# 2.6.4 对象检测、实例分割与人流量估算和对象统计

在影像和视频处理方面，深度学习主要涉及到图像分类（Image classification）、对象检测（Object detection），迁移学习（Transfer learnig），对抗生成（Adversarial generation：DCGAN/deep convolutional generative adversarial network）等。在城市空间分析研究中，每一个方向都能为之提供新的分析技术方法。对象检测是计算机视觉下，分析图像或影像，将其中的对象（例如车辆、行人、动物、桌椅、植被等，通常包括室外环境，室内环境，甚至某一对象的具体再分，例如人脸识别等）标记出来。通过图像对城市环境内对象的识别，可以统计对象的空间分布情况。同时，因为无人驾驶项目大量影像数据的获取，不仅可以对城市不连续的影像分析，亦可以通过连续影像的分析进一步获取目标对象的空间分布变化情况。在下文的分析中，对于对象的分析锁定在两个方向，一个是仅识别行人，实现人流量估计；再者是尽量多的识别对象，分析城市空间下各个对象的分布变化情况，及对象之间的联系。

目前PyTorch已经集合了大量分析算法模型，且提供预训练模型的参数，一定程度上可以跨过训练阶段，直接用于预测。PyTorch也提供了大量参考的代码（还有大量的著作教程），这都为研究者更快速和轻松的应用已有模型研究提供了便利，从而能够快速将其应用到各自的研究领域当中，而不是计算机视觉、深度学习算法本身研究的专业。

### 2.6.4.1 对象/目标检测（行人）与人流量估算

该部分代码是直接迁移应用[TorchVision Object Detection Finetuning Tutorial](https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html)[1]。通常PyTorch也提供了[Googel Colab版本](https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/torchvision_finetuning_instance_segmentation.ipynb)①，可以直接在线运行代码，解决当前没有配置GPU，或者GPU算力较低的情况，使得深度学习快速的结果反馈成为可能。

微调[Penn-Fudan Database （for Pedestrian Detection and Segmentation）](https://www.cis.upenn.edu/~jshi/ped_html/)②数据库中预训练的[Mask R-CNN](https://arxiv.org/abs/1703.06870)[2]，用于行人检测和分割。该数据库包含170张图像和345个行人实例。 在人流量估算研究中，直接应用“Mask R-CNN”训练的模型。对该方法的解释直接参看官方内容，仅是保留关键信息的解释。重点在于，通过对行人的目标检测，以[KITTI数据集](https://www.cvlibs.net/datasets/kitti/)③为例，计算行人的空间分布核密度，估算人流量。

#### 1）对象检测（行人）

用!符号直接运行conda(作为shell命令)。有些终端命令，如果无法在JupyterLab下直接实现，则打开Anaconda的终端（terminal）执行。

! pip install cython

Requirement already satisfied: cython in c:\users\richi\anaconda3\envs\usda\_database\lib\site-packages (0.29.26)

可以直接在浏览器下输入https://www.cis.upenn.edu/~jshi/ped\_html/PennFudanPed.zip，下载，也可以在终端执行wget https://www.cis.upenn.edu/~jshi/ped\_html/PennFudanPed.zip，并且解压缩unzip PennFudanPed.zip，获取[PennFudan数据集](https://www.cis.upenn.edu/~jshi/ped_html/)④。其数据库文件夹结构如下：

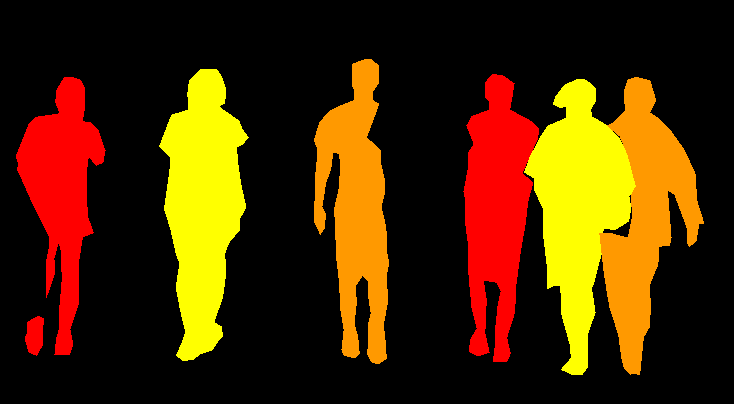
PennFudanPed/  
 PedMasks/  
 FudanPed00001\_mask.png  
 FudanPed00002\_mask.png  
 FudanPed00003\_mask.png  
 FudanPed00004\_mask.png  
 ...  
 PNGImages/  
 FudanPed00001.png  
 FudanPed00002.png  
 FudanPed00003.png  
 FudanPed00004.png

直接打开一张图像，和其对应的对象分割掩码。

from PIL import Image  
import os  
  
PennFudanPed\_fp='./dataset/PennFudanPed'  
Image.open(os.path.join(PennFudanPed\_fp,'PNGImages/PennPed00019.png'))



mask=Image.open(os.path.join(PennFudanPed\_fp,'PedMasks/PennPed00019\_mask.png')).convert('P') # Open mask with Pillow, and convert to mode 'P'  
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



定义继承父类torch.utils.data.Dataset，建立数据集（tensor数据类型）作为torch.utils.data.DataLoader输入生成可迭代对象，用于训练模型数据加载的类。

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)

返回定义的数据集包含一个PIL.Image和一个字典，包含boxes锚框，labels类标，masks分割掩码，image\_id图像索引，及area锚框面积和iscrowd。

dataset=PennFudanDataset(PennFudanPed\_fp)  
dataset[0]

(<PIL.Image.Image image mode=RGB size=559x536 at 0x1D3DBC2BD00>,  
 {'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])})

torchvision.models.detection提供了fasterrcnn\_resnet50\_fpn，maskrcnn\_resnet50\_fpn（含对象分割/对象掩码（mask））对象检测模型，包含基于[COCO数据集](https://cocodataset.org/#home)⑤预先训练的参数。针对特定类别对其微调（finetune）。

COCO数据集，是一个大规模对象检测（object detection）、分割（segmentation）和标注（captioning）数据集，其特征有：

1. 对象分割（Object segmentation）
2. 上下文识别（Recognition in context）
3. 超像素分割（Superpixel stuff segmentation）
4. 330K张图片（>200K个标签）（330K images （>200K labeled））
5. 150万个对象实例（1.5 million object instances）
6. 80个对象类（80 object categories）
7. [91个分割类（91 stuff categories）](https://github.com/nightrome/cocostuff)⑥
8. 每张图像5个标注（5 captions per image）
9. 25万人带有关键点（250,000 people with keypoints）

可以从COCO官网下载数据查看其对象的分类，可以从http://images.cocodataset.org/annotations/annotations\_trainval2014.zip处下载2014年，及从http://images.cocodataset.org/annotations/annotations\_trainval2017.zip下载2017年。解压后打开“instances\_val2017.json”，打印查看对象分类。可以看到其中分类“person”，其ID为1。

annotations\_trainval2017\_fp=r'./data/instances\_val2017.json'  
import json  
with open(annotations\_trainval2017\_fp,'r') as f:  
 annotations=json.loads(f.read())  
 object\_categories=json.dumps(annotations['categories'])  
print('COCO对象分类：\n',object\_categories)

