:00:40 (0.6691 s / it)
Epoch: [1]  [ 0/60]  eta: 0:01:07  lr: 0.005000  loss: 0.3843 (0.3843)  loss_classifier: 0.0526 (0.0526)
    loss_box_reg: 0.1679 (0.1679)  loss_mask: 0.1554 (0.1554)  loss_objectness: 0.0003 (0.0003)
    loss_rpn_box_reg: 0.0081 (0.0081)  time: 1.1258  data: 0.5763  max mem: 3320
Epoch: [1]  [10/60]  eta: 0:00:29  lr: 0.005000  loss: 0.2921 (0.2999)  loss_classifier: 0.0359 (0.0398)
    loss_box_reg: 0.1132 (0.0989)  loss_mask: 0.1438 (0.1530)  loss_objectness: 0.0010 (0.0020)
    loss_rpn_box_reg: 0.0058 (0.0063)  time: 0.5822  data: 0.0619  max mem: 3320
Epoch: [1]  [20/60]  eta: 0:00:23  lr: 0.005000  loss: 0.2389 (0.2900)  loss_classifier: 0.0358 (0.0390)
    loss_box_reg: 0.0766 (0.0950)  loss_mask: 0.1369 (0.1479)  loss_objectness: 0.0012 (0.0025)
    loss_rpn_box_reg: 0.0046 (0.0056)  time: 0.5504  data: 0.0108  max mem: 3320
```

Problem 5 continued on next page. . .

```
Epoch: [1]  [30/60]  eta: 0:00:17  lr: 0.005000  loss: 0.3070 (0.3076)  loss_classifier: 0.0380 (0.0421)
    loss_box_reg: 0.0953 (0.1044)  loss_mask: 0.1385 (0.1524)  loss_objectness: 0.0013 (0.0026)
    loss_rpn_box_reg: 0.0046 (0.0061)  time: 0.5820  data: 0.0124  max mem: 3320
Epoch: [1]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.3070 (0.3003)  loss_classifier: 0.0380 (0.0408)
    loss_box_reg: 0.0794 (0.1003)  loss_mask: 0.1420 (0.1510)  loss_objectness: 0.0006 (0.0024)
    loss_rpn_box_reg: 0.0057 (0.0058)  time: 0.5662  data: 0.0120  max mem: 3320
Epoch: [1]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2550 (0.2947)  loss_classifier: 0.0331 (0.0395)
    loss_box_reg: 0.0709 (0.0949)  loss_mask: 0.1420 (0.1527)  loss_objectness: 0.0006 (0.0023)
    loss_rpn_box_reg: 0.0037 (0.0053)  time: 0.5447  data: 0.0123  max mem: 3320
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.3055 (0.2972)  loss_classifier: 0.0358 (0.0396)
    loss_box_reg: 0.0753 (0.0948)  loss_mask: 0.1549 (0.1547)  loss_objectness: 0.0011 (0.0025)
    loss_rpn_box_reg: 0.0047 (0.0056)  time: 0.5559  data: 0.0119  max mem: 3320
Epoch: [1] Total time: 0:00:34 (0.5708 s / it)
Epoch: [2]  [ 0/60]  eta: 0:00:51  lr: 0.005000  loss: 0.3595 (0.3595)  loss_classifier: 0.0293 (0.0293)
    loss_box_reg: 0.0862 (0.0862)  loss_mask: 0.2358 (0.2358)  loss_objectness: 0.0010 (0.0010)
    loss_rpn_box_reg: 0.0072 (0.0072)  time: 0.8561  data: 0.3386  max mem: 3320
Epoch: [2]  [10/60]  eta: 0:00:30  lr: 0.005000  loss: 0.2915 (0.2671)  loss_classifier: 0.0353 (0.0352)
    loss_box_reg: 0.0756 (0.0725)  loss_mask: 0.1503 (0.1528)  loss_objectness: 0.0010 (0.0017)
    loss_rpn_box_reg: 0.0046 (0.0048)  time: 0.6186  data: 0.0416  max mem: 3320
Epoch: [2]  [20/60]  eta: 0:00:24  lr: 0.005000  loss: 0.2858 (0.2811)  loss_classifier: 0.0414 (0.0433)
    loss_box_reg: 0.0756 (0.0840)  loss_mask: 0.1367 (0.1469)  loss_objectness: 0.0011 (0.0016)
    loss_rpn_box_reg: 0.0048 (0.0053)  time: 0.5967  data: 0.0115  max mem: 3320
Epoch: [2]  [30/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2372 (0.2560)  loss_classifier: 0.0373 (0.0380)
    loss_box_reg: 0.0632 (0.0735)  loss_mask: 0.1367 (0.1385)  loss_objectness: 0.0005 (0.0013)
    loss_rpn_box_reg: 0.0033 (0.0046)  time: 0.5927  data: 0.0133  max mem: 3320
Epoch: [2]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.1995 (0.2480)  loss_classifier: 0.0249 (0.0365)
    loss_box_reg: 0.0459 (0.0708)  loss_mask: 0.1199 (0.1352)  loss_objectness: 0.0003 (0.0012)
    loss_rpn_box_reg: 0.0022 (0.0042)  time: 0.5795  data: 0.0126  max mem: 3320
Epoch: [2]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2178 (0.2453)  loss_classifier: 0.0324 (0.0357)
    loss_box_reg: 0.0540 (0.0697)  loss_mask: 0.1232 (0.1345)  loss_objectness: 0.0006 (0.0013)
    loss_rpn_box_reg: 0.0026 (0.0041)  time: 0.5692  data: 0.0115  max mem: 3320
```

```
Epoch: [2]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2120 (0.2412)  loss_classifier: 0.0290 (0.0345)
    loss_box_reg: 0.0531 (0.0675)  loss_mask: 0.1282 (0.1341)  loss_objectness: 0.0004 (0.0013)
    loss_rpn_box_reg: 0.0022 (0.0039)  time: 0.5510  data: 0.0111  max mem: 3320
Epoch: [2] Total time: 0:00:34 (0.5825 s / it)
Epoch: [3]  [ 0/60]  eta: 0:00:54  lr: 0.000500  loss: 0.2435 (0.2435)  loss_classifier: 0.0499 (0.0499)
    loss_box_reg: 0.0648 (0.0648)  loss_mask: 0.1255 (0.1255)  loss_objectness: 0.0023 (0.0023)
    loss_rpn_box_reg: 0.0010 (0.0010)  time: 0.9023  data: 0.3088  max mem: 3320
Epoch: [3]  [10/60]  eta: 0:00:31  lr: 0.000500  loss: 0.2206 (0.2460)  loss_classifier: 0.0300 (0.0339)
    loss_box_reg: 0.0597 (0.0639)  loss_mask: 0.1291 (0.1426)  loss_objectness: 0.0007 (0.0012)
    loss_rpn_box_reg: 0.0043 (0.0045)  time: 0.6361  data: 0.0380  max mem: 3320
Epoch: [3]  [20/60]  eta: 0:00:24  lr: 0.000500  loss: 0.2158 (0.2326)  loss_classifier: 0.0297 (0.0328)
    loss_box_reg: 0.0597 (0.0616)  loss_mask: 0.1202 (0.1333)  loss_objectness: 0.0007 (0.0011)
    loss_rpn_box_reg: 0.0038 (0.0039)  time: 0.5939  data: 0.0110  max mem: 3320
Epoch: [3]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1909 (0.2232)  loss_classifier: 0.0293 (0.0305)
    loss_box_reg: 0.0534 (0.0570)  loss_mask: 0.1128 (0.1306)  loss_objectness: 0.0005 (0.0013)
    loss_rpn_box_reg: 0.0026 (0.0037)  time: 0.5725  data: 0.0108  max mem: 3320
Epoch: [3]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1956 (0.2158)  loss_classifier: 0.0233 (0.0294)
    loss_box_reg: 0.0470 (0.0546)  loss_mask: 0.1174 (0.1271)  loss_objectness: 0.0004 (0.0011)
    loss_rpn_box_reg: 0.0029 (0.0036)  time: 0.5741  data: 0.0106  max mem: 3320
Epoch: [3]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1823 (0.2084)  loss_classifier: 0.0233 (0.0292)
    loss_box_reg: 0.0374 (0.0512)  loss_mask: 0.1132 (0.1235)  loss_objectness: 0.0003 (0.0012)
    loss_rpn_box_reg: 0.0023 (0.0034)  time: 0.5779  data: 0.0101  max mem: 3320
Epoch: [3]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1814 (0.2100)  loss_classifier: 0.0235 (0.0300)
    loss_box_reg: 0.0334 (0.0509)  loss_mask: 0.1119 (0.1246)  loss_objectness: 0.0003 (0.0011)
    loss_rpn_box_reg: 0.0020 (0.0034)  time: 0.5755  data: 0.0103  max mem: 3320
Epoch: [3] Total time: 0:00:35 (0.5885 s / it)
Epoch: [4]  [ 0/60]  eta: 0:00:50  lr: 0.000500  loss: 0.1429 (0.1429)  loss_classifier: 0.0145 (0.0145)
    loss_box_reg: 0.0155 (0.0155)  loss_mask: 0.1108 (0.1108)  loss_objectness: 0.0003 (0.0003)
    loss_rpn_box_reg: 0.0018 (0.0018)  time: 0.8337  data: 0.3439  max mem: 3320
Epoch: [4]  [10/60]  eta: 0:00:29  lr: 0.000500  loss: 0.1837 (0.1830)  loss_classifier: 0.0249 (0.0255)
    loss_box_reg: 0.0379 (0.0411)  loss_mask: 0.1135 (0.1128)  loss_objectness: 0.0005 (0.0013)
```

