

华中科技大学  
课程实验报告

课程名称： 计算机视觉结课报告

专业班级 CS2003

学 号 U202015374

姓 名 张隽翊

指导教师 李贤芝

报告日期 2023 年 5 月 4 日

计算机科学与技术学院

## 目 录

<b>1 目标检测任务概述 .....</b>	<b>1</b>
1.1 目标检测任务简介 .....	1
1.2 目标检测历史研究 .....	1
<b>2 两阶段目标检测典型算法.....</b>	<b>3</b>
2.1 R-CNN .....	3
2.2 Fast R-CNN.....	5
2.3 Faster R-CNN .....	7
2.4 Mask R-CNN .....	12
<b>3 Faster R-CNN 源码分析 .....</b>	<b>17</b>
3.1 General Structure .....	17
3.2 GeneralizedRCNN .....	17
3.3 FasterRCNN .....	22
3.4 Training Model.....	28
<b>4 目标检测算法实践 .....</b>	<b>29</b>
4.1 Faster R-CNN 和 Mask R-CNN 对比.....	29
4.2 视频中人体的姿态估计 .....	29
<b>5 专题实验心得 .....</b>	<b>31</b>
<b>A Faster R-CNN 和 Mask R-CNN 效果对比代码.....</b>	<b>33</b>
<b>B 视频人体姿态估计代码.....</b>	<b>39</b>

# 1 目标检测任务概述

## 1.1 目标检测任务简介

目标检测 (Detection) 是计算机视觉的基本任务之一，旨在从一副图像上用矩形框把一些物体框定出来。不同于分类 (Classification) 任务关注整体图片类别，目标检测关注的是特定的物体目标，要求在图片中同时识别出目标物的类别信息和位置信息，是一个 classification + localization 的问题。分类给出的是整张图片的内容描述，而检测给出的则是对图片前景和背景的理解。对于检测人物，我们需要从背景中分离出感兴趣的目标，并确定这一目标的描述，即类别和位置。检测模型的输出形式通常是一个列表，列表的每一项使用一个数组给出检出目标的类别和位置，常用矩形检测框的坐标表示。

## 1.2 目标检测历史研究

### 1.2.1 深度学习时代之前

早期的目标检测流程分为三步：候选框生成、特征向量提取和区域分类。

- 1) 第一阶段：候选框生成。这一阶段的目标是搜索图像中可能包含对象的位置，又叫感兴趣区域 (RoI, region of interest)。直观的思路是用滑动窗口扫描整幅图像。为了捕捉不同尺寸和不同宽高比对象的信息，输入图像被重新分割为不同的尺寸，然后用不同尺寸的窗口滑动经过输入图像；
- 2) 第二阶段：特征向量提取。在图像的每个位置上利用滑动窗口获取固定长度的特征向量，从而捕捉该区域的判别语义信息。该特征向量通常由低级视觉描述子编码而成，包括 SIFT (Scale Invariant Feature Transform)、HOG (Histogram of Gradients)、SURF (Speeded Up Robust Features) 等，它们对缩放、光线变化和旋转具备一定的鲁棒性；
- 3) 第三阶段：区域分类。学习区域分类器，为特定区域分配类别标签。

通常会使用支持向量机 (SVM, support vector machine)，因为它在小规模训练数据上性能优异。此外，Bagging、Cascade Learning 和 Adaboost 等分类技术也会用在区域分类阶段，帮助提高目标检测的准确率。

### 1.2.2 深度学习时代之后

在将深度卷积神经网络成功应用于图像分类后，基于深度学习技术的目标检测也取得了巨大进步，并且基于深度学习的新算法显著优于传统的目标检测算法。

目前，基于深度学习的目标检测框架可以分为两大类：

- 1) 二阶检测器 (Two-stage): 如基于区域的 CNN (R-CNN) 及其变体；
- 2) 一阶检测器 (One-stage): 如 YOLO 及其变体。

二阶检测器首先使用候选框生成器生成稀疏的候选框集，并从每个候选框中提取特征；然后使用区域分类器预测候选框区域的类别。一阶检测器直接对特征图上每个位置的对象进行类别预测，不经过二阶检测中的区域分类步骤。两者的结构如图 1-1 所示。

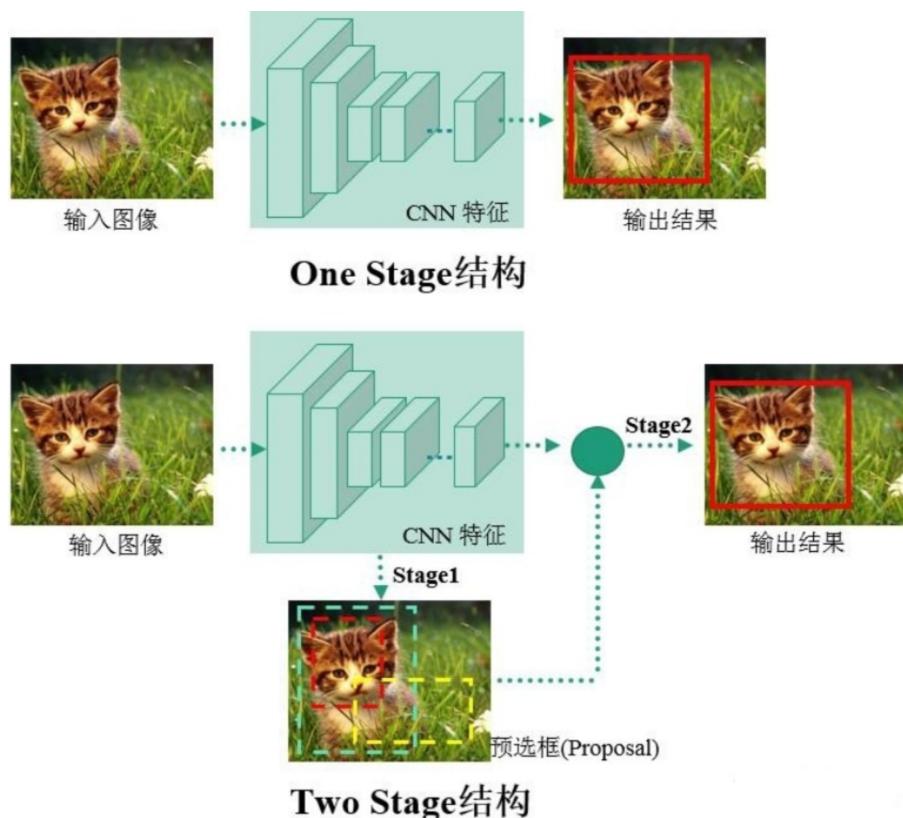


图 1-1 One Stage 和 Two Stage 结构对比

通常而言，二阶检测器的性能更好，在公开基准上取得了当前最优结果，而一阶检测器更省时，在实时目标检测方面具备更强的适用性。

## 2 两阶段目标检测典型算法

两阶段模型也称为基于区域 (Region-based) 的方法，我们选取 R-CNN 系列工作作为这一类型的代表，典型网络包括 R-CNN、Fast R-CNN、Faster R-CNN 以及 Mask R-CNN。下面我将结合原始论文和自己的理解简述每种网络的基本原理，以及后一种网络相比前一种网络有哪些技术上的改进。

### 2.1 R-CNN

R-CNN (Regions with Convolutional Neural Networks) 是一种经典的目标检测算法，也是首个将 CNN 引入目标检测领域的算法。由于诞生时间较早，R-CNN 的步子迈得不算太大，主要是在特征提取阶段使用 CNN，其它阶段使用的还是传统目标检测的方法。

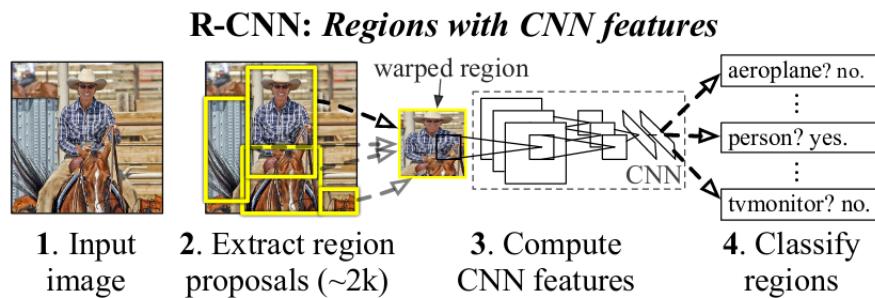


图 2-1 R-CNN 算法流程

原论文中 R-CNN 算法流程图如图 2-1 所示。R-CNN 算法流程可以分为以下四个步骤：

#### 1) 生成候选区域

对于输入的图片，使用 Selective Search 方法选出 2000 个左右的候选区域，然后使用一些合并策略将这些区域合并，得到一个层次化的区域结构。

#### 2) 特征提取

将候选区域缩放到  $227 \times 227$ ，输入到 AlexNet 网络中进行特征提取，得到  $2000 \times 4096$  维的特征向量。这一步调整图像的方法比较暴力，无论图片大小如何都直接缩放到了  $227 \times 227$ 。

#### 3) 判断候选区域的类别

将  $2000 \times 4096$  维的特征向量送入到 21 个 (20 个类别 + 1 个背景) SVM 分类器中，得到每个候选区域属于某个类别的概率。这一步可以看作两个矩阵相乘，即： $W1_{2000 \times 4096} \times W2_{4096 \times 21} = W3_{2000 \times 21}$ 。由于一张图片上的物体数量有限，一定会有大量的候选区域重叠，因此作者采用了非极大值抑制 (NMS) 方法来去除冗余的候选框，步骤如图 2-2 所示。

