DLHW1

yz5946 Sky Zhou

October 2024

1 Problem 1

a

First we revisit the definition of IOU:

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

Then by definition, $A \subseteq A \cup B, B \subseteq A \cup B, A \cap B \subseteq A, A \cap B \subseteq B$.

By transitivity of subset, we have $(A \cap B) \subseteq (A \cup B)$.

Therefore, $0 \leq |A \cap B| \leq |A \cup B|$

$$0 \leq J(A, B) = \frac{|A \cap B|}{|A \cup B|} \leq 1$$

b

Suppose we have 2 boxes, A and B, their coordinates are:

$$A : (x^A_{min},\ y^A_{min},\ x^A_{max},\ y^A_{max})$$

$$B : (x^B_{min},\ y^B_{min},\ x^B_{max},\ y^B_{max})$$

Then we have:

$$\begin{aligned} x^{min}_{int} &= \max(x^A_{min},\ x^B_{min}) \\ y^{min}_{int} &= \max(y^A_{min},\ y^B_{min}) \\ x^{max}_{int} &= \min(x^A_{max},\ x^B_{max}) \\ y^{max}_{int} &= \min(y^A_{max},\ y^B_{max}) \end{aligned}$$

With these pieces of information, we can compute intersection:

$$A \cap B = \max(0,\ x^{max}_{int} - x^{min}_{int}) \times \max(0,\ y^{max}_{int} - y^{min}_{int})$$

By definition of Union and Intersection, we also have

$$A \cup B = A + B - A \cap B$$

where:

$$\begin{aligned} A &= (x^A_{max} - x^A_{min}) \times (y^A_{max} - y^A_{min}) \\ B &= (x^B_{max} - x^B_{min}) \times (y^B_{max} - y^B_{min}) \end{aligned}$$

Therefore, $IOU(A, B)$ can be expressed as:

$$\frac{\max(0,\ x^{max}_{int} - x^{min}_{int}) \times \max(0,\ y^{max}_{int} - y^{min}_{int})}{(x^A_{max} - x^A_{min}) \times (y^A_{max} - y^A_{min}) + (x^B_{max} - x^B_{min}) \times (y^B_{max} - y^B_{min}) - \max(0,\ x^{max}_{int} - x^{min}_{int}) \times \max(0,\ y^{max}_{int} - y^{min}_{int})}$$

It is obvious that the calculation involves Max and Min functions. Min and Max themselves are continuous but not differentiable at points where their arguments are equal. For example, $\max(a, b)$ is not differentiable when $a = b$. Therefore, the IOU function expressed with Min and Max is also not differentiable.

The entire IOU expressed with Min and Max function also may not be continuous at all. When boxes do not overlap ($\text{Area}_{\text{int}} = 0$), small changes in coordinates do not affect the IoU.

2 Problem 2

Convolutional Layers

For Convolutional Layers, the formula to compute the number of weights is:

$$\textbf{Number of Weights} = \textbf{Kernel Height} \times \textbf{Kernel Width} \times \textbf{Number of Input Channels} \times \textbf{Number of Filters}$$

And

$$\textbf{Number of biases} = \textbf{Number of Filters}$$

Therefore,

$$\textbf{Number of learnable parameters} = \textbf{Number of weights} + \textbf{Number of biases}$$

We have the kernel size for our convolution layers to be 3×3

Layer 1: Conv 3 to 64

$$\text{Weights} = 3 \times 3 \times 3 \times 64 = 1728$$

$$\text{Biases} = 64$$

$$\text{Total Parameters} = 1728 + 64 = 1792$$

Layer 2: Conv 64 to 64

$$\text{Weights} = 3 \times 3 \times 64 \times 64 = 36864$$

$$\text{Biases} = 64$$

$$\text{Total Parameters} = 36864 + 64 = 36928$$

Layer 3: Conv 64 to 128

$$\text{Weights} = 3 \times 3 \times 64 \times 128 = 73728$$

$$\text{Biases} = 128$$

$$\text{Total Parameters} = 73728 + 128 = 73856$$

Layer 4: Conv 128 to 128

$$\text{Weights} = 3 \times 3 \times 128 \times 128 = 147456$$

$$\text{Biases} = 128$$

$$\text{Total Parameters} = 147456 + 128 = 147584$$

Layer 5: Conv 128 to 256

$$\text{Weights} = 3 \times 3 \times 128 \times 256 = 294912$$

$$\text{Biases} = 256$$

$$\text{Total Parameters} = 294912 + 256 = 295168$$

Layer 6 - 7: Conv 256 to 256

$$\text{Weights} = 3 \times 3 \times 256 \times 256 = 589824$$

$$\text{Biases} = 256$$

$$\text{Total Parameters} = 589824 + 256 = 590080$$

Layer 8: Conv 256 to 512

$$\text{Weights} = 3 \times 3 \times 256 \times 512 = 1179648$$

$$\text{Biases} = 512$$

$$\text{Total Parameters} = 1179648 + 512 = 1180160$$

Layer 9 - 13: Conv 512 to 512

$$\text{Weights} = 3 \times 3 \times 512 \times 512 = 2359296$$

$$\text{Biases} = 512$$

$$\text{Total Parameters} = 2359296 + 512 = 2359808$$

Fully Connected Layers

For Fully Connected Layers, the formula to compute the number of weights is:

$$\textbf{Number of Weights} = \textbf{Number of Input Neurons} \times \textbf{Number of Output Neuron}$$

