第 5 章

卷积神经网络理论与实践(二)——目标检测

5.1 什么是目标检测：从“是什么”到“在哪里”

在第 4 章中，我们学会让神经网络看一张图片，并回答一个核心问题：**“这张图片中主要是什么?”** 这是一个图像识别任务。它的输出是一个单一的标签，如“猫”、“狗”或“汽车”。

但在现实生活中，世界远比这复杂。请看来自 COCO 数据集的典型图片(图 5-1)。

图 5-1 经典目标检测原图

现在，如果向一个训练好的分类模型提问：“这张图片中是什么?”，我们可能会得到一个令人困惑的答案。因为图片中包含多个显著物体，模型可能会选择一个它认为最突出的，如“汽车”或“人”，但这显然丢失了大量信息。

人们更希望模型能告诉我们：

(1) **图片中有哪些物体**？(不止一个)

(2) **它们分别在哪里**？(用框标出位置)

(3) **它属于哪个类别的信心有多大**？(置信度)

这个能同时解决“是什么”和“在哪里”问题的任务，就是目标检测。

5.1.1　快速案例：用几行代码实现目标检测

暂时抛开复杂的训练过程，本案例直接使用一个现成的、强大的预训练模型(YOLO)来感受目标检测的强大能力。这个案例的目的是先看到结果，建立直观感受。

```
# 导入必要的库
import cv2
from ultralytics import YOLO
# 1. 加载一个预训练好的 YOLOv8 模型(就像一个经验丰富的"视觉助手")
model = YOLO('yolov5s.pt')        # 这里我们使用轻量版的 YOLO 模型
# 2. 读取我们的示例图片
image_path = "street_scene.jpg"
img = cv2.imread(image_path)
# 3. 运行检测!让模型告诉我们图片中有什么,在哪里
results = model(img)
# 4. 将检测结果(边框、标签、置信度)可视化到图片上
annotated_img = results[0].plot()
# 5. 显示并保存结果
cv2.imshow("Detection Results", annotated_img)
cv2.waitKey(0)
cv2.imwrite("detected_street_scene.jpg", annotated_img)
```

输出结果如图 5-2 所示。

图 5-2　经典目标检测结果图

看！通过这短短的几行代码，案例程序不再只是笼统地给图片贴标签，而是像一位观察者一样，清晰地指出了一个人和一辆汽车，并精确地指出了它们的位置，甚至还告诉我们对这个判断的把握有多大（置信度）。这就是目标检测的魔力。

5.1.2 从分类到检测：思维的转变

我们可以把**分类网络**想象成一个只关心“主旨大意”的读者。而**检测网络**则像一个严谨的编辑，不仅要知道文章讲了什么（分类），还要把重点句子一个个**高亮标注**出来（定位）。

接下来将逐步深入，亲手搭建和训练这些能同时完成“识别”和“定位”任务的强大网络，探索它们是如何被设计出来以解决更复杂的问题的。

5.2 目标检测的解决思路：二阶段与一阶段的哲学

在 5.1 节的案例体现了目标检测的强大能力。那么，工程师和科学家是如何设计算法来解决这个“是什么”和“在哪里”的复杂问题的呢？纵观其发展历程，主要衍生出了两种宏观思路：**二阶段（Two-Stage）方法**和**一阶段（One-Stage）方法**。

5.2.1 二阶段方法（RCNN 系列）：分段优化的范式与固有瓶颈

二阶段方法就是“先找可能的区域再确认”。RCNN 方法是第一个用 CNN 做目标检测的方法，用传统图像处理算法（Selective Search，图 5-3）把图像分割成不同尺度、可能存在目标的区域。

第一阶段：寻找“在哪里”的候选答案（Region Proposal，区域提议），让计算机先快速地、粗略地“扫一眼”图片，找出所有可能包含物体的区域，这个过程与物体类别无关，只是找出像物体的区域。