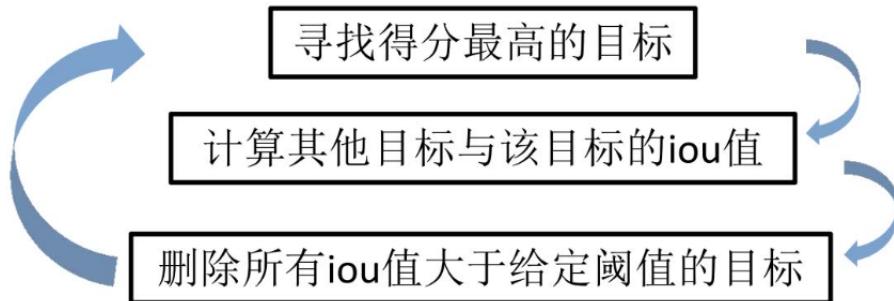


图 2-2 NMS 去除冗余候选框

#### 4) 使用回归器调整候选框位置

对 NMS 处理后剩余的候选框进一步筛选，用回归器对上述类别中剩余的候选框进行回归操作，最终得到每个类别修正后得分最高的 bounding box。

综上，R-CNN 的框架如图 2-3 所示。

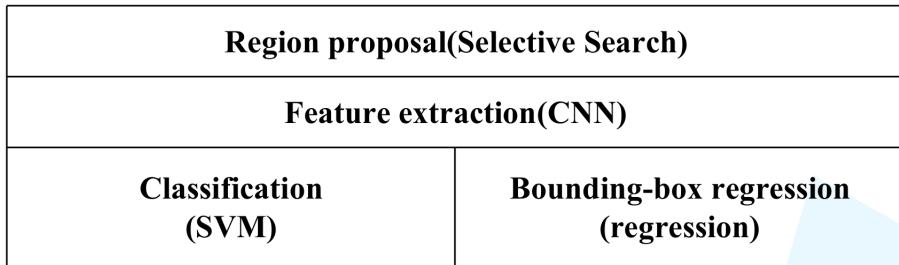


图 2-3 R-CNN 框架

论文中，作者认为 R-CNN 较之前的算法能够取得 30% 的改进有两个主要原因：

- 1) 使用了卷积神经网络来提取候选区域的特征；
- 2) 使用了迁移学习的方法。目标检测领域的数据相比于图像分类任务要少很多，使用在图像分类上训练好的模型经过 fine-tune 后可以很好地用于目标检测。

当然，作为 R-CNN 系列的第一个算法模型，R-CNN 的不足之处在于：

- 1) 速度较慢。因为 R-CNN 需要使用 Selective Search 方法生成 2000 个候选区域分别提取特征，而这些区域有着很多重叠部分，存在大量重复计算；
- 2) 训练不便。需要先预训练 CNN 并进行微调，再训练 SVM 和回归器，中间还需要使用 NMS 去除冗余候选框，步骤繁琐。

## 2.2 Fast R-CNN

Fast R-CNN 是在 R-CNN 和 SPPNet 的基础上改进得到的。在 R-CNN 和 SPPNet 中，需要先通过卷积网络提取特征，然后根据特征训练 SVM 做分类和回归器用于位置矫正。这种多阶段的流程存在以下两个问题：

- 1) 保存中间变量需要占用大量磁盘空间；
- 2) 不能根据分类和矫正结果调整卷积网络权值，一定程度上会限制网络精度。

在 Fast R-CNN 中，作者通过多任务的方式将 R-CNN 和 SPPNet 整合成一个流程，同时带来了分类和检测精度的提升，并通过 Softmax 替代 SVM 的分类任务省去了中间变量的使用。同 SPPNet 一样，Fast R-CNN 将整张图片输入到卷积网络用于提取特征，将 Selective Search 选定的候选区域坐标映射到卷积层，再使用 ROI 池化层将不同尺寸的候选区域特征窗口映射成相同尺寸的特征向量，经过两层全连接后将得到的特征分支成两个输出层，一个 Softmax 用于分类，一个 bbox 回归器用于位置精校。这两个任务的损失共同用于调整网络的参数。和 SPPNet 对比，Fast R-CNN 最大的优点是多任务的引进，在优化训练过程的同时也避免了额外存储空间的使用并在一定程度上提升了精度。

Fast R-CNN 算法流程主要分为三个步骤：

- 1) 使用 Selective Search 方法生成 2K 个图片候选区域；
- 2) 对整张图片进行特征提取得到相应的特征图，并将上一步生成的候选区域映射到特征图中；
- 3) 使用 ROI Pooling 将所有的候选区域特征统一缩放到  $7 \times 7$  大小，然后将这 2K 个特征向量展开并连接到全连接层上，得到两个输出结果，一个是  $K + 1$  类 (类别数量 + 背景类别) 的概率，另一个是每个类的预测边框。

Fast R-CNN 的网络结构如图 2-4 所示。

与 R-CNN 相比，Fast R-CNN 主要有以下几点不同：

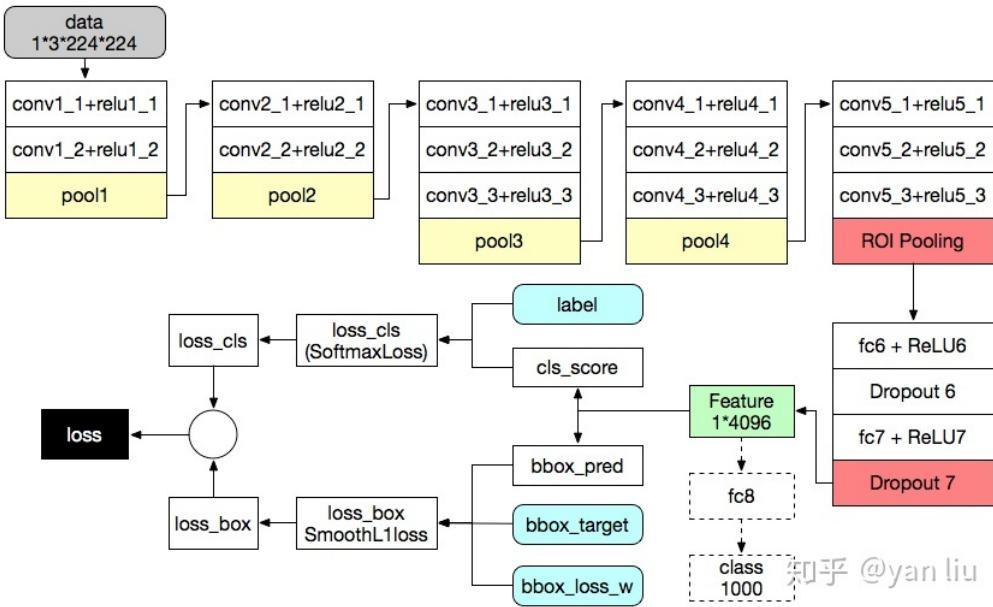


图 2-4 Fast R-CNN 网络结构

- 1) 不再对每一个候选区域单独提取特征，而是在提取整个图像的特征后，将每一个候选区域映射到特征图上。这一映射是基于图像经过多次卷积与池化后相对位置不变的特性实现的。由于 Fast R-CNN 的 backbone 是 VGG 网络，而在 VGG 中卷积操作不改变图像尺寸，每进行一次 max pooling 都会将特征图的尺寸缩小为原来的  $\frac{1}{2}$ ，因此经过四次 max pooling 后图像大小变为原始图像的  $\frac{1}{2^4} = \frac{1}{16}$ ，相应的候选区域的坐标也应该按比例缩放。也就是说，映射到特征图上的坐标是原始坐标的  $\frac{1}{16}$ 。
- 2) 在 R-CNN 中，为了统一输入图像的尺寸采用了比较暴力的方法，而在 Fast R-CNN 中使用的是 RoI Pooling，这一方法参考了 SPPNet 的空间金字塔池化。简单来说，RoI Pooling 就是将每一个候选区域的特征图分割为  $7 \times 7$  个区域，然后对每个小区域进行 max pooling 操作。
- 3) 使用了多任务的损失函数来简化 R-CNN 中的多阶段训练，如图 2-5 所示。

## Multi-task loss

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda[u \geq 1]L_{loc}(t^u, v)$$

图 2-5 Fast R-CNN 网络结构

其中,  $p$  是分类器预测的 Softmax 概率分布  $p = (p_0, \dots, p_k)$ ,  $u$  对应目标真实类别标签,  $t^u$  对应边界框回归器预测的对应类别  $u$  的回归参数  $(t_x^u, t_y^u, t_w^u, t_h^u)$ ,  $v$  对应真实目标的边界框回归参数  $(v_x, v_y, v_w, v_h)$ 。

Fast R-CNN 框架如图 2-6 所示。

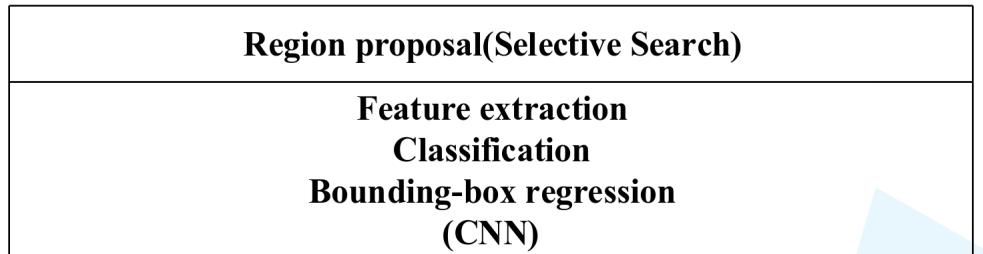


图 2-6 Fast R-CNN 框架

与 R-CNN 和 SPPNet 相比, Fast R-CNN 使用多任务训练, 取得了比分阶段训练更好的效果, 且训练时间更少, 有一定的进步。但 Fast R-CNN 中候选区域的选取还是通过 Selective Search 方法, 网络在这一阶段消耗了大量的时间。

