# 基于深度学习的目标检测实验报告

**实验目标：**

1、 理解常见的目标检测、语义分割方法原理，调通相应程序，并完成目标检测实践任务

**实验内容：**

在本实验中，我们将微调预训练的 Mask 宾夕法尼亚大学-复旦大学的 R-CNN 行人检测模型和分段。它包含 170 张图像，其中包含 345 个行人实例，我们将使用它来演示如何使用 TorchVision 中的新功能进行训练自定义数据集上的对象检测和实例分段模型。

首先下载数据集（PennFudan 数据集），然后解压缩 zip 文件至 'PennFudanPed' 目录。

```
# 解压数据集
import zipfile

zip_file_path = 'PennFudanPed.zip'  # 输入 zip 文件名
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall('./PennFudanPed')  # 解压到 'PennFudanPed' 目录

print("Done!")
```

```
Done!
```

图 1 解压缩数据集文件

以下是一对图像和分割蒙版的一个示例，见图 11。

```
# 一对图像和分割蒙版的一个示例

import matplotlib.pyplot as plt
from torchvision.io import read_image

image = read_image("PennFudanPed/PennFudanPed/PNGImages/FudanPed00046.png")
mask = read_image("PennFudanPed/PennFudanPed/PedMasks/FudanPed00046_mask.png")

plt.figure(figsize=(16, 8))
plt.subplot(121)
plt.title("Image")
plt.imshow(image.permute(1, 2, 0))
plt.subplot(122)
plt.title("Mask")
plt.imshow(mask.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x2d4c0dbe1a0>
```

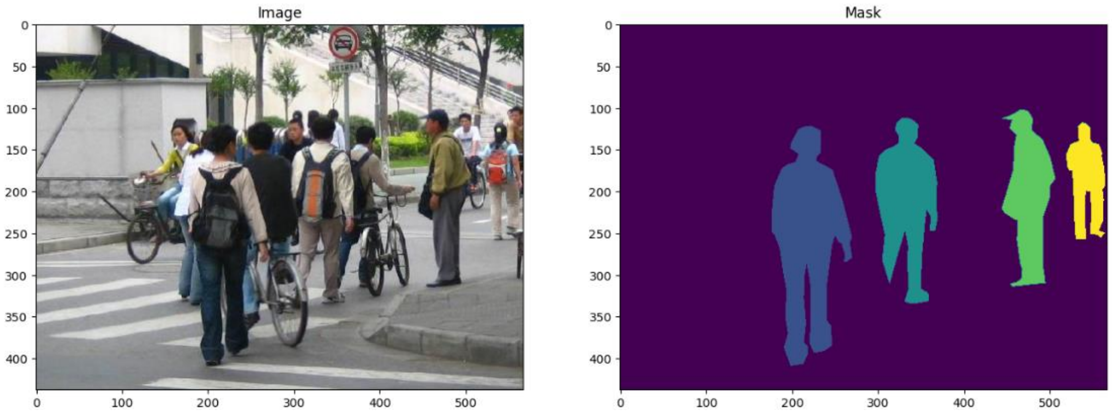


图 2 图像和分割蒙版示例

因此，每幅图像都有一个相应的分割掩码（Segmentation Mask），其中每种颜色对应一个不同的实例。让我们为这个数据集编写一个 torch.utils.data.Dataset 类。

