FPN

陈乐偲

FPN（区域生成网络），用于目标检测中生成ROI（感兴趣区域），在ROI的基础上再加上类似于图像分类的方法，便可以做完整的目标检测。

数据集选用cityscapes数据集，目标是检测出行人。

1.代码

1.0 prepare

完成准备工作（可略过）

- 自定义数据集，继承torchvision的Cityscape数据集，加以person方式读取做行人检测
- 自定义数据集transform方式，主要实现从json格式储存的检测框转变为tensor方式储存的bounding box

详见data.py

- jsonTransform 舍弃长宽太小的目标框，并且将一张图片的目标框数目限制在一定范围内，并返回检测框和标签（作为掩码使用）
- show_bboxes_img 用于显示目标检测结果

```
def jsonTransform(x):
    #将json格式读取入bbox，返回bbox和cls，bbox最多为MAX_OB个，bbox归一化至[0,1),clss为0/1
    obs = x['objects']
    boxes = torch.Tensor(np.zeros((MAX_OB,4))).to(device)
    clses = torch.Tensor(np.zeros(MAX_OB)).to(device)

    cnt = 0
    for ob in obs:
        if  ob['label'] == 'ignore':
            continue
        #舍弃太小的物体
        eps = 1e-3
        if ob['bbox'][2] / 2048 < eps and  ob['bbox'][3] / 1024 < eps:
            continue
        else:
            boxes[cnt] = torch.Tensor(ob['bbox']).to(device)
            boxes[cnt,0] /= 2048
            boxes[cnt,1] /= 1024
            boxes[cnt,2] /= 2048
            boxes[cnt,3] /= 1024
```

```
            clses[cnt] = 1
            cnt += 1
            if cnt == MAX_OB:
                break

    return boxes,clses

#在图片上显示检测框
def show_bboxes_img(img, boxes, savepath):
    img = img.float().detach().cpu().permute(1,2,0).numpy()
    h,w,_ = img.shape
    boxes = boxes.detach().cpu().numpy()
    ax =plt.subplot(1,1,1)
    plt.imshow(img)

    for box in boxes:
        box[0] = box[0]*2048
        box[1] = box[1]*1024
        box[2] = box[2]*2048
        box[3] = box[3]*1024

        ret = patches.Rectangle((box[0],box[1]),box[2],box[3],linewidth=1,
edgecolor='r',facecolor='none')
        ax.add_patch(ret)

    plt.axis('off')
    plt.savefig(savepath)
```

图片标注显示

1.1 box

实现和bouding box相关的函数，box有几种储存方式

- (x1,y1,x2,y2) 用左上角和右下角两点坐标表示
- (x,y,w,h) 用左上角坐标和大小（宽高）表示
- (cx,cy,w,h) 用中心点坐标和大小表示

在matplotlib于cityscape中使用第二种即xywh方式，代码中也与采用xywh的方式，但torchvision中使用的是第一种即xyxy的形式，调用函数时需要转换。我统一采用xywh格式储存，再需要调用torchvision中相关函数时，如极大值抑制，调用以下的bbox_wh2xy函数进行格式转换。

详见bbox.py

```
def bbox_wh2xy(box1):
    box2 = torch.zeros(box1.shape).to(device)
    x,y,w,h = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    box2[:,0] = x
    box2[:,1] = y
    box2[:,2] = x + w
    box2[:,3] = y + h
    return box2
```