### 2.3 Faster R-CNN

Fast R-CNN 虽然实现了端到端的训练, 且通过共享卷积的方式大幅提升了 R-CNN 的计算速度, 但仍然难以做到实时, 其中性能瓶颈在于候选区域的计算。在之前的物体检测系统中, Selective Search 是最常用的候选区域提取方法, 另外一个更快速的版本是 EdgeBoxes, 但 EdgeBoxes 为了速度牺牲了提取效果。Selective Search 之所以较慢是因为它使用的是 CPU, 而检测网络使用的是 GPU。

针对这一痛点, Faster R-CNN 在 Fast R-CNN 基础上进一步改进, 提出了一种名为 Region Proposal Network (RPN) 的子网络, 用于生成候选区域。在此基础上, Faster R-CNN 使用 ROI Pooling 和全连接层进行目标分类和位置回归。Faster R-CNN 相对于 Fast R-CNN, 减少了候选区域的生成时间, 同时提高了目标检测的准确率。

在结构上, Faster R-CNN 已经将特征提取 (feature extraction)、proposal 抽取、bounding box regression、classification 都整合在了一个网络中, 使得综合性能有较大提高, 在检测速度方面尤为明显。

由原论文的基本结构 (图 2-7), Faster R-CNN 可以分为四个主要部分:

- 1) Conv layers。作为一种 CNN 目标检测方法, Faster R-CNN 首先使用一组基础

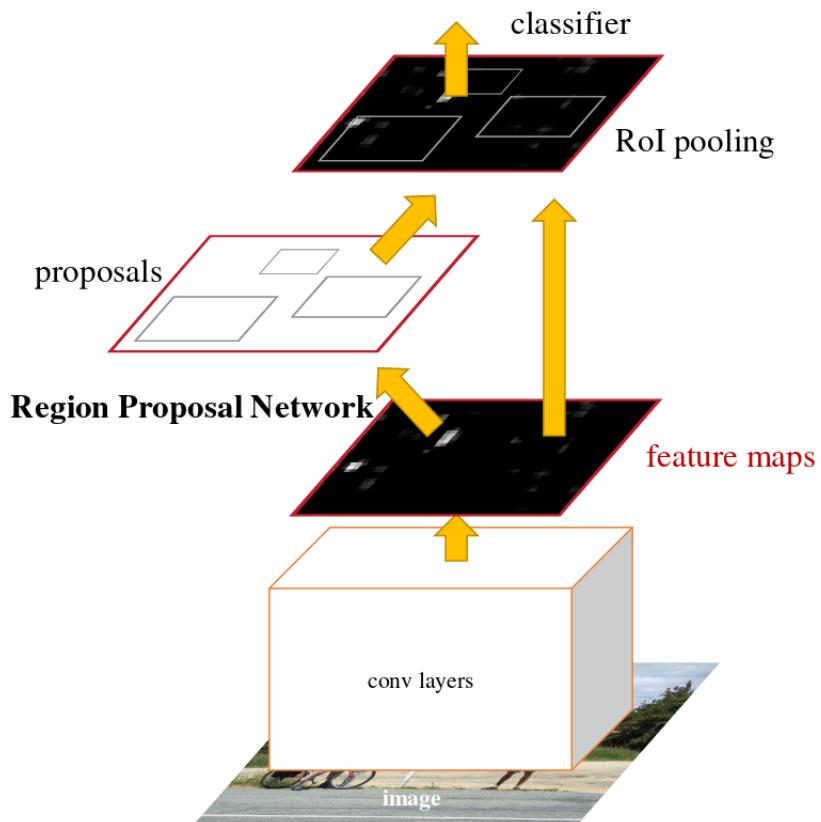


图 2-7 Faster R-CNN 基本结构

的 conv + relu + pooling 层提取图像的特征图，并共享特征图用于后续 RPN 层和全连接层；

- 2) Region Proposal Networks。RPN 网络用于生成 region proposals，通过 softmax 判断 anchors (锚框) 属于 positive 还是 negative，再利用 bounding box regression 修正 anchors 获得精确的 proposals；
- 3) ROI Pooling。这一层收集输入的特征图和 proposals，综合信息提取 proposal feature maps，送入后续全连接层判定目标类别；
- 4) Classification。利用 proposal feature maps 计算 proposal 的类别，同时再次 bounding box regression 获得检测框最终的精确位置。

图 2-8 展示了 Faster R-CNN 网络的结构。对于一幅任意大小  $P \times Q$  的图片，Faster R-CNN 首先将图片缩放至固定大小  $M \times N$ ，然后将  $M \times N$  的图像送入网络；经过 Conv layers 的一系列层后得到 Feature Map，RPN 网络首先经过  $3 \times 3$  卷积，再分别生成 positive anchors 和对应 bounding box regression 偏移量，然后计算出 proposals；RoI Pooling 层利用 proposals 从 Feature Map 中提取 proposal

feature 送入后续全连接和 Softmax 网络作 classification，即分类 proposal 属于哪类类型。

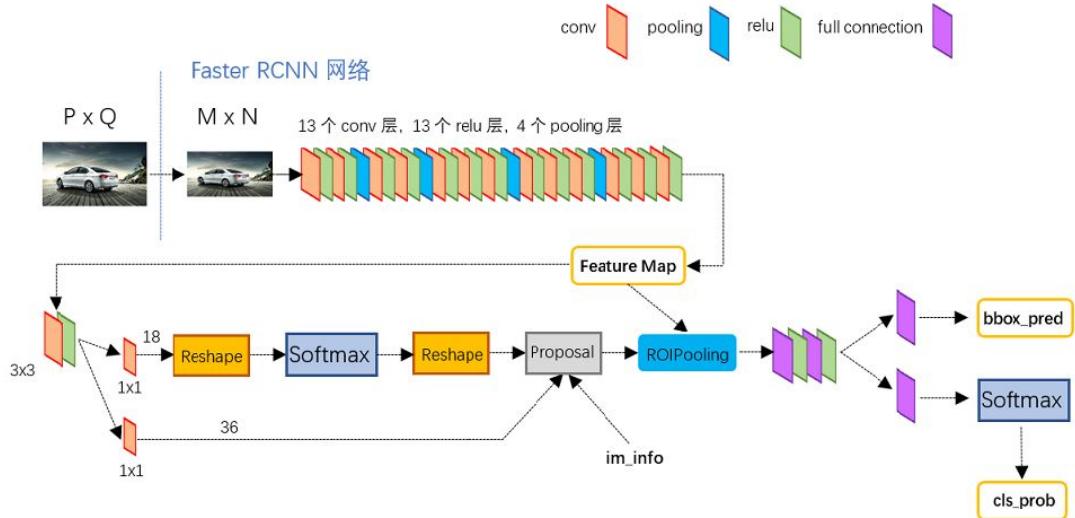


图 2-8 Faster R-CNN 基本结构

### 2.3.1 Conv layers

Conv layers 包括 conv, pooling, relu 三种网络层。在 Conv layers 中，

- 1) conv: kernel\_size=3, padding=1, stride=1
- 2) pooling: kernel\_size=2, padding=0, stride=2

在经过图 2-8 上部所示的  $13 + 13 + 4$  层网络后，图片由  $M \times N$  变为  $\frac{M}{16} \times \frac{N}{16}$ 。

### 2.3.2 Region Proposal Networks(RPN)

Faster R-CNN 抛弃了传统的滑动窗口和 Selective Search 方法，直接使用 RPN 生成检测框，这也是 Faster R-CNN 的巨大优势，能极大提升检测框的生成速度。

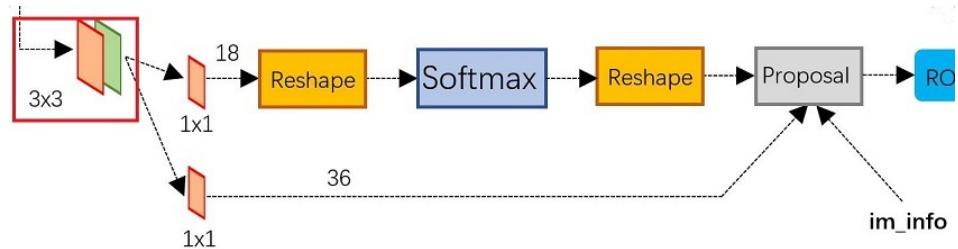


图 2-9 Faster R-CNN 基本结构

图 2-9 展示了 RPN 网络的具体结构。RPN 网络分为两条线，上面的线路通过 Softmax 分类 anchors 获得 positive 和 negative 分类，下面的线路计算对于 anchors 的 bounding box regression 偏移量，以获得精确的 proposal。最后的 Proposal 层负责综合 positive anchors 和对应 bounding box regression 偏移量获取 proposals，同时去除太小和超出边界的 proposals。

RPN 网络中的 anchors 就是一组矩形，9 个矩形共有 3 种形状，长宽比  $\frac{width}{height} \in \{1.0, 0.5, 2.0\}$ 。通过 anchors 引入了检测中常用的多尺度方法。如图 2-10 所示，遍历 Conv layers 计算获得的 feature maps，为每一个点都配备这 9 中 anchors 作为初始的检测框。