第二阶段：对候选答案进行“是什么”和“精修在哪里”的判别（分类与回归），对第一阶段提出的成千上万个候选区域进行精细分类和边框微调，追求“精确度”。

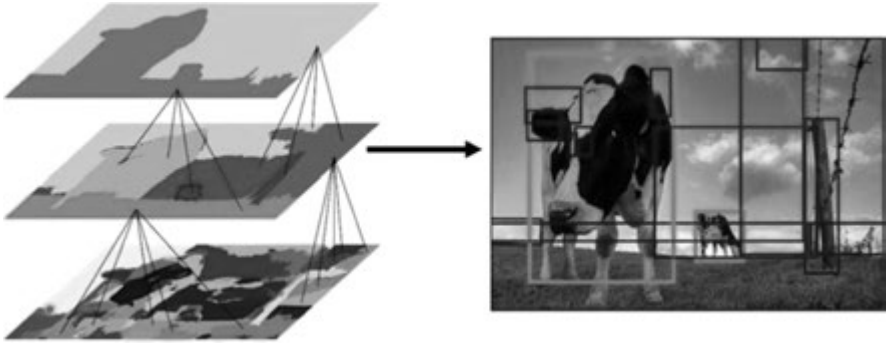

图 5-3 Selective Search 方法示意图

在 RCNN 的基础上，后来又发展出了 Fast RCNN 与 Faster RCNN 方法。

RCNN 对每个 Region Proposal 分别应用 CNN 模型需要极大的运算量，而 Fast RCNN 则先对整张图用一个 CNN 提取一次丰富的特征图（Feature Map），然后从这张“特征图”上

裁剪出各个提议区域对应的部分来进行分类和回归。这种“**只对原图做一次特征提取**”的策略好比先给整个病人做一次全身的造影，然后所有医生都直接看这张影像来诊断不同区域，无须来回对病人各个部位进行拍片检查。这大大减少了计算量，提高了效率，所以冠名为Fast，如图 5-4 所示。

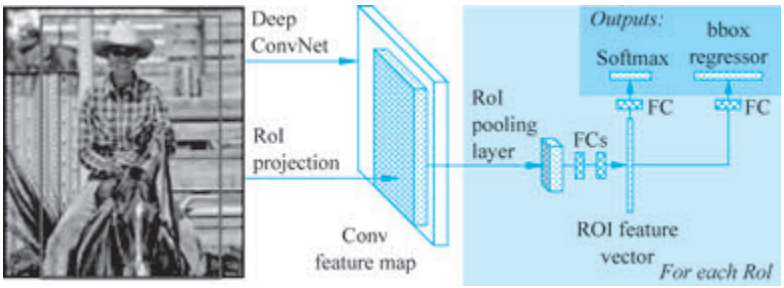

图 5-4　只做一次特征提取的 Fast RCNN 方法示意图

思考：从 CNN 特征感知的角度看，为什么在提取特征后做定位也是正确的？

Faster RCNN 用神经网络学习如何“观察”。其 Region Proposal Network（RPN）是一次重要革新，直接从 CNN 提取的特征图上快速生成提议区域，将区域提议变成了一个可学习的、轻量级的网络模块，使得“观察”和“细看”两个步骤可以共享同一个特征提取网络，进一步实现了端到端的训练和速度的提升（图 5-5）。然而，RPN 的本质仍然是一个粗糙的、需要海量预选的初步筛选器。它的任务不是做出最终判断，而是生成大量可能存在目标的预选框（Anchor），为后续的 Refinement 阶段提供素材。