在判断生成框是否有效时，需要计算交并比。

- 批交并比，输入两批次数目相同的检测框，输出数目与批次数量相同的检测框两两的交并比
- 联合交并比，输入两组检测框A,B，返回A中的每一个元素和B中的每一个元素分别的交并比，可以通过批交并比reshape后实现，或者如下直接实现

```
#批交并比
def bbox_batch_iou(box1, box2):
    #将xywh格式的bbox转化为xyxy两点式

    box1 = bbox_wh2xy(box1)
    box2 = bbox_wh2xy(box2)

    #使用两点式计算iou
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]

    inter_rect_x1 = torch.max(b1_x1, b2_x1)
    inter_rect_y1 = torch.max(b1_y1, b2_y1)
    inter_rect_x2 = torch.min(b1_x2, b2_x2)
    inter_rect_y2 = torch.min(b1_y2, b2_y2)

    inter_area = torch.max(inter_rect_x2 - inter_rect_x1,
torch.zeros(inter_rect_x2.shape).to(device)) * torch.max(
        inter_rect_y2 - inter_rect_y1,
torch.zeros(inter_rect_x2.shape).to(device))

    b1_area = (b1_x2 - b1_x1) * (b1_y2 - b1_y1)
    b2_area = (b2_x2 - b2_x1) * (b2_y2 - b2_y1)
    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou

#联合交并比，转化为xyxyx格式

# REF: https://github.com/chenyuntc/simple-faster-rcnn-pytorch
def bbox_union_iou(bbox_a, bbox_b):
    bbox_a = bbox_wh2xy(bbox_a)
```

```
    bbox_b = bbox_wh2xy(bbox_b)

    #利用广播机制计算交并比
    tl = torch.maximum(bbox_a[:, None, :2], bbox_b[:, :2])
    br = torch.minimum(bbox_a[:, None, 2:], bbox_b[:, 2:])

    area_i = torch.prod(br - tl, axis=2) * (tl < br).all(axis=2)
    area_a = torch.prod(bbox_a[:, 2:] - bbox_a[:, :2], axis=1)
    area_b = torch.prod(bbox_b[:, 2:] - bbox_b[:, :2], axis=1)
    return area_i / (area_a[:, None] + area_b - area_i)
```

计算偏移与bbox之间的转换，需要两个函数

- bbox2loc 给定两个检测框，返回原始框（锚框）到目标框（真实框）之间的偏移loc
- loc2bbox 给定原始框和偏移量，计算得到目标框

```
#输入xywh格式的box，输出loc
def bbox2loc(src, dst):

    x,y,w,h = src[:,0], src[:,1], src[:,2], src[:,3]
    bx,by,bw,bh = dst[:,0], dst[:,1], dst[:,2], dst[:,3]

    eps = torch.finfo(h.dtype).eps
    h[h<eps] = eps
    w[w<eps] = eps

    dx = (bx-x) / w
    dy = (by-y) / h
    dw = torch.log(bw / w)
    dh = torch.log(bh / h)

    loc = torch.stack((dx,dy,dw,dh),1)
    return loc

def loc2bbox(src, loc):
    dst = torch.zeros(src.shape).to(device)
    dx,dy,dw,dh = loc[:,0], loc[:,1], loc[:,2], loc[:,3]
    x,y,w,h = src[:,0], src[:,1], src[:,2], src[:,3]

    bx = dx*w + x
    by = dy*h + y
    bw = torch.exp(dw) * w
    bh = torch.exp(dh) * h

    dst[:,0], dst[:,1], dst[:,2], dst[:,3] = bx, by, bw, bh
    return dst
```

