Deep Learning HW2 - Spring 23

Aniket Sinha - aas10120

Deadline: 10th March 2023

1. (3 points) *Designing convolution filters by hand.* Consider an input 2D image and a 3×3 filter (say w) applied to this image. The goal is to guess good filters which implement each of the following elementary image processing operations.

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

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

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

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

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

ANSWER:

(a) The **Mean filter** can be used as a blurring filter. The below matrix show the weights of w for a blurring filter

$$w = \frac{1}{9} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(b) For a sharpening filter in the horizontal direction, **Prewitt or Sobel** operator can be used. The below matrix show the weights of w for a sharpening filter in the horizontal direction.

$$w = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

(c) For a sharpening filter in the vertical direction, **Prewitt** operator can be used. The below matrix show the weights of w for a sharpening filter in the vertical direction.

$$w = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

(d) For a sharpening filter in a diagonal (bottom-left to top-right) direction, the weighrs of w would be:

$$w = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

2. (3 points) *Weight decay.* The use of l_2 regularization for training multilayer neural networks has a special name: weight decay. Assume an arbitrary dataset $(x_i, y_i)_{i=1}^n$ and a loss function $L(w)$ where w is a vector that represents all the trainable weights (and biases).

   (a) Write down the l_2 regularized loss, using a weighting parameter λ for the regularizer.

   (b) Derive the gradient descent update rules for this loss.

   (c) Conclude that in each update, the weights are "shrunk" or "decayed" by a multiplicative factor before applying the descent update.

   (d) What does increasing λ achieve algorithmically, and how should the learning rate be chosen to make the updates stable?

ANSWER:

Given: Dataset: $\{(x_i, y_i)\}_{i=1}^n$ and Loss function $L(w)$

(a) The l_2 regularized loss, $\overline{L}(w)$, using weighting parameter(λ)) is given by:

$$\overline{L}(w) = L(w) + \lambda \parallel w \parallel_2^2$$

(b) Gradient descent equations for the l_2 regularized loss. Here, we are taking the derivative of the above loss function w.r.t. w:

$$\Delta\overline{L}(w) = \Delta L(w) + 2.\lambda.w$$

$$w_{i+1} = w_i - \eta.\Delta\overline{L}(w_i)$$

$$w_{i+1} = w_i - \eta.(\Delta L(w) + 2.\lambda.w)$$

$$w_{i+1} = w_i(1 - 2\eta\lambda) - \eta\Delta L(w)$$

(c) From the above derived equation:

$$w_{i+1} = w_i(1 - 2\eta\lambda) - \eta\Delta L(w)$$

we can see that the weight is being updated by the factor of

$$(1 - 2\eta\lambda)$$

where η is the learning rate and the new weight contains the decaying old weight. Therefore, we can see that the weights are actually getting "shrunk" or "decayed" by a multiplicative factor before applying the descent update.

(d) As per the equation for updating the gradient, we can observe that the optimal value is achieved when the gradient gets closer to zero. Hence, with that in consideration, we can arrive at the following conclusion

$$\mid \Delta w \mid = \mid \Delta L(w) \mid /2\lambda$$

, Based on this, we can infer that

$$\mid \Delta w \mid \propto \frac{1}{2\lambda}$$

.

Therefore, we can deduce from this, that as the value of lambda increases, the weight decreases.

Also, based on the equation $1 - 2\eta\lambda$, we can observe that the relationship

$$\eta \propto \frac{1}{2\lambda}$$

must exist in order to ensure a stable update. Specifically, when we have a larger value of λ, a smaller value of η is required.

3. (2 points) *The IoU metric.* In class we defined the IoU metric (or the Jaccard similarity index) for comparing bounding boxes.

(a) Using elementary properties of sets, argue that the IoU metric between any two pair of bounding boxes is always a non-negative real number in [0, 1].

(b) If we represent each bounding box as a function of the top-left and bottom-right coordinates (assume all coordinates are real numbers) then argue that the IoU metric is non-differentiable and hence cannot be directly optimized by gradient descent.

ANSWER:

(a) Suppose we have two bounding boxes, A and B, where Box A is larger than Box B. Without the loss of generality, we can define the IOU as follows:

$$IOU(A, B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

Now, from above, we know that :

$$0 <\mid A \cap B \mid \leq \mid B \mid$$

$$\mid B \mid \leq \mid A \cup B \mid \leq \mid A \mid + \mid B \mid$$

Using the above two equations, we get,

$$\frac{0}{\mid A \mid + \mid B \mid} \leq \frac{\mid A \cap B \mid}{\mid A \cup B \mid} < \frac{\mid B \mid}{\mid B \mid}$$

Hence, we can say that :

$$0 \leq IOU(A, B) \leq 1$$

Therefore,

$$\Rightarrow IOU(A, B) \in [0, 1]$$

(b) Consider two boxes, A and B, defined by their corner points $(x_1, y_1), (x_2, y_2)$ and $(x_3, y_3), (x_4, y_4)$, respectively. Suppose that Box A moves from left to right towards Box B. When the two boxes overlap, we can calculate the intersection as follows:

$$Top - left = (min(x_1, x_3), min(y_1, y_3))$$

$$Bottom - right = (min(x_2, x_4), min(y_2, y_4))$$

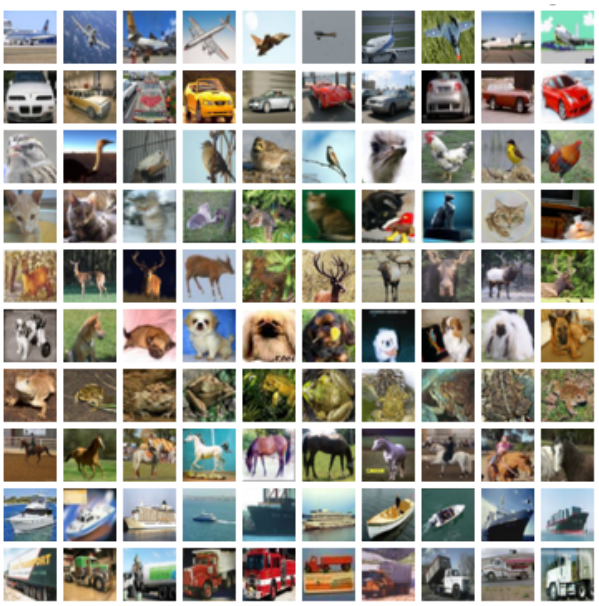
If there is full overlap between the two boxes, the IoU will reach its maximum value, while if there is no overlap, that will result in an IoU close to 0. However, because the numerator of the IoU calculation depends on the non-differentiable minima function, the IoU function itself is also non-differentiable. Hence, gradient descent cannot be used to directly optimize the IoU at every point, as differentiation is not possible for the non-differentiable parts of the function.