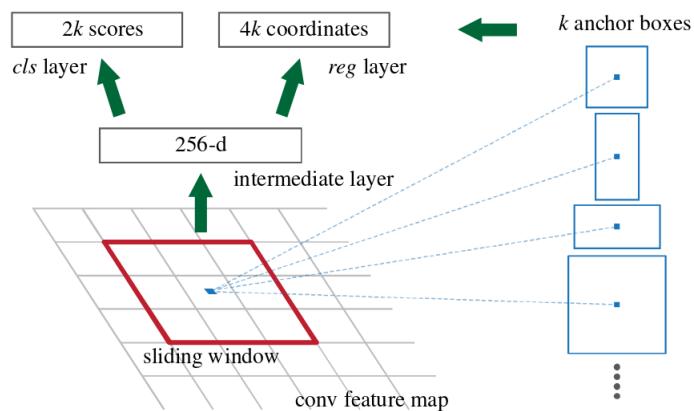


图 2-10 Anchors 示意图

对于  $800 \times 600$  的原图，一共有： $ceil(800/16) \times ceil(600/16) \times 9 = 50 \times 38 \times 17100$  个 anchor

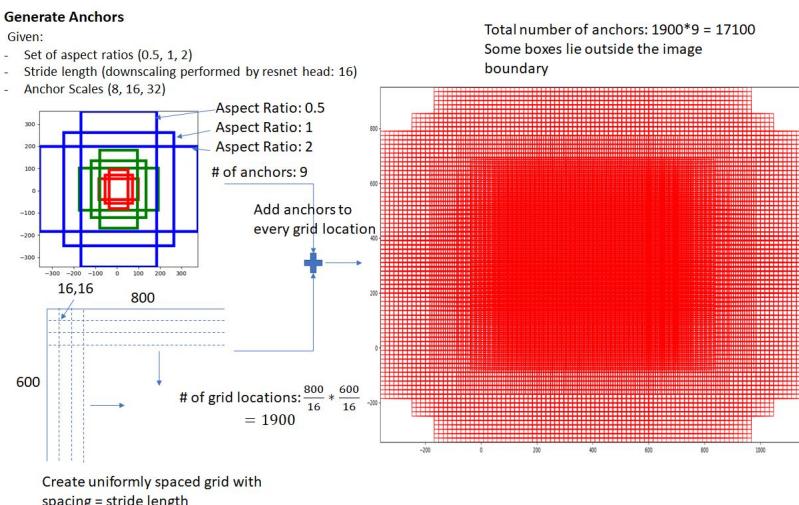


图 2-11 Anchors 生成

bounding box regression 用于修正 anchor 的位置，采用回归网络输出每个 Anchor 的平移量和变换尺度  $(t_x, t_y, t_w, t_h)$ 。图 2-9 的下面一条线路完成 positive / negative 分类和 bounding box regression 坐标回归。

Proposal layer 的处理流程为：生成 anchors，利用 bounding box regression 的  $[d_x(A), d_y(A), d_w(A), d_h(A)]$  对所有的 anchors 做 bounding box regression，按照输入的 positive softmax scores 从大到小排序，提取前 pre\_nms\_topN 个 anchors 即提取修正位置后的 anchors，限定超出边界的部分，去除尺寸很小的 anchors，对剩余的 anchors 进行 NMS，最后送入 RoI Pooling 层。

### 2.3.3 ROI Pooling

ROI Pooling 层收集 proposals，并计算出 feature maps，送入后续网络。ROI Pooling 层有两个输入：原始的 feature maps 和 RPN 输出的 proposal boxes。ROI Pooling 用于解决大小不同的 proposals 输出结果大小不一致的问题。ROI Pooling layer forward 过程如下：

- 1) 使用 spatial\_scale 参数将大小为  $M \times N$  的 proposal 映射回大小为  $\frac{M}{16} \times \frac{N}{16}$  的 feature map 尺度；
- 2) 再将每个 proposal 对应的 feature map 区域水平分为  $pool\_w * pool\_h$  的网格；
- 3) 对每个网格做 max pooling。

经过 ROI Pooling 的处理后，大小不同的 proposals 输出结果都是  $pool\_w * pool\_h$  固定大小，实现了固定长度的输出。

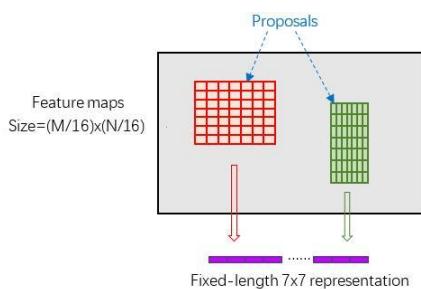


图 2-12 proposal 输出

### 2.3.4 Classification

Classification 部分利用已经获得的 proposal feature maps，通过全连接层和 softmax 计算每个 proposal 具体属于哪个类别，输出 cls\_prob 概率向量；同时再

次利用 bounding box regression 获得每个 proposal 的位置偏移量 bbox\_pred，用于回归预测更加精确的目标检测框。

### 2.3.5 Training Faster R-CNN

Faster R-CNN 的训练是在已经训练好的模型基础上继续进行训练。实际训练大致分为 6 个步骤：

- 1) 在已经训练好的模型上训练 RPN 网络，对应 stage1\_rpn\_train.pt；
- 2) 利用步骤 1 中训练好的 RPN 网络收集 proposals，对应 rpn\_test.pt；
- 3) 第一次训练 Faster R-CNN 网络，对应 stage1\_fast\_rcnn\_train.pt；
- 4) 第二次训练 RPN 网络，对应 stage2\_rpn\_train.pt；
- 5) 再次利用步骤 4 中训练好的 RPN 网络收集 proposals，对应 rpn\_test.pt；
- 6) 第二次训练 Faster R-CNN 网络，对应 stage2\_fast\_rcnn\_train.pt。

Faster R-CNN 还有一种端到端的训练方式，可以一次完成训练过程。

## 2.4 Mask R-CNN

Mask R-CNN 算法在 Faster R-CNN 的基础上进一步改进，在进行目标检测的同时还可以进行实例分割。Mask-RCNN 的算法实现思路非常直接，针对目标检测算法 Faster-RCNN 加入语义分割算法 FCN，使得完成目标检测的同时也得到语义分割的结果，算法对 Faster-RCNN 的一些细节做了调整，最终的组成部分是 RPN + RoIAlign + Fast R-CNN + FCN。

Mask R-CNN 的结构示意图如图 2-13 所示。

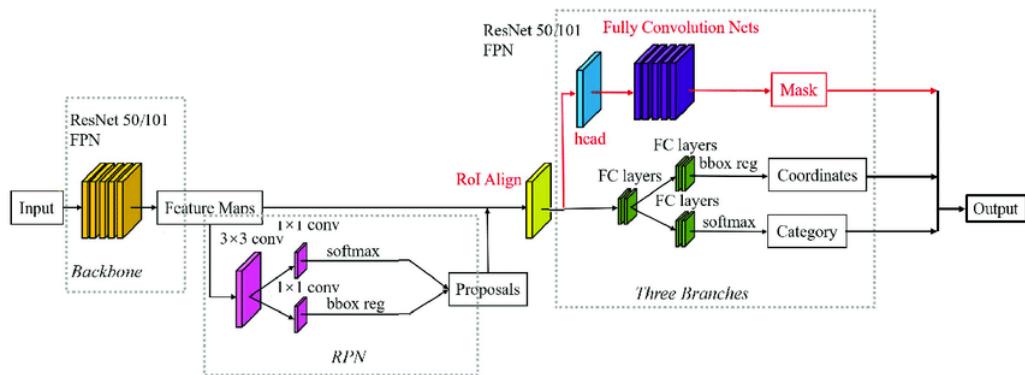


图 2-13 Mask R-CNN 结构示意图

### 2.4.1 Backbone

和 Faster R-CNN 一样，Mask R-CNN 的第一部分是一个标准的 CNN 网络（一般是 ResNet50 或者 ResNet101），用于提取图像中的信息。Mask R-CNN 还使用了 FPN (Feature Pyramid Networks，特征金字塔网络) 来提升网络的性能。

ResNet 残差网络的结构如图 2-14 所示。Mask R-CNN 中所使用的 ResNet 50/101 就是常用的两种残差网络结构。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
conv2_x	56×56			3×3 max pool, stride 2		
conv3_x	28×28	$\left[ \begin{array}{l} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 2$	$\left[ \begin{array}{l} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 3$	$\left[ \begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[ \begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[ \begin{array}{l} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$
conv4_x	14×14	$\left[ \begin{array}{l} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 2$	$\left[ \begin{array}{l} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 4$	$\left[ \begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[ \begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[ \begin{array}{l} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 8$
conv5_x	7×7	$\left[ \begin{array}{l} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 2$	$\left[ \begin{array}{l} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 6$	$\left[ \begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 6$	$\left[ \begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 23$	$\left[ \begin{array}{l} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 36$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

图 2-14 ResNet 残差网络架构

### 2.4.2 Feature Pyramid Networks(FPN)

FPN (Feature Pyramid Networks) 意为特征金字塔网络，基本结构如图 2-15 所示。

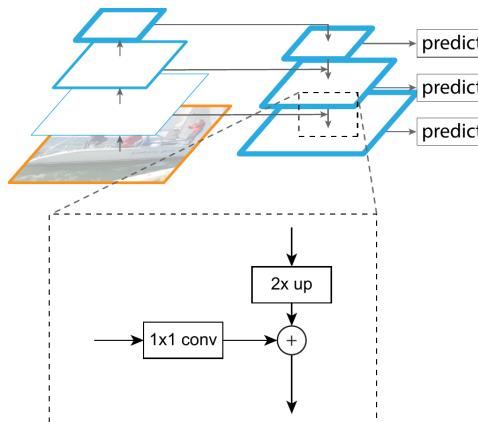


图 2-15 FPN 基本结构示意图

FPN 的具体思想是将尺寸小但特征信息多的层做反卷积与尺寸大但特征提

取得少的前几层结果做融合，将整合后的数据做后续预测。FPN 会输出多个结果，这些结果的大小依次差距 2 倍，将这些结果分别输入到下一步的 RPN 网络结构中，同一张图片的各个输出结果可以共享后续的检测框参数。