And

$$\textbf{Number of biases} = \textbf{Number of Output Neuron}$$

Therefore,

$$\textbf{Number of learnable parameters} = \textbf{Number of weights} + \textbf{Number of biases}$$

Layer 14: FC (Input = $7 \times 7 \times 512$ = 25088, Output = 4096)

$$\text{Weights} = 25088 \times 4096 = 102760192$$

$$\text{Biases} = 4096$$

$$\text{Total Parameters} = 102760192 + 4096 = 102764544$$

Layer 15: FC (Input = 4096, Output = 4096)

$$\text{Weights} = 4096 \times 4096 = 16777216$$

$$\text{Biases} = 4096$$

$$\text{Total Parameters} = 16777216 + 4096 = 16781312$$

Layer 16: FC (Input = 4096, Output = 1000)

$$\text{Weights} = 4096 \times 1,000 = 4096000$$

$$\text{Biases} = 1000$$

$$\text{Total Parameters} = 4096,000 + 1000 = 4097000$$

Total Number of Learnable Parameters

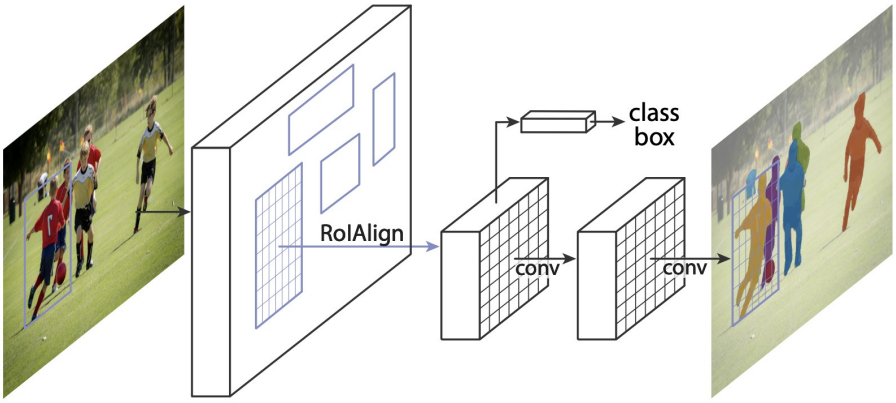
$$\begin{aligned}\text{Total Parameters for Convolutional Layers} &= 1792 + 36928 + 73856 + 147584 + 295168 \\ &\quad + 2 \times 590080 + 1180160 + 5 \times 2359,808 \\ &= 14,714,688\end{aligned}$$

$$\text{Total Parameters for Fully Connected Layers} = 102764544{+}16781312{+}4097000 = 123,642,856$$

$$\text{Total} = 14,714,688 + 123,642,856 = 138,357,544$$

3 Problem 3

I only incorporate necessary codes that are directly relevant to the training results. I am happy to provide the complete ipynb file if necessary.



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

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

1 - Finetuning from a pretrained model

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

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

# load a model pre-trained on COCO
model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT"
)

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

2 - Modifying the model to add a different backbone

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

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

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

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

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

Object detection and instance segmentation model for PennFudan Dataset

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

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

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


def get_model_instance_segmentation(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model =
torchvision.models.detection.maskrcnn_resnet50_fpn(weights="DEFAULT")

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

    # now get the number of input features for the mask classifier
    in_features_mask =
model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(
        in_features_mask,
        hidden_layer,
        num_classes
    )

    return model
```

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

Putting everything together

In `references/detection/`, we have a number of helper functions to simplify training and evaluating detection models. Here, we will use `references/detection/engine.py` and `references/detection/utils.py`. Just download everything under `references/detection` to your folder and use them here. On Linux if you have `wget`, you can download them using below commands:

Let's write some helper functions for data augmentation / transformation:

```
from torchvision.transforms import v2 as T


def get_transform(train):
    transforms = []
    if train:
        transforms.append(T.RandomHorizontalFlip(0.5))
    transforms.append(T.ToDtype(torch.float, scale=True))
    transforms.append(T.ToPureTensor())
    return T.Compose(transforms)
```

Testing `forward()` method (Optional)

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

```
import os
print(os.getcwd())

import utils

model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT"
)
dataset = PennFudanDataset('Downloads/PennFudanPed/PennFudanPed',
get_transform(train=True))
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True,
    collate_fn=utils.collate_fn
)

# For Training
images, targets = next(iter(data_loader))
images = list(image for image in images)
targets = [{k: v for k, v in t.items()} for t in targets]
output = model(images, targets)  # Returns losses and detections
print(output)

# For inference
model.eval()
x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
predictions = model(x)  # Returns predictions
print(predictions[0])
```

```
C:\Users\Sky
{'loss_classifier': tensor(0.2733, grad_fn=<NllLossBackward0>),
'loss_box_reg': tensor(0.0791, grad_fn=<DivBackward0>),
'loss_objectness': tensor(0.0342,
grad_fn=<BinaryCrossEntropyWithLogitsBackward0>), 'loss_rpn_box_reg':
tensor(0.0070, grad_fn=<DivBackward0>)}
{'boxes': tensor([], size=(0, 4), grad_fn=<StackBackward0>), 'labels':
tensor([], dtype=torch.int64), 'scores': tensor([],
grad_fn=<IndexBackward0>)}
```

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