Question 4

AlexNet

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

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



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

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

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

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

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

import copy
import random
import time
```

```
SEED = 1234

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

Loading and Preparing the Data

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

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

```
Files already downloaded and verified
```

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

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

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

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

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

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

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

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

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

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

```
VALID_RATIO = 0.9

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

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

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

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

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

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

    n_images = len(images)

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

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

    for i in range(rows*cols):

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

        image = images[i]

        if normalize:
            image_min = image.min()
            image_max = image.max()
            image.clamp_(min = image_min, max = image_max)
```

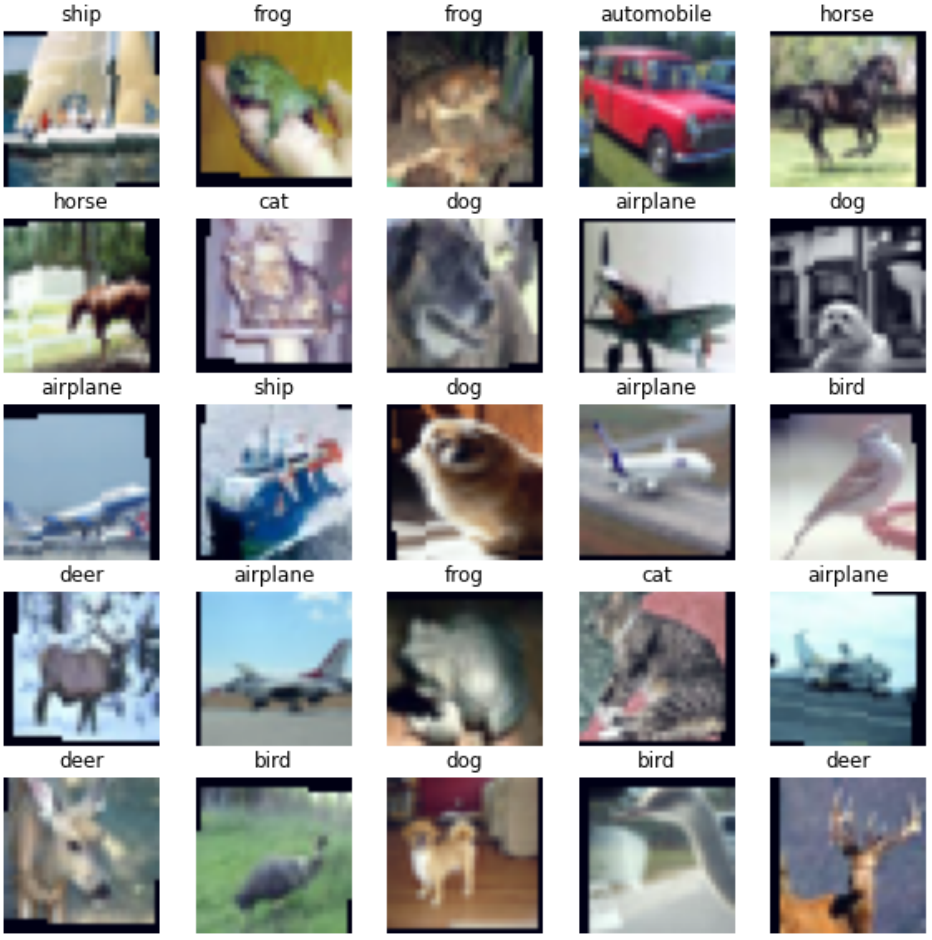
```
            image.add_(-image_min).div_(image_max - image_min + 1e-5)

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

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

adding this text cell for pdf alignment

```
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,   # Load the training data
                                 batch_size=BATCH_SIZE,   # Setting batch size for training data
                                 shuffle=True,   # Shuffling the data for each epoch
                                 num_workers=2);   # Using 2 workers for parallel data loading

valid_iterator = data.DataLoader(valid_data,   # Load the validation data
                                 batch_size=BATCH_SIZE,   # Setting batch size for validation data
                                 shuffle=True,   # Shuffling the data for each epoch
                                 num_workers=2);   # Using 2 workers for parallel data loading

test_iterator = data.DataLoader(test_data,   # Load the test data
                                batch_size=BATCH_SIZE,   # Setting batch size for test data
                                shuffle=True,   # Shuffling the data for each epoch
                                num_workers=2);   # Using 2 workers for parallel data loading
```

Defining the Model

Next up is defining the model.

AlexNet will have the following architecture:

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

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

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

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

```
        self.features = nn.Sequential(
            # Define according to the steps described above

            # 3 input channels, 64 output channels, 3x3 kernel, padding=1, stride=2
            nn.Conv2d(3,64,kernel_size=3,padding=1,stride=2),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True),   # ReLU activation function

            # 64 input channels, 192 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(64,192,kernel_size=3,padding=1,stride=1),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True),   # ReLU activation function

            # 192 input channels, 384 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(192,384,kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),   # ReLU activation function

            # 384 input channels, 256 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(384,256,kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),   # ReLU activation function

            # 256 input channels, 256 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(256,256,kernel_size=3,padding=1,stride=1),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True)   # ReLU activation function

        )

        self.classifier = nn.Sequential(
            # define according to the steps described above

            # Dropout regularization to prevent overfitting
            nn.Dropout(),

            # Fully connected layer with 1024 input features and 4096 output features
            nn.Linear(1024,4096),

            nn.ReLU(inplace=True),   # ReLU activation function

            # Dropout regularization to prevent overfitting
            nn.Dropout(),

            # Fully connected layer with 4096 input features and 4096 output features
            nn.Linear(4096,4096),
            nn.ReLU(inplace=True),   # ReLU activation function

            # Fully connected layer with 4096 input features and 10 output features (for classificat
            nn.Linear(4096,10)

        )

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

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

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

Training the Model

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

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

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

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

```
model.apply(initialize_parameters)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): 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()

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

```
# This is formatted as code
```

We define a function to calculate accuracy...

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

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

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

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in iterator:

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

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

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

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

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

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

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in iterator:

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

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
```

```
            epoch_acc += acc.item()

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

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

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

Then, finally, we train our model.

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

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

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

EPOCHS = 25

# Fill training code here
best_accuracy = 0  # initialize the best validation accuracy to 0
for ep in np.arange(EPOCHS):  # loop over the specified number of epochs
    st_time = time.time()  # record the start time of the epoch
    tr_loss, tr_acc = train(model, train_iterator, optimizer, criterion, device)  # train the model o
    val_loss, validation_accuracy = evaluate(model, valid_iterator, criterion, device)  # evaluate th
    if validation_accuracy > best_accuracy:  # if the validation accuracy is better than the best so
        torch.save(model, 'best_model.pth')  # save the model as the new best model
        best_accuracy = validation_accuracy  # update the best validation accuracy
    end_time = time.time()  # record the end time of the epoch
    elapsed_mins, elapsed_secs = epoch_time(st_time, end_time)  # calculate the elapsed time of the e
    # print the epoch number, elapsed time, training loss/accuracy, validation loss/accuracy
    print("Epoch-{0}[ Elapsed Time: {1}min {2}s] Train [ Loss={3:.2f} Acc={4:.2f} ] Validation [ Loss
          .format(ep + 1, elapsed_mins, elapsed_secs, tr_loss, tr_acc, val_loss, validation_accuracy)
```

```
Epoch-1[ Elapsed 0min 29s] Train [ Loss=2.32 Acc=0.24 ] Validation [ Loss=1.59 Acc=0.40 ]
Epoch-2[ Elapsed 0min 28s] Train [ Loss=1.52 Acc=0.44 ] Validation [ Loss=1.33 Acc=0.52 ]
Epoch-3[ Elapsed 0min 28s] Train [ Loss=1.35 Acc=0.51 ] Validation [ Loss=1.19 Acc=0.58 ]
Epoch-4[ Elapsed 0min 28s] Train [ Loss=1.25 Acc=0.55 ] Validation [ Loss=1.17 Acc=0.59 ]
Epoch-5[ Elapsed 0min 29s] Train [ Loss=1.16 Acc=0.59 ] Validation [ Loss=1.09 Acc=0.62 ]
Epoch-6[ Elapsed 0min 28s] Train [ Loss=1.11 Acc=0.61 ] Validation [ Loss=1.05 Acc=0.63 ]
Epoch-7[ Elapsed 0min 28s] Train [ Loss=1.05 Acc=0.63 ] Validation [ Loss=0.99 Acc=0.66 ]
Epoch-8[ Elapsed 0min 28s] Train [ Loss=1.00 Acc=0.64 ] Validation [ Loss=0.94 Acc=0.67 ]
Epoch-9[ Elapsed 0min 29s] Train [ Loss=0.97 Acc=0.66 ] Validation [ Loss=0.95 Acc=0.66 ]
Epoch-10[ Elapsed 0min 28s] Train [ Loss=0.93 Acc=0.68 ] Validation [ Loss=0.91 Acc=0.68 ]
Epoch-11[ Elapsed 0min 28s] Train [ Loss=0.90 Acc=0.68 ] Validation [ Loss=0.88 Acc=0.69 ]
Epoch-12[ Elapsed 0min 28s] Train [ Loss=0.88 Acc=0.69 ] Validation [ Loss=0.83 Acc=0.71 ]
Epoch-13[ Elapsed 0min 29s] Train [ Loss=0.84 Acc=0.71 ] Validation [ Loss=0.82 Acc=0.72 ]
Epoch-14[ Elapsed 0min 28s] Train [ Loss=0.82 Acc=0.71 ] Validation [ Loss=0.81 Acc=0.72 ]
Epoch-15[ Elapsed 0min 28s] Train [ Loss=0.80 Acc=0.72 ] Validation [ Loss=0.83 Acc=0.71 ]
Epoch-16[ Elapsed 0min 28s] Train [ Loss=0.79 Acc=0.73 ] Validation [ Loss=0.77 Acc=0.74 ]
Epoch-17[ Elapsed 0min 28s] Train [ Loss=0.76 Acc=0.74 ] Validation [ Loss=0.77 Acc=0.75 ]
Epoch-18[ Elapsed 0min 27s] Train [ Loss=0.75 Acc=0.74 ] Validation [ Loss=0.77 Acc=0.75 ]
Epoch-19[ Elapsed 0min 27s] Train [ Loss=0.74 Acc=0.75 ] Validation [ Loss=0.77 Acc=0.74 ]
Epoch-20[ Elapsed 0min 28s] Train [ Loss=0.71 Acc=0.76 ] Validation [ Loss=0.74 Acc=0.76 ]
Epoch-21[ Elapsed 0min 27s] Train [ Loss=0.69 Acc=0.76 ] Validation [ Loss=0.76 Acc=0.75 ]
Epoch-22[ Elapsed 0min 27s] Train [ Loss=0.70 Acc=0.76 ] Validation [ Loss=0.74 Acc=0.75 ]
Epoch-23[ Elapsed 0min 28s] Train [ Loss=0.67 Acc=0.77 ] Validation [ Loss=0.73 Acc=0.76 ]
```

```
Epoch-24[ Elapsed 0min 27s] Train [ Loss=0.66 Acc=0.77 ] Validation [ Loss=0.72 Acc=0.76 ]
Epoch-25[ Elapsed 0min 27s] Train [ Loss=0.65 Acc=0.78 ] Validation [ Loss=0.72 Acc=0.76 ]
```