1.2 anchor

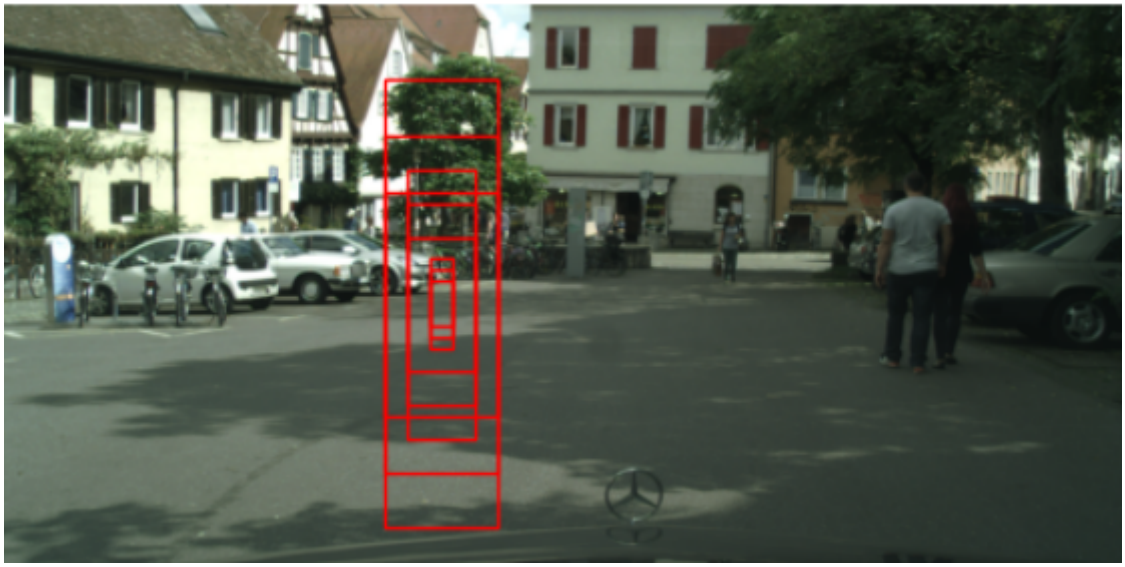
锚框的定义时RPN中重要的一环，RPN的任务是在基准锚框的基础上，学习得到基准锚框到真实检测框之间的偏移量，尽可能将基准锚框接近检测框，从而生成感兴趣区域ROI。

锚框生成分为两步

- 给定中心点，生成一系列围绕中心点的锚框
- 给定图片，将图片划分为网络，对每个网络的中心点生成锚框，在前一步对的基础上用矩阵运算可以轻松实现。

```
#给定长宽比例ratios和尺寸scales,生成以原点(0,0)为中心的一列点
def gererate_base_anchor(ratios=[4,6,8],scales=[0.1,0.06,0.02]):
    anchor = torch.Tensor(len(ratios)*len(scales),4).to(device)
    #使用循环遍历，也可以使用meshgrid优雅实现
    for i,r in enumerate(ratios):
        for j,s in enumerate(scales):
            w = s
            h = r * w
            x0 = -w/2
            y0 = -h/2
            anchor[i*len(ratios)+j,:] = torch.Tensor([x0,y0,w,h]).to(device)
    return anchor

#给定网格分割数，生成一张图片的所有锚框
def gererate_all_anchor(num_gridx=32,num_gridy=16):

    x = (torch.arange(0,num_gridx).to(device) + 0.5) / num_gridx
    y = (torch.arange(0,num_gridy).to(device) + 0.5) / num_gridy

    x,y = torch.meshgrid(x,y)
    x = x.reshape(-1)
    y = y.reshape(-1)
    w = torch.zeros(x.shape).to(device)
    h = torch.zeros(x.shape).to(device)

    shift = torch.stack([x,y,w,h],axis=1)
    shift = shift.unsqueeze(0).permute(1,0,2)

    anchor = gererate_base_anchor()
    anchor = anchor.unsqueeze(0)

    anchors = anchor + shift
    anchors = anchors.reshape(-1,4)
    return anchors
```