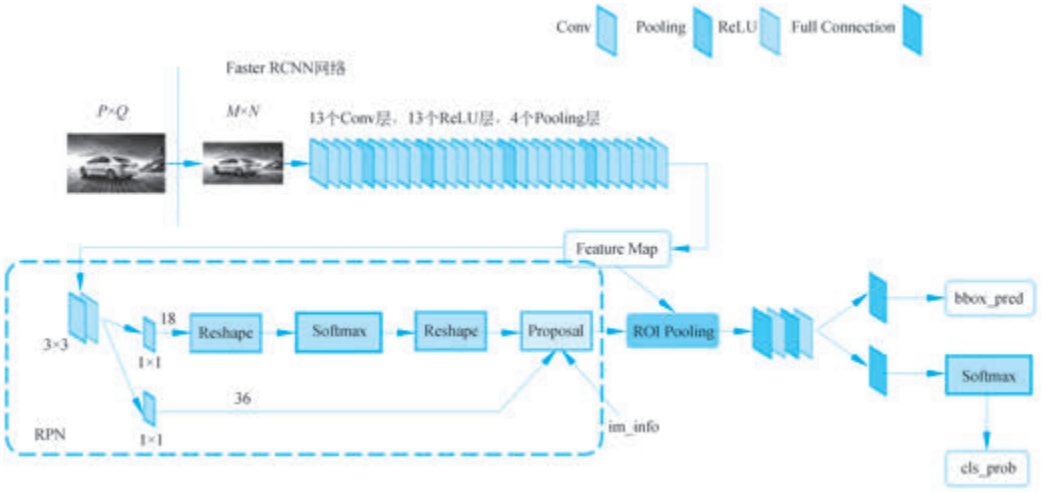

图 5-5　Faster RCNN 与 RPN 示意图

然而，二阶段方法的弱点是其架构与生俱来的：它对“在哪里”和“是什么”两个问题的解答是分步、异构的。即便 Faster RCNN 通过 RPN 引入了部分学习能力，但其核心思路仍然是通过一个相对简陋的模块海量生成假设，再由另一个复杂模块进行验证。这种设计在计算上存在冗余，在性能潜力上也受到了第一阶段的限制。

5.2.2 一阶段方法（YOLO 系列）：统一建模的范式革命

一阶段方法的核心思想是：**“一眼看完 You Look Only Once（YOLO）”**。它摒弃了“先提议，再检测”的复杂流程，它认为“在哪里”“是什么”“有多大可信度”这三个问题应该被**同步、联合**地解答。

YOLO 将输入图像均匀地划分成 $S \times S$ 个网格（Grid Cell）。让每个网格单元直接、同时地预测：

（1）位置：预测多个边界框的坐标和尺寸。

（2）类别：预测每个边界框内物体属于各个类别的概率。

（3）置信度：预测每个边界框包含目标且预测准确的可信程度。

这是一个高度集成的输出。网络的一次前向传播，直接产出了对上述三个问题的完整答案集合。**在每个网格单元中以不同的尺度寻找：如果某个物体的中心落入了这个网格，那么这个物体是什么？它的准确边界框是怎样的？**

一阶段方法的网络参数在训练时，需要同时学习定位、分类和置信度评估这三个任务。这意味着用于判断“在哪里”的特征，自始至终都受到“是什么”这个最终任务的直接反馈和调节。模型学会了提取对分类和定位都最有效的、共享的高层特征，实现了真正的端到端优化。

一阶段方法并非一开始就在精度上领先，但其**原理上的先进性**为其后来的统治地位奠定了基础。通过一系列技术改进（如引入 Anchor 机制、设计更强大的特征金字塔网络 FPN、使用更科学的损失函数），在保持高速度的同时，其精度也实现了反超。

5.3 深入经典检测器 YOLO——以 YOLOv5 为例

YOLOv5 由于其卓越的性能、简洁的代码和庞大的用户群体，是近年来最受欢迎的 YOLO 实现之一。它集成了此前 YOLO 系列及业界各种最佳实践，代表了单阶段目标检测的工程化水平。

5.3.1 YOLOv5 的训练数据：COCO 数据集及其加载

COCO（Common Objects in Context）数据集是计算机视觉领域一个至关重要的大型数据集，其全称为“常见物体情景数据集”。它由微软团队构建，核心价值在于推动图像识别、目标检测、分割和图像描述生成等任务的研究。该数据集规模庞大，包含超过 33 万张图像，其中超过 20 万张带有精细标注，共计标注了 150 万个目标实例。这些图像涵盖了 80 种日常生活中常见的物体类别，如人、车辆、动物和各种日常用品。

COCO 数据集的标注极其丰富，不仅提供了标准的物体边界框，还包括了用于实例分割的像素级掩码、用于人体姿态估计的关键点坐标以及描述图像内容的自然语言字幕。数据集通常按年份版本（如 2017 版）组织，主要分为训练集（约 12 万张图像）、验证集（5000 张图像）和测试集（约 4 万张图像，标注未公开）三个子集，其标注信息以统一的 JSON 格式文件存储。