### 2.4.3 Region Proposal Network

RPN (Region Proposal Network) 的主要功能是产生物品检测框，核心操作是根据特征图按照一定规律生成一系列锚框 (anchor)。这一步与 Faster R-CNN 基本相同。

### 2.4.4 ROI Process

ROI (Region of Interest) 即感兴趣的区域。从 Mask-RCNN 的架构可以看出，ROI 有两部分输入，第一部分是由 backbone 生成的特征图像 (feature map)，另一部分是由 RPN 生成的检测框 (proposal)，ROI 这一部分的主要任务就是把生成的检测框对应到特征图像上。

RoI Pooling 是 Faster R-CNN 提取特征的模块。整个 RoI Pooling 可以分成两步，第一步将候选框缩放到特征层上，对于非整数的情况四舍五入取整。第二步操作类似 Max Pooling，将刚得到的区域缩放到一个预定义的大小，无法整除时各部分大小再次做取整操作，每个区域内选取最大值作为输出。

RoI Align 是 Mask R-CNN 对 RoI Pooling 的改进，主要是处理 RoI Pooling 中取整操作带来的误差。RoI Pooling 主要有两部分取整带来的误差，一个是候选框对应到特征层上时的取整，另一个是区域缩放不能整除时对边界的取整。Mask R-CNN 原论文中 RoI Align 的示意图如图 2-16 所示，利用双线性差值计算每个子区域的 4 个点，将最大值作为这一部分的输出。

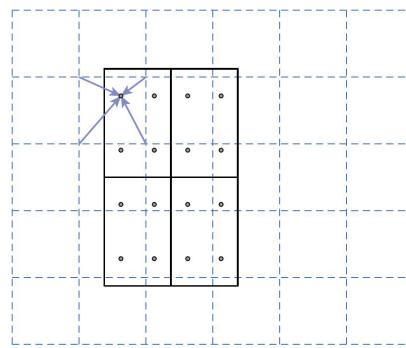


图 2-16 FPN 基本结构示意图

### 2.4.5 Loss Calculation

经过 RoI 处理后的特征矩阵一般有确定大小（原论文是  $7 \times 7$ ），将这样的矩阵展平经过一系列全链接层得到预测结果，对于 Mask R-CNN 网络有三个预测输出结构，分别输出预测预测的种类类别、预测物品的矩形展示框以及待预测物品的准确边界（像素级分类）。

输出种类类别的预测器非常简单，与 CNN 模型类似，用一系列全连接层激活函数层最后加一层 softmax 层预测各个种类的概率，损失函数使用的是多类别的交叉熵损失函数。

第二部分损失是用来物品边界框回归器用来计算边界框的损失。对于每种类别的候选边界框有 4 个回归参数： $d_x, d_y, d_w, d_h$ ，这些参数对于候选框位置的影响可以使用下列公式推导：

$$\begin{aligned}\hat{G}_x &= P_w d_x + P_x \\ \hat{G}_y &= P_h d_y + P_y \\ \hat{G}_w &= P_w e^{d_w} \\ \hat{G}_h &= P_h e^{d_h}\end{aligned}\tag{2.1}$$

其中  $P_x, P_y, P_w, P_h$  分别是候选框的中心  $x, y$  坐标以及候选框的宽高，而  $\hat{G}_x, \hat{G}_y, \hat{G}_w, \hat{G}_h$  分别为最终预测的边界框中心  $x, y$  坐标以及宽高。这 4 个回归参数  $(d_x, d_y, d_w, d_h)$  就相当于将前一步生成的候选框平移缩放到离真实候选框最近损失最小的位置。对于这 4 个回归参数，我们将边界框回归器预测的对应类别  $u$ （这里的类别不包括背景）的回归参数记为  $t^u$ ，具体为  $(t_x^u, t_y^u, t_w^u, t_h^u)$ ，真实目标的边界框回归参数记为  $v$ ，具体为  $(v_x, v_y, v_w, v_h)$ ，则边界框回归的损失函数可以表示为：

$$L_{loc}(t^u, v) = \sum_{i \in \{x, y, w, h\}} smooth_{L_\alpha}(t_i^u - v_i)\tag{2.2}$$

第三部分损失是物品准确边界的计算损失。在这一部分中，Mask R-CNN 选取了 FCN 算法作为 Mask 的预测方法。在 Mask 分支计算损失时，模型有两种不同的结构，如图 2-17 所示。

可以看到第二种方法针对 Mask 使用了不同的 RoI，这样 Mask 使用的 RoI

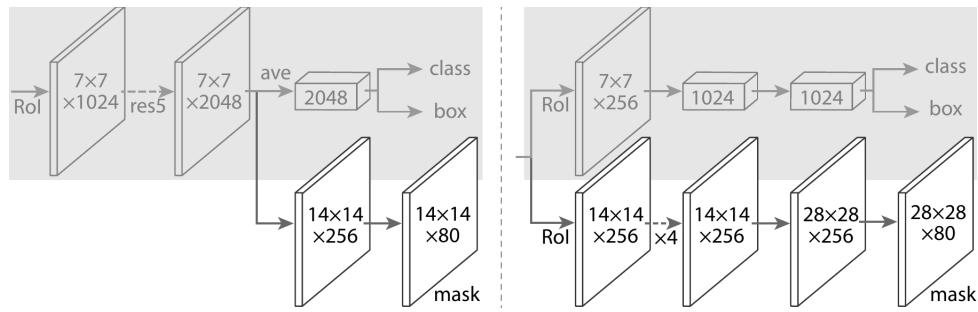


图 2-17 FPN 基本结构示意图

输出大小更大，预测可以做的更精细。另外，Mask 分支在训练和预测时也有一些小差异，训练网络时的目标是由 RPN 提供的 (Proposal)，但在预测时的目标是由其他两个分支 (Faster R-CNN) 提供的，这样做辅助提高了训练模型的能力以及预测的准确度 (RPN 提供的类别都是准确的正样本但是完整边界不准确，而预测时 Faster-RCNN 提供的边界是准确的)。

对于标准的 Mask R-CNN，总损失有 5 部分：类别误差、回归框误差、Mask 预测误差、RPN 类别误差损失、RPN 边界框误差损失，将这些误差损失求和就是整体误差，用来训练评估模型。

#### 2.4.6 Sum Up

Mask R-CNN 是一个很多算法的合成体：

- 1) 使用残差网络作为卷积结构；
- 2) 使用 FPN 作为骨干架构；
- 3) 使用 Faster R-CNN 的物体检测流程：RPN + Fast R-CNN；
- 4) 增加 FCN 用于语义分割。

Mask R-CNN 的特点有：

- 1) 将 FCN 和 Faster R-CNN 合并，通过构建一个三任务的损失函数来优化模型；
- 2) 使用 RoI Align 优化了 RoI Pooling，解决了 Faster R-CNN 在语义分割中的区域不匹配问题。

### 3 Faster R-CNN 源码分析

PyTorch 已经在 torchvision 模块集成了 Faster R-CNN 和 Mask R-CNN 代码。这里简单分析一下 Faster R-CNN 代码部分。torchvision 中对应源码文档在 [torchvision faster\\_rcnn](#)。

#### 3.1 General Structure

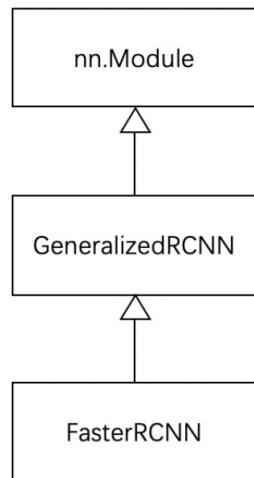


图 3-1 Python Class 继承关系

#### 3.2 GeneralizedRCNN

GeneralizedRCNN 是 torchvision 中目标检测的基类，集成了 torch.nn.Module，后续的 FasterRCNN、MaskRCNN 都继承自 GeneralizedRCNN。GeneralizedRCNN 的代码如下：

```

class GeneralizedRCNN(nn.Module):
    """
    Main class for Generalized R-CNN.

    Args:
        backbone (nn.Module):
        rpn (nn.Module):
        roi_heads (nn.Module): takes the features + the proposals from the
    
```

```
RPN and computes
detections / masks from it.

transform (nn.Module): performs the data transformation from the
inputs to feed into
the model

"""

def __init__(self, backbone: nn.Module, rpn: nn.Module, roi_heads:
    nn.Module, transform: nn.Module) -> None:
    super().__init__()
    _log_api_usage_once(self)
    self.transform = transform
    self.backbone = backbone
    self.rpn = rpn
    self.roi_heads = roi_heads
    # used only on torchscript mode
    self._has_warned = False

@torch.jit.unused
def eager_outputs(self, losses, detections):
    # type: (Dict[str, Tensor], List[Dict[str, Tensor]]) ->
        Union[Dict[str, Tensor], List[Dict[str, Tensor]]]
    if self.training:
        return losses

    return detections

def forward(self, images, targets=None):
    # type: (List[Tensor], Optional[List[Dict[str, Tensor]]]) ->
        Tuple[Dict[str, Tensor], List[Dict[str, Tensor]]]
"""

Args:
    images (list[Tensor]): images to be processed
    targets (list[Dict[str, Tensor]]): ground-truth boxes present in
        the image (optional)
```