```
    loss_rpn_box_reg: 0.0016 (0.0022)  time: 0.5834  data: 0.0396  max mem: 3320
Epoch: [4]  [20/60]  eta: 0:00:23  lr: 0.000500  loss: 0.1867 (0.2062)  loss_classifier: 0.0298 (0.0327)
    loss_box_reg: 0.0379 (0.0467)  loss_mask: 0.1149 (0.1224)  loss_objectness: 0.0005 (0.0014)
    loss_rpn_box_reg: 0.0016 (0.0031)  time: 0.5725  data: 0.0106  max mem: 3320
Epoch: [4]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1867 (0.2043)  loss_classifier: 0.0358 (0.0322)
    loss_box_reg: 0.0426 (0.0460)  loss_mask: 0.1203 (0.1215)  loss_objectness: 0.0004 (0.0012)
    loss_rpn_box_reg: 0.0028 (0.0034)  time: 0.5962  data: 0.0112  max mem: 3320
Epoch: [4]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1772 (0.1971)  loss_classifier: 0.0227 (0.0300)
    loss_box_reg: 0.0349 (0.0438)  loss_mask: 0.1148 (0.1191)  loss_objectness: 0.0004 (0.0011)
    loss_rpn_box_reg: 0.0020 (0.0032)  time: 0.5807  data: 0.0121  max mem: 3320
Epoch: [4]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1575 (0.1912)  loss_classifier: 0.0187 (0.0286)
    loss_box_reg: 0.0285 (0.0419)  loss_mask: 0.1104 (0.1167)  loss_objectness: 0.0002 (0.0010)
    loss_rpn_box_reg: 0.0019 (0.0030)  time: 0.5492  data: 0.0130  max mem: 3320
Epoch: [4]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1653 (0.1873)  loss_classifier: 0.0196 (0.0279)
    loss_box_reg: 0.0296 (0.0404)  loss_mask: 0.1097 (0.1152)  loss_objectness: 0.0002 (0.0010)
    loss_rpn_box_reg: 0.0020 (0.0029)  time: 0.5691  data: 0.0112  max mem: 3320
Epoch: [4] Total time: 0:00:34 (0.5804 s / it)
Epoch: [5]  [ 0/60]  eta: 0:00:57  lr: 0.000500  loss: 0.1310 (0.1310)  loss_classifier: 0.0131 (0.0131)
    loss_box_reg: 0.0213 (0.0213)  loss_mask: 0.0950 (0.0950)  loss_objectness: 0.0001 (0.0001)
    loss_rpn_box_reg: 0.0016 (0.0016)  time: 0.9573  data: 0.3209  max mem: 3320
Epoch: [5]  [10/60]  eta: 0:00:28  lr: 0.000500  loss: 0.1633 (0.1947)  loss_classifier: 0.0207 (0.0277)
    loss_box_reg: 0.0325 (0.0452)  loss_mask: 0.1095 (0.1180)  loss_objectness: 0.0005 (0.0016)
    loss_rpn_box_reg: 0.0016 (0.0021)  time: 0.5658  data: 0.0379  max mem: 3320
Epoch: [5]  [20/60]  eta: 0:00:23  lr: 0.000500  loss: 0.1681 (0.1876)  loss_classifier: 0.0210 (0.0264)
    loss_box_reg: 0.0325 (0.0424)  loss_mask: 0.1095 (0.1154)  loss_objectness: 0.0005 (0.0011)
    loss_rpn_box_reg: 0.0018 (0.0022)  time: 0.5605  data: 0.0108  max mem: 3502
Epoch: [5]  [30/60]  eta: 0:00:17  lr: 0.000500  loss: 0.1723 (0.1936)  loss_classifier: 0.0235 (0.0273)
    loss_box_reg: 0.0349 (0.0441)  loss_mask: 0.1086 (0.1184)  loss_objectness: 0.0004 (0.0010)
    loss_rpn_box_reg: 0.0025 (0.0029)  time: 0.5931  data: 0.0115  max mem: 3502
Epoch: [5]  [40/60]  eta: 0:00:11  lr: 0.000500  loss: 0.1674 (0.1893)  loss_classifier: 0.0235 (0.0265)
    loss_box_reg: 0.0320 (0.0421)  loss_mask: 0.1080 (0.1171)  loss_objectness: 0.0004 (0.0008)
```

Problem 5 continued on next page. . .

```
    loss_rpn_box_reg: 0.0026 (0.0028)  time: 0.5918  data: 0.0121  max mem: 3502
Epoch: [5]  [50/60]  eta: 0:00:05  lr: 0.000500  loss: 0.1656 (0.1929)  loss_classifier: 0.0194 (0.0274)
    loss_box_reg: 0.0254 (0.0436)  loss_mask: 0.1074 (0.1181)  loss_objectness: 0.0002 (0.0011)
    loss_rpn_box_reg: 0.0019 (0.0028)  time: 0.5911  data: 0.0120  max mem: 3502
Epoch: [5]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.1656 (0.1924)  loss_classifier: 0.0194 (0.0274)
    loss_box_reg: 0.0321 (0.0433)  loss_mask: 0.1074 (0.1178)  loss_objectness: 0.0003 (0.0010)
    loss_rpn_box_reg: 0.0023 (0.0029)  time: 0.5882  data: 0.0108  max mem: 3502
Epoch: [5] Total time: 0:00:35 (0.5912 s / it)
Epoch: [6]  [ 0/60]  eta: 0:01:13  lr: 0.000050  loss: 0.1655 (0.1655)  loss_classifier: 0.0326 (0.0326)
    loss_box_reg: 0.0281 (0.0281)  loss_mask: 0.0993 (0.0993)  loss_objectness: 0.0043 (0.0043)
    loss_rpn_box_reg: 0.0012 (0.0012)  time: 1.2214  data: 0.6604  max mem: 3502
Epoch: [6]  [10/60]  eta: 0:00:32  lr: 0.000050  loss: 0.2183 (0.2191)  loss_classifier: 0.0304 (0.0331)
    loss_box_reg: 0.0422 (0.0537)  loss_mask: 0.1325 (0.1283)  loss_objectness: 0.0008 (0.0013)
    loss_rpn_box_reg: 0.0031 (0.0027)  time: 0.6473  data: 0.0730  max mem: 3502
Epoch: [6]  [20/60]  eta: 0:00:24  lr: 0.000050  loss: 0.1709 (0.1974)  loss_classifier: 0.0263 (0.0298)
    loss_box_reg: 0.0350 (0.0455)  loss_mask: 0.1086 (0.1180)  loss_objectness: 0.0005 (0.0015)
    loss_rpn_box_reg: 0.0026 (0.0026)  time: 0.5931  data: 0.0127  max mem: 3502
Epoch: [6]  [30/60]  eta: 0:00:18  lr: 0.000050  loss: 0.1611 (0.1851)  loss_classifier: 0.0207 (0.0269)
    loss_box_reg: 0.0265 (0.0403)  loss_mask: 0.1022 (0.1145)  loss_objectness: 0.0002 (0.0011)
    loss_rpn_box_reg: 0.0018 (0.0023)  time: 0.5922  data: 0.0111  max mem: 3502
Epoch: [6]  [40/60]  eta: 0:00:12  lr: 0.000050  loss: 0.1757 (0.1900)  loss_classifier: 0.0236 (0.0274)
    loss_box_reg: 0.0359 (0.0424)  loss_mask: 0.1055 (0.1166)  loss_objectness: 0.0004 (0.0011)
    loss_rpn_box_reg: 0.0019 (0.0025)  time: 0.6014  data: 0.0122  max mem: 3502
Epoch: [6]  [50/60]  eta: 0:00:06  lr: 0.000050  loss: 0.1819 (0.1885)  loss_classifier: 0.0230 (0.0268)
    loss_box_reg: 0.0383 (0.0413)  loss_mask: 0.1144 (0.1165)  loss_objectness: 0.0004 (0.0013)
    loss_rpn_box_reg: 0.0026 (0.0026)  time: 0.5908  data: 0.0145  max mem: 3502
Epoch: [6]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1558 (0.1880)  loss_classifier: 0.0156 (0.0268)
    loss_box_reg: 0.0255 (0.0408)  loss_mask: 0.1106 (0.1168)  loss_objectness: 0.0004 (0.0012)
    loss_rpn_box_reg: 0.0021 (0.0025)  time: 0.5713  data: 0.0134  max mem: 3502
Epoch: [6] Total time: 0:00:35 (0.5997 s / it)
Epoch: [7]  [ 0/60]  eta: 0:00:57  lr: 0.000050  loss: 0.1362 (0.1362)  loss_classifier: 0.0213 (0.0213)
```

Problem 5 continued on next page. . .