COCO对象分类：  
 [{"supercategory": "person", "id": 1, "name": "person"}, {"supercategory": "vehicle", "id": 2, "name": "bicycle"}, {"supercategory": "vehicle", "id": 3, "name": "car"}, {"supercategory": "vehicle", "id": 4, "name": "motorcycle"}, {"supercategory": "vehicle", "id": 5, "name": "airplane"}, {"supercategory": "vehicle", "id": 6, "name": "bus"}, {"supercategory": "vehicle", "id": 7, "name": "train"}, {"supercategory": "vehicle", "id": 8, "name": "truck"}, {"supercategory": "vehicle", "id": 9, "name": "boat"}, {"supercategory": "outdoor", "id": 10, "name": "traffic light"}, {"supercategory": "outdoor", "id": 11, "name": "fire hydrant"}, {"supercategory": "outdoor", "id": 13, "name": "stop sign"}, {"supercategory": "outdoor", "id": 14, "name": "parking meter"}, {"supercategory": "outdoor", "id": 15, "name": "bench"}, {"supercategory": "animal", "id": 16, "name": "bird"}, {"supercategory": "animal", "id": 17, "name": "cat"}, {"supercategory": "animal", "id": 18, "name": "dog"}, {"supercategory": "animal", "id": 19, "name": "horse"}, {"supercategory": "animal", "id": 20, "name": "sheep"}, {"supercategory": "animal", "id": 21, "name": "cow"}, {"supercategory": "animal", "id": 22, "name": "elephant"}, {"supercategory": "animal", "id": 23, "name": "bear"}, {"supercategory": "animal", "id": 24, "name": "zebra"}, {"supercategory": "animal", "id": 25, "name": "giraffe"}, {"supercategory": "accessory", "id": 27, "name": "backpack"}, {"supercategory": "accessory", "id": 28, "name": "umbrella"}, {"supercategory": "accessory", "id": 31, "name": "handbag"}, {"supercategory": "accessory", "id": 32, "name": "tie"}, {"supercategory": "accessory", "id": 33, "name": "suitcase"}, {"supercategory": "sports", "id": 34, "name": "frisbee"}, {"supercategory": "sports", "id": 35, "name": "skis"}, {"supercategory": "sports", "id": 36, "name": "snowboard"}, {"supercategory": "sports", "id": 37, "name": "sports ball"}, {"supercategory": "sports", "id": 38, "name": "kite"}, {"supercategory": "sports", "id": 39, "name": "baseball bat"}, {"supercategory": "sports", "id": 40, "name": "baseball glove"}, {"supercategory": "sports", "id": 41, "name": "skateboard"}, {"supercategory": "sports", "id": 42, "name": "surfboard"}, {"supercategory": "sports", "id": 43, "name": "tennis racket"}, {"supercategory": "kitchen", "id": 44, "name": "bottle"}, {"supercategory": "kitchen", "id": 46, "name": "wine glass"}, {"supercategory": "kitchen", "id": 47, "name": "cup"}, {"supercategory": "kitchen", "id": 48, "name": "fork"}, {"supercategory": "kitchen", "id": 49, "name": "knife"}, {"supercategory": "kitchen", "id": 50, "name": "spoon"}, {"supercategory": "kitchen", "id": 51, "name": "bowl"}, {"supercategory": "food", "id": 52, "name": "banana"}, {"supercategory": "food", "id": 53, "name": "apple"}, {"supercategory": "food", "id": 54, "name": "sandwich"}, {"supercategory": "food", "id": 55, "name": "orange"}, {"supercategory": "food", "id": 56, "name": "broccoli"}, {"supercategory": "food", "id": 57, "name": "carrot"}, {"supercategory": "food", "id": 58, "name": "hot dog"}, {"supercategory": "food", "id": 59, "name": "pizza"}, {"supercategory": "food", "id": 60, "name": "donut"}, {"supercategory": "food", "id": 61, "name": "cake"}, {"supercategory": "furniture", "id": 62, "name": "chair"}, {"supercategory": "furniture", "id": 63, "name": "couch"}, {"supercategory": "furniture", "id": 64, "name": "potted plant"}, {"supercategory": "furniture", "id": 65, "name": "bed"}, {"supercategory": "furniture", "id": 67, "name": "dining table"}, {"supercategory": "furniture", "id": 70, "name": "toilet"}, {"supercategory": "electronic", "id": 72, "name": "tv"}, {"supercategory": "electronic", "id": 73, "name": "laptop"}, {"supercategory": "electronic", "id": 74, "name": "mouse"}, {"supercategory": "electronic", "id": 75, "name": "remote"}, {"supercategory": "electronic", "id": 76, "name": "keyboard"}, {"supercategory": "electronic", "id": 77, "name": "cell phone"}, {"supercategory": "appliance", "id": 78, "name": "microwave"}, {"supercategory": "appliance", "id": 79, "name": "oven"}, {"supercategory": "appliance", "id": 80, "name": "toaster"}, {"supercategory": "appliance", "id": 81, "name": "sink"}, {"supercategory": "appliance", "id": 82, "name": "refrigerator"}, {"supercategory": "indoor", "id": 84, "name": "book"}, {"supercategory": "indoor", "id": 85, "name": "clock"}, {"supercategory": "indoor", "id": 86, "name": "vase"}, {"supercategory": "indoor", "id": 87, "name": "scissors"}, {"supercategory": "indoor", "id": 88, "name": "teddy bear"}, {"supercategory": "indoor", "id": 89, "name": "hair drier"}, {"supercategory": "indoor", "id": 90, "name": "toothbrush"}]

精调模型（finetuning）。

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):  
 # 01-load an instance segmentation model pre-trained on COCO  
 model = torchvision.models.detection.maskrcnn\_resnet50\_fpn(pretrained=True)   
  
 # 02-get the number of input features for the classifier  
 in\_features = model.roi\_heads.box\_predictor.cls\_score.in\_features  
 # 03-replace the pre-trained head with a new one  
 model.roi\_heads.box\_predictor = FastRCNNPredictor(in\_features, num\_classes) #因为仅检测行人分类，包含背景总共2类，用其替换原输入特征数in\_features(1024)  
  
 # 04-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  
 # 05-and replace the mask predictor with a new one  
 model.roi\_heads.mask\_predictor = MaskRCNNPredictor(in\_features\_mask,hidden\_layer,num\_classes)  
 return model

可以下载https://github.com/pytorch/vision.git代码，在“references”下包含有大量帮助函数，可以简化对象/目标检测训练和评估。将对应的utils.py，transforms.py，coco\_eval.py，engine.py，coco\_utils.py这5个文件，直接复制到该文件（.ipynb）所在的目录下，执行调入。定义的get\_transform函数，可以实现对图像指定方式的变换操作，即图像增广（image augmentation），PyTorh的[transforms](https://pytorch.org/vision/0.9/transforms.html)⑦提供了大量图像增广的方法。这里使用了RandomHorizontalFlip实现图像的随机水平向翻转。并用ToTensor()方法将图像转换为tensor数据类型。

from engine import train\_one\_epoch, evaluate #pip install pycocotools-windows 安装关联库  
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)

将建立的数据集通过torch.utils.data.DataLoader方法加载为可迭代对象，用于训练模型的输入数据。

dataset = PennFudanDataset(PennFudanPed\_fp, get\_transform(train=True))  
dataset\_test = PennFudanDataset(PennFudanPed\_fp, get\_transform(train=False))  
  
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:])  
  
data\_loader = torch.utils.data.DataLoader(dataset, batch\_size=2, shuffle=True, num\_workers=0,collate\_fn=utils.collate\_fn)  
data\_loader\_test = torch.utils.data.DataLoader(dataset\_test, batch\_size=1, shuffle=False, num\_workers=0,collate\_fn=utils.collate\_fn)

定义训练模型，优化函数，学习率。指定GPU或者CPU，训练模型。

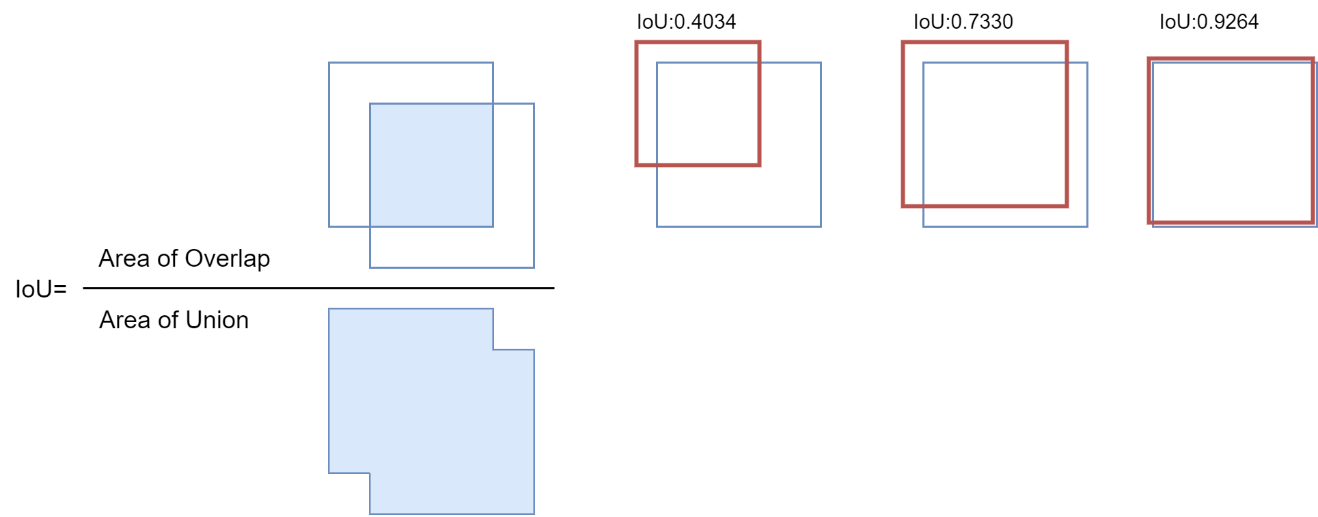
device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')  
num\_classes = 2  
model = get\_instance\_segmentation\_model(num\_classes)  
model.to(device)  
params = [p for p in model.parameters() if p.requires\_grad] # 提取含梯度的参数  
optimizer = torch.optim.SGD(params, lr=0.005,momentum=0.9, weight\_decay=0.0005) # 对含梯度的参数执行梯度下降法-SGD  
lr\_scheduler = torch.optim.lr\_scheduler.StepLR(optimizer,step\_size=3,gamma=0.1) # 根据epoch训练次数调整学习率的方法。PyTorch也提供了torch.optim.lr\_scheduler.ReduceLROnPlateau，基于训练中某些测量值使学习率下降的方法。