```
from utils import *
from engine import train_one_epoch, evaluate
import torch


# train on the GPU or on the CPU, if a GPU is not available
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
print(device)

# our dataset has two classes only - background and person
num_classes = 2
# use our dataset and defined transformations
dataset = PennFudanDataset('Downloads/PennFudanPed/PennFudanPed',
get_transform(train=True))
dataset_test = PennFudanDataset('Downloads/PennFudanPed/PennFudanPed',
get_transform(train=False))

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

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

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

```python
# get the model using our helper function
model = get_model_instance_segmentation(num_classes)

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

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

# and a learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.1
)

# let's train it just for 2 epochs
num_epochs = 10

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

print("That's it!")
```

```
cpu
Epoch: [0]  [ 0/60]  eta: 0:03:44  lr: 0.000090  loss: 4.9049 (4.9049)
loss_classifier: 0.7998 (0.7998)  loss_box_reg: 0.2327 (0.2327)
loss_mask: 3.8580 (3.8580)  loss_objectness: 0.0095 (0.0095)
loss_rpn_box_reg: 0.0048 (0.0048)  time: 3.7488  data: 0.0110
Epoch: [0]  [10/60]  eta: 0:03:08  lr: 0.000936  loss: 1.7355 (2.7553)
loss_classifier: 0.4271 (0.4706)  loss_box_reg: 0.2327 (0.2438)
loss_mask: 1.0918 (2.0143)  loss_objectness: 0.0178 (0.0221)
loss_rpn_box_reg: 0.0042 (0.0044)  time: 3.7660  data: 0.0101
Epoch: [0]  [20/60]  eta: 0:02:26  lr: 0.001783  loss: 1.1201 (1.7939)
loss_classifier: 0.2250 (0.3326)  loss_box_reg: 0.2105 (0.2345)
loss_mask: 0.5033 (1.1987)  loss_objectness: 0.0160 (0.0223)
loss_rpn_box_reg: 0.0045 (0.0059)  time: 3.6523  data: 0.0119
```

```
loss_mask: 0.1087 (0.1097)  loss_objectness: 0.0003 (0.0006)
loss_rpn_box_reg: 0.0014 (0.0023)  time: 3.8517  data: 0.0103
Epoch: [9]  [50/60]  eta: 0:00:37  lr: 0.000005  loss: 0.1446 (0.1699)
loss_classifier: 0.0178 (0.0226)  loss_box_reg: 0.0225 (0.0332)
loss_mask: 0.1070 (0.1109)  loss_objectness: 0.0003 (0.0008)
loss_rpn_box_reg: 0.0014 (0.0024)  time: 3.7888  data: 0.0106
Epoch: [9]  [59/60]  eta: 0:00:03  lr: 0.000005  loss: 0.1454 (0.1727)
loss_classifier: 0.0147 (0.0230)  loss_box_reg: 0.0201 (0.0340)
loss_mask: 0.1067 (0.1123)  loss_objectness: 0.0003 (0.0009)
loss_rpn_box_reg: 0.0011 (0.0025)  time: 3.7858  data: 0.0109
Epoch: [9] Total time: 0:03:47 (3.7843 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:02:36  model_time: 3.1184 (3.1184)
evaluator_time: 0.0020 (0.0020)  time: 3.1286  data: 0.0083
Test:  [49/50]  eta: 0:00:02  model_time: 2.5000 (2.5547)
evaluator_time: 0.0020 (0.0025)  time: 2.5618  data: 0.0067
Test: Total time: 0:02:08 (2.5645 s / it)
Averaged stats: model_time: 2.5000 (2.5547)  evaluator_time: 0.0020
(0.0025)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.802
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.986
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.921
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.314
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.417
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.814
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.318
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.853
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.853
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.800
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.778
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.859
```

```
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.740
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.977
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.868
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.217
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.262
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.757
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.296
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.782
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.782
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.550
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.794
That's it!
```

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

But what do the predictions look like? Let's take one image in the dataset and verify

```
import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes,
draw_segmentation_masks


image =
read_image("Downloads/PennFudanPed/PennFudanPed/PNGImages/FudanPed0004
6.png")
eval_transform = get_transform(train=False)

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

```
image = (255.0 * (image - image.min()) / (image.max() -
image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels,
colors="red")
masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5,
colors="blue")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
plt.show()
```

Implementation of Training another model (model2) with option2, a different backbone

```
import torch
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator
from torchvision import models
from torchvision.models import resnet50
import torch.nn as nn

# 1. Load a pre-trained ResNet-50 model as the backbone
backbone = torchvision.models.mobilenet_v2(weights="DEFAULT").features
# ``FasterRCNN`` needs to know the number of
# output channels in a backbone. For mobilenet_v2, it's 1280
# so we need to add it here
backbone.out_channels = 1280

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

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

# put the pieces together inside a Faster-RCNN model
model2 = FasterRCNN(
    backbone,
    num_classes=2,
    rpn_anchor_generator=anchor_generator,
    box_roi_pool=roi_pooler
)
```

```
# Reuse the training in option 1
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
model2.to(device)

dataset = PennFudanDataset('Downloads/PennFudanPed/PennFudanPed',
get_transform(train=True))
dataset_test = PennFudanDataset('Downloads/PennFudanPed/PennFudanPed',
get_transform(train=False))
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True,
    collate_fn=collate_fn
)