1、图像张量将由 torchvision.tv_tensors.Image 封装；

2、边界框将由 torchvision.tv_tensors.BoundingBoxes 封装；

3、遮罩将由 torchvision.tv_tensors.Mask 封装。

在本教程中，我们将使用 Mask R-CNN，它基于 Faster R-CNN 之上。

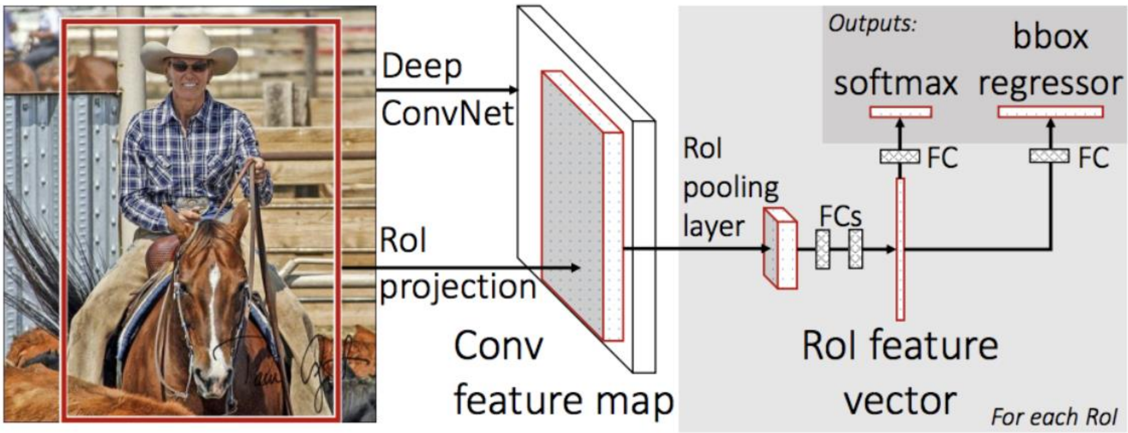


图 3 Faster R-CNN 示例

Mask R-CNN 添加了一个额外的分支即全卷积网络和掩码预测，并将 Faster R-CNN 中的兴趣区域池化层替换为了兴趣区域对齐层，使用双线性插值来保留特征图上的空间信息，从而更适用于像素级预测。因此如果训练集中标注了每个目标在图像上的像素级位置，那么 Mask R-CNN 能够有效地利用这些详尽的标注信息进一步提升目标检测的精度。

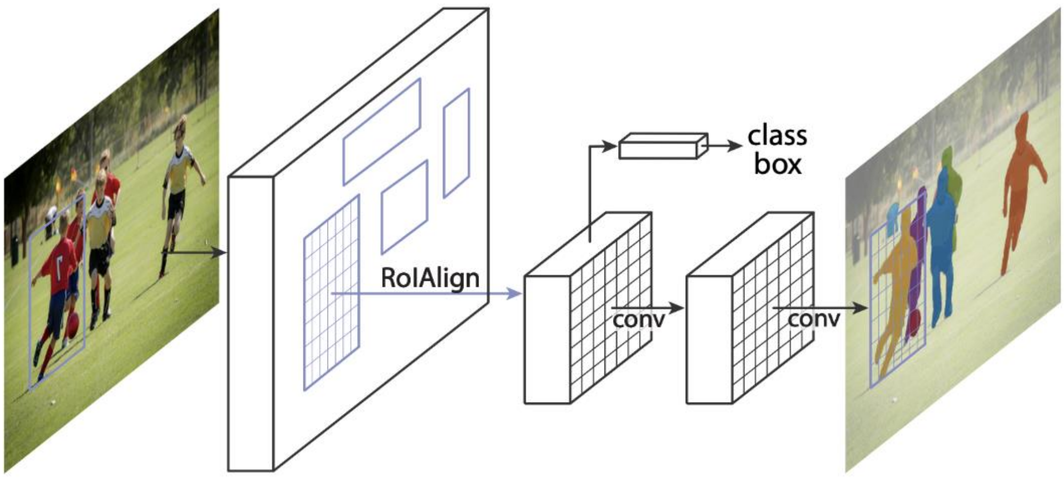


图 4 Mask R-CNN 示例

有两种常见的情况当人们可能想要修改 TorchVision Model Zoo 中的可用模型之一。第一种是当我们想要从预先训练的模型开始，然后微调最后一层。另一种是当我们想替换模型主干为其他模型（例如：为了更快地进行预测）。

假设您想从 COCO（目标检测上的经典数据集，类似于图像分类中的 ImageNet）上预训练的模型开始并希望针对您的特定类对其进行微调。这是一个可能的方法：

```python
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)
```

图 5 使用 COCO 预训练模型

修改模型以添加不同的主干（部分修改代码）。

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

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

图 6 添加不同的主干

然后我们进行 PennFudan 数据集的目标检测和实例分割模型，在我们的例子中，我们希望从预训练模型进行微调，因为我们的数据集非常小，因此我们将遵循第 1 种方法。

在这里，我们还想计算实例分段掩码，因此我们将使用 Mask R-CNN：

```python
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
```