from tqdm.auto import tqdm  
  
num\_epochs = 10  
for epoch in tqdm(range(num\_epochs)):  
 # train for one epoch, printing every 10 iterations  
 train\_one\_epoch(model, optimizer, data\_loader, device, epoch, print\_freq=999)  
 # update the learning rate  
 lr\_scheduler.step()  
 # evaluate on the test dataset  
 if epoch==5 or epoch==9:  
 evaluate(model, data\_loader\_test, device=device)  
 torch.save(model.state\_dict(),'./model/mask\_R\_CNN\_person/mask\_R\_CNN\_person\_stateDict\_{}.pth'.format(epoch)) # 仅保存模型的状态字典(state\_dict)，state\_dict由PyTorch自动生成，包含各层可训练参数（通常为卷积层、线性层），例如权值、偏置等。  
torch.save(model,'./model/mask\_R\_CNN\_person/mask\_R\_CNN\_person\_final.pth') # 保存整个模型

0%| | 0/10 [00:00<?, ?it/s]  
  
  
Epoch: [0] [ 0/60] eta: 0:00:26 lr: 0.000090 loss: 0.2183 (0.2183) loss\_classifier: 0.0226 (0.0226) loss\_box\_reg: 0.0326 (0.0326) loss\_mask: 0.1577 (0.1577) loss\_objectness: 0.0002 (0.0002) loss\_rpn\_box\_reg: 0.0053 (0.0053) time: 0.4401 data: 0.0229 max mem: 3501  
Epoch: [0] [59/60] eta: 0:00:00 lr: 0.005000 loss: 0.1799 (0.1872) loss\_classifier: 0.0271 (0.0261) loss\_box\_reg: 0.0388 (0.0417) loss\_mask: 0.1068 (0.1153) loss\_objectness: 0.0002 (0.0012) loss\_rpn\_box\_reg: 0.0025 (0.0028) time: 0.6631 data: 0.0290 max mem: 3501  
Epoch: [0] Total time: 0:00:38 (0.6371 s / it)  
Epoch: [1] [ 0/60] eta: 0:00:42 lr: 0.005000 loss: 0.2146 (0.2146) loss\_classifier: 0.0234 (0.0234) loss\_box\_reg: 0.0671 (0.0671) loss\_mask: 0.1167 (0.1167) loss\_objectness: 0.0007 (0.0007) loss\_rpn\_box\_reg: 0.0066 (0.0066) time: 0.7067 data: 0.0303 max mem: 3501  
Epoch: [1] [59/60] eta: 0:00:01 lr: 0.005000 loss: 0.1690 (0.1890) loss\_classifier: 0.0212 (0.0244) loss\_box\_reg: 0.0357 (0.0470) loss\_mask: 0.1072 (0.1140) loss\_objectness: 0.0001 (0.0005) loss\_rpn\_box\_reg: 0.0019 (0.0030) time: 1.2584 data: 0.0286 max mem: 3598  
Epoch: [1] Total time: 0:01:06 (1.1008 s / it)  
Epoch: [2] [ 0/60] eta: 0:01:09 lr: 0.005000 loss: 0.2185 (0.2185) loss\_classifier: 0.0276 (0.0276) loss\_box\_reg: 0.0569 (0.0569) loss\_mask: 0.1319 (0.1319) loss\_objectness: 0.0003 (0.0003) loss\_rpn\_box\_reg: 0.0018 (0.0018) time: 1.1653 data: 0.0833 max mem: 3598  
Epoch: [2] [59/60] eta: 0:00:01 lr: 0.005000 loss: 0.1705 (0.1888) loss\_classifier: 0.0199 (0.0228) loss\_box\_reg: 0.0416 (0.0506) loss\_mask: 0.1055 (0.1117) loss\_objectness: 0.0004 (0.0009) loss\_rpn\_box\_reg: 0.0025 (0.0029) time: 1.4117 data: 0.0296 max mem: 3598  
Epoch: [2] Total time: 0:01:21 (1.3545 s / it)  
Epoch: [3] [ 0/60] eta: 0:00:44 lr: 0.000500 loss: 0.1718 (0.1718) loss\_classifier: 0.0272 (0.0272) loss\_box\_reg: 0.0404 (0.0404) loss\_mask: 0.1031 (0.1031) loss\_objectness: 0.0003 (0.0003) loss\_rpn\_box\_reg: 0.0008 (0.0008) time: 0.7485 data: 0.0269 max mem: 3598  
Epoch: [3] [59/60] eta: 0:00:01 lr: 0.000500 loss: 0.1513 (0.1641) loss\_classifier: 0.0188 (0.0208) loss\_box\_reg: 0.0262 (0.0325) loss\_mask: 0.1087 (0.1078) loss\_objectness: 0.0002 (0.0006) loss\_rpn\_box\_reg: 0.0018 (0.0025) time: 1.5931 data: 0.0354 max mem: 3598  
Epoch: [3] Total time: 0:01:28 (1.4723 s / it)  
Epoch: [4] [ 0/60] eta: 0:01:39 lr: 0.000500 loss: 0.1058 (0.1058) loss\_classifier: 0.0058 (0.0058) loss\_box\_reg: 0.0104 (0.0104) loss\_mask: 0.0880 (0.0880) loss\_objectness: 0.0003 (0.0003) loss\_rpn\_box\_reg: 0.0013 (0.0013) time: 1.6625 data: 0.0223 max mem: 3598  
Epoch: [4] [59/60] eta: 0:00:01 lr: 0.000500 loss: 0.1613 (0.1578) loss\_classifier: 0.0222 (0.0206) loss\_box\_reg: 0.0284 (0.0280) loss\_mask: 0.1038 (0.1065) loss\_objectness: 0.0003 (0.0006) loss\_rpn\_box\_reg: 0.0013 (0.0021) time: 1.5290 data: 0.0344 max mem: 3598  
Epoch: [4] Total time: 0:01:28 (1.4679 s / it)  
Epoch: [5] [ 0/60] eta: 0:00:34 lr: 0.000500 loss: 0.1003 (0.1003) loss\_classifier: 0.0062 (0.0062) loss\_box\_reg: 0.0095 (0.0095) loss\_mask: 0.0829 (0.0829) loss\_objectness: 0.0001 (0.0001) loss\_rpn\_box\_reg: 0.0015 (0.0015) time: 0.5715 data: 0.0189 max mem: 3598  
Epoch: [5] [59/60] eta: 0:00:01 lr: 0.000500 loss: 0.1335 (0.1501) loss\_classifier: 0.0148 (0.0191) loss\_box\_reg: 0.0209 (0.0252) loss\_mask: 0.0936 (0.1035) loss\_objectness: 0.0004 (0.0004) loss\_rpn\_box\_reg: 0.0018 (0.0020) time: 1.5362 data: 0.0314 max mem: 3598  
Epoch: [5] Total time: 0:01:28 (1.4678 s / it)  
creating index...  
index created!  
Test: [ 0/50] eta: 0:00:07 model\_time: 0.1326 (0.1326) evaluator\_time: 0.0020 (0.0020) time: 0.1435 data: 0.0080 max mem: 3598  
Test: [49/50] eta: 0:00:00 model\_time: 0.1875 (0.1653) evaluator\_time: 0.0030 (0.0056) time: 0.2308 data: 0.0173 max mem: 3598  
Test: Total time: 0:00:09 (0.1905 s / it)  
Averaged stats: model\_time: 0.1875 (0.1653) evaluator\_time: 0.0030 (0.0056)  
Accumulating evaluation results...  
DONE (t=0.01s).  
Accumulating evaluation results...  
DONE (t=0.01s).  
IoU metric: bbox  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.842  
 Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.987  
 Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.936  
 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 ] = 0.511  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.855  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.386  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.884  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.884  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000  
 Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.775  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.892  
IoU metric: segm  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.754  
 Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.987  
 Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.880  
 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 ] = 0.315  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.767  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.345  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.802  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.802  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000  
 Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.688  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.810  
Epoch: [6] [ 0/60] eta: 0:00:56 lr: 0.000050 loss: 0.2243 (0.2243) loss\_classifier: 0.0388 (0.0388) loss\_box\_reg: 0.0630 (0.0630) loss\_mask: 0.1197 (0.1197) loss\_objectness: 0.0005 (0.0005) loss\_rpn\_box\_reg: 0.0024 (0.0024) time: 0.9385 data: 0.0548 max mem: 3598  
Epoch: [6] [59/60] eta: 0:00:01 lr: 0.000050 loss: 0.1289 (0.1505) loss\_classifier: 0.0152 (0.0198) loss\_box\_reg: 0.0174 (0.0247) loss\_mask: 0.0923 (0.1034) loss\_objectness: 0.0002 (0.0006) loss\_rpn\_box\_reg: 0.0013 (0.0019) time: 1.7674 data: 0.0334 max mem: 3598  
Epoch: [6] Total time: 0:01:48 (1.8028 s / it)  
Epoch: [7] [ 0/60] eta: 0:02:45 lr: 0.000050 loss: 0.1358 (0.1358) loss\_classifier: 0.0173 (0.0173) loss\_box\_reg: 0.0277 (0.0277) loss\_mask: 0.0882 (0.0882) loss\_objectness: 0.0005 (0.0005) loss\_rpn\_box\_reg: 0.0021 (0.0021) time: 2.7606 data: 0.0469 max mem: 3598  
Epoch: [7] [59/60] eta: 0:00:01 lr: 0.000050 loss: 0.1500 (0.1498) loss\_classifier: 0.0183 (0.0192) loss\_box\_reg: 0.0231 (0.0244) loss\_mask: 0.1026 (0.1037) loss\_objectness: 0.0002 (0.0006) loss\_rpn\_box\_reg: 0.0017 (0.0019) time: 1.6621 data: 0.0355 max mem: 3598  
Epoch: [7] Total time: 0:01:40 (1.6707 s / it)  
Epoch: [8] [ 0/60] eta: 0:01:21 lr: 0.000050 loss: 0.1167 (0.1167) loss\_classifier: 0.0123 (0.0123) loss\_box\_reg: 0.0065 (0.0065) loss\_mask: 0.0967 (0.0967) loss\_objectness: 0.0001 (0.0001) loss\_rpn\_box\_reg: 0.0011 (0.0011) time: 1.3542 data: 0.0170 max mem: 3598  
Epoch: [8] [59/60] eta: 0:00:01 lr: 0.000050 loss: 0.1445 (0.1495) loss\_classifier: 0.0183 (0.0190) loss\_box\_reg: 0.0230 (0.0252) loss\_mask: 0.1005 (0.1030) loss\_objectness: 0.0003 (0.0004) loss\_rpn\_box\_reg: 0.0019 (0.0020) time: 1.6169 data: 0.0341 max mem: 3598  
Epoch: [8] Total time: 0:01:34 (1.5714 s / it)  
Epoch: [9] [ 0/60] eta: 0:01:41 lr: 0.000005 loss: 0.1187 (0.1187) loss\_classifier: 0.0102 (0.0102) loss\_box\_reg: 0.0147 (0.0147) loss\_mask: 0.0924 (0.0924) loss\_objectness: 0.0000 (0.0000) loss\_rpn\_box\_reg: 0.0014 (0.0014) time: 1.6946 data: 0.0209 max mem: 3598  
Epoch: [9] [59/60] eta: 0:00:01 lr: 0.000005 loss: 0.1484 (0.1511) loss\_classifier: 0.0188 (0.0194) loss\_box\_reg: 0.0263 (0.0252) loss\_mask: 0.0994 (0.1037) loss\_objectness: 0.0003 (0.0007) loss\_rpn\_box\_reg: 0.0018 (0.0020) time: 1.5774 data: 0.0319 max mem: 3598  
Epoch: [9] Total time: 0:01:36 (1.6148 s / it)  
creating index...  
index created!  
Test: [ 0/50] eta: 0:00:07 model\_time: 0.1326 (0.1326) evaluator\_time: 0.0030 (0.0030) time: 0.1476 data: 0.0110 max mem: 3598  
Test: [49/50] eta: 0:00:00 model\_time: 0.1865 (0.1704) evaluator\_time: 0.0040 (0.0056) time: 0.2054 data: 0.0176 max mem: 3598  
Test: Total time: 0:00:09 (0.1954 s / it)  
Averaged stats: model\_time: 0.1865 (0.1704) evaluator\_time: 0.0040 (0.0056)  
Accumulating evaluation results...  
DONE (t=0.01s).  
Accumulating evaluation results...  
DONE (t=0.01s).  
IoU metric: bbox  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.838  
 Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.987  
 Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.936  
 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 ] = 0.515  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.850  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.388  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.881  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.881  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000  
 Average Recall (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.787  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.887  
IoU metric: segm  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.757  
 Average Precision (AP) @[ IoU=0.50 | area= all | maxDets=100 ] = 0.987  
 Average Precision (AP) @[ IoU=0.75 | area= all | maxDets=100 ] = 0.879  
 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 ] = 0.321  
 Average Precision (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.772  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 1 ] = 0.345  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets= 10 ] = 0.804  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= all | maxDets=100 ] = 0.804  
 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 ] = 0.688  
 Average Recall (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.813