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

params = [p for p in model2.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005, momentum=0.9,
weight_decay=0.0005)

lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.1)

num_epochs = 10
for epoch in range(num_epochs):
    train_one_epoch(model2, optimizer, data_loader, device, epoch,
print_freq=10)
    lr_scheduler.step()
    # Evaluate on the test dataset
    evaluate(model2, data_loader_test, device=device)
print("That's it!")
```

```
Epoch: [0]  [ 0/85]  eta: 0:05:57  lr: 0.000064  loss: 1.4703 (1.4703)
loss_classifier: 0.6877 (0.6877)  loss_box_reg: 0.0408 (0.0408)
loss_objectness: 0.6924 (0.6924)  loss_rpn_box_reg: 0.0494 (0.0494)
time: 4.2085  data: 0.0125
Epoch: [0]  [10/85]  eta: 0:05:02  lr: 0.000659  loss: 1.3922 (1.4011)
loss_classifier: 0.6617 (0.6344)  loss_box_reg: 0.0408 (0.0434)
loss_objectness: 0.6847 (0.6826)  loss_rpn_box_reg: 0.0368 (0.0407)
time: 4.0306  data: 0.0118
Epoch: [0]  [20/85]  eta: 0:04:20  lr: 0.001254  loss: 1.2415 (1.2378)
```

```
loss_objectness: 0.0248 (0.0391)  loss_rpn_box_reg: 0.0139 (0.0187)
time: 3.9765  data: 0.0105
Epoch: [9]  [84/85]  eta: 0:00:04  lr: 0.000005  loss: 0.2061 (0.2687)
loss_classifier: 0.0632 (0.0800)  loss_box_reg: 0.1033 (0.1314)
loss_objectness: 0.0292 (0.0387)  loss_rpn_box_reg: 0.0144 (0.0186)
time: 3.9928  data: 0.0106
Epoch: [9] Total time: 0:05:41 (4.0161 s / it)
creating index...
index created!
Test:  [  0/170]  eta: 0:02:21  model_time: 0.8221 (0.8221)
evaluator_time: 0.0010 (0.0010)  time: 0.8321  data: 0.0090
Test:  [100/170]  eta: 0:00:53  model_time: 0.7347 (0.7569)
evaluator_time: 0.0010 (0.0009)  time: 0.7469  data: 0.0088
Test:  [169/170]  eta: 0:00:00  model_time: 0.6956 (0.7316)
evaluator_time: 0.0010 (0.0009)  time: 0.6942  data: 0.0050
Test: Total time: 0:02:05 (0.7392 s / it)
Averaged stats: model_time: 0.6956 (0.7316)  evaluator_time: 0.0010
(0.0009)
Accumulating evaluation results...
DONE (t=0.02s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.399
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.844
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.279
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.087
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.426
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.221
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.522
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.536
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small |
maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium |
maxDets=100 ] = 0.281
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.563
That's it!
```

```
import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes,
```

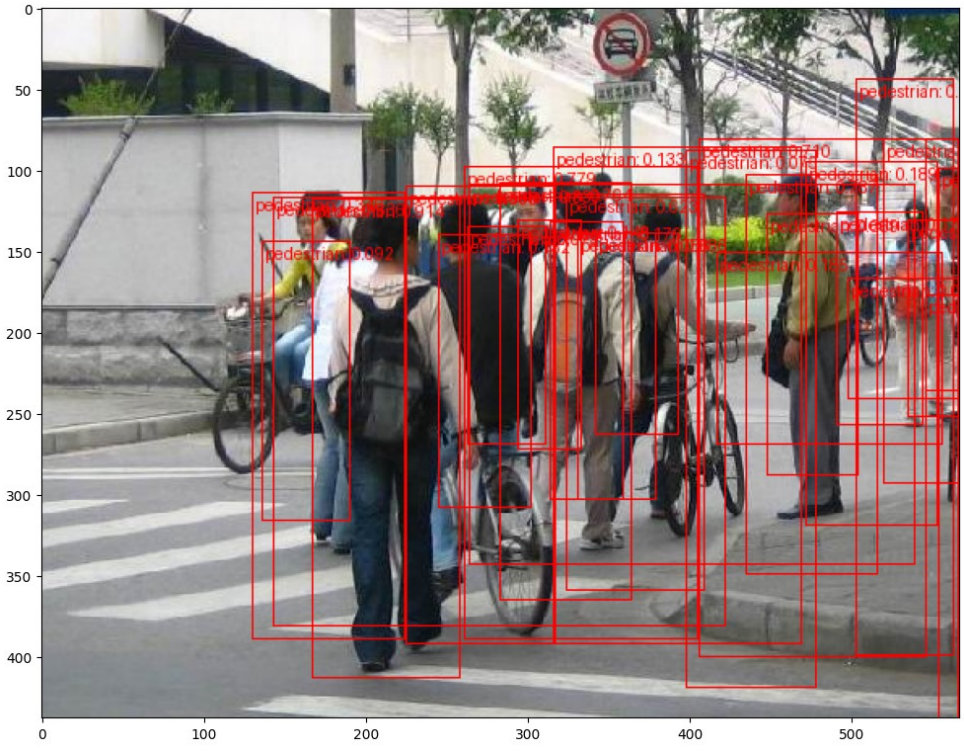
```
draw_segmentation_masks