图 7 使用 Mask R-CNN

在 references/detection/ 中，我们有许多辅助函数来简化检测模型的训练和评估。在这里，我们将使用 references/detection/engine.py 和 references/detection/utils.py 。只需将下的所有内容下载到您的文件夹中并在此处使用它们（我个人觉得使用 Python 的 request 库更具可移植性和跨平台性，因此将代码进行了适当修改）。

```
import requests

urls = [
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/engine.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/utils.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_utils.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_eval.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/transforms.py"
]

for url in urls:
    response = requests.get(url)
    filename = url.split("/")[-1]  # 从URL中提取文件名
    with open(filename, "wb") as f:
        f.write(response.content)  # 写入文件

print("Done!")
```

```
Done!
```

图 8 下载并写入辅助函数

从 v0.15.0 开始，torchvision 提供了新的 Transforms API，可以轻松地为对象检测和分割任务编写数据增强管道。我们编写一些用于数据增强的辅助函/转型（get_transfrom）。

现在让我们编写执行训练的 main 函数和验证，并通过输出来展示模型训练效果：

```
import torch
from engine import train_one_epoch, evaluate

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

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

# 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
)
```

```python
# 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 = 2

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

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

```
Downloading: "https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-bf2d0c1e.pth" to C:\Users\WDMX/.cache\torch\hub\checkpoints\maskrcnn_resnet5
0_fpn_coco-bf2d0c1e.pth
100.0%
C:\Users\WDMX\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: 0:01:42  lr: 0.000090  loss: 4.3386 (4.3386)  loss_classifier: 0.7486 (0.7486)  loss_box_reg: 0.2996 (0.2996)  loss_mask: 3.264
0 (3.2640)  loss_objectness: 0.0240 (0.0240)  loss_rpn_box_reg: 0.0024 (0.0024)  time: 1.7091  data: 0.0199  max mem: 2146
Epoch: [0]  [10/60]  eta: 0:00:24  lr: 0.000936  loss: 1.3656 (2.1830)  loss_classifier: 0.4359 (0.4453)  loss_box_reg: 0.2296 (0.2480)  loss_mask: 0.690
1 (1.4679)  loss_objectness: 0.0188 (0.0173)  loss_rpn_box_reg: 0.0034 (0.0045)  time: 0.4827  data: 0.0237  max mem: 2667
Epoch: [0]  [20/60]  eta: 0:00:17  lr: 0.001783  loss: 0.9414 (1.5001)  loss_classifier: 0.2338 (0.3179)  loss_box_reg: 0.2126 (0.2677)  loss_mask: 0.314
8 (0.8871)  loss_objectness: 0.0188 (0.0206)  loss_rpn_box_reg: 0.0035 (0.0068)  time: 0.3807  data: 0.0250  max mem: 3304
Epoch: [0]  [30/60]  eta: 0:00:12  lr: 0.002629  loss: 0.5953 (1.2200)  loss_classifier: 0.0908 (0.2483)  loss_box_reg: 0.2502 (0.2687)  loss_mask: 0.227
5 (0.6795)  loss_objectness: 0.0095 (0.0164)  loss_rpn_box_reg: 0.0054 (0.0071)  time: 0.3837  data: 0.0247  max mem: 3304
Epoch: [0]  [40/60]  eta: 0:00:07  lr: 0.003476  loss: 0.4476 (1.0315)  loss_classifier: 0.0600 (0.2003)  loss_box_reg: 0.1822 (0.2456)  loss_mask: 0.216
5 (0.5660)  loss_objectness: 0.0034 (0.0130)  loss_rpn_box_reg: 0.0054 (0.0067)  time: 0.3536  data: 0.0223  max mem: 3304
Epoch: [0]  [50/60]  eta: 0:00:03  lr: 0.004323  loss: 0.4145 (0.9125)  loss_classifier: 0.0511 (0.1704)  loss_box_reg: 0.1612 (0.2346)  loss_mask: 0.182
0 (0.4899)  loss_objectness: 0.0020 (0.0108)  loss_rpn_box_reg: 0.0050 (0.0067)  time: 0.3491  data: 0.0220  max mem: 3304
Epoch: [0]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.4038 (0.8291)  loss_classifier: 0.0423 (0.1517)  loss_box_reg: 0.1584 (0.2206)  loss_mask: 0.172
4 (0.4408)  loss_objectness: 0.0013 (0.0095)  loss_rpn_box_reg: 0.0050 (0.0065)  time: 0.3697  data: 0.0232  max mem: 3304
Epoch: [0] Total time: 0:00:23 (0.3901 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:05  model_time: 0.0912 (0.0912)  evaluator_time: 0.0070 (0.0070)  time: 0.1100  data: 0.0100  max mem: 3304
Test:  [49/50]  eta: 0:00:00  model_time: 0.0616 (0.0658)  evaluator_time: 0.0020 (0.0049)  time: 0.0773  data: 0.0078  max mem: 3304
Test: Total time: 0:00:03 (0.0791 s / it)