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?

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

# Also, print out the confusion matrox.

model=torch.load("best_model.pth") #loading the best performing model
test_loss,test_accuracy=evaluate(model,test_iterator,criterion,device) #evaluating it on test dataset
print("Test [ Loss=%.2f Acc=%.2f} ]" % (test_loss,test_accuracy)) #print the test loss and accuracy
```

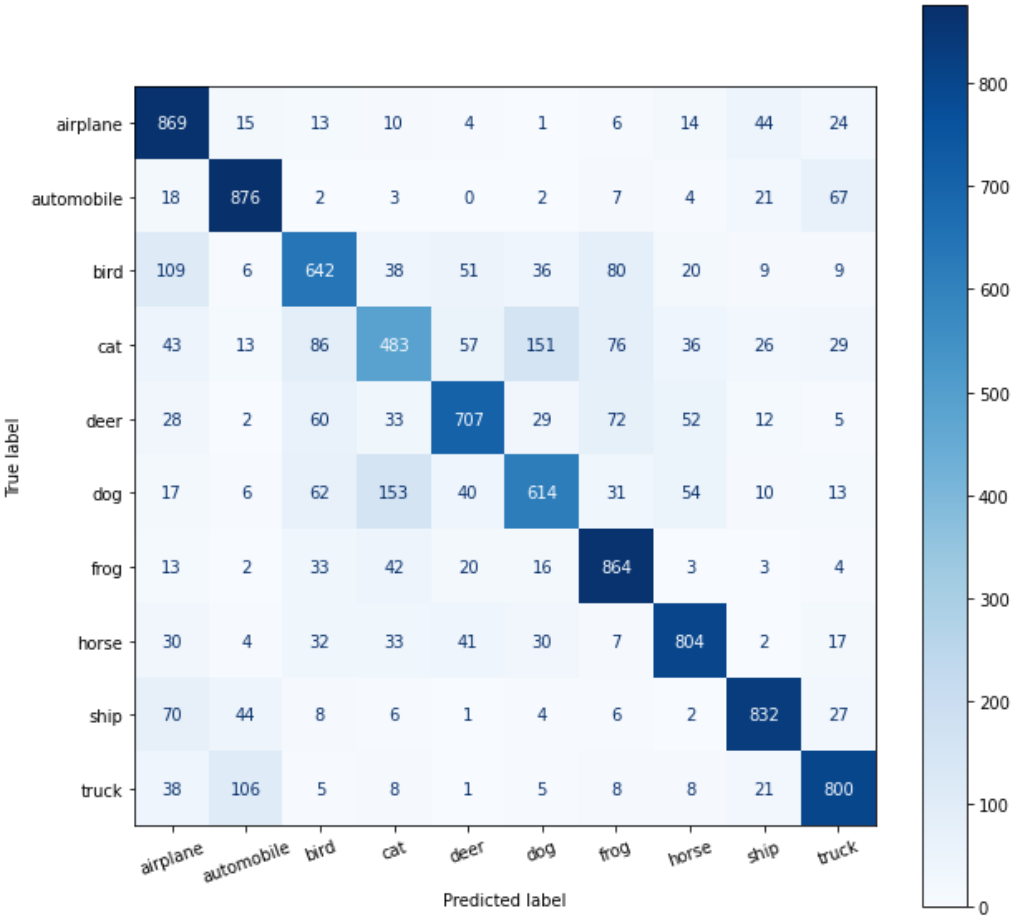
```
Test [ Loss=0.72 Acc=0.75} ]
```

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

    model.eval()

    labels = []
    probs = []

    # Q4: Fill code here.

    #I'm using a PyTorch model to predict on data from a loader, concatenating the
    #targets and predictions into tensors, and moving them to device.
    #The torch.no_grad() block disables gradient computation for efficiency.

    with torch.no_grad():

        for (x, y) in iterator:
            # moving x,y to device
            x = x.to(device)
            y = y.to(device)
            #forward pass through the model
            y_pred, _ = model(x)
            # add target to list of targets
            labels.append(y)
            # add predicted output to list of predictions
            probs.append(y_pred)

    labels = torch.cat(labels, dim = 0) # concatenate list of targets into one tensor
    probs = torch.cat(probs, dim = 0) # concatenating list of predictions into one tensor
    labels = labels.to(device)
    probs = probs.to(device)

    return labels, probs
```

```
labels, probs = get_predictions(model, test_iterator, device) # getting predictions using the model a
labels = labels.cpu().numpy() #converting to numpy array
```

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

```
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)
```



From the Confustion Matrix : We can see that the values are quite high for all, except for cats and dogs. Even then, the images were correctly labelled by the model. Hence, we can say that the model performed really well.

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.)

Question 5

In this, we are going forward with option 1 model (FInetuning) as , mentioned in the tutorial doc.