```
    loss_box_reg: 0.0217 (0.0217)  loss_mask: 0.0909 (0.0909)  loss_objectness: 0.0009 (0.0009)
    loss_rpn_box_reg: 0.0014 (0.0014)  time: 0.9515  data: 0.3533  max mem: 3502
Epoch: [7]  [10/60]  eta: 0:00:29  lr: 0.000050  loss: 0.1661 (0.1890)  loss_classifier: 0.0232 (0.0290)
    loss_box_reg: 0.0328 (0.0385)  loss_mask: 0.1108 (0.1188)  loss_objectness: 0.0003 (0.0004)
    loss_rpn_box_reg: 0.0021 (0.0023)  time: 0.5984  data: 0.0443  max mem: 3502
Epoch: [7]  [20/60]  eta: 0:00:23  lr: 0.000050  loss: 0.1684 (0.1928)  loss_classifier: 0.0232 (0.0273)
    loss_box_reg: 0.0328 (0.0403)  loss_mask: 0.1180 (0.1222)  loss_objectness: 0.0002 (0.0003)
    loss_rpn_box_reg: 0.0021 (0.0027)  time: 0.5674  data: 0.0118  max mem: 3502
Epoch: [7]  [30/60]  eta: 0:00:17  lr: 0.000050  loss: 0.1791 (0.1899)  loss_classifier: 0.0228 (0.0278)
    loss_box_reg: 0.0301 (0.0408)  loss_mask: 0.1161 (0.1181)  loss_objectness: 0.0002 (0.0006)
    loss_rpn_box_reg: 0.0019 (0.0026)  time: 0.5950  data: 0.0120  max mem: 3502
Epoch: [7]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1677 (0.1861)  loss_classifier: 0.0231 (0.0268)
    loss_box_reg: 0.0338 (0.0395)  loss_mask: 0.1100 (0.1165)  loss_objectness: 0.0003 (0.0007)
    loss_rpn_box_reg: 0.0026 (0.0026)  time: 0.6054  data: 0.0122  max mem: 3502
Epoch: [7]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1652 (0.1857)  loss_classifier: 0.0235 (0.0266)
    loss_box_reg: 0.0341 (0.0396)  loss_mask: 0.1062 (0.1161)  loss_objectness: 0.0003 (0.0008)
    loss_rpn_box_reg: 0.0026 (0.0027)  time: 0.5934  data: 0.0117  max mem: 3502
Epoch: [7]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1715 (0.1867)  loss_classifier: 0.0270 (0.0267)
    loss_box_reg: 0.0329 (0.0398)  loss_mask: 0.1116 (0.1167)  loss_objectness: 0.0002 (0.0007)
    loss_rpn_box_reg: 0.0019 (0.0026)  time: 0.5712  data: 0.0110  max mem: 3502
Epoch: [7] Total time: 0:00:35 (0.5880 s / it)
Epoch: [8]  [ 0/60]  eta: 0:00:59  lr: 0.000050  loss: 0.2865 (0.2865)  loss_classifier: 0.0580 (0.0580)
    loss_box_reg: 0.0720 (0.0720)  loss_mask: 0.1505 (0.1505)  loss_objectness: 0.0012 (0.0012)
    loss_rpn_box_reg: 0.0049 (0.0049)  time: 0.9995  data: 0.3814  max mem: 3502
Epoch: [8]  [10/60]  eta: 0:00:30  lr: 0.000050  loss: 0.1997 (0.2095)  loss_classifier: 0.0277 (0.0314)
    loss_box_reg: 0.0463 (0.0488)  loss_mask: 0.1209 (0.1248)  loss_objectness: 0.0004 (0.0007)
    loss_rpn_box_reg: 0.0034 (0.0038)  time: 0.6164  data: 0.0449  max mem: 3502
Epoch: [8]  [20/60]  eta: 0:00:24  lr: 0.000050  loss: 0.1850 (0.2041)  loss_classifier: 0.0279 (0.0306)
    loss_box_reg: 0.0408 (0.0452)  loss_mask: 0.1157 (0.1247)  loss_objectness: 0.0003 (0.0007)
    loss_rpn_box_reg: 0.0022 (0.0029)  time: 0.5852  data: 0.0110  max mem: 3502
Epoch: [8]  [30/60]  eta: 0:00:18  lr: 0.000050  loss: 0.1732 (0.2025)  loss_classifier: 0.0288 (0.0301)
```

Problem 5 continued on next page. . .

```
    loss_box_reg: 0.0408 (0.0461)  loss_mask: 0.1126 (0.1224)  loss_objectness: 0.0003 (0.0009)
    loss_rpn_box_reg: 0.0020 (0.0030)  time: 0.5965  data: 0.0125  max mem: 3502
Epoch: [8]  [40/60]  eta: 0:00:11  lr: 0.000050  loss: 0.1583 (0.1962)  loss_classifier: 0.0219 (0.0287)
    loss_box_reg: 0.0342 (0.0437)  loss_mask: 0.1055 (0.1202)  loss_objectness: 0.0003 (0.0008)
    loss_rpn_box_reg: 0.0024 (0.0029)  time: 0.5761  data: 0.0120  max mem: 3502
Epoch: [8]  [50/60]  eta: 0:00:05  lr: 0.000050  loss: 0.1523 (0.1886)  loss_classifier: 0.0207 (0.0272)
    loss_box_reg: 0.0241 (0.0406)  loss_mask: 0.1015 (0.1174)  loss_objectness: 0.0002 (0.0007)
    loss_rpn_box_reg: 0.0012 (0.0027)  time: 0.5625  data: 0.0106  max mem: 3502
Epoch: [8]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.1590 (0.1897)  loss_classifier: 0.0245 (0.0272)
    loss_box_reg: 0.0258 (0.0407)  loss_mask: 0.1005 (0.1184)  loss_objectness: 0.0002 (0.0006)
    loss_rpn_box_reg: 0.0017 (0.0028)  time: 0.5649  data: 0.0110  max mem: 3502
Epoch: [8] Total time: 0:00:35 (0.5856 s / it)
Epoch: [9]  [ 0/60]  eta: 0:00:51  lr: 0.000005  loss: 0.1334 (0.1334)  loss_classifier: 0.0115 (0.0115)
    loss_box_reg: 0.0185 (0.0185)  loss_mask: 0.1027 (0.1027)  loss_objectness: 0.0003 (0.0003)
    loss_rpn_box_reg: 0.0003 (0.0003)  time: 0.8519  data: 0.2903  max mem: 3502
Epoch: [9]  [10/60]  eta: 0:00:30  lr: 0.000005  loss: 0.1661 (0.1952)  loss_classifier: 0.0255 (0.0257)
    loss_box_reg: 0.0370 (0.0407)  loss_mask: 0.1086 (0.1237)  loss_objectness: 0.0003 (0.0011)
    loss_rpn_box_reg: 0.0044 (0.0039)  time: 0.6142  data: 0.0361  max mem: 3502
Epoch: [9]  [20/60]  eta: 0:00:24  lr: 0.000005  loss: 0.1924 (0.2076)  loss_classifier: 0.0308 (0.0292)
    loss_box_reg: 0.0454 (0.0485)  loss_mask: 0.1188 (0.1246)  loss_objectness: 0.0006 (0.0015)
    loss_rpn_box_reg: 0.0029 (0.0038)  time: 0.6057  data: 0.0130  max mem: 3502
Epoch: [9]  [30/60]  eta: 0:00:18  lr: 0.000005  loss: 0.1712 (0.1939)  loss_classifier: 0.0256 (0.0265)
    loss_box_reg: 0.0339 (0.0422)  loss_mask: 0.1153 (0.1209)  loss_objectness: 0.0004 (0.0012)
    loss_rpn_box_reg: 0.0022 (0.0031)  time: 0.5977  data: 0.0129  max mem: 3601
Epoch: [9]  [40/60]  eta: 0:00:11  lr: 0.000005  loss: 0.1612 (0.1870)  loss_classifier: 0.0205 (0.0261)
    loss_box_reg: 0.0229 (0.0386)  loss_mask: 0.1069 (0.1183)  loss_objectness: 0.0002 (0.0011)
    loss_rpn_box_reg: 0.0015 (0.0029)  time: 0.5673  data: 0.0123  max mem: 3601
Epoch: [9]  [50/60]  eta: 0:00:05  lr: 0.000005  loss: 0.1615 (0.1865)  loss_classifier: 0.0239 (0.0272)
    loss_box_reg: 0.0288 (0.0387)  loss_mask: 0.1023 (0.1167)  loss_objectness: 0.0004 (0.0012)
    loss_rpn_box_reg: 0.0014 (0.0027)  time: 0.5674  data: 0.0122  max mem: 3601
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.1733 (0.1867)  loss_classifier: 0.0238 (0.0275)
```

Problem 5 continued on next page. . .

```
    loss_box_reg: 0.0291 (0.0389)  loss_mask: 0.1071 (0.1163)  loss_objectness: 0.0004 (0.0012)
    loss_rpn_box_reg: 0.0014 (0.0028)  time: 0.5804  data: 0.0106  max mem: 3601
Epoch: [9] Total time: 0:00:35 (0.5911 s / it)
```

Evaluate the model (Option 1: ResNet)

```
evaluate(model_1, data_loader_test, device=device)
```

```
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:19  model_time: 0.1760 (0.1760)  evaluator_time: 0.0051 (0.0051)
       time: 0.3944  data: 0.2121  max mem: 3601
Test:  [49/50]  eta: 0:00:00  model_time: 0.1104 (0.1096)  evaluator_time: 0.0046 (0.0062)
       time: 0.1239  data: 0.0066  max mem: 3601
Test: Total time: 0:00:06 (0.1300 s / it)
Averaged stats: model_time: 0.1104 (0.1096)  evaluator_time: 0.0046 (0.0062)
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 ] = 0.834
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.990
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.960
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.508
 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 ] = 0.380
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.876
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.876
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 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 ] = 0.883
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 ] = 0.990
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.918
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.452
 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 ] = 0.345
 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 ] = -1.000
 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 ] = 0.807
```

Problem 5 continued on next page. . .

Now let's instantiate the model (Option 2: MobileNet) and the optimizer.