Returns:

```
    result (list[BoxList] or dict[Tensor]): the output from the
        model.

        During training, it returns a dict[Tensor] which contains
        the losses.

        During testing, it returns list[BoxList] contains additional
        fields
        like `scores`, `labels` and `mask` (for Mask R-CNN models).

    """
    if self.training:
        if targets is None:
            torch._assert(False, "targets should not be none when in
                training mode")
        else:
            for target in targets:
                boxes = target["boxes"]
                if isinstance(boxes, torch.Tensor):
                    torch._assert(
                        len(boxes.shape) == 2 and boxes.shape[-1] == 4,
                        f"Expected target boxes to be a tensor of shape
                            [N, 4], got {boxes.shape}.",
                    )
                else:
                    torch._assert(False, f"Expected target boxes to be of
                        type Tensor, got {type(boxes)}.")

    original_image_sizes: List[Tuple[int, int]] = []
    for img in images:
        val = img.shape[-2:]
        torch._assert(
            len(val) == 2,
            f"expecting the last two dimensions of the Tensor to be H
                and W instead got {img.shape[-2:]}",
```

```
)  
original_image_sizes.append((val[0], val[1]))  
  
images, targets = self.transform(images, targets)  
  
# Check for degenerate boxes  
# TODO: Move this to a function  
if targets is not None:  
    for target_idx, target in enumerate(targets):  
        boxes = target["boxes"]  
        degenerate_boxes = boxes[:, 2:] <= boxes[:, :2]  
        if degenerate_boxes.any():  
            # print the first degenerate box  
            bb_idx = torch.where(degenerate_boxes.any(dim=1))[0][0]  
            degen_bb: List[float] = boxes[bb_idx].tolist()  
            torch._assert(  
                False,  
                "All bounding boxes should have positive height and  
                width."  
                f" Found invalid box {degen_bb} for target at index  
                {target_idx}.",  
            )  
  
features = self.backbone(images.tensors)  
if isinstance(features, torch.Tensor):  
    features = OrderedDict([("0", features)])  
proposals, proposal_losses = self.rpn(images, features, targets)  
detections, detector_losses = self.roi_heads(features, proposals,  
                                              images.image_sizes, targets)  
detections = self.transform.postprocess(detections,  
                                         images.image_sizes, original_image_sizes) # type:  
                                         ignore[operator]  
  
losses = {}  
losses.update(detector_losses)
```

```
losses.update(proposal_losses)

if torch.jit.is_scripting():
    if not self._has_warned:
        warnings.warn("RCNN always returns a (Losses, Detections)
                      tuple in scripting")
    self._has_warned = True
    return losses, detections
else:
    return self.eager_outputs(losses, detections)
```

关键接口包括 transform、backbone、rpn 和 roi\_heads。

### 3.2.1 transform

transform 对应的代码如下所示。

```
for img in images:
    val = img.shape[-2:]
    torch._assert(
        len(val) == 2,
        f"expecting the last two dimensions of the Tensor to be H and W
        instead got {img.shape[-2:]}",
    )
    original_image_sizes.append((val[0], val[1]))

images, targets = self.transform(images, targets)
```

transform 主要用于将输入进行标准化和将图像缩放到固定大小。由于把缩放后的图像输入网络，那么网络输出的检测框也是在缩放后的图像上的。但是实际中我们需要的是在原始图像的检测框，为了对应起来，所以需要记录变换前 original\_image\_sizes。尽管 Faster R-CNN 理论上可以支持任意大小的图片，但太大的图片可能无法在有限的内存下加载，考虑到工程因素缩放是一个比较稳妥的选择。

### 3.2.2 backbone + rpn + roi\_heads

完成图像缩放后，正式进入网络进行处理。

- 将 transform 后的图像输入到 backbone 模块提取特征图。backbone 一般为 VGG、ResNet、MobileNet 等网络；
- 

```
features = self.backbone(images.tensors)
```

---

- 经过 rpn 模块生成 proposals 和 proposals\_losses；
- 

```
proposals, proposal_losses = self.rpn(images, features, targets)
```

---

- 进入 roi\_heads 模块，即 roi\_pooling + classification；
- 

```
detections, detector_losses = self.roi_heads(features, proposals,
                                                images.image_sizes, targets)
```

---

- 最后经 postprocess 模块进行 NMS，同时将 box 通过 original\_image\_size 映射回原图。
- 

```
detections = self.transform.postprocess(detections,
                                         images.image_sizes, original_image_sizes) # type:
                                         ignore[operator]
```

---

## 3.3 FasterRCNN

FasterRCNN 继承自基类 GeneralizedRCNN，实现了 GeneralizedRCNN 中的 transform、backbone、rpn、roi\_heads 接口。

---

```
super(FasterRCNN, self).__init__(backbone, rpn, roi_heads, transform)
```

---

### 3.3.1 transform

---

```
if image_mean is None:
```

---

```
image_mean = [0.485, 0.456, 0.406]
if image_std is None:
    image_std = [0.229, 0.224, 0.225]
transform = GeneralizedRCNNTransform(min_size, max_size, image_mean,
    image_std, **kwargs)
```

transform 接口使用 GeneralizedRCNNTransform 实现，包含与缩放相关的系数 min\_size、max\_size 以及与归一化相关的系数 image\_mean、image\_std。

### 3.3.2 backbone

---

```
def fasterrcnn_resnet50_fpn(pretrained=False, progress=True,
    num_classes=91, pretrained_backbone=True, **kwargs):
    if pretrained:
        # no need to download the backbone if pretrained is set
        pretrained_backbone = False
    backbone = resnet_fpn_backbone('resnet50', pretrained_backbone)
    model = FasterRCNN(backbone, num_classes, **kwargs)
    if pretrained:
        state_dict =
            load_state_dict_from_url(model_urls['fasterrcnn_resnet50_fpn_coco'],
            progress=progress)
        model.load_state_dict(state_dict)
    return model
```

backbone 部分使用 ResNet50 + FPN 结构。

### 3.3.3 rpn

---

首先是 rpn\_anchor\_generator 方法：

```
if rpn_anchor_generator is None:
    anchor_sizes = ((32,), (64,), (128,), (256,), (512,))
    aspect_ratios = ((0.5, 1.0, 2.0,),) * len(anchor_sizes)
    rpn_anchor_generator = AnchorGenerator(
        anchor_sizes, aspect_ratios)
```

)

---

由于加入了 FPN，需要将多个 feature\_map 逐个输入到 rpn 网络。

AnchorGenerator 的具体实现如下。

---

```
class AnchorGenerator(nn.Module):
    ...
    def generate_anchors(
        self,
        scales: List[int],
        aspect_ratios: List[float],
        dtype: torch.dtype = torch.float32,
        device: torch.device = torch.device("cpu"),
    ) -> Tensor:
        scales = torch.as_tensor(scales, dtype=dtype, device=device)
        aspect_ratios = torch.as_tensor(aspect_ratios, dtype=dtype,
                                       device=device)
        h_ratios = torch.sqrt(aspect_ratios)
        w_ratios = 1 / h_ratios

        ws = (w_ratios[:, None] * scales[None, :]).view(-1)
        hs = (h_ratios[:, None] * scales[None, :]).view(-1)

        base_anchors = torch.stack([-ws, -hs, ws, hs], dim=1) / 2
        return base_anchors.round()

    def set_cell_anchors(self, dtype: torch.dtype, device: torch.device):
        self.cell_anchors = [cell_anchor.to(dtype=dtype, device=device) for
                            cell_anchor in self.cell_anchors]
```

---

接下来关注 AnchorGenerator.grid\_anchors 函数的实现。

---

```
def grid_anchors(self, grid_sizes: List[List[int]], strides:
    List[List[Tensor]]) -> List[Tensor]:
    anchors = []
    cell_anchors = self.cell_anchors
```

```
torch._assert(cell_anchors is not None, "cell_anchors should not be
None")
torch._assert(
    len(grid_sizes) == len(strides) == len(cell_anchors),
    "Anchors should be Tuple[Tuple[int]] because each feature "
    "map could potentially have different sizes and aspect ratios. "
    "There needs to be a match between the number of "
    "feature maps passed and the number of sizes / aspect ratios
    specified.",
)

for size, stride, base_anchors in zip(grid_sizes, strides,
                                       cell_anchors):
    grid_height, grid_width = size
    stride_height, stride_width = stride
    device = base_anchors.device

    # For output anchor, compute [x_center, y_center, x_center,
    # y_center]
    shifts_x = torch.arange(0, grid_width, dtype=torch.int32,
                           device=device) * stride_width
    shifts_y = torch.arange(0, grid_height, dtype=torch.int32,
                           device=device) * stride_height
    shift_y, shift_x = torch.meshgrid(shifts_y, shifts_x,
                                       indexing="ij")
    shift_x = shift_x.reshape(-1)
    shift_y = shift_y.reshape(-1)
    shifts = torch.stack((shift_x, shift_y, shift_x, shift_y), dim=1)

    # For every (base anchor, output anchor) pair,
    # offset each zero-centered base anchor by the center of the
    # output anchor.
    anchors.append((shifts.view(-1, 1, 4) + base_anchors.view(1, -1,
                                                               4)).reshape(-1, 4))
```

```

    return anchors

def forward(self, image_list: ImageList, feature_maps: List[Tensor]) ->
    List[Tensor]:
    grid_sizes = [feature_map.shape[-2:] for feature_map in
                  feature_maps]
    image_size = image_list.tensors.shape[-2:]
    dtype, device = feature_maps[0].dtype, feature_maps[0].device
    strides = [
        [
            torch.empty(() , dtype=torch.int64,
                       device=device).fill_(image_size[0] // g[0]),
            torch.empty(() , dtype=torch.int64,
                       device=device).fill_(image_size[1] // g[1]),
        ]
        for g in grid_sizes
    ]
    self.set_cell_anchors(dtype, device)
    anchors_over_all_feature_maps = self.grid_anchors(grid_sizes,
                                                       strides)
    anchors: List[List[torch.Tensor]] = []
    for _ in range(len(image_list.image_sizes)):
        anchors_in_image = [anchors_per_feature_map for
                            anchors_per_feature_map in anchors_over_all_feature_maps]
        anchors.append(anchors_in_image)
    anchors = [torch.cat(anchors_per_image) for anchors_per_image in
              anchors]
    return anchors