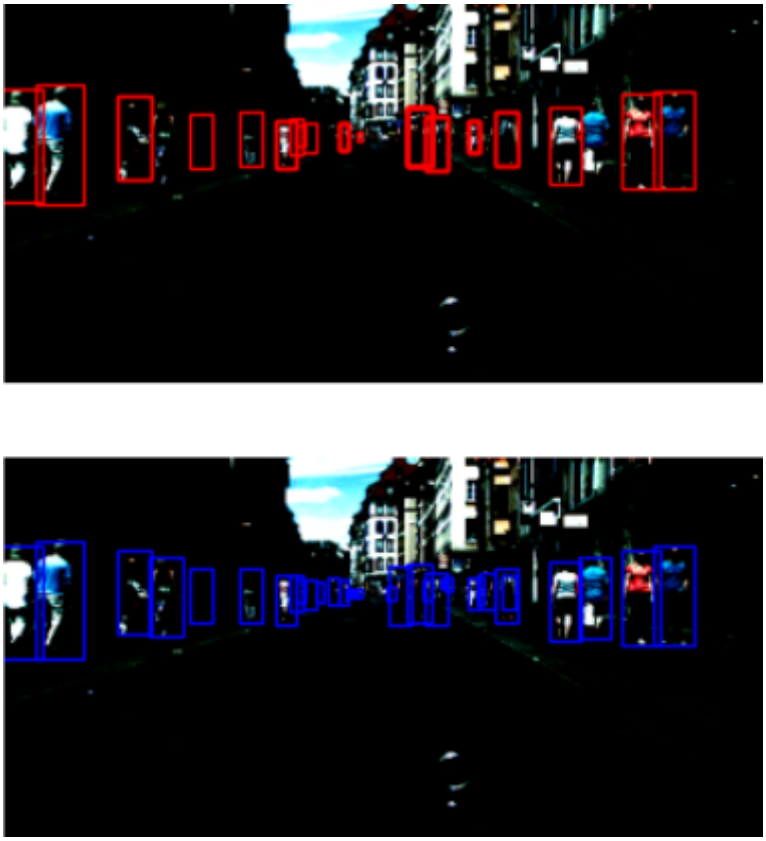
给定一个中心点，我们将生成如下的锚框。



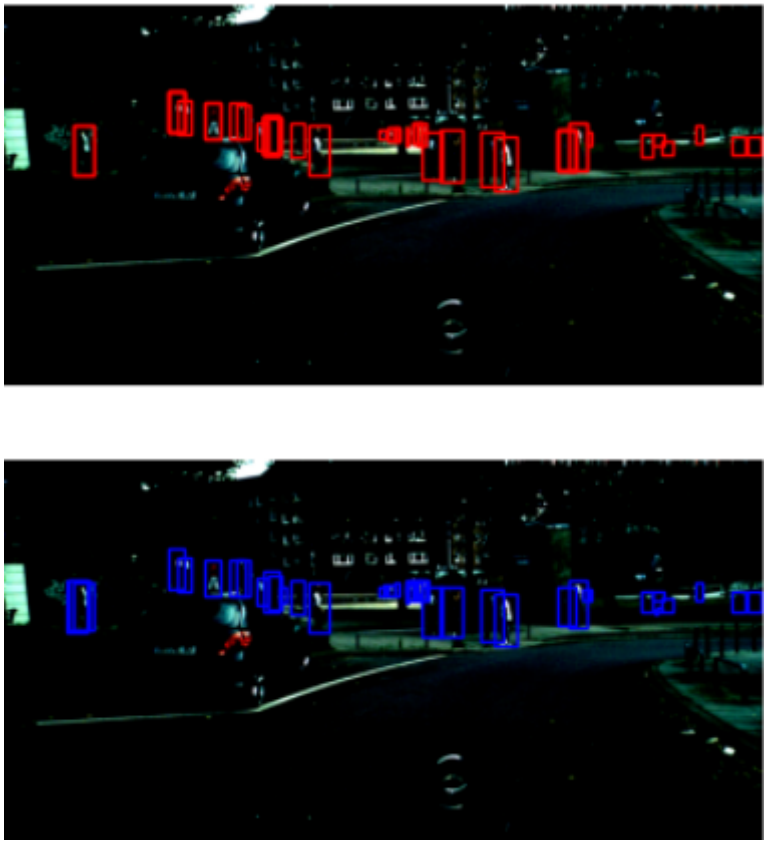
锚框示例图

在原始锚框的基础上，选取和目标框接近（交并比大）的作为正样本，将其抽取出来，利用bbox2loc计算将其偏移至目标所需要的偏移量，变换锚框后得到下面的结果。

NOTE:红色框表示用锚框偏移后的结果，蓝色框表示真实的检测框标注



偏移锚框得到真实检测框示例



采用更小的分辨率偏移示例

NOTE:图像在dataloader中采用了归一化，导致输出结果观感较差。

1.3 model

模型，FPN的主干，由如下部分组成

- 使用vgg提取图片特征，特征为原图下采样1/16之后的结果
- 使用卷积层提取隐层
- 在隐层的基础上，定义loc头，输出锚框到检测框之间的偏移和缩放比
- 在隐层的基础上，定义score头，输出每个锚框含有物体的概率

```
class FPN(nn.Module):
    #FPN,输出每个锚点的归一化之后的loc和pred,
    def __init__(self,n_anchors=9,in_channel=512,out_channel=512):
        super().__init__()
        self.extracter = get_vgg16_extracter()
        self.conv = nn.Conv2d(in_channel,out_channel,3,1,1)
        self.loc = nn.Conv2d(out_channel, n_anchors*4 ,3,1,1)
        self.score = nn.Conv2d(out_channel,n_anchors, 3,1,1)

    def forward(self,x):
        x = self.extracter(x)
        n_batch = x.shape[0]
        x = F.relu(x)
        x = self.conv(x)
        pred_score = torch.sigmoid(self.score(x))
        pred_loc = torch.sigmoid(self.loc(x))
```

```
        pred_score = pred_score.reshape(n_batch,-1)
        pred_loc = pred_loc.reshape(n_batch,-1,4)
        return pred_loc, pred_score
```

其中使用到了get_vgg_extracter函数，加载预训练的vgg网络，并冻结某些层

```
def get_vgg16_extracter():
    model = vgg16(pretrained=True)
    features = list(model.features)[:30]

    #冻结某些层
    for layer in features[:10]:
        for p in layer.parameters():
            p.requires_grad = False

    features = nn.Sequential(*features)
    #用vgg16提取特征返回16倍下采样后的结果
    return features
```

同时定义FPNTrainer用于训练整个FPN，使用方法如主函数：

```