```
# get the model using our helper function
model_2 = get_instance_segmentation_model_2(num_classes)

# move model to the right device
model_2.to(device)



# construct an optimizer
params = [p for p in model_2.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005,
                            momentum=0.9, weight_decay=0.0005)

# and a learning rate scheduler which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
```

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

```
# let's train it for 10 epochs
from torch.optim.lr_scheduler import StepLR
num_epochs = 10

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model_2, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
```

Problem 5 continued on next page. . .

```
Epoch: [0]  [ 0/60]  eta: 0:01:05  lr: 0.000090  loss: 6.5203 (6.5203)  loss_classifier: 0.6889 (0.6889)
    loss_box_reg: 0.0088 (0.0088)  loss_mask: 5.0657 (5.0657)  loss_objectness: 0.6994 (0.6994)
    loss_rpn_box_reg: 0.0575 (0.0575)  time: 1.0968  data: 0.2465  max mem: 3693
Epoch: [0]  [10/60]  eta: 0:00:21  lr: 0.000936  loss: 4.7205 (4.6260)  loss_classifier: 0.3987 (0.3992)
    loss_box_reg: 0.0289 (0.0326)  loss_mask: 3.6863 (3.4848)  loss_objectness: 0.6767 (0.6646)
    loss_rpn_box_reg: 0.0352 (0.0449)  time: 0.4322  data: 0.0316  max mem: 5262
Epoch: [0]  [20/60]  eta: 0:00:16  lr: 0.001783  loss: 2.6217 (3.3756)  loss_classifier: 0.2578 (0.3371)
    loss_box_reg: 0.0701 (0.0947)  loss_mask: 1.6175 (2.3442)  loss_objectness: 0.5531 (0.5599)
    loss_rpn_box_reg: 0.0245 (0.0397)  time: 0.3894  data: 0.0102  max mem: 5724
Epoch: [0]  [30/60]  eta: 0:00:12  lr: 0.002629  loss: 1.5354 (2.7402)  loss_classifier: 0.2467 (0.3286)
    loss_box_reg: 0.1486 (0.1191)  loss_mask: 0.6964 (1.7874)  loss_objectness: 0.3265 (0.4687)
    loss_rpn_box_reg: 0.0254 (0.0363)  time: 0.4320  data: 0.0132  max mem: 6265
Epoch: [0]  [40/60]  eta: 0:00:08  lr: 0.003476  loss: 1.2901 (2.3946)  loss_classifier: 0.2437 (0.3174)
    loss_box_reg: 0.1729 (0.1362)  loss_mask: 0.6119 (1.5038)  loss_objectness: 0.2244 (0.4024)
    loss_rpn_box_reg: 0.0261 (0.0347)  time: 0.4314  data: 0.0137  max mem: 6265
Epoch: [0]  [50/60]  eta: 0:00:04  lr: 0.004323  loss: 1.0495 (2.1153)  loss_classifier: 0.1992 (0.2912)
    loss_box_reg: 0.1809 (0.1415)  loss_mask: 0.5276 (1.3044)  loss_objectness: 0.1576 (0.3460)
    loss_rpn_box_reg: 0.0220 (0.0323)  time: 0.4102  data: 0.0104  max mem: 6265
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.9402 (1.9310)  loss_classifier: 0.1703 (0.2699)
    loss_box_reg: 0.1507 (0.1416)  loss_mask: 0.4977 (1.1843)  loss_objectness: 0.0928 (0.3037)
    loss_rpn_box_reg: 0.0220 (0.0314)  time: 0.3988  data: 0.0101  max mem: 6265
Epoch: [0] Total time: 0:00:25 (0.4208 s / it)
Epoch: [1]  [ 0/60]  eta: 0:00:46  lr: 0.005000  loss: 0.7883 (0.7883)  loss_classifier: 0.1263 (0.1263)
    loss_box_reg: 0.1246 (0.1246)  loss_mask: 0.3986 (0.3986)  loss_objectness: 0.0752 (0.0752)
    loss_rpn_box_reg: 0.0636 (0.0636)  time: 0.7824  data: 0.3804  max mem: 6265
Epoch: [1]  [10/60]  eta: 0:00:21  lr: 0.005000  loss: 0.7559 (0.7262)  loss_classifier: 0.1164 (0.1175)
    loss_box_reg: 0.1168 (0.1245)  loss_mask: 0.3986 (0.3876)  loss_objectness: 0.0550 (0.0532)
    loss_rpn_box_reg: 0.0359 (0.0433)  time: 0.4286  data: 0.0415  max mem: 6265
Epoch: [1]  [20/60]  eta: 0:00:16  lr: 0.005000  loss: 0.7559 (0.7564)  loss_classifier: 0.1161 (0.1238)
    loss_box_reg: 0.1168 (0.1292)  loss_mask: 0.3963 (0.3975)  loss_objectness: 0.0549 (0.0569)
    loss_rpn_box_reg: 0.0373 (0.0490)  time: 0.4060  data: 0.0089  max mem: 6265
```

Problem 5 continued on next page. . .

```
Epoch: [1]  [30/60]  eta: 0:00:12  lr: 0.005000  loss: 0.6814 (0.7417)  loss_classifier: 0.1022 (0.1179)
    loss_box_reg: 0.1128 (0.1267)  loss_mask: 0.3963 (0.3971)  loss_objectness: 0.0494 (0.0532)
    loss_rpn_box_reg: 0.0368 (0.0467)  time: 0.4115  data: 0.0132  max mem: 6265
Epoch: [1]  [40/60]  eta: 0:00:08  lr: 0.005000  loss: 0.6708 (0.7500)  loss_classifier: 0.0879 (0.1127)
    loss_box_reg: 0.1205 (0.1293)  loss_mask: 0.4019 (0.4090)  loss_objectness: 0.0428 (0.0510)
    loss_rpn_box_reg: 0.0344 (0.0480)  time: 0.4071  data: 0.0137  max mem: 6265
Epoch: [1]  [50/60]  eta: 0:00:04  lr: 0.005000  loss: 0.6906 (0.7310)  loss_classifier: 0.0879 (0.1058)
    loss_box_reg: 0.1198 (0.1218)  loss_mask: 0.3924 (0.4079)  loss_objectness: 0.0366 (0.0504)
    loss_rpn_box_reg: 0.0344 (0.0451)  time: 0.4002  data: 0.0120  max mem: 6265
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.6447 (0.7225)  loss_classifier: 0.0808 (0.1027)
    loss_box_reg: 0.1198 (0.1254)  loss_mask: 0.3478 (0.4033)  loss_objectness: 0.0321 (0.0483)
    loss_rpn_box_reg: 0.0320 (0.0427)  time: 0.4088  data: 0.0128  max mem: 6265
Epoch: [1] Total time: 0:00:24 (0.4160 s / it)
Epoch: [2]  [ 0/60]  eta: 0:00:39  lr: 0.005000  loss: 0.5779 (0.5779)  loss_classifier: 0.0611 (0.0611)
    loss_box_reg: 0.0954 (0.0954)  loss_mask: 0.3804 (0.3804)  loss_objectness: 0.0189 (0.0189)
    loss_rpn_box_reg: 0.0221 (0.0221)  time: 0.6631  data: 0.2427  max mem: 6265
Epoch: [2]  [10/60]  eta: 0:00:22  lr: 0.005000  loss: 0.6993 (0.6658)  loss_classifier: 0.0873 (0.0937)
    loss_box_reg: 0.1159 (0.1298)  loss_mask: 0.3804 (0.3670)  loss_objectness: 0.0348 (0.0413)
    loss_rpn_box_reg: 0.0323 (0.0341)  time: 0.4453  data: 0.0312  max mem: 6368
Epoch: [2]  [20/60]  eta: 0:00:17  lr: 0.005000  loss: 0.5693 (0.6020)  loss_classifier: 0.0727 (0.0824)
    loss_box_reg: 0.1030 (0.1183)  loss_mask: 0.3188 (0.3357)  loss_objectness: 0.0347 (0.0355)
    loss_rpn_box_reg: 0.0259 (0.0300)  time: 0.4172  data: 0.0101  max mem: 6368
Epoch: [2]  [30/60]  eta: 0:00:12  lr: 0.005000  loss: 0.5262 (0.6008)  loss_classifier: 0.0684 (0.0802)
    loss_box_reg: 0.0967 (0.1210)  loss_mask: 0.3032 (0.3357)  loss_objectness: 0.0293 (0.0339)
    loss_rpn_box_reg: 0.0246 (0.0300)  time: 0.4152  data: 0.0121  max mem: 6368
Epoch: [2]  [40/60]  eta: 0:00:08  lr: 0.005000  loss: 0.5486 (0.6025)  loss_classifier: 0.0691 (0.0778)
    loss_box_reg: 0.1041 (0.1192)  loss_mask: 0.3394 (0.3415)  loss_objectness: 0.0303 (0.0347)
    loss_rpn_box_reg: 0.0246 (0.0292)  time: 0.4140  data: 0.0124  max mem: 6368
Epoch: [2]  [50/60]  eta: 0:00:04  lr: 0.005000  loss: 0.5385 (0.5909)  loss_classifier: 0.0642 (0.0767)
    loss_box_reg: 0.0951 (0.1167)  loss_mask: 0.3180 (0.3340)  loss_objectness: 0.0334 (0.0345)
    loss_rpn_box_reg: 0.0221 (0.0290)  time: 0.4067  data: 0.0103  max mem: 6368
```

Problem 5 continued on next page. . .