model\_= get\_instance\_segmentation\_model(num\_classes)  
model\_.load\_state\_dict(torch.load('./model/mask\_R\_CNN\_person/mask\_R\_CNN\_person\_stateDict\_9.pth'))

<All keys matched successfully>



Intersection over Union(IoU)交并比，用于评价对象检测、图像分割的精度，计算公式如上图[3]所示。由于模型的参数变化，例如图像金子塔尺度（image pyramid scale），滑动窗口大小（sliding window size，即卷积核）、特征提取方法（feature extraction method）等，预测的边界（或锚框）与实际情况（ground-truth）完全匹配是不现实的。而IoU评估指标能很好的表述预测的精度。

除了IoU评价指标，同时提取一幅图像测试，测试结果显示能够很好的提取图像场景中的行人对象。

* 注意将模型和数据均转化到指定的同一device（GPU或CPU）下，否则提示数据类型错误。

img, \_ = dataset\_test[34]  
model\_.to(device)  
model\_.eval()  
with torch.no\_grad():  
 prediction = model\_([img.to(device)])

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



行人掩码包含在返回预测值的masks键下，其数据形状为，其中12为预测的行人数量，这与实际的基本吻合（行人中往往存在互相遮掩的情况），可以用于人流量的统计。

import torchvision.transforms as transforms  
print('掩码形状：',prediction[0]['masks'].shape)  
transforms.ToPILImage()(torch.sum(prediction[0]['masks'],dim=0)).convert("RGB") #Image.fromarray(prediction[0]['masks'][0, 0].mul(255).byte().cpu().numpy())

掩码形状： torch.Size([10, 1, 378, 745])