```

---

放置好 anchors 后，接下来就要调整网络，使网络输出能够判断每个 anchor 是否有目标，同时还要有 bounding box regression 需要的 4 个值 ( $d_x, d_y, d_w, d_h$ )。

---

```

class RPNHead(nn.Module):
    def __init__(self, in_channels, num_anchors):
        super(RPNHead, self).__init__()
        self.conv = nn.Conv2d(

```

```

    in_channels, in_channels, kernel_size=3, stride=1, padding=1
)
self.cls_logits = nn.Conv2d(in_channels, num_anchors,
    kernel_size=1, stride=1)
self.bbox_pred = nn.Conv2d(
    in_channels, num_anchors * 4, kernel_size=1, stride=1
)

def forward(self, x):
    logits = []
    bbox_reg = []
    for feature in x:
        t = F.relu(self.conv(feature))
        logits.append(self.cls_logits(t))
        bbox_reg.append(self.bbox_pred(t))
    return logits, bbox_reg

```

首先进行卷积，然后对 feature 进行卷积输出 cls\_logits，同时对 feature 进行卷积输出 bbox\_pred，对应每个点的 4 个框的位置回归信息  $(d_x, d_y, d_w, d_h)$ 。

```

self.cls_logits = nn.Conv2d(in_channels, num_anchors, kernel_size=1,
    stride=1)
self.bbox_pred = nn.Conv2d(in_channels, num_anchors * 4, kernel_size=1,
    stride=1)

```

上述过程只是单个 feature\_map 的处理流程。对于 FPN 网络的输出的多个大小不同的 feature\_maps，每个特征图都会按照上述过程计算 stride 和网格，并设置 anchors。当处理完后获得密密麻麻的各种 anchors 了。

接下来进入 RegionProposalNetwork 类，在 test 阶段计算有目标的 anchor 并进行回归生成 proposals，然后采用 NMS 处理，在 train 阶段除此之外还计算 rpn loss。

首先计算有目标的 anchor 并进行框回归生成 proposals：

```

objectness, pred_bbox_deltas = self.head(features)
anchors = self.anchor_generator(images, features)

```

---

```

...
proposals = self.box_coder.decode(pred_bbox_deltas.detach(), anchors)
proposals = proposals.view(num_images, -1, 4)

```

---

然后依照 objectness 置信由大到小度排序（优先提取更可能包含目标的的），并采用 NMS 生成 boxes（即 NMS 后的 proposal boxes）：

---

```

boxes, scores = self.filter_proposals(proposals, objectness,
images.image_sizes, num_anchors_per_level)

```

---

如果是训练阶段，还要将 boxes 与 anchors 进行匹配，计算 cls\_logits 的损失 loss\_objectness，同时计算 bbox\_pred 的损失 loss\_rpn\_box\_reg。

在 RegionProposalNetwork 之后已经生成了 boxes，接下来就要提取 boxes 内的特征进行 roi\_pooling：

---

```

roi_heads = RoIHeads(
    # Box
    box_roi_pool, box_head, box_predictor,
    box_fg_iou_thresh, box_bg_iou_thresh,
    box_batch_size_per_image, box_positive_fraction,
    bbox_reg_weights,
    box_score_thresh, box_nms_thresh, box_detections_per_img
)

```

---

### 3.3.4 MultiScaleRoIAlign

RoI Align 部分，重点关注 MultiScaleRoIAlign 类的实现，此处不再赘述。

## 3.4 Training Model

FasterRCNN 模型在两处地方有损失函数：

在 RegionProposalNetwork 类，需要判别 anchor 中是否包含目标从而生成 proposals，这里需要计算 loss。在 RoIHeads 类，对 roi\_pooling 后的全连接生成的 cls\_score 和 bbox\_pred 进行训练，也需要计算 loss。

## 4 目标检测算法实践

### 4.1 Faster R-CNN 和 Mask R-CNN 对比

使用 Faster R-CNN 和 Mask R-CNN 对同一幅图片进行处理，得到的结果如图 4-1 所示。



图 4-1 FPN 基本结构示意图

从结果来看，Mask R-CNN 的效果要优于 Faster R-CNN，识别出了更多的目标，且结果更加准确。

### 4.2 视频中人体的姿态估计

这一部分，我参考网上的一些实现，使用 PyTorch 库中的预训练模型尝试对视频中人体姿态进行估计，效果如图 4-2、图 4-3、图 4-4 所示。

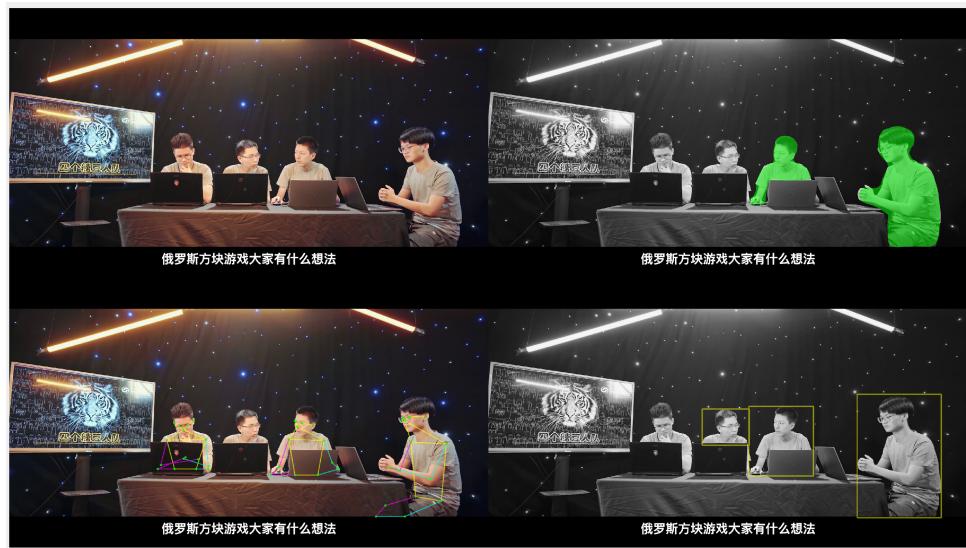


图 4-2 视频姿态估计-1

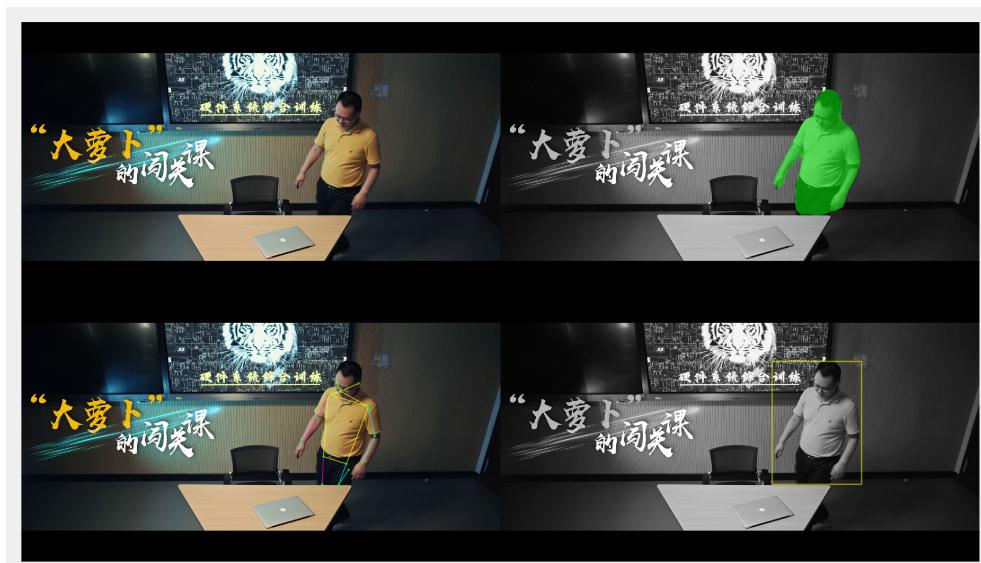


图 4-3 视频姿态估计-2



图 4-4 视频姿态估计-3

可以看到，检测结果还是比较令人满意的。但算法运行过程中，视频处理的速度明显较慢，这不仅与电脑硬件性能有关，也和算法的复杂度关系密切。通过查阅资料我了解到，目前在视频目标检测中应用更广泛的算法是 YOLO 系列算法，这也是可以改进的地方。

## 5 专题实验心得

本次专题实验让我进一步加深了对深度学习理论知识的理解。在阅读论文的过程中，我既惊叹前人的智慧，也在思考如果换作是我，有没有其他的想法？当然，在深度学习领域和计算机视觉领域已经有许多“巨人的肩膀”存在，而他们提出的算法之所以被广泛学习和应用，一定是在某个领域取得了重大突破，像我这样的初学者并不奢望能够赶上他们。记得斯坦福大学 CS231n 课程讲义中有这样一句话：I like to summarize this point as “don't be a hero” : Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on ImageNet, download a pretrained model and finetune it on your data. You should rarely ever have to train a ConvNet from scratch or design one from scratch. 既然已经有“巨人的肩膀”，与其自己一个人埋头苦干，不如好好借助他们的力量，向更深更难更广阔的领域进发。