```
Epoch: [2]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.4718 (0.5893)  loss_classifier: 0.0654 (0.0767)
    loss_box_reg: 0.0712 (0.1134)  loss_mask: 0.2928 (0.3354)  loss_objectness: 0.0289 (0.0339)
    loss_rpn_box_reg: 0.0264 (0.0299)  time: 0.4024  data: 0.0104  max mem: 6368
Epoch: [2] Total time: 0:00:25 (0.4179 s / it)
Epoch: [3]  [ 0/60]  eta: 0:00:48  lr: 0.000500  loss: 0.4911 (0.4911)  loss_classifier: 0.0379 (0.0379)
    loss_box_reg: 0.0909 (0.0909)  loss_mask: 0.3092 (0.3092)  loss_objectness: 0.0176 (0.0176)
    loss_rpn_box_reg: 0.0355 (0.0355)  time: 0.8160  data: 0.3878  max mem: 6368
Epoch: [3]  [10/60]  eta: 0:00:22  lr: 0.000500  loss: 0.4911 (0.5296)  loss_classifier: 0.0602 (0.0648)
    loss_box_reg: 0.1060 (0.1102)  loss_mask: 0.2897 (0.2908)  loss_objectness: 0.0255 (0.0283)
    loss_rpn_box_reg: 0.0355 (0.0355)  time: 0.4529  data: 0.0444  max mem: 6368
Epoch: [3]  [20/60]  eta: 0:00:17  lr: 0.000500  loss: 0.5026 (0.5378)  loss_classifier: 0.0602 (0.0671)
    loss_box_reg: 0.0975 (0.1103)  loss_mask: 0.2814 (0.2987)  loss_objectness: 0.0259 (0.0318)
    loss_rpn_box_reg: 0.0302 (0.0298)  time: 0.4139  data: 0.0107  max mem: 6368
Epoch: [3]  [30/60]  eta: 0:00:12  lr: 0.000500  loss: 0.5142 (0.5310)  loss_classifier: 0.0598 (0.0677)
    loss_box_reg: 0.0889 (0.1087)  loss_mask: 0.2754 (0.2980)  loss_objectness: 0.0270 (0.0306)
    loss_rpn_box_reg: 0.0216 (0.0259)  time: 0.4158  data: 0.0136  max mem: 6368
Epoch: [3]  [40/60]  eta: 0:00:08  lr: 0.000500  loss: 0.5317 (0.5300)  loss_classifier: 0.0677 (0.0680)
    loss_box_reg: 0.1121 (0.1105)  loss_mask: 0.2721 (0.2980)  loss_objectness: 0.0220 (0.0292)
    loss_rpn_box_reg: 0.0169 (0.0243)  time: 0.4232  data: 0.0128  max mem: 6368
Epoch: [3]  [50/60]  eta: 0:00:04  lr: 0.000500  loss: 0.4661 (0.5222)  loss_classifier: 0.0666 (0.0673)
    loss_box_reg: 0.1014 (0.1087)  loss_mask: 0.2741 (0.2957)  loss_objectness: 0.0196 (0.0280)
    loss_rpn_box_reg: 0.0158 (0.0225)  time: 0.4169  data: 0.0100  max mem: 6368
Epoch: [3]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.4759 (0.5158)  loss_classifier: 0.0616 (0.0667)
    loss_box_reg: 0.0944 (0.1087)  loss_mask: 0.2616 (0.2901)  loss_objectness: 0.0266 (0.0286)
    loss_rpn_box_reg: 0.0160 (0.0217)  time: 0.4142  data: 0.0104  max mem: 6368
Epoch: [3] Total time: 0:00:25 (0.4259 s / it)
Epoch: [4]  [ 0/60]  eta: 0:00:43  lr: 0.000500  loss: 0.3010 (0.3010)  loss_classifier: 0.0314 (0.0314)
    loss_box_reg: 0.0476 (0.0476)  loss_mask: 0.1832 (0.1832)  loss_objectness: 0.0346 (0.0346)
    loss_rpn_box_reg: 0.0042 (0.0042)  time: 0.7263  data: 0.2824  max mem: 6368
Epoch: [4]  [10/60]  eta: 0:00:21  lr: 0.000500  loss: 0.4187 (0.4983)  loss_classifier: 0.0492 (0.0556)
    loss_box_reg: 0.0775 (0.0988)  loss_mask: 0.2722 (0.3019)  loss_objectness: 0.0230 (0.0269)
```

Problem 5 continued on next page. . .

```
    loss_rpn_box_reg: 0.0118 (0.0151)  time: 0.4382  data: 0.0351  max mem: 6368
Epoch: [4]  [20/60]  eta: 0:00:17  lr: 0.000500  loss: 0.4466 (0.4932)  loss_classifier: 0.0567 (0.0602)
    loss_box_reg: 0.0989 (0.1112)  loss_mask: 0.2742 (0.2837)  loss_objectness: 0.0161 (0.0225)
    loss_rpn_box_reg: 0.0136 (0.0156)  time: 0.4196  data: 0.0104  max mem: 6368
Epoch: [4]  [30/60]  eta: 0:00:12  lr: 0.000500  loss: 0.4962 (0.4995)  loss_classifier: 0.0616 (0.0615)
    loss_box_reg: 0.1056 (0.1104)  loss_mask: 0.2745 (0.2856)  loss_objectness: 0.0184 (0.0264)
    loss_rpn_box_reg: 0.0140 (0.0156)  time: 0.4253  data: 0.0118  max mem: 6368
Epoch: [4]  [40/60]  eta: 0:00:08  lr: 0.000500  loss: 0.4710 (0.4949)  loss_classifier: 0.0586 (0.0625)
    loss_box_reg: 0.0913 (0.1102)  loss_mask: 0.2623 (0.2793)  loss_objectness: 0.0268 (0.0272)
    loss_rpn_box_reg: 0.0145 (0.0157)  time: 0.4139  data: 0.0115  max mem: 6368
Epoch: [4]  [50/60]  eta: 0:00:04  lr: 0.000500  loss: 0.4934 (0.5086)  loss_classifier: 0.0725 (0.0661)
    loss_box_reg: 0.1135 (0.1151)  loss_mask: 0.2645 (0.2837)  loss_objectness: 0.0270 (0.0271)
    loss_rpn_box_reg: 0.0174 (0.0166)  time: 0.4203  data: 0.0100  max mem: 6368
Epoch: [4]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.4700 (0.4996)  loss_classifier: 0.0607 (0.0641)
    loss_box_reg: 0.0973 (0.1102)  loss_mask: 0.2639 (0.2817)  loss_objectness: 0.0240 (0.0270)
    loss_rpn_box_reg: 0.0185 (0.0165)  time: 0.4144  data: 0.0109  max mem: 6368
Epoch: [4] Total time: 0:00:25 (0.4244 s / it)
Epoch: [5]  [ 0/60]  eta: 0:00:47  lr: 0.000500  loss: 0.6510 (0.6510)  loss_classifier: 0.0944 (0.0944)
    loss_box_reg: 0.1744 (0.1744)  loss_mask: 0.3214 (0.3214)  loss_objectness: 0.0378 (0.0378)
    loss_rpn_box_reg: 0.0230 (0.0230)  time: 0.7955  data: 0.2694  max mem: 6368
Epoch: [5]  [10/60]  eta: 0:00:23  lr: 0.000500  loss: 0.6000 (0.6002)  loss_classifier: 0.0740 (0.0825)
    loss_box_reg: 0.1513 (0.1520)  loss_mask: 0.3200 (0.3163)  loss_objectness: 0.0251 (0.0270)
    loss_rpn_box_reg: 0.0197 (0.0225)  time: 0.4691  data: 0.0342  max mem: 6368
Epoch: [5]  [20/60]  eta: 0:00:17  lr: 0.000500  loss: 0.4647 (0.5521)  loss_classifier: 0.0652 (0.0739)
    loss_box_reg: 0.1036 (0.1313)  loss_mask: 0.2695 (0.3030)  loss_objectness: 0.0251 (0.0253)
    loss_rpn_box_reg: 0.0157 (0.0185)  time: 0.4220  data: 0.0104  max mem: 6368
Epoch: [5]  [30/60]  eta: 0:00:13  lr: 0.000500  loss: 0.4639 (0.5414)  loss_classifier: 0.0635 (0.0712)
    loss_box_reg: 0.0942 (0.1242)  loss_mask: 0.2873 (0.3011)  loss_objectness: 0.0206 (0.0265)
    loss_rpn_box_reg: 0.0140 (0.0185)  time: 0.4181  data: 0.0131  max mem: 6368
Epoch: [5]  [40/60]  eta: 0:00:08  lr: 0.000500  loss: 0.4536 (0.5163)  loss_classifier: 0.0610 (0.0680)
    loss_box_reg: 0.0942 (0.1185)  loss_mask: 0.2631 (0.2875)  loss_objectness: 0.0166 (0.0250)
```