Averaged stats: model_time: 0.0616 (0.0658)  evaluator_time: 0.0020 (0.0049)
Accumulating evaluation results...
DONE (t=0.01s).
Accumulating evaluation results...
DONE (t=0.01s).
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.599
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.978
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.757
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.350
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.350
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.612
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.292
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.655
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.655
 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 ] = 0.613
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.660
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.673
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.968
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.875
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.300
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.414
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.686
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.301
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.717
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.720
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.600
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.738
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.721
Epoch: [1]  [ 0/60]  eta: 0:00:19  lr: 0.005000  loss: 0.2007 (0.2007)  loss_classifier: 0.0171 (0.0171)  loss_box_reg: 0.0536 (0.0536)  loss_mask: 0.123
9 (0.1239)  loss_objectness: 0.0038 (0.0038)  loss_rpn_box_reg: 0.0022 (0.0022)  time: 0.3186  data: 0.0262  max mem: 3304
Epoch: [1]  [10/60]  eta: 0:00:18  lr: 0.005000  loss: 0.2993 (0.2930)  loss_classifier: 0.0363 (0.0375)  loss_box_reg: 0.0897 (0.1032)  loss_mask: 0.135
3 (0.1445)  loss_objectness: 0.0010 (0.0021)  loss_rpn_box_reg: 0.0046 (0.0057)  time: 0.3748  data: 0.0202  max mem: 3304
Epoch: [1]  [20/60]  eta: 0:00:14  lr: 0.005000  loss: 0.3074 (0.2988)  loss_classifier: 0.0369 (0.0418)  loss_box_reg: 0.1077 (0.1020)  loss_mask: 0.135
3 (0.1471)  loss_objectness: 0.0006 (0.0016)  loss_rpn_box_reg: 0.0047 (0.0063)  time: 0.3715  data: 0.0194  max mem: 3304
Epoch: [1]  [30/60]  eta: 0:00:11  lr: 0.005000  loss: 0.2649 (0.2835)  loss_classifier: 0.0345 (0.0393)  loss_box_reg: 0.0860 (0.0955)  loss_mask: 0.133
1 (0.1407)  loss_objectness: 0.0010 (0.0020)  loss_rpn_box_reg: 0.0044 (0.0059)  time: 0.3653  data: 0.0192  max mem: 3304
Epoch: [1]  [40/60]  eta: 0:00:07  lr: 0.005000  loss: 0.2489 (0.2838)  loss_classifier: 0.0325 (0.0404)  loss_box_reg: 0.0676 (0.0936)  loss_mask: 0.130
7 (0.1419)  loss_objectness: 0.0008 (0.0020)  loss_rpn_box_reg: 0.0044 (0.0059)  time: 0.3712  data: 0.0193  max mem: 3304
Epoch: [1]  [50/60]  eta: 0:00:03  lr: 0.005000  loss: 0.2384 (0.2741)  loss_classifier: 0.0345 (0.0392)  loss_box_reg: 0.0609 (0.0873)  loss_mask: 0.123
6 (0.1401)  loss_objectness: 0.0005 (0.0020)  loss_rpn_box_reg: 0.0037 (0.0054)  time: 0.3595  data: 0.0187  max mem: 3304
Epoch: [1]  [59/60]  eta: 0:00:00  lr: 0.005000  loss: 0.2630 (0.2795)  loss_classifier: 0.0404 (0.0405)  loss_box_reg: 0.0652 (0.0885)  loss_mask: 0.131
5 (0.1431)  loss_objectness: 0.0010 (0.0019)  loss_rpn_box_reg: 0.0038 (0.0056)  time: 0.3622  data: 0.0191  max mem: 3304
Epoch: [1] Total time: 0:00:22 (0.3680 s / it)
creating index...
index created!
Test:  [ 0/50]  eta: 0:00:04  model_time: 0.0732 (0.0732)  evaluator_time: 0.0046 (0.0046)  time: 0.0857  data: 0.0080  max mem: 3304
Test:  [49/50]  eta: 0:00:00  model_time: 0.0636 (0.0633)  evaluator_time: 0.0020 (0.0035)  time: 0.0765  data: 0.0083  max mem: 3304
Test: Total time: 0:00:03 (0.0756 s / it)
Averaged stats: model_time: 0.0636 (0.0633)  evaluator_time: 0.0020 (0.0035)
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