Part c of Q5 - Beatles image is also included in this

source: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

TorchVision Instance Segmentation Finetuning Tutorial

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

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

```
%%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'
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: cython in /usr/local/lib/python3.9/dist-packages (0.29.33)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI
  Cloning https://github.com/cocodataset/cocoapi.git to /tmp/pip-req-build-dcpd2c47
  Running command git clone --filter=blob:none --quiet https://github.com/cocodataset/cocoapi.git /tmp/pip-req-build-dcpd2c47
  Resolved https://github.com/cocodataset/cocoapi.git to commit 8c9bcc3cf640524c4c20a9c40e89cb6a2f2fa0e9
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools>=18.0 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (57.4.0)
Requirement already satisfied: cython>=0.27.3 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (0.29.33)
Requirement already satisfied: matplotlib>=2.1.0 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (3.5.3)
Requirement already satisfied: pyparsing>=2.2.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (3.0.9)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (0.11.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (2.8.2)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (4.39.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (8.4.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (23.0)
```

```
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.22.4)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.4.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.7->matplotlib>=2.1.0->pycocotools==2.0) (1.15.0)
Building wheels for collected packages: pycocotools
  Building wheel for pycocotools (setup.py) ... done
  Created wheel for pycocotools: filename=pycocotools-2.0-cp39-cp39-linux_x86_64.whl size=397886 sha256=0a58103fd1bd2b366f7e1f9c343350eb1020391f198d8ff7c825303fc
  Stored in directory: /tmp/pip-ephem-wheel-cache-nr9u91w_/wheels/13/c1/d6/a321055f7089f1a6af654fbf794536b196999f082a9cb68a37
Successfully built pycocotools
Installing collected packages: pycocotools
  Attempting uninstall: pycocotools
    Found existing installation: pycocotools 2.0.6
    Uninstalling pycocotools-2.0.6:
      Successfully uninstalled pycocotools-2.0.6
Successfully installed pycocotools-2.0
```

Defining the Dataset

The [torchvision reference scripts for training object detection, instance segmentation and person keypoint detection] allows for easily supporting adding new custom datasets. The dataset should inherit from the standard `torch.utils.data.Dataset` class, and implement `__len__` and `__getitem__`.

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

- image: a PIL Image of size (H, W)
- target: a dict containing the following fields
  - `boxes` (`FloatTensor[N, 4]`): the coordinates of the `N` bounding boxes in `[x0, y0, x1, y1]` format, ranging from `0` to `W` and `0` to `H`
  - `labels` (`Int64Tensor[N]`): the label for each bounding box
  - `image_id` (`Int64Tensor[1]`): an image identifier. It should be unique between all the images in the dataset, and is used during evaluation
  - `area` (`Tensor[N]`): The area of the bounding box. This is used during evaluation with the COCO metric, to separate the metric scores between small, medium and large boxes.
  - `iscrowd` (`UInt8Tensor[N]`): instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks` (`UInt8Tensor[N, H, W]`): The segmentation masks for each one of the objects
  - (optionally) `keypoints` (`FloatTensor[N, K, 3]`): For each one of the `N` objects, it contains the `K` keypoints in `[x, y, visibility]` format, defining the object. `visibility=0` means that the keypoint is not visible. Note that for data augmentation, the notion of flipping a keypoint is dependent on the data representation, and you should probably adapt `references/detection/transforms.py` for your new keypoint representation

If your model returns the above methods, they will make it work for both training and evaluation, and will use the evaluation scripts from pycocotools.

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

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

Writing a custom dataset for Penn-Fudan

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

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

```
%%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
```

```
inflating: PennFudanPed/Annotation/FudanPed00020.txt
inflating: PennFudanPed/Annotation/FudanPed00021.txt
inflating: PennFudanPed/Annotation/FudanPed00022.txt
inflating: PennFudanPed/Annotation/FudanPed00023.txt
inflating: PennFudanPed/Annotation/FudanPed00024.txt
inflating: PennFudanPed/Annotation/FudanPed00025.txt
inflating: PennFudanPed/Annotation/FudanPed00026.txt
inflating: PennFudanPed/Annotation/FudanPed00027.txt
inflating: PennFudanPed/Annotation/FudanPed00028.txt
inflating: PennFudanPed/Annotation/FudanPed00029.txt
inflating: PennFudanPed/Annotation/FudanPed00030.txt
inflating: PennFudanPed/Annotation/FudanPed00031.txt
inflating: PennFudanPed/Annotation/FudanPed00032.txt
inflating: PennFudanPed/Annotation/FudanPed00033.txt
inflating: PennFudanPed/Annotation/FudanPed00034.txt
inflating: PennFudanPed/Annotation/FudanPed00035.txt
inflating: PennFudanPed/Annotation/FudanPed00036.txt
inflating: PennFudanPed/Annotation/FudanPed00037.txt
inflating: PennFudanPed/Annotation/FudanPed00038.txt
inflating: PennFudanPed/Annotation/FudanPed00039.txt
inflating: PennFudanPed/Annotation/FudanPed00040.txt
inflating: PennFudanPed/Annotation/FudanPed00041.txt
inflating: PennFudanPed/Annotation/FudanPed00042.txt
inflating: PennFudanPed/Annotation/FudanPed00043.txt
inflating: PennFudanPed/Annotation/FudanPed00044.txt
inflating: PennFudanPed/Annotation/FudanPed00045.txt
inflating: PennFudanPed/Annotation/FudanPed00046.txt
inflating: PennFudanPed/Annotation/FudanPed00047.txt
inflating: PennFudanPed/Annotation/FudanPed00048.txt
inflating: PennFudanPed/Annotation/FudanPed00049.txt
inflating: PennFudanPed/Annotation/FudanPed00050.txt
inflating: PennFudanPed/Annotation/FudanPed00051.txt
inflating: PennFudanPed/Annotation/FudanPed00052.txt
inflating: PennFudanPed/Annotation/FudanPed00053.txt
inflating: PennFudanPed/Annotation/FudanPed00054.txt
inflating: PennFudanPed/Annotation/FudanPed00055.txt
inflating: PennFudanPed/Annotation/FudanPed00056.txt
inflating: PennFudanPed/Annotation/FudanPed00057.txt
```

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

The data is structured as follows

```
PennFudanPed/
  PedMasks/
    FudanPed00001_mask.png
    FudanPed00002_mask.png
    FudanPed00003_mask.png
    FudanPed00004_mask.png
```

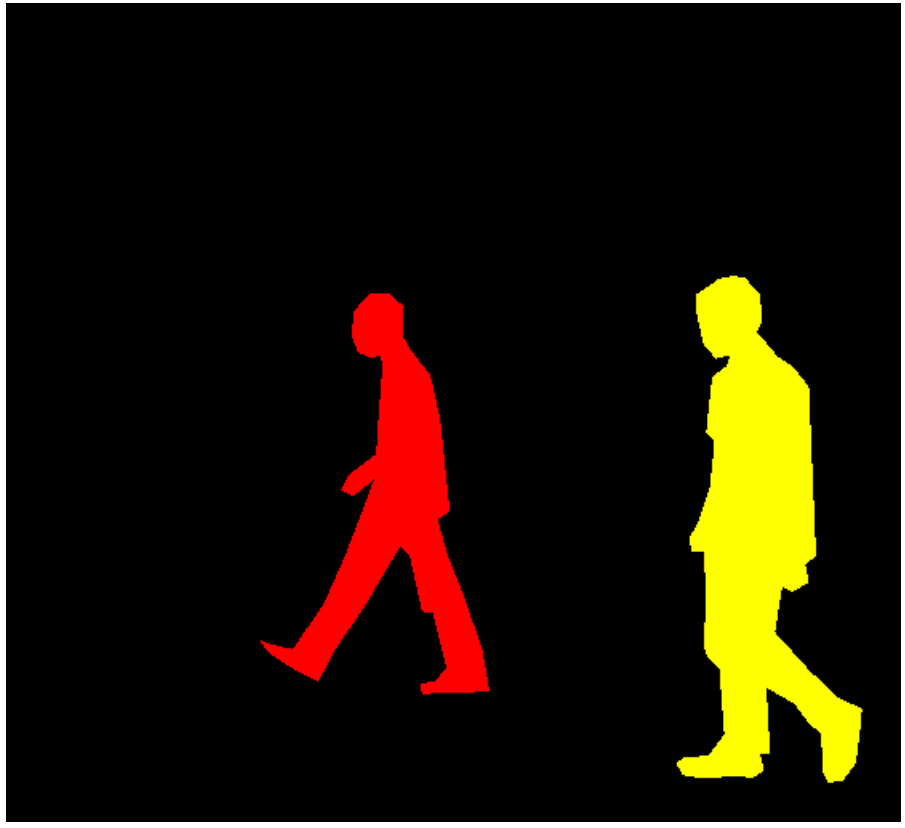
```
    ...
  PNGImages/
    FudanPed00001.png
    FudanPed00002.png
    FudanPed00003.png
    FudanPed00004.png
```