```
    loss_rpn_box_reg: 0.0160 (0.0173)  time: 0.4228  data: 0.0128  max mem: 6368
Epoch: [5]  [50/60]  eta: 0:00:04  lr: 0.000500  loss: 0.4536 (0.5046)  loss_classifier: 0.0550 (0.0660)
    loss_box_reg: 0.0980 (0.1129)  loss_mask: 0.2425 (0.2835)  loss_objectness: 0.0166 (0.0249)
    loss_rpn_box_reg: 0.0137 (0.0172)  time: 0.4131  data: 0.0101  max mem: 6368
Epoch: [5]  [59/60]  eta: 0:00:00  lr: 0.000500  loss: 0.4460 (0.4953)  loss_classifier: 0.0523 (0.0643)
    loss_box_reg: 0.0980 (0.1105)  loss_mask: 0.2582 (0.2804)  loss_objectness: 0.0181 (0.0235)
    loss_rpn_box_reg: 0.0147 (0.0166)  time: 0.4101  data: 0.0110  max mem: 6368
Epoch: [5] Total time: 0:00:25 (0.4258 s / it)
Epoch: [6]  [ 0/60]  eta: 0:00:52  lr: 0.000050  loss: 0.5759 (0.5759)  loss_classifier: 0.0769 (0.0769)
    loss_box_reg: 0.0943 (0.0943)  loss_mask: 0.3320 (0.3320)  loss_objectness: 0.0525 (0.0525)
    loss_rpn_box_reg: 0.0202 (0.0202)  time: 0.8732  data: 0.4139  max mem: 6368
Epoch: [6]  [10/60]  eta: 0:00:22  lr: 0.000050  loss: 0.4275 (0.4858)  loss_classifier: 0.0617 (0.0650)
    loss_box_reg: 0.0877 (0.1023)  loss_mask: 0.2712 (0.2770)  loss_objectness: 0.0219 (0.0259)
    loss_rpn_box_reg: 0.0127 (0.0155)  time: 0.4511  data: 0.0446  max mem: 6368
Epoch: [6]  [20/60]  eta: 0:00:17  lr: 0.000050  loss: 0.4976 (0.5114)  loss_classifier: 0.0680 (0.0707)
    loss_box_reg: 0.1021 (0.1109)  loss_mask: 0.2766 (0.2875)  loss_objectness: 0.0209 (0.0255)
    loss_rpn_box_reg: 0.0156 (0.0167)  time: 0.4148  data: 0.0089  max mem: 6368
Epoch: [6]  [30/60]  eta: 0:00:13  lr: 0.000050  loss: 0.5398 (0.5245)  loss_classifier: 0.0816 (0.0727)
    loss_box_reg: 0.1180 (0.1138)  loss_mask: 0.2892 (0.2923)  loss_objectness: 0.0257 (0.0294)
    loss_rpn_box_reg: 0.0161 (0.0163)  time: 0.4282  data: 0.0128  max mem: 6368
Epoch: [6]  [40/60]  eta: 0:00:08  lr: 0.000050  loss: 0.4732 (0.5100)  loss_classifier: 0.0675 (0.0696)
    loss_box_reg: 0.0876 (0.1101)  loss_mask: 0.2790 (0.2863)  loss_objectness: 0.0210 (0.0277)
    loss_rpn_box_reg: 0.0149 (0.0163)  time: 0.4216  data: 0.0131  max mem: 6368
Epoch: [6]  [50/60]  eta: 0:00:04  lr: 0.000050  loss: 0.4599 (0.5037)  loss_classifier: 0.0620 (0.0694)
    loss_box_reg: 0.0991 (0.1108)  loss_mask: 0.2568 (0.2803)  loss_objectness: 0.0210 (0.0267)
    loss_rpn_box_reg: 0.0153 (0.0165)  time: 0.4163  data: 0.0102  max mem: 6368
Epoch: [6]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.4204 (0.4877)  loss_classifier: 0.0586 (0.0670)
    loss_box_reg: 0.0933 (0.1073)  loss_mask: 0.2284 (0.2729)  loss_objectness: 0.0127 (0.0246)
    loss_rpn_box_reg: 0.0127 (0.0159)  time: 0.4146  data: 0.0105  max mem: 6368
Epoch: [6] Total time: 0:00:25 (0.4272 s / it)
Epoch: [7]  [ 0/60]  eta: 0:00:47  lr: 0.000050  loss: 0.6768 (0.6768)  loss_classifier: 0.0698 (0.0698)
```

Problem 5 continued on next page. . .

```
    loss_box_reg: 0.1491 (0.1491)  loss_mask: 0.3956 (0.3956)  loss_objectness: 0.0404 (0.0404)
    loss_rpn_box_reg: 0.0219 (0.0219)  time: 0.7872  data: 0.3537  max mem: 6368
Epoch: [7]  [10/60]  eta: 0:00:22  lr: 0.000050  loss: 0.4558 (0.5038)  loss_classifier: 0.0560 (0.0591)
    loss_box_reg: 0.1125 (0.1134)  loss_mask: 0.2634 (0.2896)  loss_objectness: 0.0229 (0.0263)
    loss_rpn_box_reg: 0.0177 (0.0153)  time: 0.4417  data: 0.0403  max mem: 6368
Epoch: [7]  [20/60]  eta: 0:00:17  lr: 0.000050  loss: 0.4571 (0.5102)  loss_classifier: 0.0609 (0.0649)
    loss_box_reg: 0.1125 (0.1206)  loss_mask: 0.2608 (0.2846)  loss_objectness: 0.0209 (0.0246)
    loss_rpn_box_reg: 0.0140 (0.0156)  time: 0.4193  data: 0.0104  max mem: 6368
Epoch: [7]  [30/60]  eta: 0:00:13  lr: 0.000050  loss: 0.4571 (0.4965)  loss_classifier: 0.0609 (0.0645)
    loss_box_reg: 0.1012 (0.1124)  loss_mask: 0.2607 (0.2784)  loss_objectness: 0.0178 (0.0255)
    loss_rpn_box_reg: 0.0127 (0.0156)  time: 0.4288  data: 0.0138  max mem: 6368
Epoch: [7]  [40/60]  eta: 0:00:08  lr: 0.000050  loss: 0.4190 (0.4798)  loss_classifier: 0.0553 (0.0617)
    loss_box_reg: 0.0843 (0.1073)  loss_mask: 0.2428 (0.2716)  loss_objectness: 0.0161 (0.0238)
    loss_rpn_box_reg: 0.0123 (0.0153)  time: 0.4207  data: 0.0130  max mem: 6368
Epoch: [7]  [50/60]  eta: 0:00:04  lr: 0.000050  loss: 0.4190 (0.4785)  loss_classifier: 0.0551 (0.0631)
    loss_box_reg: 0.0856 (0.1066)  loss_mask: 0.2387 (0.2702)  loss_objectness: 0.0188 (0.0235)
    loss_rpn_box_reg: 0.0125 (0.0151)  time: 0.4141  data: 0.0101  max mem: 6368
Epoch: [7]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.4685 (0.4824)  loss_classifier: 0.0655 (0.0637)
    loss_box_reg: 0.0901 (0.1066)  loss_mask: 0.2520 (0.2733)  loss_objectness: 0.0221 (0.0239)
    loss_rpn_box_reg: 0.0122 (0.0148)  time: 0.4190  data: 0.0104  max mem: 6368
Epoch: [7] Total time: 0:00:25 (0.4267 s / it)
Epoch: [8]  [ 0/60]  eta: 0:00:51  lr: 0.000050  loss: 0.6706 (0.6706)  loss_classifier: 0.0777 (0.0777)
    loss_box_reg: 0.1448 (0.1448)  loss_mask: 0.4002 (0.4002)  loss_objectness: 0.0295 (0.0295)
    loss_rpn_box_reg: 0.0184 (0.0184)  time: 0.8535  data: 0.4206  max mem: 6368
Epoch: [8]  [10/60]  eta: 0:00:23  lr: 0.000050  loss: 0.5050 (0.5125)  loss_classifier: 0.0693 (0.0705)
    loss_box_reg: 0.1053 (0.1213)  loss_mask: 0.2656 (0.2845)  loss_objectness: 0.0216 (0.0211)
    loss_rpn_box_reg: 0.0146 (0.0151)  time: 0.4670  data: 0.0438  max mem: 6368
Epoch: [8]  [20/60]  eta: 0:00:17  lr: 0.000050  loss: 0.4650 (0.5070)  loss_classifier: 0.0617 (0.0686)
    loss_box_reg: 0.0953 (0.1107)  loss_mask: 0.2535 (0.2800)  loss_objectness: 0.0260 (0.0303)
    loss_rpn_box_reg: 0.0151 (0.0174)  time: 0.4206  data: 0.0081  max mem: 6368
Epoch: [8]  [30/60]  eta: 0:00:13  lr: 0.000050  loss: 0.4650 (0.5035)  loss_classifier: 0.0625 (0.0670)
```

Problem 5 continued on next page. . .