image =
read_image("Downloads/PennFudanPed/PennFudanPed/PNGImages/FudanPed0004
6.png")
eval_transform = get_transform(train=False)

model2.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model2([x, ])
    pred = predictions[0]


image = (255.0 * (image - image.min()) / (image.max() -
image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels,
colors="red")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
plt.show()
```


0
50
100
150
200
250
300
350
400
0
100
200
300
400
500

Comparison of result

Option 1

The fine-tuned model detects bounding boxes, but the density is visibly reduced compared to option 2. And it seems identifies the object more accurately and the noise is lower. This is because this model makes use of segmentation masks, making it easier to visually distinguish each person, and the bounding boxes are fewer, which might be a result of more improved feature extraction.

Option 2

The backbone substitution solution detects a large number of bounding boxes around the pedestrians. There are many overlapping boxes, which makes it difficult to clearly distinguish individual instances. This might be due to the lack of incorporation of masked images.

Therefore, I trained a new model with different backbone but also incorporates Masked segementation.

```
import torch
import torchvision
from torchvision.models.detection import FasterRCNN, MaskRCNN
from torchvision.models.detection.rpn import AnchorGenerator
from torchvision import models
from torchvision.models import resnet50
import torch.nn as nn
from engine import train_one_epoch, evaluate
# train on the GPU or on the CPU, if a GPU is not available
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
print(device)

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

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

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

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

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

# Modify the Mask R-CNN model to use MobileNetV2 as the backbone
def get_model_instance_segmentation_with_mobilenet(num_classes):
    # Load the pre-trained MobileNetV2 model as the backbone
    backbone = models.mobilenet_v2(weights="DEFAULT").features

    # Mask R-CNN needs to know the number of output channels in the
backbone
    # For MobileNetV2, it's 1280, which is the output channel size of
the last layer
    backbone.out_channels = 1280

    # Define the AnchorGenerator for the Region Proposal Network (RPN)
    anchor_generator = AnchorGenerator(
        sizes=((32, 64, 128, 256, 512),),
        aspect_ratios=((0.5, 1.0, 2.0),)
    )

    # Define the RoI Pooling
    roi_pooler = torchvision.ops.MultiScaleRoIAlign(
        featmap_names=['0'], output_size=7, sampling_ratio=2
    )

    # Put together the Mask R-CNN model using the new backbone
    model = MaskRCNN(
```

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

    # Replace the box predictor
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features,
num_classes)

    # Replace the mask predictor with the correct number of classes
    in_features_mask =
model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    model.roi_heads.mask_predictor =
MaskRCNNPredictor(in_features_mask, hidden_layer, num_classes)

    return model

# Get the Mask R-CNN model with MobileNetV2 as the backbone
model3 = get_model_instance_segmentation_with_mobilenet(num_classes)

# Move model to the appropriate device
model3.to(device)

# Construct an optimizer
params = [p for p in model3.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(
    params,
    lr=0.005,
    momentum=0.9,
    weight_decay=0.0005
)

# Construct a learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.1
)

# Train the model for 10 epochs
num_epochs = 10

for epoch in range(num_epochs):
    # Train for one epoch, printing every 10 iterations
    model3.train()  # Set model to training mode
    train_one_epoch(model3, optimizer, data_loader, device, epoch,
print_freq=10)
    lr_scheduler.step()  # Update the learning rate
```

```
    # Evaluate on the test dataset
    model3.eval()  # Set model to evaluation mode
    evaluate(model3, data_loader_test, device=device)

print("That's it!")
```

```
cpu
```

```
C:\Users\Sky\engine.py:30: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.cuda.amp.autocast(enabled=scaler is not None):
```

```
Epoch: [0]  [ 0/60]  eta: 0:06:54  lr: 0.000090  loss: 4.2752 (4.2752)
loss_classifier: 0.6914 (0.6914)  loss_box_reg: 0.0532 (0.0532)
loss_mask: 2.7791 (2.7791)  loss_objectness: 0.7050 (0.7050)
loss_rpn_box_reg: 0.0466 (0.0466)  time: 6.9126  data: 0.0170
Epoch: [0]  [10/60]  eta: 0:05:35  lr: 0.000936  loss: 3.7530 (3.6396)
loss_classifier: 0.6485 (0.6210)  loss_box_reg: 0.0543 (0.0592)
loss_mask: 2.2463 (2.2279)  loss_objectness: 0.6928 (0.6903)
loss_rpn_box_reg: 0.0372 (0.0412)  time: 6.7116  data: 0.0147
Epoch: [0]  [20/60]  eta: 0:04:36  lr: 0.001783  loss: 2.6636 (2.9552)
loss_classifier: 0.4085 (0.4663)  loss_box_reg: 0.0652 (0.0871)
loss_mask: 1.4787 (1.7132)  loss_objectness: 0.6575 (0.6487)
loss_rpn_box_reg: 0.0372 (0.0399)  time: 6.9209  data: 0.0142
Epoch: [0]  [30/60]  eta: 0:03:34  lr: 0.002629  loss: 1.8659 (2.5565)
loss_classifier: 0.2707 (0.4079)  loss_box_reg: 0.1533 (0.1217)
loss_mask: 0.7972 (1.4019)  loss_objectness: 0.5173 (0.5874)
loss_rpn_box_reg: 0.0336 (0.0376)  time: 7.4109  data: 0.0160
Epoch: [0]  [40/60]  eta: 0:02:24  lr: 0.003476  loss: 1.5896 (2.2944)
loss_classifier: 0.2709 (0.3774)  loss_box_reg: 0.1687 (0.1347)
loss_mask: 0.6738 (1.2219)  loss_objectness: 0.3895 (0.5247)
loss_rpn_box_reg: 0.0299 (0.0357)  time: 7.5561  data: 0.0165
Epoch: [0]  [50/60]  eta: 0:01:13  lr: 0.004323  loss: 1.3738 (2.1047)
loss_classifier: 0.2533 (0.3550)  loss_box_reg: 0.1666 (0.1437)
loss_mask: 0.6354 (1.1004)  loss_objectness: 0.2936 (0.4715)
loss_rpn_box_reg: 0.0267 (0.0341)  time: 7.5487  data: 0.0153
Epoch: [0]  [59/60]  eta: 0:00:07  lr: 0.005000  loss: 1.2887 (1.9553)
loss_classifier: 0.2301 (0.3310)  loss_box_reg: 0.1560 (0.1449)
loss_mask: 0.5813 (1.0186)  loss_objectness: 0.2142 (0.4281)
loss_rpn_box_reg: 0.0257 (0.0327)  time: 7.5624  data: 0.0145
Epoch: [0] Total time: 0:07:19 (7.3241 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:01:57  model_time: 2.3349 (2.3349)
evaluator_time: 0.0165 (0.0165)  time: 2.3594  data: 0.0080
Test:  [49/50]  eta: 0:00:02  model_time: 2.6204 (2.7708)
evaluator_time: 0.0180 (0.0212)  time: 2.6658  data: 0.0097
Test: Total time: 0:02:20 (2.8019 s / it)
```