#### 2）人流量估算

直接将上述训练后的模型用于无人驾驶场景KITTI数据集，先随机提取一幅含有行人的图像，用该模型预测，查看预测结果。如果基本吻合，则说明该模型可以用于进一步的人流量分析。从观察结果来看，基本能够提取出场景内的行人。

img\_kitti\_fp=r'./data/0000000181.png'  
img\_kitti=Image.open(img\_kitti\_fp)  
img\_kitti



import torchvision.transforms as transforms  
device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')  
print('device:{}'.format(device))  
  
trans\_2tensor=transforms.Compose([transforms.ToTensor(),]) # 将图像转换为tensor  
img\_kitti\_tensor=trans\_2tensor(img\_kitti)  
  
model\_.to(device)  
model\_.eval()  
with torch.no\_grad():  
 img\_kitti\_pred= model\_([img\_kitti\_tensor.to(device)])  
print('估计行人数量={}'.format(img\_kitti\_pred[0]['masks'].shape[0]))  
transforms.ToPILImage()(torch.sum(img\_kitti\_pred[0]['masks'],dim=0)).convert("RGB")

device:cuda  
估计行人数量=16



1. 提取KITTI数据集图像的位置坐标。

import util\_misc,util\_A  
KITTI\_info\_fp=r'G:\data\2011\_09\_29\_drive\_0071\_sync\oxts\data'  
timestamps\_fp=r'G:\data\2011\_09\_29\_drive\_0071\_sync\image\_03\timestamps.txt'  
drive\_29\_0071\_info=util\_A.KITTI\_info(KITTI\_info\_fp,timestamps\_fp)  
drive\_29\_0071\_info\_coordi=drive\_29\_0071\_info[['lat','lon','timestamps\_']]  
util\_misc.print\_html(drive\_29\_0071\_info\_coordi)

|  | **lat** | **lon** | **timestamps\_** |
| --- | --- | --- | --- |
| **0** | 49.008650 | 8.398092 | 2011-09-29 13:54:59.990872576 |
| **1** | 49.008777 | 8.397611 | 2011-09-29 13:55:00.094612992 |
| **2** | 49.009162 | 8.396541 | 2011-09-29 13:55:00.198486528 |
| **3** | 49.008962 | 8.397075 | 2011-09-29 13:55:00.302340864 |
| **4** | 49.009505 | 8.395251 | 2011-09-29 13:55:00.406079232 |

1. 提取KITTI数据集图像（’2011\_09\_29\_drive\_0071\_sync’数据子集），并转换为tensor

import os  
from PIL import Image  
import torchvision.transforms as transforms  
from tqdm.auto import tqdm  
  
drive\_29\_0071\_img\_fp=util\_misc.filePath\_extraction(r'G:\data\2011\_09\_29\_drive\_0071\_sync\image\_03\data',['png'])  
drive\_29\_0071\_img\_fp\_list=util\_misc.flatten\_lst([[os.path.join(k,f) for f in drive\_29\_0071\_img\_fp[k]] for k,v in drive\_29\_0071\_img\_fp.items()])  
print("2011\_09\_29\_drive\_0071\_sync-数据子集图像数据：",len(drive\_29\_0071\_img\_fp\_list))  
  
trans\_2tensor=transforms.Compose([transforms.ToTensor(),]) # 将图像转换为tensor  
drive\_29\_0071\_img\_tensor=[trans\_2tensor(Image.open(i)) for i in tqdm(drive\_29\_0071\_img\_fp\_list)]

2011\_09\_29\_drive\_0071\_sync-数据子集图像数据： 1059  
  
  
  
 0%| | 0/1059 [00:00<?, ?it/s]

1. 加载已训练的模型，并预测数据集图像

import torch  
device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')  
model\_entire=torch.load('./model/mask\_R\_CNN\_person/mask\_R\_CNN\_person\_final.pth')  
model\_entire.eval()  
with torch.no\_grad():  
 drive\_29\_0071\_img\_pred=[model\_entire([i.to(device)])[0]['masks'].shape[0] for i in tqdm(drive\_29\_0071\_img\_tensor)]

0%| | 0/1059 [00:00<?, ?it/s]  
  
  
C:\Users\richi\anaconda3\envs\USDA\_database\lib\site-packages\torch\functional.py:445: UserWarning: torch.meshgrid: in an upcoming release, it will be required to pass the indexing argument. (Triggered internally at ..\aten\src\ATen\native\TensorShape.cpp:2157.)  
 return \_VF.meshgrid(tensors, \*\*kwargs) # type: ignore[attr-defined]

1. 显示人流分布密度

drive\_29\_0071\_info\_coordi['person\_num']=drive\_29\_0071\_img\_pred  
  
import plotly.express as px  
fig = px.density\_mapbox(drive\_29\_0071\_info\_coordi, lat='lat', lon='lon', z='person\_num', radius=10,  
 center=dict(lat=49.008645, lon=8.398104), zoom=18,  
 mapbox\_style="stamen-terrain")  
fig.show()



### 2.6.4.2 对象实例分割与对象统计

PyTorch图像/语义分割模型（semantic segmentation），包括[Faster R-CNN ResNet-50 FPN](https://arxiv.org/abs/1506.01497)[4]，[Mask R-CNN ResNet-50 FPN](https://arxiv.org/abs/1703.06870)[2]，其预先训练的模型采用的数据集为[COCO train2017](https://cocodataset.org/#home)⑤的子集[Pascal VOC](http://host.robots.ox.ac.uk/pascal/VOC/)⑧，包括有20个分类。['\_\_background\_\_', 'aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor']等。使用预先训练的模型，输入的图像期望已训练时相同的方式归一化处理。图像映射到[0,1]区间，使用mean =[0.485, 0.456, 0.406]和std =[0.229, 0.224, 0.225]进行归一化。图像变换的处理使用torchvision.transforms实现。

通过图像分割可以获取一些城市对象（Pascal VOC有20个分类），对于KITTI数据集而言，就可以获取每一位置的城市对象内容，这样就可以对城市空间的内容加以统计，并通过建立关联结构分析对象之间的关系（即在多处场景中，同时出现某些对象的可能性大小）。

代码的书写参考[FCN-RESNET101](https://pytorch.org/hub/pytorch_vision_fcn_resnet101/)[5]和[PyTorch for Beginners: Semantic Segmentation using torchvision](https://learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/)[6]。

* 关于[TORCH.SQUEEZE](https://pytorch.org/docs/stable/generated/torch.squeeze.html)⑨

可以根据指定轴，移除轴尺寸为1的轴，与torch.unsqueeze互逆。

x = torch.zeros(2, 1, 2, 1, 2)  
print('x.size:{}\ndim=default:{}\ndim=0:{}\ndim=1:{}'.format(x.size(),torch.squeeze(x).size(),torch.squeeze(x,dim=0).size(),torch.squeeze(x,dim=1).size()))

x.size:torch.Size([2, 1, 2, 1, 2])  
dim=default:torch.Size([2, 2, 2])  
dim=0:torch.Size([2, 1, 2, 1, 2])  
dim=1:torch.Size([2, 2, 1, 2])

#### 1）FCN-RESNET101对象实例分割

直接读取PyTorch预训练的模型，用于新场景的应用。

import torch  
x=torch.tensor([1, 2, 3, 4])  
print('dim=0',torch.unsqueeze(x, 0))  
print('dim=1',torch.unsqueeze(x, 1))

dim=0 tensor([[1, 2, 3, 4]])  
dim=1 tensor([[1],  
 [2],  
 [3],  
 [4]])

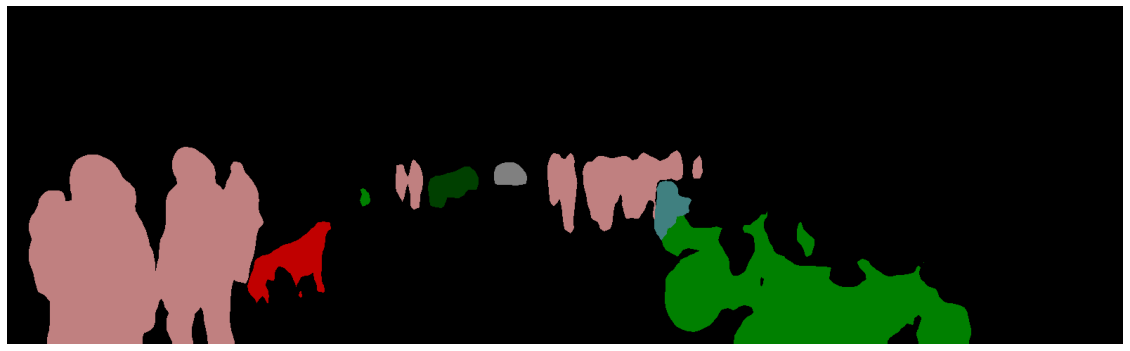
import torch,os  
import util\_misc  
# A-加载模型  
from torchvision import models  
fcn=models.segmentation.fcn\_resnet101(pretrained=True).eval()   
  
# B-加载数据集  
drive\_29\_0071\_img\_fp=util\_misc.filePath\_extraction(r'G:\data\2011\_09\_29\_drive\_0071\_sync\image\_03\data',['png'])  
drive\_29\_0071\_img\_fp\_list=util\_misc.flatten\_lst([[os.path.join(k,f) for f in drive\_29\_0071\_img\_fp[k]] for k,v in drive\_29\_0071\_img\_fp.items()])  
# 指定运算设备,GPU or CPU  
device=torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')