```
    loss_box_reg: 0.0953 (0.1113)  loss_mask: 0.2566 (0.2809)  loss_objectness: 0.0261 (0.0275)
    loss_rpn_box_reg: 0.0174 (0.0169)  time: 0.4195  data: 0.0116  max mem: 6368
Epoch: [8]  [40/60]  eta: 0:00:08  lr: 0.000050  loss: 0.4551 (0.4893)  loss_classifier: 0.0607 (0.0647)
    loss_box_reg: 0.0977 (0.1070)  loss_mask: 0.2736 (0.2753)  loss_objectness: 0.0209 (0.0261)
    loss_rpn_box_reg: 0.0135 (0.0161)  time: 0.4188  data: 0.0114  max mem: 6368
Epoch: [8]  [50/60]  eta: 0:00:04  lr: 0.000050  loss: 0.4350 (0.4874)  loss_classifier: 0.0607 (0.0639)
    loss_box_reg: 0.0896 (0.1066)  loss_mask: 0.2736 (0.2770)  loss_objectness: 0.0152 (0.0245)
    loss_rpn_box_reg: 0.0119 (0.0154)  time: 0.4115  data: 0.0101  max mem: 6368
Epoch: [8]  [59/60]  eta: 0:00:00  lr: 0.000050  loss: 0.4047 (0.4767)  loss_classifier: 0.0543 (0.0628)
    loss_box_reg: 0.0841 (0.1048)  loss_mask: 0.2356 (0.2699)  loss_objectness: 0.0177 (0.0241)
    loss_rpn_box_reg: 0.0113 (0.0151)  time: 0.4120  data: 0.0107  max mem: 6368
Epoch: [8] Total time: 0:00:25 (0.4263 s / it)
Epoch: [9]  [ 0/60]  eta: 0:00:53  lr: 0.000005  loss: 0.7721 (0.7721)  loss_classifier: 0.1039 (0.1039)
    loss_box_reg: 0.2174 (0.2174)  loss_mask: 0.3776 (0.3776)  loss_objectness: 0.0391 (0.0391)
    loss_rpn_box_reg: 0.0340 (0.0340)  time: 0.8837  data: 0.4085  max mem: 6368
Epoch: [9]  [10/60]  eta: 0:00:22  lr: 0.000005  loss: 0.4864 (0.5357)  loss_classifier: 0.0601 (0.0711)
    loss_box_reg: 0.1295 (0.1232)  loss_mask: 0.2786 (0.2975)  loss_objectness: 0.0300 (0.0271)
    loss_rpn_box_reg: 0.0168 (0.0167)  time: 0.4527  data: 0.0444  max mem: 6368
Epoch: [9]  [20/60]  eta: 0:00:17  lr: 0.000005  loss: 0.4764 (0.5096)  loss_classifier: 0.0678 (0.0685)
    loss_box_reg: 0.1135 (0.1182)  loss_mask: 0.2575 (0.2840)  loss_objectness: 0.0215 (0.0234)
    loss_rpn_box_reg: 0.0166 (0.0156)  time: 0.4192  data: 0.0097  max mem: 6368
Epoch: [9]  [30/60]  eta: 0:00:12  lr: 0.000005  loss: 0.4386 (0.4874)  loss_classifier: 0.0575 (0.0649)
    loss_box_reg: 0.0879 (0.1082)  loss_mask: 0.2436 (0.2752)  loss_objectness: 0.0169 (0.0249)
    loss_rpn_box_reg: 0.0108 (0.0141)  time: 0.4190  data: 0.0130  max mem: 6368
Epoch: [9]  [40/60]  eta: 0:00:08  lr: 0.000005  loss: 0.4371 (0.4878)  loss_classifier: 0.0575 (0.0661)
    loss_box_reg: 0.0819 (0.1087)  loss_mask: 0.2608 (0.2741)  loss_objectness: 0.0168 (0.0243)
    loss_rpn_box_reg: 0.0140 (0.0147)  time: 0.4219  data: 0.0160  max mem: 6368
Epoch: [9]  [50/60]  eta: 0:00:04  lr: 0.000005  loss: 0.4522 (0.4903)  loss_classifier: 0.0597 (0.0657)
    loss_box_reg: 0.0946 (0.1086)  loss_mask: 0.2651 (0.2769)  loss_objectness: 0.0174 (0.0241)
    loss_rpn_box_reg: 0.0142 (0.0150)  time: 0.4261  data: 0.0140  max mem: 6368
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.4792 (0.4862)  loss_classifier: 0.0574 (0.0654)
```

```
    loss_box_reg: 0.0962 (0.1089)  loss_mask: 0.2667 (0.2742)  loss_objectness: 0.0156 (0.0228)
    loss_rpn_box_reg: 0.0135 (0.0149)  time: 0.4234  data: 0.0116  max mem: 6368
Epoch: [9] Total time: 0:00:25 (0.4304 s / it)
```

Problem 5 continued on next page. . .

Evaluate the model (Option 2: Mobilenet)

```
evaluate(model_2, data_loader_test, device=device)
```

```
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:18  model_time: 0.1628 (0.1628)  evaluator_time: 0.0078 (0.0078)
       time: 0.3725  data: 0.2006  max mem: 6368
Test:  [49/50]  eta: 0:00:00  model_time: 0.0530 (0.0586)  evaluator_time: 0.0085 (0.0116)
       time: 0.0707  data: 0.0049  max mem: 6368
Test: Total time: 0:00:04 (0.0826 s / it)
Averaged stats: model_time: 0.0530 (0.0586)  evaluator_time: 0.0085 (0.0116)
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 ] = 0.441
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.907
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.307
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.130
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.462
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.234
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.551
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.559
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.375
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.572
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.356
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.837
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.260
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.035
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.379
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.214
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.442
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.448
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.150
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.469
```

Problem 5 continued on next page. . .

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

```
from PIL import Image
import requests
from torchvision import transforms

im = Image.open(requests.get(
        'https://upload.wikimedia.org/wikipedia/en/4/42/Beatles_-_Abbey_Road.jpg',
        stream=True).raw)
convert_tensor = transforms.ToTensor()
x = convert_tensor(im)

Image.fromarray(x.mul(255).permute(1, 2, 0).byte().numpy())
```

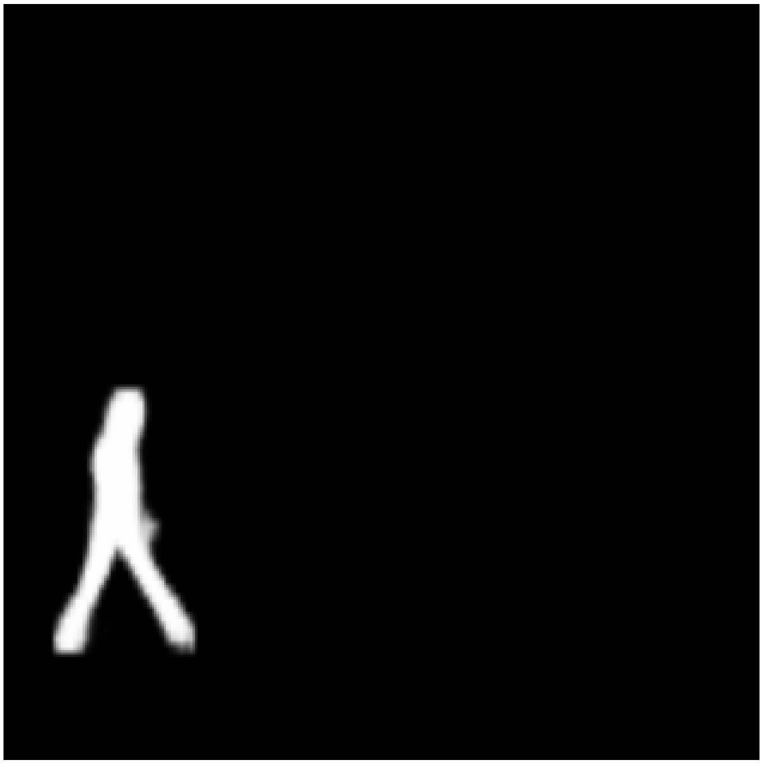
And let's now visualize the top predicted segmentation mask. The masks are predicted as `[N, 1, H, W]`, where `N` is the number of predictions, and are probability maps between 0-1.

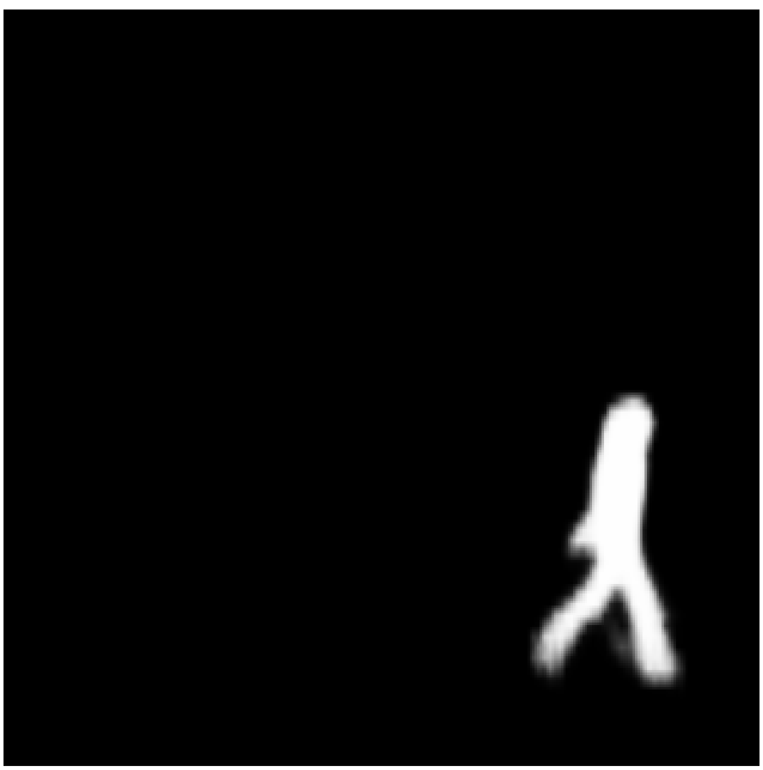
```
model_1.eval()
model_2.eval()
with torch.no_grad():
    prediction_1 = model_1([x.to(device)])
    prediction_2 = model_2([x.to(device)])
```

Model 1 Output Visualization (Top 3 Masks).