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.299
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.729
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.143
 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 ] = 0.088
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.318
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.156
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.450
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.466
 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 ] = 0.220
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.488
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.269
 Average Precision  (AP) @[ IoU=0.50      | area=   all |
maxDets=100 ] = 0.687
 Average Precision  (AP) @[ IoU=0.75      | area=   all |
maxDets=100 ] = 0.111
 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 ] = 0.009
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large |
maxDets=100 ] = 0.302
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
1 ] = 0.148
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=
10 ] = 0.368
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all |
maxDets=100 ] = 0.375
 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 ] = 0.090
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large |
```

```
maxDets=100 ] = 0.400
That's it!
```

```
import matplotlib.pyplot as plt
from torchvision.utils import draw_bounding_boxes,
draw_segmentation_masks
image =
read_image("Downloads/PennFudanPed/PennFudanPed/PNGImages/FudanPed0004
6.png")
eval_transform = get_transform(train=False)

model3.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model3([x, ])
    pred = predictions[0]


image = (255.0 * (image - image.min()) / (image.max() -
image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels,
colors="red")
masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5,
colors="blue")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
plt.show()
```

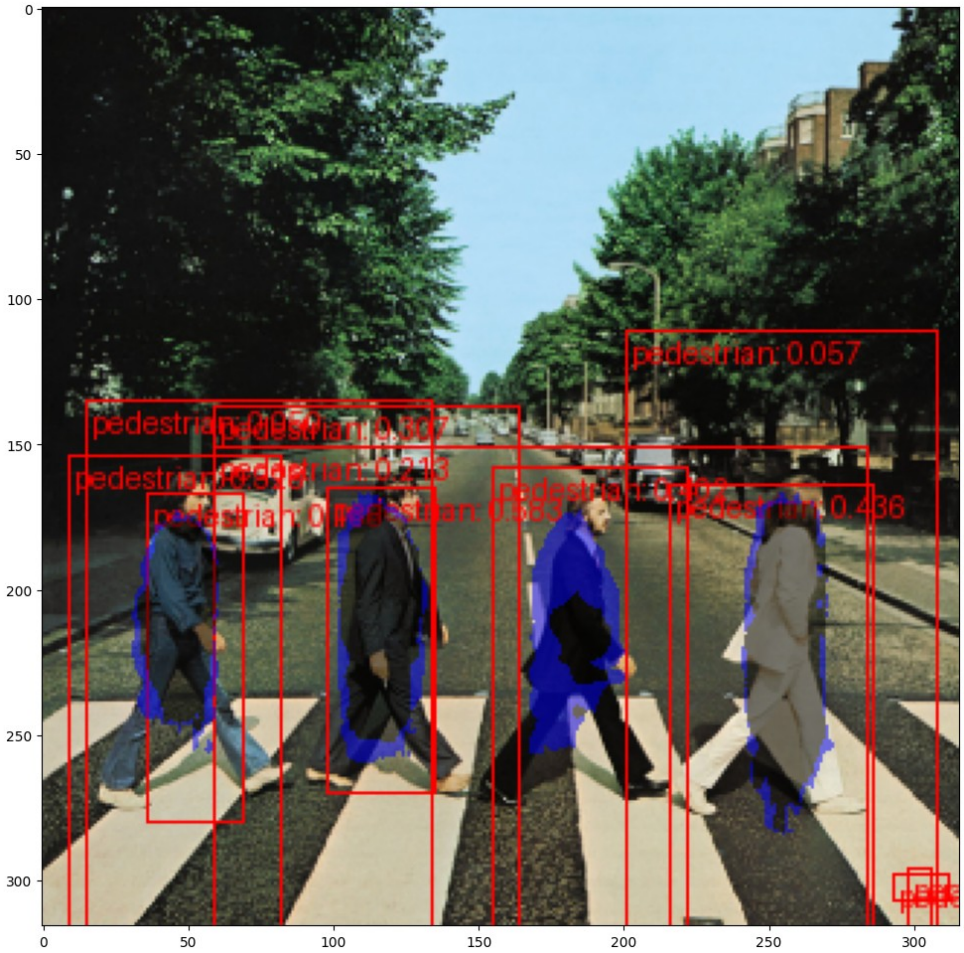
Although the the number of boxes is not significantly reduces, the overlapping of boxes makes more sense than the previous verions that without Masked. For example, some of the overlapping boxes seems captures the same object, but with slightly different boarder coordinates. This might be because that the MobileNetV2 backbone is a lightweight architecture primarily designed for speed and efficiency. Compared to the original ResNet50 backbone, it may lack the feature extraction capability needed for complex segmentation tasks, which can lead to decreased detection and segmentation accuracy. Lightweight backbones may struggle with fine details or distinguishing between multiple instances of similar objects, which might explain the over-detection and poorly defined masks. Therefore it fails to reduce the number of overlapping boxes that contains the same object.