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

```
from PIL import Image
Image.open('PennFudanPed/PNGImages/FudanPed00001.png')
```

```
mask = Image.open('PennFudanPed/PedMasks/FudanPed00001_mask.png')
# each mask instance has a different color, from zero to N, where
# N is the number of instances. In order to make visualization easier,
```

```
# let's adda color palette to the mask.
mask.convert('L')
mask.putpalette([
    0, 0, 0, # black background
    255, 0, 0, # index 1 is red
    255, 255, 0, # index 2 is yellow
    255, 153, 0, # index 3 is orange
])
mask
```



So each image has a corresponding segmentation mask, where each color correspond to a different instance. Let's write a `torch.utils.data.Dataset` class for this dataset.

```
import os
import numpy as np
import torch
```

```
import torch.utils.data
from PIL import Image


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

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

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

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

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

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

```
        masks = torch.as_tensor(masks, dtype=torch.uint8)

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

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

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

        return img, target

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

That's all for the dataset. Let's see how the outputs are structured for this dataset

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

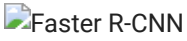
```
(<PIL.Image.Image image mode=RGB size=559x536 at 0x7FB5EC8E7130>,
 {'boxes': tensor([[159., 181., 301., 430.],
          [419., 170., 534., 485.]]),
  'labels': tensor([1, 1]),
  'masks': tensor([[[0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           ...,
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0]],

          [[0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           ...,
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0],
           [0, 0, 0,  ..., 0, 0, 0]]], dtype=torch.uint8),
  'image_id': tensor([0]),
```

```
          'area': tensor([35358., 36225.]),
          'iscrowd': tensor([0, 0])})
```

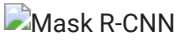
So we can see that by default, the dataset returns a `PIL.Image` and a dictionary containing several fields, including `boxes`, `labels` and `masks`.

Defining your model

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



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



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

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

1 - Finetuning from a pretrained model

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

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

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

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

2 - Modifying the model to add a different backbone

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

Here is how we would go into leveraging the functions provided by torchvision to modify a backbone.

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

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

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

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

# put the pieces together inside a FasterRCNN model
model = FasterRCNN(backbone,
```

```
                   num_classes=2,
                   rpn_anchor_generator=anchor_generator,
                   box_roi_pool=roi_pooler)
```

An Instance segmentation model for PennFudan Dataset

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

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

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


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

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

Training and evaluation functions

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

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

```
%%shell

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

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

```
Cloning into 'vision'...
remote: Enumerating objects: 314390, done.
remote: Counting objects: 100% (9147/9147), done.
remote: Compressing objects: 100% (533/533), done.
remote: Total 314390 (delta 8641), reused 9080 (delta 8594), pack-reused 305243
Receiving objects: 100% (314390/314390), 643.67 MiB | 18.13 MiB/s, done.
Resolving deltas: 100% (288778/288778), done.
Note: switching to 'v0.8.2'.

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

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 2f40a483d7 [v0.8.X] .circleci: Add Python 3.9 to CI (#3063)
```

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

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

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

Testing forward() method

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

```
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
dataset = PennFudanDataset('PennFudanPed', get_transform(train=True))
data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=2, shuffle=True, num_workers=4,
    collate_fn=utils.collate_fn
)
# For Training
images,targets = next(iter(data_loader))
images = list(image for image in images)
targets = [{k: v for k, v in t.items()} for t in targets]
output = model(images,targets)   # Returns losses and detections
# For inference
model.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
predictions = model(x)           # Returns predictions
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:208: User
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223: User
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coc
100%                                          160M/160M [00:00<00:00, 287MB/s]
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:554: Us
  warnings.warn( create warning msg(
```

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

Putting everything together

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

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

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

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

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

Now let's instantiate the model and the optimizer

```
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 = get_instance_segmentation_model(num_classes)
# move model to the right device
model.to(device)

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

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

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223: User
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-
100%                                          170M/170M [00:00<00:00, 281MB/s]
```

And now let's train the model for 10 epochs, evaluating at the end of every 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, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)
```

```
Epoch: [0]  [40/60]  eta: 0:00:15  lr: 0.003476  loss: 0.4563 (0.8822)  loss_classifier: 0.0628 (0.2069)  loss_box_reg: 0.2232 (0.2689)  loss_mask: 0.1694 (0.3
Epoch: [0]  [50/60]  eta: 0:00:07  lr: 0.004323  loss: 0.3723 (0.7839)  loss_classifier: 0.0431 (0.1753)  loss_box_reg: 0.1632 (0.2458)  loss_mask: 0.1642 (0.3
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.3096 (0.7128)  loss_classifier: 0.0394 (0.1551)  loss_box_reg: 0.1121 (0.2260)  loss_mask: 0.1585 (0.3
Epoch: [0] Total time: 0:00:41 (0.6936 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:32  model_time: 0.2476 (0.2476)  evaluator_time: 0.0142 (0.0142)  time: 0.6525  data: 0.3897  max mem: 3320
Test:  [49/50]  eta: 0:00:00  model_time: 0.1100 (0.1131)  evaluator_time: 0.0041 (0.0084)  time: 0.1234  data: 0.0041  max mem: 3320
Test: Total time: 0:00:06 (0.1383 s / it)
Averaged stats: model_time: 0.1100 (0.1131)  evaluator_time: 0.0041 (0.0084)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.685
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.703
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.312
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.743
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.743
 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 ] = 0.625
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.751
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.723
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.978
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.894
 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 ] = 0.472
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.742
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.326
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.761
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.761
 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 ] = 0.637
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.770
Epoch: [1]  [ 0/60]  eta: 0:01:07  lr: 0.005000  loss: 0.4306 (0.4306)  loss_classifier: 0.0616 (0.0616)  loss_box_reg: 0.1474 (0.1474)  loss_mask: 0.1975 (0.1
Epoch: [1]  [10/60]  eta: 0:00:30  lr: 0.005000  loss: 0.2812 (0.3042)  loss_classifier: 0.0305 (0.0375)  loss_box_reg: 0.0839 (0.1008)  loss_mask: 0.1639 (0.1
Epoch: [1]  [20/60]  eta: 0:00:24  lr: 0.005000  loss: 0.2662 (0.2952)  loss_classifier: 0.0331 (0.0386)  loss_box_reg: 0.0684 (0.0942)  loss_mask: 0.1547 (0.1
Epoch: [1]  [30/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2721 (0.3051)  loss_classifier: 0.0385 (0.0418)  loss_box_reg: 0.0794 (0.1011)  loss_mask: 0.1463 (0.1
Epoch: [1]  [40/60]  eta: 0:00:11  lr: 0.005000  loss: 0.2619 (0.2936)  loss_classifier: 0.0374 (0.0397)  loss_box_reg: 0.0794 (0.0954)  loss_mask: 0.1391 (0.1
Epoch: [1]  [50/60]  eta: 0:00:05  lr: 0.005000  loss: 0.2418 (0.2847)  loss_classifier: 0.0311 (0.0379)  loss_box_reg: 0.0732 (0.0924)  loss_mask: 0.1330 (0.1
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2710 (0.2833)  loss_classifier: 0.0369 (0.0396)  loss_box_reg: 0.0871 (0.0920)  loss_mask: 0.1336 (0.1
Epoch: [1] Total time: 0:00:35 (0.5944 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:22  model_time: 0.1580 (0.1580)  evaluator_time: 0.0036 (0.0036)  time: 0.4473  data: 0.2848  max mem: 3360
Test:  [49/50]  eta: 0:00:00  model_time: 0.1094 (0.1135)  evaluator_time: 0.0051 (0.0079)  time: 0.1279  data: 0.0065  max mem: 3360
Test: Total time: 0:00:06 (0.1372 s / it)
Averaged stats: model_time: 0.1094 (0.1135)  evaluator_time: 0.0051 (0.0079)
Accumulating evaluation results...
DONE (t=0.01s).
```

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

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

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