Downloading: "https://download.pytorch.org/models/fcn\_resnet101\_coco-7ecb50ca.pth" to C:\Users\richi/.cache\torch\hub\checkpoints\fcn\_resnet101\_coco-7ecb50ca.pth  
  
  
  
 0%| | 0.00/208M [00:00<?, ?B/s]

# C-映射图像分割的类标（类别）  
def decode\_segmap\_FCN\_RESNET101(image, nc=21):   
 '''  
 function - fcn\_resnet101模型，图像分割的类别给与颜色标识  
 '''   
 import numpy as np  
  
 label\_colors = np.array([(0, 0, 0), # 0=background  
 # 1=aeroplane, 2=bicycle, 3=bird, 4=boat, 5=bottle  
 (128, 0, 0), (0, 128, 0), (128, 128, 0), (0, 0, 128), (128, 0, 128),  
 # 6=bus, 7=car, 8=cat, 9=chair, 10=cow  
 (0, 128, 128), (128, 128, 128), (64, 0, 0), (192, 0, 0), (64, 128, 0),  
 # 11=dining table, 12=dog, 13=horse, 14=motorbike, 15=person  
 (192, 128, 0), (64, 0, 128), (192, 0, 128), (64, 128, 128), (192, 128, 128),  
 # 16=potted plant, 17=sheep, 18=sofa, 19=train, 20=tv/monitor  
 (0, 64, 0), (128, 64, 0), (0, 192, 0), (128, 192, 0), (0, 64, 128)])  
  
 r = np.zeros\_like(image).astype(np.uint8)  
 g = np.zeros\_like(image).astype(np.uint8)  
 b = np.zeros\_like(image).astype(np.uint8)  
  
 for l in range(0, nc):  
 idx = image == l  
 r[idx] = label\_colors[l, 0]  
 g[idx] = label\_colors[l, 1]  
 b[idx] = label\_colors[l, 2]  
  
 rgb = np.stack([r, g, b], axis=2)  
 return rgb

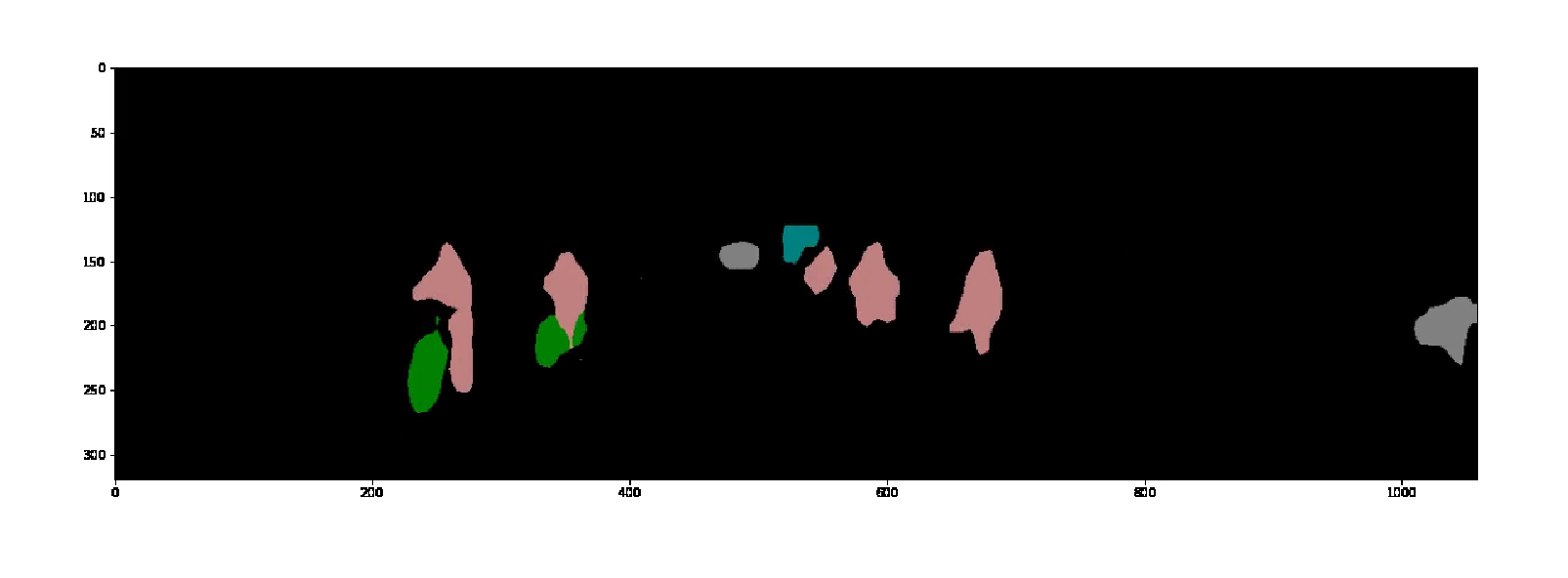
# D-预测图像分割结果，并打印  
def segmentation\_FCN\_RESNET101\_plot(net, path, show\_orig=True, dev='cuda',img\_resize=640,figsize=(20, 20)):  
 '''  
 function - 应用 torchvision.models.segmentation.fcn\_resnet101预测图像，并打印显示分割预测结果  
 '''  
 import matplotlib.pyplot as plt  
 from PIL import Image  
 import torch  
 import torchvision.transforms as T  
   
 img = Image.open(path)  
 plt.figure(figsize=figsize)  
 if show\_orig: plt.imshow(img); plt.axis('off'); plt.show()  
 # Comment the Resize and CenterCrop for better inference results  
 trf = T.Compose([  
 T.Resize(img\_resize),   
 # T.CenterCrop(224),   
 T.ToTensor(),   
 T.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])])  
 inp = trf(img).unsqueeze(0).to(dev)  
 out = net.to(dev)(inp)['out']  
 om = torch.argmax(out.squeeze(), dim=0).detach().cpu().numpy()  
 rgb = decode\_segmap\_FCN\_RESNET101(om)  
 plt.figure(figsize=figsize)  
 plt.imshow(rgb); plt.axis('off'); plt.show()  
  
# E-提取一副图像预测  
segmentation\_FCN\_RESNET101\_plot(fcn, drive\_29\_0071\_img\_fp\_list[550],dev=device,img\_resize=480) # 可以通过调整img\_resize参数，即调整图像大小来减少GPU使用量，避免GPU溢出





# F-计算KITTI-drive\_29\_0071\_img子集的所有图像分割，返回结果并动态显示  
def segmentation\_FCN\_RESNET101\_animation(net, paths,save\_path='./animation.mp4' ,dev='cuda',img\_resize=640,interval=150,figsize=(20, 20)):  
 '''  
 function - 应用 torchvision.models.segmentation.fcn\_resnet101预测图像，并打印显示分割预测结果的动画  
 '''  
 import matplotlib.pyplot as plt  
 from PIL import Image  
 import torch  
 import torchvision.transforms as T  
 import matplotlib.animation as animation  
 from tqdm.auto import tqdm  
   
 plt.figure(figsize=figsize)  
 imgs=[]  
 fig=plt.figure(figsize=figsize)  
 for path in tqdm(paths):  
 img=Image.open(path)  
 trf=T.Compose([ T.Resize(img\_resize), T.ToTensor(), T.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])])  
 inp=trf(img).unsqueeze(0).to(dev)  
 out=net.to(dev)(inp)['out']  
 sementic\_seg=torch.argmax(out.squeeze(), dim=0).detach().cpu().numpy()  
 rgb=decode\_segmap\_FCN\_RESNET101(sementic\_seg)   
 imgs.append([plt.imshow(rgb,animated=True,)])   
 anima=animation.ArtistAnimation(fig,imgs,interval=interval, blit=True,repeat\_delay=1000)  
 anima.save(save\_path)  
 print(".mp4 saved.")  
 return anima,imgs  
  
from tqdm.auto import tqdm  
import matplotlib.pyplot as plt  
from IPython.display import HTML  
  
save\_path=r'./results/segmentation\_FCN\_RESNET101\_animation.mp4'  
drive\_29\_0071\_img\_fp\_list.sort()  
anima,\_=segmentation\_FCN\_RESNET101\_animation(fcn, drive\_29\_0071\_img\_fp\_list, save\_path,dev='cuda',img\_resize=320,figsize=(20,8))  
HTML(anima.to\_html5\_video())

0%| | 0/1059 [00:00<?, ?it/s]  
  
  
.mp4 saved.