```
image = read_image("Beatles_-_Abbey_Road.jpg")
eval_transform = get_transform(train=False)
```

```
model3.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model3([x, ])
    pred = predictions[0]


image = (255.0 * (image - image.min()) / (image.max() -
image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels,
colors="red")
masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5,
colors="blue")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
plt.show()
```



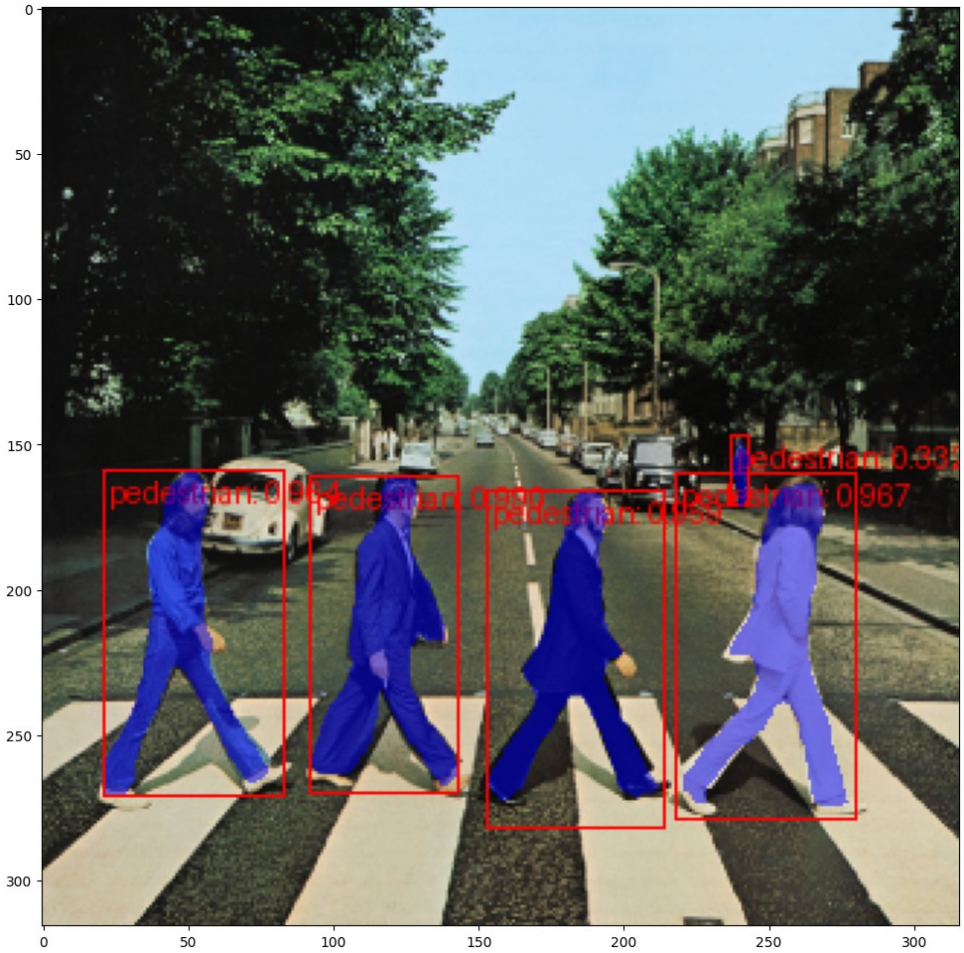

```
image = read_image("Beatles_-_Abbey_Road.jpg")
eval_transform = get_transform(train=False)