```
prediction
```

```
[{'boxes': tensor([[ 61.8545,  37.3171, 197.8835, 325.6659],
          [276.5266,  23.4003, 290.7968,  73.5139]], device='cuda:0'),
  'labels': tensor([1, 1], device='cuda:0'),
  'scores': tensor([0.9986, 0.5032], device='cuda:0'),
  'masks': tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            ...,
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.],
            [0., 0., 0.,  ..., 0., 0., 0.]]],


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

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

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

adding this for pdf alignment

```
Image.fromarray(img.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.

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

Looks pretty good!

Wrapping up

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

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

part 5(c) - Beatles Image

```
# downloading the image and testing the model on this image
from PIL import Image
import requests
from io import BytesIO

#Getting image from the URL
response = requests.get('https://upload.wikimedia.org/wikipedia/en/4/42/Beatles_-_Abbey_Road.jpg')
img = Image.open(BytesIO(response.content)) #Converting image to PIL Image object
img
from torchvision import transforms
cvt_im = transforms.ToTensor() #Defining transformation to convert PIL Image to Tensor
im = cvt_im(img) #Converting PIL Image to Tensor
model.eval()
with torch.no_grad(): # Disabling gradient calculation to save memory and computation
    prediction = model([im.to(device)]) ## Making prediction using the model and the input Tensor image
```

```
Image.fromarray(im.mul(255).permute(1, 2, 0).byte().numpy())
#Converting Tensor image to PIL Image by reversing normalization and changing dimensions before creating a PIL Image
```

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

```
#Converting segmentation mask to a PIL Image by reversing normalization, converting to byte array, converting to CPU and then creating PIL Image
```

This model performs really well, as after looking at the masked image, it does appear to be a human walking. Even the masking is sharpened.
After seeing the actual image, we can see that first person from left is shown in the masked result.

Comparison with backbone model results, is mentioned at the end of this pdf

Question 5

In this, we are going forward with option 2 model (Backbone) as , mentioned in the tutorial doc.

Part c of Q5 - Beatles image is also included in this

Part b of Q5 is also mentioned in this pdf

TorchVision Instance Segmentation

Source: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

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

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

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

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: cython in /usr/local/lib/python3.9/dist-packages (0.29.33)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI
  Cloning https://github.com/cocodataset/cocoapi.git to /tmp/pip-req-build-sfe014rt
  Running command git clone --filter=blob:none --quiet https://github.com/cocodataset/cocoapi.git /tmp/pip-req-build-sfe014rt
  Resolved https://github.com/cocodataset/cocoapi.git to commit 8c9bcc3cf640524c4c20a9c40e89cb6a2f2fa0e9
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools>=18.0 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (57.4.0)
Requirement already satisfied: cython>=0.27.3 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (0.29.33)
Requirement already satisfied: matplotlib>=2.1.0 in /usr/local/lib/python3.9/dist-packages (from pycocotools==2.0) (3.5.3)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (4.39.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (2.8.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (8.4.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (23.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.22.4)
```

```
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (1.4.4)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (0.11.0)
Requirement already satisfied: pyparsing>=2.2.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib>=2.1.0->pycocotools==2.0) (3.0.9)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.7->matplotlib>=2.1.0->pycocotools==2.0) (1.15.0)
Building wheels for collected packages: pycocotools
  Building wheel for pycocotools (setup.py) ... done
  Created wheel for pycocotools: filename=pycocotools-2.0-cp39-cp39-linux_x86_64.whl size=397886 sha256=2215f13f371f5eb5cb2becb8bf4197183945065822175 2c745e8a335b7c08ca9
  Stored in directory: /tmp/pip-ephem-wheel-cache-jv5gmth4/wheels/13/c1/d6/a321055f7089f1a6af654fbf794536b196999f082a9cb68a37
Successfully built pycocotools
Installing collected packages: pycocotools
  Attempting uninstall: pycocotools
    Found existing installation: pycocotools 2.0.6
    Uninstalling pycocotools-2.0.6:
      Successfully uninstalled pycocotools-2.0.6
Successfully installed pycocotools-2.0
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/gautamchitnis/cocoapi.git@cocodataset-master#subdirectory=PythonAPI
  Cloning https://github.com/gautamchitnis/cocoapi.git (to revision cocodataset-master) to /tmp/pip-req-build-rklpo0wa
  Running command git clone --filter=blob:none --quiet https://github.com/gautamchitnis/cocoapi.git /tmp/pip-req-build-rklpo0wa
  Running command git checkout -b cocodataset-master --track origin/cocodataset-master
  Switched to a new branch 'cocodataset-master'
  Branch 'cocodataset-master' set up to track remote branch 'cocodataset-master' from 'origin'.
  Resolved https://github.com/gautamchitnis/cocoapi.git to commit 20291f19c46a8d11935862bc9e449a1b72ec25ed
  Preparing metadata (setup.py) ... done
```

Defining the Dataset

The [torchvision reference scripts for training object detection, instance segmentation and person keypoint detection](https://github.com/pytorch/vision/tree/main/references/detection) allows for easily supporting adding new custom datasets. The dataset should inherit from the standard `torch.utils.data.Dataset` class, and implement `__len__` and `__getitem__`.

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

- image: a PIL Image of size (H, W)
- target: a dict containing the following fields
  - `boxes` (`FloatTensor[N, 4]`): the coordinates of the `N` bounding boxes in `[x0, y0, x1, y1]` format, ranging from `0` to `W` and `0` to `H`
  - `labels` (`Int64Tensor[N]`): the label for each bounding box
  - `image_id` (`Int64Tensor[1]`): an image identifier. It should be unique between all the images in the dataset, and is used during evaluation
  - `area` (`Tensor[N]`): The area of the bounding box. This is used during evaluation with the COCO metric, to separate the metric scores between small, medium and large boxes.
  - `iscrowd` (`UInt8Tensor[N]`): instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks` (`UInt8Tensor[N, H, W]`): The segmentation masks for each one of the objects

- (optionally) `keypoints` (`FloatTensor[N, K, 3]`): For each one of the `N` objects, it contains the `K` keypoints in `[x, y, visibility]` format, defining the object. `visibility=0` means that the keypoint is not visible. Note that for data augmentation, the notion of flipping a keypoint is dependent on the data representation, and you should probably adapt `references/detection/transforms.py` for your new keypoint representation

If your model returns the above methods, they will make it work for both training and evaluation, and will use the evaluation scripts from pycocotools.