实验过程中，我不仅对 PyTorch 框架的掌握更加熟练，也深刻感受到了一句玩笑话：既是“Burning Brain”，也是“Burning NVIDIA”。在深度学习中如何高效使用 GPU 至关重要。对于我个人而言，刚开始实验的时候使用 CPU 运行和 GPU 运行，效率完全不同。ChatGPT 出来后，深度学习领域对算力的需求进一步增加，但这并不意味着追求大模型和高算力就是唯一的出路，精巧、高效的模型也是一个发展方向，因为世界上还有着许多资源并不充裕的应用场景，例如移动端。作为一名计算机专业的学生，我能作的就是多看、多读、多想，努力了解前沿领域，大胆尝试自己感兴趣的事物。

最后，感谢老师在理论课和实验课上对我们的悉心指导，您辛苦了！

## 参 考 文 献

- [1] Girshick, R., Donahue, J., Darrell, T. & Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings Of The IEEE Conference On Computer Vision And Pattern Recognition*. pp. 580-587 (2014)
- [2] Girshick, R. Fast r-cnn. *Proceedings Of The IEEE International Conference On Computer Vision*. pp. 1440-1448 (2015)
- [3] Ren, S., He, K., Girshick, R. & Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances In Neural Information Processing Systems*. **28** (2015)
- [4] He, K., Gkioxari, G., Dollár, P. & Girshick, R. Mask r-cnn. *Proceedings Of The IEEE International Conference On Computer Vision*. pp. 2961-2969 (2017)
- [5] He, K., Zhang, X., Ren, S. & Sun, J. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions On Pattern Analysis And Machine Intelligence*. **37**, 1904-1916 (2015)

## 附录

### A Faster R-CNN 和 Mask R-CNN 效果对比代码

---

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import os
import random
import cv2
import numpy as np
import matplotlib.pyplot as plt

import torch
from torchvision import transforms as T
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection import maskrcnn_resnet50_fpn

from PIL import Image


# 设置 torch.device。
# In[2]:


device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 从数据集中随机选择一张图片，分别使用 Faster R-CNN 和 Mask R-CNN
# 处理。这里采用 VOC2012 数据集。
```

# In[3]:

```
voc_path = "./data/VOCdevkit/VOC2012"
image_set = 'train'
# 读取图像列表
with open(os.path.join(voc_path, 'ImageSets', 'Main', f'{image_set}.txt')) as f:
    image_list = f.read().splitlines()

# 随机选择一个图像
random_image = random.choice(image_list)
img_pil = Image.open(os.path.join(voc_path, 'JPEGImages',
f'{random_image}.jpg')).convert('RGB')
# 展示图像
img_pil
```

# In[4]:

```
# 预处理图像
img_transform = T.Compose(
[T.ToTensor()])
img = img_transform(img_pil).unsqueeze(0)
img = img.to(device)
img
```

# In[5]:

```
# 加载预训练的 Faster R-CNN 和 Mask R-CNN 模型
faster_rcnn_model = fasterrcnn_resnet50_fpn(weights=True)
mask_rcnn_model = maskrcnn_resnet50_fpn(weights=True)

faster_rcnn_model.cuda()
mask_rcnn_model.cuda()

# 将模型置为 eval 模式
faster_rcnn_model.eval()
mask_rcnn_model.eval()

# In[6]：

# 使用模型进行目标检测
with torch.no_grad():
    faster_rcnn_pred = faster_rcnn_model(img)
    mask_rcnn_pred = mask_rcnn_model(img)

# 可视化结果

# In[7]：

# COCO 数据集标签对照表。PyTorch 的 Faster R-CNN 和 Mask R-CNN 模型是在
# COCO 数据集上训练的，需要对标签进行映射
COCO_CLASSES = {1: 'person', 2: 'bicycle', 3: 'car', 4: 'motorcycle', 5:
    'airplane',
    6: 'bus', 7: 'train', 8: 'truck', 9: 'boat', 10: 'traffic
        light',
    11: 'fire hydrant', 13: 'stop sign', 14: 'parking meter',
    15: 'bench',
    16: 'bird', 17: 'cat', 18: 'dog', 19: 'horse', 20: 'sheep',
```

```

21: 'cow',
22: 'elephant', 23: 'bear', 24: 'zebra', 25: 'giraffe', 27:
    'backpack',
28: 'umbrella', 31: 'handbag', 32: 'tie', 33: 'suitcase',
    34: 'frisbee',
35: 'skis', 36: 'snowboard', 37: 'sports ball', 38: 'kite',
    39: 'baseball bat',
40: 'baseball glove', 41: 'skateboard', 42: 'surfboard', 43:
    'tennis racket',
44: 'bottle', 46: 'wine glass', 47: 'cup', 48: 'fork', 49:
    'knife', 50: 'spoon',
51: 'bowl', 52: 'banana', 53: 'apple', 54: 'sandwich', 55:
    'orange',
56: 'broccoli', 57: 'carrot', 58: 'hot dog', 59: 'pizza',
    60: 'donut',
61: 'cake', 62: 'chair', 63: 'couch', 64: 'potted plant',
    65: 'bed', 67: 'dining table',
70: 'toilet', 72: 'tv', 73: 'laptop', 74: 'mouse', 75:
    'remote', 76: 'keyboard',
77: 'cell phone', 78: 'microwave', 79: 'oven', 80:
    'toaster', 81: 'sink',
82: 'refrigerator', 84: 'book', 85: 'clock', 86: 'vase', 87:
    'scissors',
88: 'teddy bear', 89: 'hair drier', 90: 'toothbrush'}

COLORS = ['#e6194b', '#3cb44b', '#ffe119', '#0082c8', '#f58231',
    '#911eb4', '#46f0f0', '#f032e6',
    '#d2f53c', '#fabebe', '#008080', '#000080', '#aa6e28', '#fffac8',
    '#800000', '#aafffc3', '#808000',
    '#ffd8b1', '#e6beff', '#808080']

def get_label(label: str):
    return COCO_CLASSES[label]

```

```
def show_result(pred, img, confidence_threshold: float = 0.8):
    for box, score, label in zip(pred[0]['boxes'], pred[0]['scores'],
                                 pred[0]['labels']):
        if score > confidence_threshold:
            box = box.cpu().numpy().astype(int)
            cv2.rectangle(img, (box[0], box[1]), (box[2], box[3]), (255, 0,
                0), 2)
            cv2.putText(img, get_label(label.item()), (box[0], box[1] - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                        (255, 0, 0), 2)
```

# In[8]:

```
faster_rcnn_img = np.array(img_pil)
mask_rcnn_img = np.array(img_pil)

show_result(faster_rcnn_pred, faster_rcnn_img)
show_result(mask_rcnn_pred, mask_rcnn_img)
```

# 作图对比两种模型的检测结果。

# In[9]:

```
fig, axes = plt.subplots(1, 2, figsize=(15, 30))
# Faster R-CNN
axes[0].imshow(cv2.cvtColor(faster_rcnn_img, cv2.COLOR_BGR2RGB))
axes[0].axis('off')
axes[0].set_title("Faster R-CNN")
# Mask R-CNN
axes[1].imshow(cv2.cvtColor(mask_rcnn_img, cv2.COLOR_BGR2RGB))
```

```
axes[1].axis('off')
axes[1].set_title("Mask R-CNN")
# 对比图
plt.show()
```

---

## B 视频人体姿态估计代码

```
import cv2
import numpy as np
from utils.utils import ConvolutionalPoseMachine,
    draw_body_connections, draw_keypoints, draw_masks, draw_body_box

# 实例化 ConvolutionalPoseMachine 类
estimator = ConvolutionalPoseMachine(pretrained=True)
# 使用 opencv 读入视频
cap = cv2.VideoCapture('data/hust.mp4')

# 读取成功意味着 cap.isOpened()==True, 持续运行
while True:
    # frame 相当于一帧一帧的图像
    _, frame = cap.read()
    # 传入视频帧至实例化后的 ConvolutionalPoseMachine 类
    pred_dict = estimator(frame, masks=True, keypoints=True)
    # 调用定义的 get_masks 静态方法获取掩膜
    masks = estimator.get_masks(pred_dict['maskrcnn'],
        score_threshold=0.99)
    # 调用定义的 get_keypoints 静态方法获取关键点
    keypoints = estimator.get_keypoints(pred_dict['keypointrcnn'],
        score_threshold=0.99)
    # 调用定义的 get_boxes 静态方法获取关键点
    boxes = estimator.get_boxes(pred_dict['fasterrcnn'],
        score_threshold=0.99)
    # BGR 转灰度图像
    frame_dst = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # 合并单通道成多通道
    frame_dst = cv2.merge([frame_dst] * 3)
    # 绘制掩膜
    overlay_m = draw_masks(frame_dst, masks, color=(0, 255, 0), alpha=0.5)
    # 绘制预测框
```

```
overlay_b = draw_body_box(frame_dst, boxes, thickness=3)
# 连接关键点
overlay_k = draw_body_connections(frame, keypoints, thickness=2,
                                    alpha=0.7)
# 绘制关键点
overlay_k = draw_keypoints(overlay_k, keypoints, radius=4, alpha=0.8)
# 将参数元组的元素数组按水平方向及垂直方向进行叠加
# 预计显示结果如下示意:
#
# | 原图 | 掩膜预测图 |
#
# | 关键点及连接绘制图 | 预测框绘制图 |
#
# 水平排列
image_h1 = np.hstack((frame, overlay_m))
image_h2 = np.hstack((overlay_k, overlay_b))
# 垂直排列
image_v_and_h = np.vstack((image_h1, image_h2))
# 处理后的视频帧显示
cv2.namedWindow("Video Show", 0) # 0 可调大小, 窗口名必须 imshow
                                中的窗口名一致
cv2.resizeWindow("Video Show", 1600, 900) # 设置窗口的长和宽
cv2.imshow('Video Show', image_v_and_h) # 显示视频
if cv2.waitKey(1) & 0xff == 27: # 按下 ESC 退出
    break

# 释放资源并关闭窗口
cap.release()
cv2.destroyAllWindows()
```

---