```
im1 = Image.fromarray(prediction_1[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
im2 = Image.fromarray(prediction_1[0]['masks'][1, 0].mul(255).byte().cpu().numpy())
im3 = Image.fromarray(prediction_1[0]['masks'][2, 0].mul(255).byte().cpu().numpy())

im1.show()
im2.show()
im3.show()
```

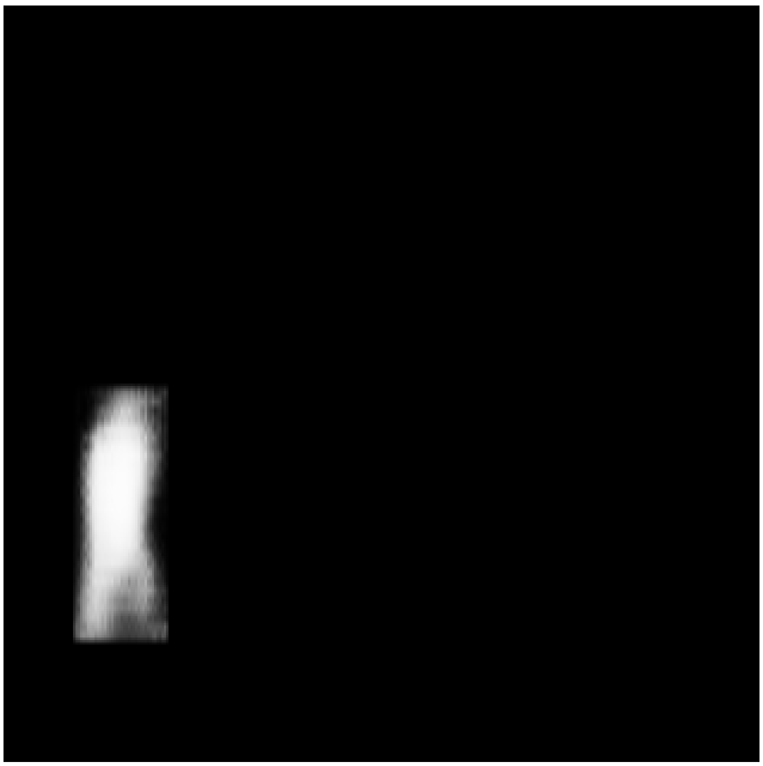
Problem 5 continued on next page. . .

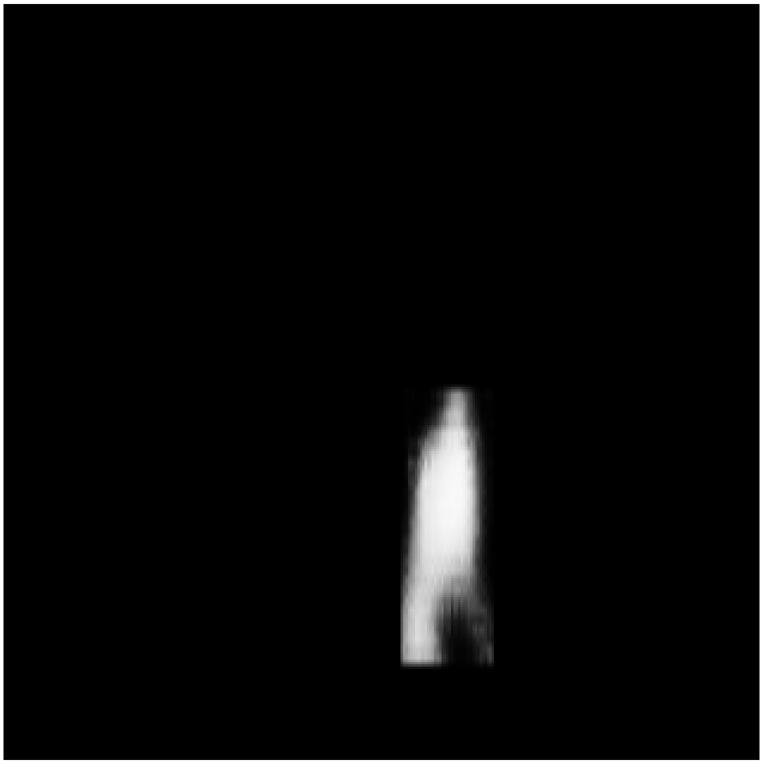


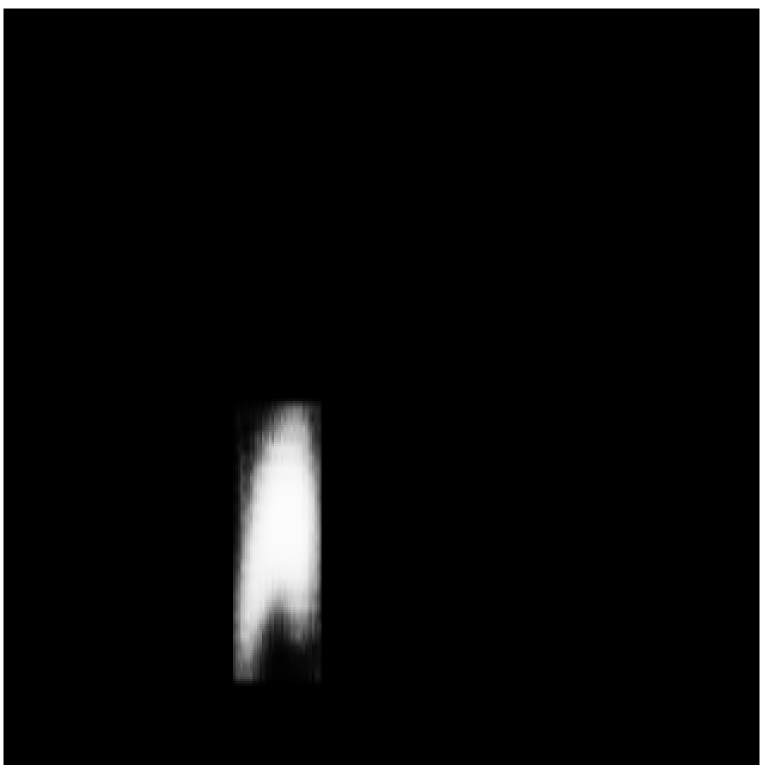


Model 2 Output Visualization (Top 3 Masks).

```
im1 = Image.fromarray(prediction_2[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
im2 = Image.fromarray(prediction_2[0]['masks'][1, 0].mul(255).byte().cpu().numpy())
im3 = Image.fromarray(prediction_2[0]['masks'][2, 0].mul(255).byte().cpu().numpy())

im1.show()
im2.show()
im3.show()
```







(b)
Performance comparison between the two object detection models based on the provided evaluation metrics:

Average Precision (AP): AP measures the accuracy of the model in localizing the objects in the image. It is the area under the precision-recall curve.

Model 1 (ResNet) outperforms Model 2 (MobileNet) in terms of AP. The ResNet model has higher AP values across all the IoU thresholds and all object sizes (small, medium, and large). The ResNet model achieves an AP of 0.990 at IoU=0.50 for all object sizes, whereas the MobileNet model achieves an AP of 0.907. Similarly, the ResNet model achieves an AP of 0.846 at IoU=0.50:0.95 for large objects, whereas the MobileNet model achieves an AP of 0.462.

Average Recall (AR): AR measures the ability of the model to detect all instances of the object in the image. It is the area under the recall-IoU curve.

Model 1 (ResNet) outperforms Model 2 (MobileNet) in terms of AR. The ResNet model has higher AR values across all the IoU thresholds and all object sizes (small, medium, and large). The ResNet model achieves an AR of 0.876 at IoU=0.50:0.95 for all object sizes, whereas the MobileNet model achieves an AR of 0.802. Similarly, the ResNet model achieves an AR of 0.883 at IoU=0.50:0.95 for large objects, whereas the MobileNet model achieves an AR of 0.807.

Intersection over Union (IoU): IoU measures the overlap between the predicted bounding box and the ground truth bounding box. It is the ratio of the intersection area to the union area.

Model 1 (ResNet) outperforms Model 2 (MobileNet) in terms of IoU. The ResNet model has higher IoU values across all the object sizes (small, medium, and large) for both bbox and segm metrics. For example, the ResNet model achieves an IoU of 0.960 at IoU=0.75 for all object sizes for bbox metric, whereas the MobileNet model achieves an IoU of 0.307. Similarly, the ResNet model achieves an IoU of 0.918 at IoU=0.75 for all object sizes for segm metric, whereas the MobileNet model achieves an IoU of 0.130.

In summary, based on the evaluation metrics provided, the ResNet model outperforms the MobileNet model in terms of object detection accuracy. The ResNet model achieves higher values of AP, AR, and IoU across all object sizes and IoU thresholds.

Problem 5 continued on next page. . .

(c)
We tested two different models for image segmentation, referred to as ”model 1” and ”model 2”. Model 1 used a more complex architecture called ”ResNet” while model 2 used a simpler architecture called ”MobileNet”.

When testing the models using a real image consisting of humans to be segmented, they found that the segmentation produced by model 1 was sharper, while the segmentation produced by model 2 was smoother. This difference in performance could be attributed to the fact that ResNet is a more complex architecture with more layers and parameters, allowing it to capture more intricate details in the image. On the other hand, MobileNet is a simpler architecture designed for mobile devices, which may not be as powerful or accurate as DeepLabV3+ in certain tasks.

Overall, the findings suggest that the choice of architecture can have a significant impact on the performance of image segmentation models. Depending on the specific task and requirements, a more complex or simpler architecture may be more appropriate.