#### 2）对象统计与关联网络结构

对象统计是确定每一位置空间下存在有哪些物，因为使用的训练模型包括20个分类，因此只能识别这些已训练的对象，而树木、建筑等不能识别；再者，该模型只返回对象掩码（不同对象，不同索引，但是同一对象不可再分，例如对人数的统计不能实现）。但是对每一位置空间已有对象的识别可以初步判断该空间的特征；并通过统计所有位置下，与每一对象同时存在的其它对象的频数，可以应用[NetworkX](https://networkx.org/)⑩库构建网络结构，直观的观察对象之间的空间存在关系。并统计每一位置出现对象种类的数量，以热力图的形式可视化，可以初步判断不同地段的对象混杂程度，通常对象种类越丰富的区域，空间表现出的活力越高。

首先调整了预测函数，使返回值为对象实例分割索引及对应的分类名称。

def segmentation\_FCN\_RESNET101(net, path, show\_orig=True, dev='cuda',img\_resize=640):  
 '''  
 function - 应用 torchvision.models.segmentation.fcn\_resnet101预测图像，返回预测结果  
 '''  
 from PIL import Image  
 import torch  
 import torchvision.transforms as T  
 import numpy as np   
   
 seg\_FCN\_RESNET101\_classi\_mapping={0:'background',1:'aeroplane', 2:'bicycle', 3:'bird', 4:'boat', 5:'bottle',6:'bus', 7:'car', 8:'cat', 9:'chair', 10:'cow',11:'dining table', 12:'dog',   
 13:'horse', 14:'motorbike', 15:'person',16:'potted plant', 17:'sheep', 18:'sofa', 19:'train', 20:'tv/monitor'}  
 img=Image.open(path)  
 trf=T.Compose([  
 T.Resize(img\_resize),   
 #T.CenterCrop(224),   
 T.ToTensor(),   
 T.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])])  
 inp=trf(img).unsqueeze(0).to(dev)  
 out=net.to(dev)(inp)['out']  
 sementic\_seg=torch.argmax(out.squeeze(), dim=0).detach().cpu().numpy()  
 sementic\_seg\_classi=[seg\_FCN\_RESNET101\_classi\_mapping[i] for i in np.unique(sementic\_seg)]  
 return (np.unique(sementic\_seg).tolist(),sementic\_seg\_classi)

sementic\_seg=segmentation\_FCN\_RESNET101(fcn, drive\_29\_0071\_img\_fp\_list[200],dev=device,img\_resize=280)  
print('预测的图像包含的对象标签：{}'.format(sementic\_seg))

预测的图像包含的对象标签：([0, 2, 7, 9, 11, 15], ['background', 'bicycle', 'car', 'chair', 'dining table', 'person'])

计算所有图像，返回预测结果。

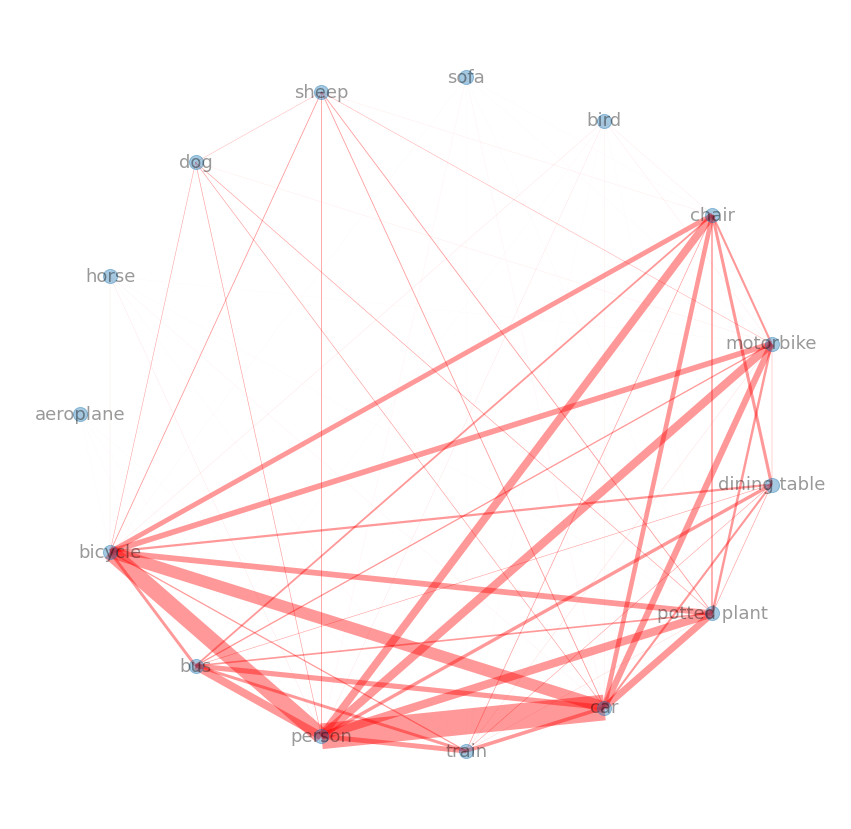
from tqdm.auto import tqdm  
drive\_29\_0071\_img\_seg\_pred=[segmentation\_FCN\_RESNET101(fcn, img,dev=device,img\_resize=280) for img in tqdm(drive\_29\_0071\_img\_fp\_list)]

0%| | 0/1059 [00:00<?, ?it/s]

通过查看预测结果，可以发现场景中存在狗和火车。而一些未出现的类，例如sheep、hourse等应该是狗在不同影像位置下识别的错误。其它的分类错误，可能与场景中出现的海报等图画有关。观察最终的网络结构，因为将每一个对象与其它对象在不同场景中共存的情况进行统计，即计算共存对象的频数，将该频数或者其倍数作为网络边的权重值，并通过粗细显示。因此可以观察到，线越细的对象在整个1059张图像所代表的位置空间下，出现的位置较少；而线越粗的则出现位置相对较多。经常同时出现的对象为’person’，‘car’和’bicycle’，次之的有’chair’，’motorbike’和’potted plant’等。有些信息的出现是合乎常理，例如场景中骑车的人，因此这些分析结果似乎价值偏弱；但是，’chair’和’potted plant’的出现，可以判定该条街道室外活动的主要内容，餐饮、休闲等。

def count\_list\_frequency(lst):  
 '''  
 function - 计算列表的频数  
 '''  
 freq={}  
 for i in lst:  
 if(i in freq):  
 freq[i]+=1  
 else:  
 freq[i]=1  
 return freq   
   
  
def objects\_network\_PascalVOC(seg\_idxs,figsize=(15,15),layout='spring\_layout',w\_ratio=0.5):  
 '''  
 function - 根据连续的图像分割数据，计算各个一对象（真实世界存在的物）与其它对象对应的数量，构建网络结构，分析相互关系  
 '''  
 import numpy as np  
 import networkx as nx  
 import matplotlib.pyplot as plt  
   
   
 seg\_FCN\_RESNET101\_classi\_mapping={0:'background',1:'aeroplane', 2:'bicycle', 3:'bird', 4:'boat', 5:'bottle',6:'bus', 7:'car', 8:'cat', 9:'chair', 10:'cow',11:'dining table', 12:'dog',   
 13:'horse', 14:'motorbike', 15:'person',16:'potted plant', 17:'sheep', 18:'sofa', 19:'train', 20:'tv/monitor'}   
 flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]  
 unique\_idxs=np.unique(flatten\_lst(seg\_idxs)).tolist()  
 unique\_idxs\_=list(filter(lambda x: x != 0, unique\_idxs))  
 print('存在的对象有：',[(i,seg\_FCN\_RESNET101\_classi\_mapping[i]) for i in unique\_idxs])  
   
 # 01-收集每一对象所有存在时刻包含的其他对象  
 object\_associations={}  
 for obj in unique\_idxs\_:  
 obj\_associations\_list=flatten\_lst([i for i in seg\_idxs if obj in i])  
 obj\_associations\_list\_=list(filter(lambda x: x != 0,obj\_associations\_list))  
 object\_associations[obj]=obj\_associations\_list\_  
 # print(object\_associations)  
   
 # 02-计算每一对象，包含其他对象的频数  
 object\_associations\_frequency={}  
 for k,v in object\_associations.items():  
 v\_=list(filter(lambda x: x != k,v))  
 object\_associations\_frequency[k]=count\_list\_frequency(v\_)  
 # print(object\_associations\_frequency)  
   