图 9 模型训练

因此，经过一个 epoch 的训练，我们获得了 COCO 风格的 mAP > 50，并且 Mask mAP 为 65。但预测是什么样的呢？让我们在应用到数据集并验证，使用 matplotlib.pyplot 可视化输出以便对验证结果有更好的理解。

```
import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes, draw_segmentation_masks


image = read_image("PennFudanPed/PennFudanPed/PNGImages/FudanPed00046.png")
eval_transform = get_transform(train=False)

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


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

masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5, colors="blue")


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

```
<matplotlib.image.AxesImage at 0x2d672f03610>
```

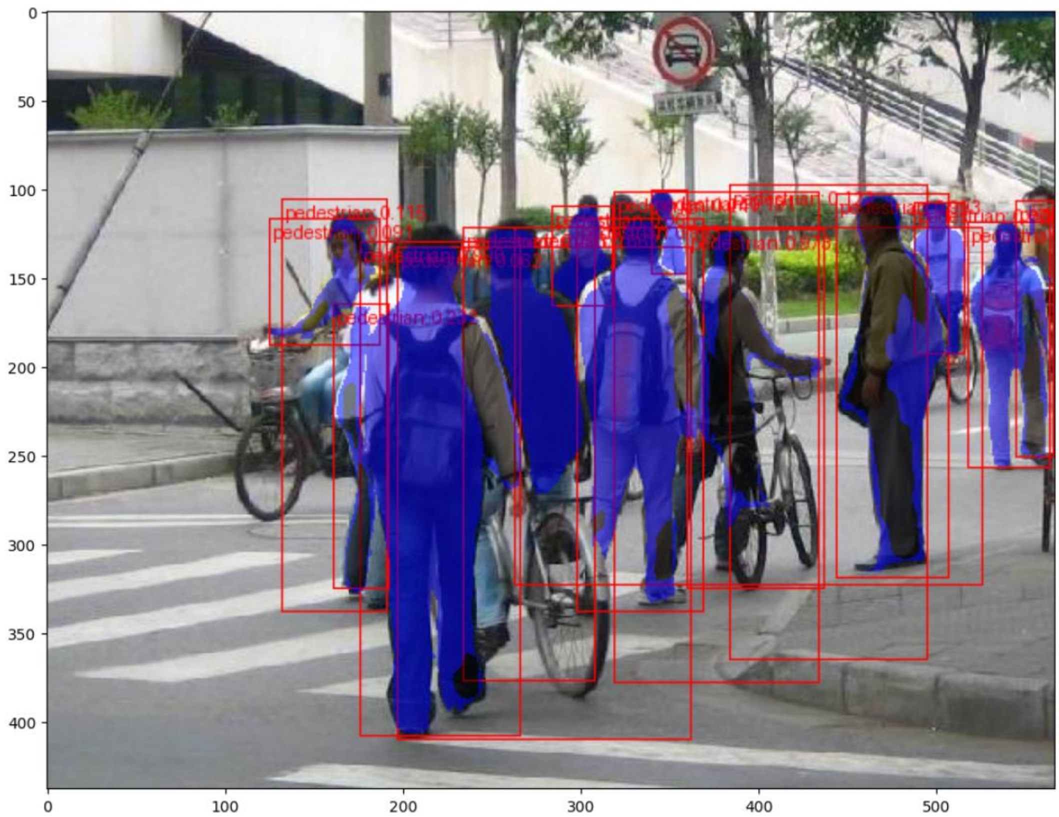


图 10 验证与结果可视化