该数据集最大的特点是其图像源自复杂的真实生活场景，如图 5-6 所示，画面中常包含

多个目标、存在遮挡且目标尺度多变，因此对模型的泛化能力提出了严峻挑战。正因如此，COCO 数据集已成为评估目标检测、实例分割等算法性能的权威基准之一。

图 5-6　COCO 数据集样本

以下是配置并加载 COCO 2017 版数据集的相关代码，先创建配置文件 coco.yaml：

```
# COCO 2017 dataset
    path: ../datasets/coco        # 替换为 COCO 数据集的存储路径
    train: train2017.txt
    val: val2017.txt
    test: test-dev2017.txt
    # 80 个类别
    names: ['person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus', 'train', 'truck', 'boat',
'traffic light', 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog',
'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat',
'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork',
'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',
'pizza', 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'dining table', 'toilet', 'tv',
'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink',
'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```

在主文件中可以通过字典来访问数据集配置，实现代码模块化：

```
from utils.general import check_dataset
# 1. 设置数据集配置文件路径
data_config = 'data/coco.yaml'                # COCO 数据集配置文件
# 2. 验证并加载数据集配置
data_dict = check_dataset(data_config)        # 检查并解析数据集配置
train_path = data_dict['train']               # 训练集路径
val_path = data_dict['val']                   # 验证集路径
nc = int(data_dict['nc'])                     # 类别数量 (COCO 为 80 类)
names = data_dict['names']                    # 类别名称列表
```

5.3.2 网络结构：模块化与精细化的设计

YOLOv5 的网络结构可以清晰地分为三个核心部分：Backbone(主干网络)、Neck(颈部)和 Head(检测头)。这种设计使得网络能够高效地提取特征并融合不同尺度的信息。