 # 03-构建网络，以频数或其倍数为权重  
 fig, ax = plt.subplots(figsize=figsize)  
 G=nx.Graph()  
 layout\_dic={  
 'spring\_layout':nx.spring\_layout,   
 'random\_layout':nx.random\_layout,  
 'circular\_layout':nx.circular\_layout,  
 'kamada\_kawai\_layout':nx.kamada\_kawai\_layout,  
 'shell\_layout':nx.shell\_layout,  
 'spiral\_layout':nx.spiral\_layout,  
 }  
   
   
 for k,v in object\_associations\_frequency.items():  
 for obj,w in v.items():  
 G.add\_edge(seg\_FCN\_RESNET101\_classi\_mapping[k],seg\_FCN\_RESNET101\_classi\_mapping[obj],weight=w\*w\_ratio)  
   
 pos=layout\_dic[layout](G)   
 weights=nx.get\_edge\_attributes(G,'weight').values()  
 nx.draw(G,pos,with\_labels=True,font\_size=18,alpha=0.4, edge\_color="r",node\_size=200,width=list(weights))  
  
seg\_idxs=[i[0] for i in drive\_29\_0071\_img\_seg\_pred]  
objects\_network\_PascalVOC(seg\_idxs,layout='shell\_layout',w\_ratio=0.03)

存在的对象有： [(0, 'background'), (1, 'aeroplane'), (2, 'bicycle'), (3, 'bird'), (6, 'bus'), (7, 'car'), (9, 'chair'), (11, 'dining table'), (12, 'dog'), (13, 'horse'), (14, 'motorbike'), (15, 'person'), (16, 'potted plant'), (17, 'sheep'), (18, 'sofa'), (19, 'train')]



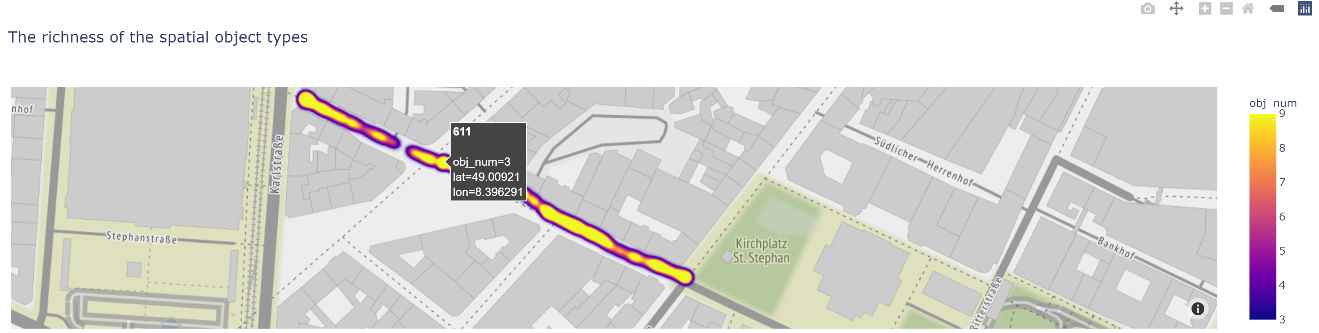
* 空间对象种类的丰富程度

import util\_A  
KITTI\_info\_fp=r'G:\data\2011\_09\_29\_drive\_0071\_sync\oxts\data'  
timestamps\_fp=r'G:\data\2011\_09\_29\_drive\_0071\_sync\image\_03\timestamps.txt'  
drive\_29\_0071\_info=util\_A.KITTI\_info(KITTI\_info\_fp,timestamps\_fp)  
drive\_29\_0071\_info\_coordi=drive\_29\_0071\_info[['lat','lon','timestamps\_']]  
  
obj\_num=[len(i) for i in seg\_idxs]  
drive\_29\_0071\_info\_coordi['obj\_num']=obj\_num  
drive\_29\_0071\_info\_coordi['idx']=drive\_29\_0071\_info\_coordi.index  
drive\_29\_0071\_info\_coordi

|  | **lat** | **lon** | **timestamps\_** | **obj\_num** | **idx** |
| --- | --- | --- | --- | --- | --- |
| **0** | 49.008650 | 8.398092 | 2011-09-29 13:54:59.990872576 | 3 | 0 |
| **1** | 49.008777 | 8.397611 | 2011-09-29 13:55:00.094612992 | 3 | 1 |
| **2** | 49.009162 | 8.396541 | 2011-09-29 13:55:00.198486528 | 3 | 2 |
| **3** | 49.008962 | 8.397075 | 2011-09-29 13:55:00.302340864 | 3 | 3 |
| **4** | 49.009505 | 8.395251 | 2011-09-29 13:55:00.406079232 | 4 | 4 |
| **...** | ... | ... | ... | ... | ... |
| **1054** | 49.009215 | 8.396286 | 2011-09-29 13:56:49.458599424 | 4 | 1054 |
| **1055** | 49.009353 | 8.395764 | 2011-09-29 13:56:49.562463744 | 4 | 1055 |
| **1056** | 49.008706 | 8.397888 | 2011-09-29 13:56:49.666327808 | 3 | 1056 |
| **1057** | 49.009215 | 8.396288 | 2011-09-29 13:56:49.770316544 | 3 | 1057 |
| **1058** | 49.009079 | 8.396812 | 2011-09-29 13:56:49.874179584 | 3 | 1058 |

1059 rows × 5 columns

import plotly.express as px  
fig = px.density\_mapbox(drive\_29\_0071\_info\_coordi, lat='lat', lon='lon', z='obj\_num', radius=10,  
 center=dict(lat=49.008645, lon=8.398104), zoom=18,  
 mapbox\_style="stamen-terrain",  
 title='The richness of the spatial object types',  
 #hover\_data=['idx'],  
 hover\_name='idx'  
 )  
fig.show()



注释（Notes）：

① TorchVision Object Detection Finetuning Tutorial的Googel Colab版本，（<https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/torchvision_finetuning_instance_segmentation.ipynb>）。

② Penn-Fudan Database （for Pedestrian Detection and Segmentation，（<https://www.cis.upenn.edu/~jshi/ped_html>）。

③ KITTI数据集，由德国卡尔斯鲁厄理工学院（Karlsruhe Institute of Technology (KIT) ）和丰田美国技术研究院（Toyota Technological Institute at Chicago (TTI-C)）联合创办，是目前国际上最大的自动驾驶场景下的计算机视觉算法评测数据集。该数据集用于评测立体图像(stereo)，光流(optical flow)，视觉测距(visual odometry)，3D物体检测(object detection)和3D跟踪(tracking)等计算机视觉技术在车载环境下的性能。KITTI包含市区、乡村和高速公路等场景采集的真实图像数据，每张图像中最多达15辆车和30个行人，还有各种程度的遮挡与截断，（<https://www.cvlibs.net/datasets/kitti>）。

④ PennFudan数据集，用于行人检测的图像数据库。取自校园和城市街道等场景，每张图片中至少有一个行人，（<https://www.cis.upenn.edu/~jshi/ped_html>）。

⑤ COCO数据集，是一个大规模对象检测（object detection）、分割（segmentation）和标注（captioning）数据集，（<https://cocodataset.org/#home>）。

⑥ COCO数据集91个分割类，（<https://github.com/nightrome/cocostuff>）。

⑦ torchvision.transforms，（<https://pytorch.org/vision/0.9/transforms.html>）。

⑧ The PASCAL Visual Object Classes Homepage，（<http://host.robots.ox.ac.uk/pascal/VOC>）。

⑨ 关于TORCH.SQUEEZE，（<https://pytorch.org/docs/stable/generated/torch.squeeze.html>）。

⑩ NetworkX， Python编程语言软件包,可用于创建、操作和学习复杂网络（图）的结构、动态和功能等，（<https://networkx.org>）。

参考文献（References）:

[1] TorchVision Object Detection Finetuning Tutorial, <https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html>.

[2] He, K., Gkioxari, G., Dollár, P. & Girshick, R. Mask R-CNN. (2017).<https://arxiv.org/abs/1703.06870>.

[3] Intersection over Union (IoU) for object detection，<https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>.

[4] Ren, S., He, K., Girshick, R. & Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. (2015). <https://arxiv.org/abs/1506.01497>.

[5] Fully-Convolutional Network model with ResNet-50 and ResNet-101 backbones, <https://pytorch.org/hub/pytorch_vision_fcn_resnet101>.

[6] PyTorch for Beginners: Semantic Segmentation using torchvision, <https://learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/>.