if __name__ == '__main__':

    BATCH = 8
    IMG_SIZE = (1024,2048)
    myTransform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize(IMG_SIZE[0])
        #transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

    dataset = myCityscapes('./datasets/', split='train', mode='person',
target_type='person',transform=myTransform,target_transform=jsonTransform)
    dataloader = DataLoader(dataset, batch_size=BATCH,shuffle=True)
    validset = myCityscapes('./datasets/', split='val', mode='person',
target_type='person',transform=myTransform,target_transform=jsonTransform)
    validloader = DataLoader(dataset, batch_size=BATCH,shuffle=True)

    fpn = FPNTrainer(IMG_SIZE)
    fpn.train(dataloader,validloader)
```

FPN主要分为训练和测试两大部分

训练时需要计算损失，损失由两部分组成

- 检测框损失：采集与检测框交并比大的锚框作为正样本，训练loc网络将正样本中的锚框尽可能回归到检测框
- 置信度损失：采集交并比较大的作为正样本，较小的作为负样本，训练score网络尽可能区分正负两类样本

```
true_loc, labels = create_anchor_target(self.anchors, bboxes)
#取出loc中正样本的部分用smoothF1计算损失，score中正负样本的部分用MSEloss计算损失
pred_loc = pred_loc[labels==1]
true_loc = true_loc[labels==1]
pred_score = pred_score[labels!=-1]
true_score = labels[labels!=-1]

box_loss = F.smooth_l1_loss(pred_loc,true_loc)
pred_loss = F.mse_loss(pred_score, true_score)
loss = box_loss + pred_loss
tot_box_loss += box_loss.item()
tot_pred_loss += pred_loss.item()

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

需要用到create_anchor_target函数，函数输入锚框和正确目标检测框，计算交并比大小并实现采样。

采样流程如下：（anchor为锚框，bbox为真实检测框标注）

- 将每个bbox最接近的anchor设置为正样例
- 计算每个anchor与所有bbox的最大交并比
- 将最大交并比大于给定阈值的anchor作为正样例
- 将最大交并比小于给定阈值的anchor作为负样例
- 由于正样例一般数目较少，在负样例中抽取与正样例等量的样本

并且计算正样本的锚框相对于检测框之间的偏移量作为loc网络学习的内容。