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

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

Writing a custom dataset for Penn-Fudan

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

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

```
%%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
```

```
inflating: PennFudanPed/PNGImages/PennPed00056.png
inflating: PennFudanPed/PNGImages/PennPed00057.png
inflating: PennFudanPed/PNGImages/PennPed00058.png
inflating: PennFudanPed/PNGImages/PennPed00059.png
inflating: PennFudanPed/PNGImages/PennPed00060.png
inflating: PennFudanPed/PNGImages/PennPed00061.png
inflating: PennFudanPed/PNGImages/PennPed00062.png
inflating: PennFudanPed/PNGImages/PennPed00063.png
inflating: PennFudanPed/PNGImages/PennPed00064.png
inflating: PennFudanPed/PNGImages/PennPed00065.png
inflating: PennFudanPed/PNGImages/PennPed00066.png
inflating: PennFudanPed/PNGImages/PennPed00067.png
inflating: PennFudanPed/PNGImages/PennPed00068.png
inflating: PennFudanPed/PNGImages/PennPed00069.png
inflating: PennFudanPed/PNGImages/PennPed00070.png
inflating: PennFudanPed/PNGImages/PennPed00071.png
inflating: PennFudanPed/PNGImages/PennPed00072.png
inflating: PennFudanPed/PNGImages/PennPed00073.png
inflating: PennFudanPed/PNGImages/PennPed00074.png
inflating: PennFudanPed/PNGImages/PennPed00075.png
inflating: PennFudanPed/PNGImages/PennPed00076.png
inflating: PennFudanPed/PNGImages/PennPed00077.png
inflating: PennFudanPed/PNGImages/PennPed00078.png
inflating: PennFudanPed/PNGImages/PennPed00079.png
inflating: PennFudanPed/PNGImages/PennPed00080.png
inflating: PennFudanPed/PNGImages/PennPed00081.png
inflating: PennFudanPed/PNGImages/PennPed00082.png
inflating: PennFudanPed/PNGImages/PennPed00083.png
inflating: PennFudanPed/PNGImages/PennPed00084.png
inflating: PennFudanPed/PNGImages/PennPed00085.png
inflating: PennFudanPed/PNGImages/PennPed00086.png
inflating: PennFudanPed/PNGImages/PennPed00087.png
inflating: PennFudanPed/PNGImages/PennPed00088.png
inflating: PennFudanPed/PNGImages/PennPed00089.png
inflating: PennFudanPed/PNGImages/PennPed00090.png
inflating: PennFudanPed/PNGImages/PennPed00091.png
inflating: PennFudanPed/PNGImages/PennPed00092.png
inflating: PennFudanPed/PNGImages/PennPed00093.png
inflating: PennFudanPed/PNGImages/PennPed00094.png
inflating: PennFudanPed/PNGImages/PennPed00095.png
inflating: PennFudanPed/PNGImages/PennPed00096.png
inflating: PennFudanPed/readme.txt
```

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

The data is structured as follows

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

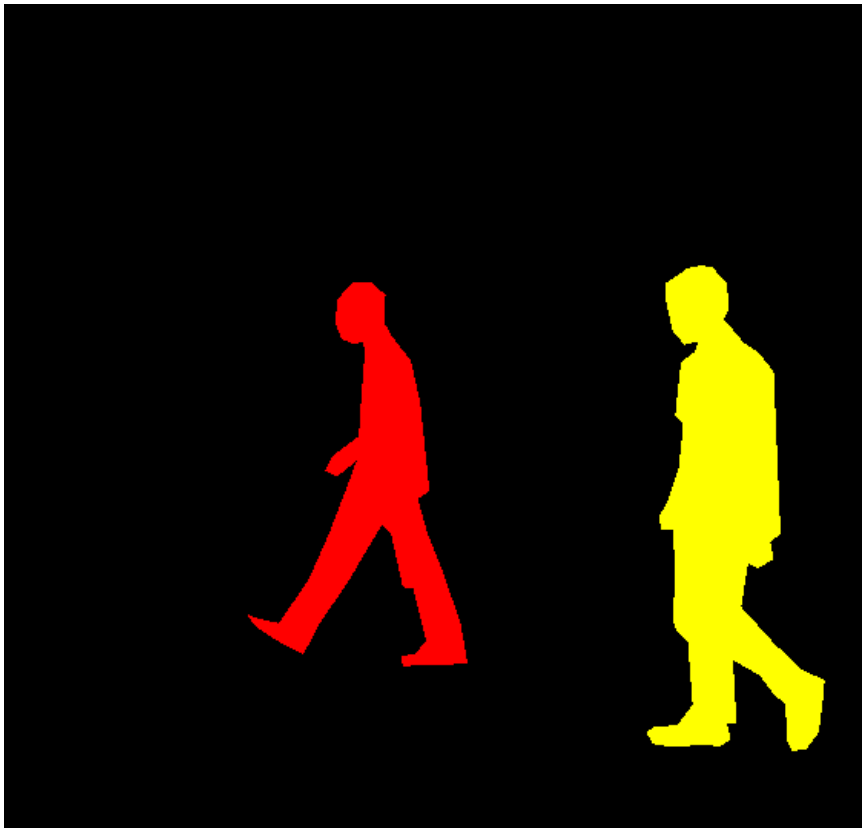
```
    FudanPed00003_mask.png
    FudanPed00004_mask.png
    ...
  PNGImages/
    FudanPed00001.png
    FudanPed00002.png
    FudanPed00003.png
    FudanPed00004.png
```

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

```
from PIL import Image
Image.open('PennFudanPed/PNGImages/FudanPed00001.png')
```

```
mask = Image.open('PennFudanPed/PedMasks/FudanPed00001_mask.png')
# each mask instance has a different color, from zero to N, where
```

```
# N is the number of instances. In order to make visualization easier,
# let's adda color palette to the mask.
mask.convert('L')
mask.putpalette([
    0, 0, 0, # black background
    255, 0, 0, # index 1 is red
    255, 255, 0, # index 2 is yellow
    255, 153, 0, # index 3 is orange
])
mask
```



So each image has a corresponding segmentation mask, where each color correspond to a different instance. Let's write a `torch.utils.data.Dataset` class for this dataset.

```
import os
import numpy as np
import torch
import torch.utils.data
```

```
from PIL import Image


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

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

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

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

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

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

        image_id = torch.tensor([idx])
```

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

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

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

        return img, target

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

That's all for the dataset. Let's see how the outputs are structured for this dataset

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

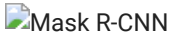
```
(<PIL.Image.Image image mode=RGB size=559x536 at 0x7F6D3D22DA90>,
 {'boxes': tensor([[159., 181., 301., 430.],
         [419., 170., 534., 485.]]),
  'labels': tensor([1, 1]),
  'masks': tensor([[[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]],

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

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

Defining your model

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

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



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

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

2 - Modifying the model to add a different backbone

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

Here is how we would go into leveraging the functions provided by torchvision to modify a backbone.

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

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

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

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

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

An Instance segmentation model for PennFudan Dataset

Here,we will be following approach number 2.

Reference: https://pytorch.org/vision/0.12/_modules/torchvision/models/detection/mask_rcnn.html

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

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection import MaskRCNN
from torchvision.models.detection.rpn import AnchorGenerator

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

    backbone = torchvision.models.mobilenet_v2(weights="DEFAULT").features
    backbone.out_channels = 1280
    anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                       aspect_ratios=((0.5, 1.0, 2.0),))
    roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0'],
                                                    output_size=7,
                                                    sampling_ratio=2)
```

```
    # put the pieces together inside a MaskRCNN model
    model = MaskRCNN(backbone,
                     num_classes=2,
                     rpn_anchor_generator=anchor_generator,
                     box_roi_pool=roi_pooler)

    return model
```

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

Training and evaluation functions

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

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

```
%%shell

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

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

```
Cloning into 'vision'...
remote: Enumerating objects: 314401, done.
remote: Counting objects: 100% (9158/9158), done.
remote: Compressing objects: 100% (552/552), done.
remote: Total 314401 (delta 8642), reused 9071 (delta 8586), pack-reused 305243
Receiving objects: 100% (314401/314401), 645.88 MiB | 18.57 MiB/s, done.
Resolving deltas: 100% (288780/288780), done.
Note: switching to 'v0.8.2'.