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


image = (255.0 * (image - image.min()) / (image.max() -
```

```
image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in
zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels,
colors="red")
masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5,
colors="blue")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
plt.show()
```



The difference in performance between these two models is similar to the picture picked from the dataset. The fine-tuned model predicts very well with significantly fewer false positives compared to the identification result of the substituted backbone model. Also, the confidence score of the fine-tuned model is very high, all obvious pedastrians have a confidence score over 0.95. MobileNetV2 backbone predicts a lot false positives and produces overlaps. Therefore, the preformance of the fine-tuned model using ResNet50 is

The results look good!

4 Problem 4

a

For orthogonal vectors, we have the following property:

$$||a + b||^2 = ||a||^2 + ||b||^2$$

We can use this property to compute the norms of z_1, z_2, z_3

$$||z_1||^2 = ||p + 2q||^2 = ||p||^2 + ||2q||^2 = \alpha^2 + (2\alpha)^2 = 5\alpha^2 \tag{1}$$

$$||z_1|| = \sqrt{5}\alpha \tag{2}$$

$$||z_2||^2 = ||r + q||^2 = ||r||^2 + ||q||^2 = \alpha^2 + \alpha^2 = 2\alpha^2 \tag{3}$$

$$||z_2|| = \sqrt{2}\alpha \tag{4}$$

$$||z_3||^2 = ||s + 3p||^2 = ||s||^2 + ||3p||^2 = \alpha^2 + (3\alpha)^2 = 10\alpha^2 \tag{5}$$

$$||z_3|| = \sqrt{10}\alpha \tag{6}$$

b

To compute w_1, w_2, w_3, we need all the dot products between z_1, z_2, z_3. Note that for p, q, r, s, since they are orthogonal basis, the product of any of the two vectors are 0.

$$z_1 \cdot z_1 = ||z_1||^2 = 5\alpha^2 \tag{1}$$

$$z_1 \cdot z_2 = (p + 2q) \cdot (r + q) \tag{2}$$

$$= p \cdot r + p \cdot q + 2q \cdot r + 2q \cdot q \tag{3}$$

$$= 2q \cdot q = 2\alpha^2 \tag{4}$$

$$z_1 \cdot z_3 = (p + 2q) \cdot (s + 3p) \tag{5}$$

$$= p \cdot 3p = 3\alpha^2 \tag{6}$$

$$z_2 \cdot z_2 = ||z_2||^2 = 2\alpha^2 \tag{7}$$

$$z_2 \cdot z_3 = (r + q) \cdot (s + 3p) = 0 \tag{8}$$

$$z_3 \cdot z_3 = ||z_3||^2 = 10\alpha^2 \tag{9}$$

Then we can start to compute attention weights:

w_1

$$w_{11} = \frac{z_1 \cdot z_1}{\sum_{i=1}^{3} z_1 \cdot z_i} \tag{1}$$

$$= \frac{5\alpha^2}{5\alpha^2 + 2\alpha^2 + 3\alpha^2} = \frac{1}{2} \tag{2}$$

$$w_{12} = \frac{z_1 \cdot z_2}{\sum} = \frac{1}{5} \tag{3}$$

$$w_{13} = \frac{z_1 \cdot z_3}{\sum} = \frac{3}{10} \tag{4}$$

Therefore,

$$w_1 = w_{11}z_1 + w_{12}z_2 + w_{13}z_3 = \frac{1}{2}z_1 + \frac{1}{5}z_2 + \frac{3}{10}z_3$$

z_1 is mostly approximated by w_1 but there are still a mix of z_2, z_3 that contributes to w_1

w_2

$$w_{21} = \frac{z_2 \cdot z_1}{\sum_{i=1}^{3} z_2 \cdot z_i} \tag{1}$$

$$= \frac{2\alpha^2}{2\alpha^2 + 2\alpha^2 + 0} = \frac{1}{2} \tag{2}$$

$$w_{22} = \frac{z_2 \cdot z_2}{\sum} = \frac{1}{2} \tag{3}$$

$$w_{23} = \frac{z_2 \cdot z_3}{\sum} = 0 \tag{4}$$

Therefore,

$$w_2 = \frac{1}{2}z_1 + \frac{1}{2}z_2$$

z_2 and z_1 are equally likely to be approximated by w_2 since they share the same contribution to w_2

w_3

$$w_{31} = \frac{z_3 \cdot z_1}{\sum_{i=1}^{3} z_3 \cdot z_i} \tag{1}$$

$$= \frac{3\alpha^2}{3\alpha^2 + 0 + 10\alpha^2} = \frac{3}{13} \tag{2}$$

$$w_{32} = \frac{z_3 \cdot z_2}{\sum} = 0 \tag{3}$$

$$w_{33} = \frac{z_3 \cdot z_3}{\sum} = \frac{10}{13} \tag{4}$$

Therefore,

$$w_3 = \frac{3}{13}z_1 + \frac{10}{13}z_3$$

z_3 is most likely to be approximated by w_3, since w_3 consists of a mix of z_1 and z_3, but

z_3 contributes the majority of likelihood to w_3.

c

We observed that $z_1 \cdot z_3 = 3\alpha^2$ and $z_2 \cdot z_2 = 2\alpha^2$. In this case, $z_1 \cdot z_3$ is greater because the component of $\mathbf{p}$ in both z_1 and z_3 contributes significantly to their similarity more than the components $\mathbf{r}$ and $\mathbf{q}$ do in z_2. This tells us that in the attention mechanism, cross-token similarity can sometimes be larger than the self-similarity because attention mechanisms are not solely influenced by the magnitude of the vectors but are also significantly affected by the alignment or similarity of the vectors.