```
def create_anchor_target(anchor,bbox, pos_iou_thresh=0.7, neg_iou_thresh=0.3,
n_sample=128):
    ious = bbox_union_iou(anchor,bbox)
    argmax_ious = ious.argmax(axis=1) #每个anchor最接近的bbox
    max_ious = ious[torch.arange(ious.shape[0]), argmax_ious]
    gt_argmax_ious = ious.argmax(axis=0) #每个bbox最接近的anchor，gt正样例
    gt_max_ious = ious[gt_argmax_ious, torch.arange(ious.shape[1])]
    gt_argmax_ious = torch.where(ious == gt_max_ious)[0]

    #产生每个anchor位移到最接近的bbox之间的loc用于训练
    loc = bbox2loc(anchor, bbox[argmax_ious])

    #产生label用于训练fpn score，使用正负采样的方式
    label = torch.zeros(len(anchor)).to(device) - 1
    label[max_ious < neg_iou_thresh] = 0
    label[gt_argmax_ious] = 1
    label[max_ious >= pos_iou_thresh] = 1

    #负采样
    n_pos = torch.sum(label==1).item()
    n_neg = n_pos
```

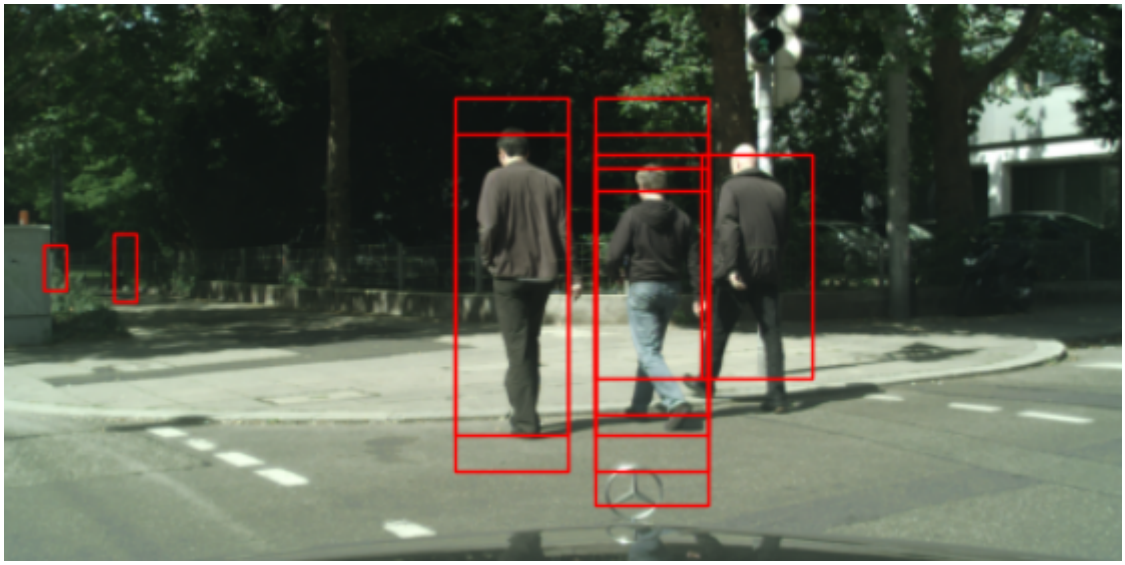
```
    neg_index = torch.where(label == 0)[0]
    if len(neg_index) > n_neg:
        disable_index = np.random.choice(neg_index.cpu(), size=(len(neg_index) -
n_neg), replace=False)
        label[disable_index] = -1