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

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

  git switch -c <new-branch-name>
```

```
Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 2f40a483d7 [v0.8.X] .circleci: Add Python 3.9 to CI (#3063)
```

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

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


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

Testing forward() method

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

```
# model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
backbone = torchvision.models.mobilenet_v2(weights="DEFAULT").features
backbone.out_channels = 1280
anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                   aspect_ratios=((0.5, 1.0, 2.0),))

roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0'],
                                                output_size=7,
                                                sampling_ratio=2)

model = FasterRCNN(backbone,
                   num_classes=2,
                   rpn_anchor_generator
                   =anchor_generator,
                   box_roi_pool=roi_pooler)
```

```
dataset = PennFudanDataset('PennFudanPed', get_transform(train=True))
data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=2, shuffle=True, num_workers=4,
    collate_fn=utils.collate_fn
)
# For Training
images,targets = next(iter(data_loader))
images = list(image for image in images)
targets = [{k: v for k, v in t.items()} for t in targets]
output = model(images,targets)   # Returns losses and detections
# For inference
model.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
predictions = model(x)           # Returns predictions
```

```
Downloading: "https://download.pytorch.org/models/mobilenet_v2-7ebf99e0.pth" to /root/.cache
100%                                      13.6M/13.6M [00:01<00:00, 14.6MB/s]
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:554: UserWarning: This
  warnings.warn(_create_warning_msg(
```

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

Putting everything together

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

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

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

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

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

Now let's instantiate the model and the optimizer

```
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 = get_instance_segmentation_model(num_classes)
# move model to the right device
model.to(device)

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

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

And now let's train the model for 10 epochs, evaluating at the end of every 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, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.015
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.241
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.149
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.335
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.342
 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 ] = 0.062
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.362
Epoch: [9]  [ 0/60]  eta: 0:01:04  lr: 0.000005  loss: 0.8007 (0.8007)  loss_classifier: 0.1243 (0.1243)  loss_box_reg: 0.2248 (0.2248)  loss_mask: 0.3517 (0.3517)  l
Epoch: [9]  [10/60]  eta: 0:00:23  lr: 0.000005  loss: 0.5882 (0.6202)  loss_classifier: 0.0875 (0.0902)  loss_box_reg: 0.1376 (0.1447)  loss_mask: 0.3050 (0.3212)  l
Epoch: [9]  [20/60]  eta: 0:00:18  lr: 0.000005  loss: 0.5785 (0.6372)  loss_classifier: 0.0875 (0.0916)  loss_box_reg: 0.1376 (0.1467)  loss_mask: 0.3050 (0.3369)  l
Epoch: [9]  [30/60]  eta: 0:00:13  lr: 0.000005  loss: 0.5785 (0.6280)  loss_classifier: 0.0897 (0.0925)  loss_box_reg: 0.1280 (0.1441)  loss_mask: 0.3281 (0.3330)  l
Epoch: [9]  [40/60]  eta: 0:00:08  lr: 0.000005  loss: 0.5284 (0.6259)  loss_classifier: 0.0740 (0.0917)  loss_box_reg: 0.1039 (0.1412)  loss_mask: 0.3281 (0.3347)  l
Epoch: [9]  [50/60]  eta: 0:00:04  lr: 0.000005  loss: 0.5610 (0.6270)  loss_classifier: 0.0855 (0.0920)  loss_box_reg: 0.1288 (0.1408)  loss_mask: 0.3249 (0.3353)  l
Epoch: [9]  [59/60]  eta: 0:00:00  lr: 0.000005  loss: 0.6110 (0.6317)  loss_classifier: 0.0894 (0.0930)  loss_box_reg: 0.1437 (0.1443)  loss_mask: 0.3248 (0.3339)  l
Epoch: [9] Total time: 0:00:26 (0.4449 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:32  model_time: 0.1167 (0.1167)  evaluator_time: 0.0047 (0.0047)  time: 0.6568  data: 0.5344  max mem: 5687
Test:  [49/50]  eta: 0:00:00  model_time: 0.0545 (0.0607)  evaluator_time: 0.0120 (0.0153)  time: 0.0753  data: 0.0042  max mem: 5687
Test: Total time: 0:00:04 (0.0986 s / it)
Averaged stats: model_time: 0.0545 (0.0607)  evaluator_time: 0.0120 (0.0153)
Accumulating evaluation results...
DONE (t=0.02s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.305
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.757
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.159
 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 ] = 0.018
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.326
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.181
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.446
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.460
 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 ] = 0.138
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.483
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.225
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.690
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.046
 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 ] = 0.006
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.246
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.145
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.326
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.339
 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 ] = 0.075
```

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

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

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

```
prediction
```

```
[{'boxes': tensor([[ 91.7069,  51.5380, 171.3164, 308.8423],
          [ 76.4305,  86.0414, 182.9175, 195.9882],
          [291.3152,   0.0000, 292.0000, 168.8823],
          [ 76.6169,  99.4069, 105.8228, 201.5214],
          [ 54.5581,  87.6692, 208.4057, 301.6768]], device='cuda:0'),
  'labels': tensor([1, 1, 1, 1, 1], device='cuda:0'),
  'scores': tensor([0.7112, 0.3475, 0.2464, 0.1384, 0.1219], device='cuda:0'),
  'masks': tensor([[[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],


          [[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],


          [[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.1665],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.2143],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.2621],
           ...,
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
           [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],
```

```
 [[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   ...,
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],


 [[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   ...,
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
   [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]]],
device='cuda:0')}]
```

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

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

```
Image.fromarray(img.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.

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

Part- 5b conclusion

After training both fine tuning and backbone models on our dataset for 10 epochs, we can see that the fine tuning model is outperforming the backbone model after just 10 epochs. Specifically, we see that the bounding box generated by the fine tuning model is superior to that of the backbone model. Additionally, we can also see that the finetuning model gives more accurate masking results than the backbone model.

Part -5c backbone

+ Code + Text

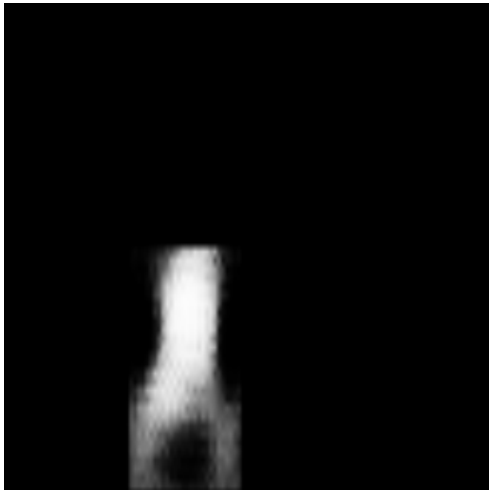
Testing the model on the Beatles Image

```
from PIL import Image
import requests
from io import BytesIO
#Getting image from the URL
response = requests.get('https://upload.wikimedia.org/wikipedia/en/4/42/Beatles_-_Abbey_Road.jpg')
img = Image.open(BytesIO(response.content)) #Converting image to PIL Image object
img
from torchvision import transforms
cvt_im = transforms.ToTensor() #Defining transformation to convert PIL Image to Tensor
```

```
im = cvt_im(img) #Converting PIL Image to Tensor
model.eval()
with torch.no_grad(): # Disabling gradient calculation to save memory and computation
    prediction = model([im.to(device)]) ## Making prediction using the model and the input Tensor image
```

```
Image.fromarray(im.mul(255).permute(1, 2, 0).byte().numpy())
#Converting Tensor image to PIL Image by reversing normalization and changing dimensions before creating a PIL Image
```

```
Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())
#Converting segmentation mask to a PIL Image by reversing normalization, converting to byte array, converting to CPU and then creating PIL Image
```



Part 5c final conclusion

Upon evaluation of the "Beatles" image masking results, the results indicate that the model with fine-tuning shows better performance compared to the backbone model. Specifically, the fine-tuned model showcased a well-defined masking of the first person in the image, resulting in a sharper and more precise masking and as a result, the masked result does appear to be of a person walking. In contrast, the backbone model lacks clarity, making it difficult to identify the white blob as a person walking due to the image's blurryness and imprecision.

✓ 0s completed at 5:43 PM