    return loc,label
```

测试时，根据loc和score生成一系列检测框。

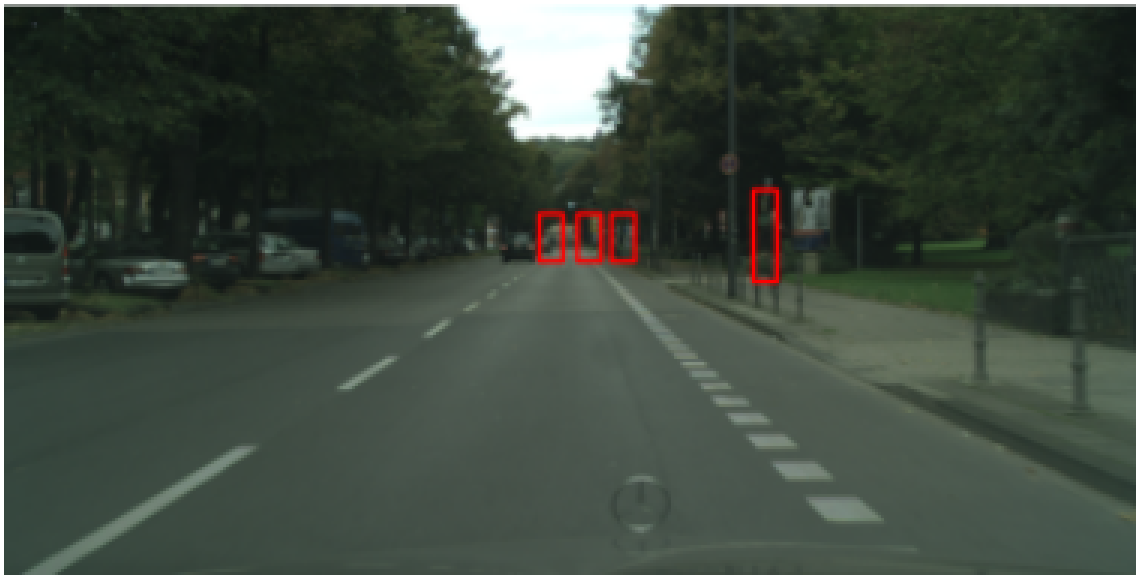
取出预测分数score大于0.5的检测框，然后调用极大值抑制算法（需由xywh格式转变为xyxy格式），选取出最终的预测框。

```
pred_bbox = loc2bbox(self.anchors, loc)
pred_bbox = pred_bbox[pred>0.5]
pred = pred[pred>0.5]
bboxes = bboxes[clses==1]

keep = nms(bbox_wh2xy(pred_bbox),pred,0.2)
pred_bbox = pred_bbox[keep]

show_bboxes_compare_img(img,pred_bbox,bboxes,savepath)
```

极大值抑制使用torchvision中的nms函数



极大值抑制前



极大值抑制后

NOTE：两张图片并非同一图片

训练过程中结果，由于数据集中人很小，检测效果有时候并不是很好

训练结果示例

1.4 structure

整体网络结构

```
FPN(
  (extracter): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
  )
  (conv): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
  )
  (loc): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Conv2d(512, 36, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (score): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
```

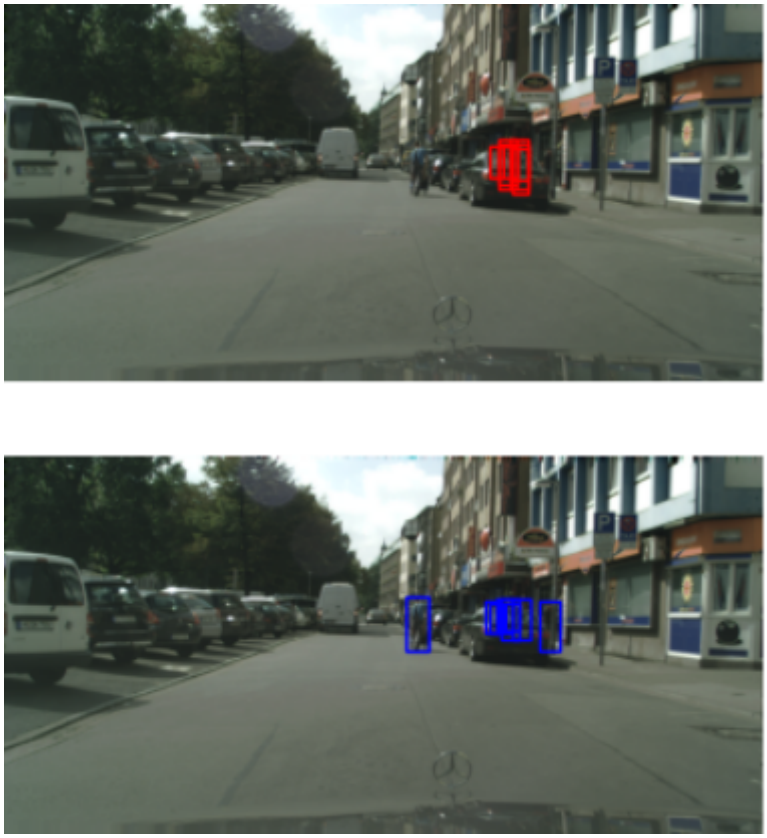
```
    (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Conv2d(512, 9, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
```

2.训练

损失由边框回归损失和置信度判断损失构成。其实等价于同时训练两个神经网络，一个神经网络用于执行边框回归，一个用于判断边框中是否有物体，而两个网络共享着提取特征的部分。

为了同时训练两个网络，先分别进行训练。先将边框回归用准确边框代替，至训练网络进行置信度判断，即二分类边框中是否有物体，结果如下：

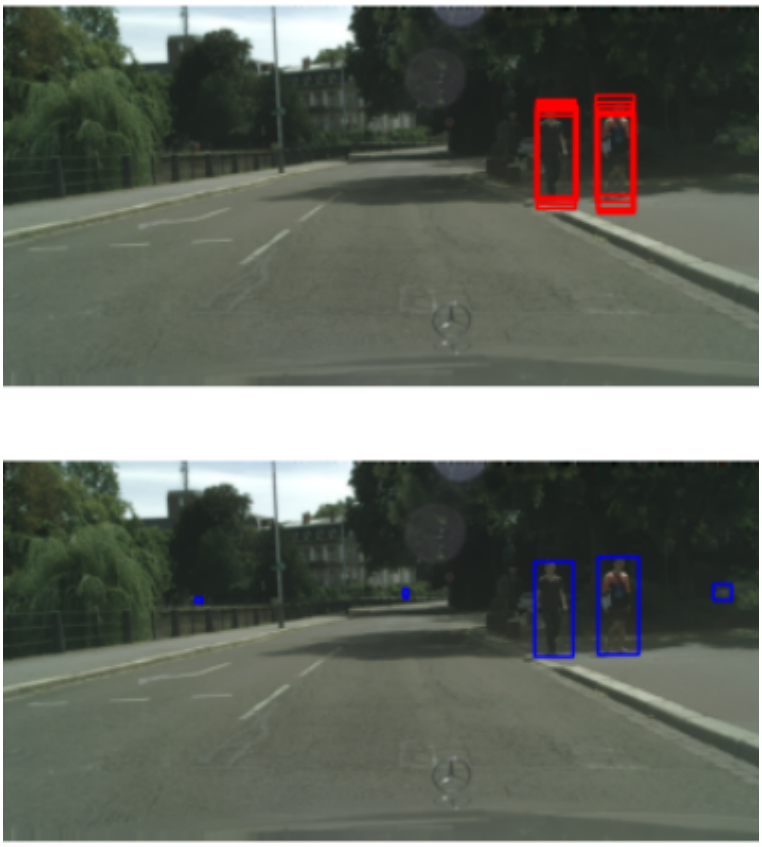
置信度判断结果展示



边框回归结果展示1

边框回归会尝试将多个候选框集中回归到同一部分，特别是对于十分密集的人群。

NOET：该边框回归结果并没有加极大值抑制



参考

[1] https://github.com/chenyuntc/simple-faster-rcnn-pytorch