如图 5-7 所示，Backbone (CSPDarknet)：负责从输入图像中提取不同层次的特征。其主要组件是 C3 模块(借鉴了 CSPNet 结构)和 SPPF 模块(快速空间金字塔池化)，能有效增强网络的学习能力并减少计算量。

Neck(PANet)：负责融合来自 Backbone 不同层次的特征图。它采用 FPN (特征金字塔网络) 和 PAN(路径聚合网络) 的结构，实现自上而下和自下而上的双向特征融合，使得网络能够同时利用深层的语义信息和浅层的定位信息，极大地提升了对不同尺度目标的检测能力。

Head (检测头)：最终进行预测的部分。它对 Neck 融合后的不同尺度的特征图进行卷积操作，直接输出最终的预测张量。

5.3.3 输出的奥秘：Anchor 与预测张量

在 YOLOv5 中，Anchor 是指先验的锚点框，用于匹配目标。Anchor 的生成发生在训练开始之前，是基于数据集的目标框标签生成的。

以在 COCO 数据集(80 个类别)上训练的 YOLOv5 模型为例，来解读其输出。

COCO 中的图像样本输入尺度为 640×640，YOLOv5 在 3 个不同尺度的特征图上进行下采样(P3/8、P4/16、P5/32)。其中 P3 是特征层的名称，8 表示下采样的尺度。生成的 9 个 Anchor 会按照尺寸大小被分配到 3 个特征层上：P3/8(80×80)负责检测小目标，分配尺寸最小的 3 个 Anchor(在 COCO 中是[10,13]，[16,30]，[33,23])，P4/16 与 P5/32 同理，分别检测中等目标与大目标。

以 P3/8(80×80)为例，输出张量形状为[batch，3，80，80，85]。其中第三、四个维度 80 将特征图视为一个 80×80 的网格，每个网格单元负责预测其中心点附近的物体；第二个维度 3 表示该尺度下的每个网格点对应 3 个先验的锚点框(Anchor)。

85 是每个锚点框预测的向量长度，由 4 个**坐标偏移量**＋1 个**物体置信度**＋80 个**类别置信度**构成。

坐标偏移量分为 tx，ty，tw，th(4 个值)，预测的是相对于锚点框的偏移量和缩放系数，并非直接的坐标。网络需要学习如何微调锚点框的位置和大小，使其更精准地覆盖目标。

物体置信度表示该锚点框内存在目标的概率(0～1 之间)。

类别置信度表示如果框内有目标，它属于 COCO 数据集中 80 个类别中每一类的概率。通常取置信度与最大类别概率的乘积作为最终的检测置信度。

因此，对于 80×80 的输出，网络一共给出了 80×80×3＝19200 个预测框。后续再通过非极大值抑制(NMS)等后处理步骤，过滤掉重叠和低置信度的框，得到最终的检测结果。

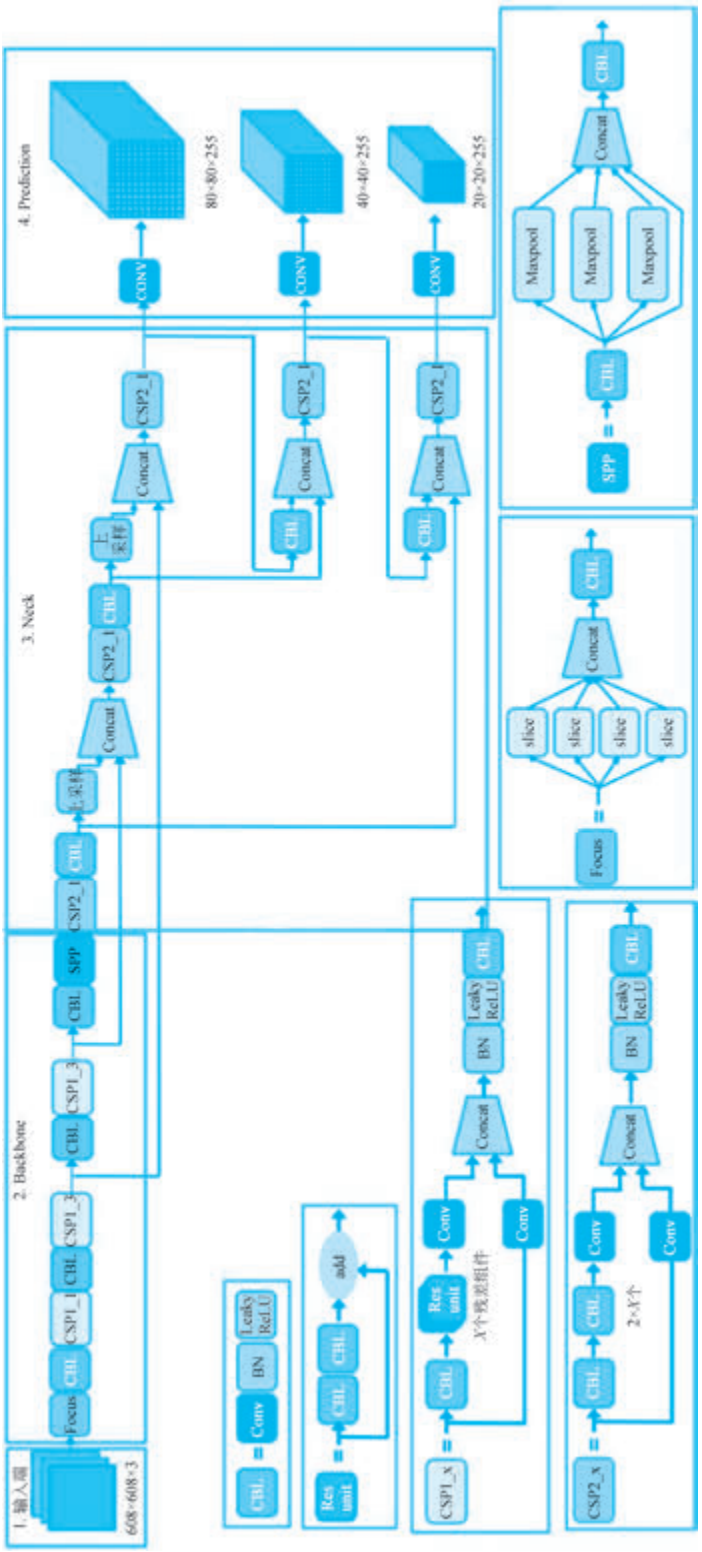

图 5-7　YOLOv5 的网络结构图

5.3.4 优化目标：损失函数的构成

与模型的输出结构相似，YOLOv5 的损失函数同样是一个多任务损失函数，由**定位损失**、**置信度损失**和**分类损失**三部分加权求和构成：

$$\text{TotalLoss} = \text{Loss}_{\text{reg}}(\text{CIoU}) + \text{Loss}_{\text{obj}}(\text{BCE}) + \text{Loss}_{\text{cls}}(\text{BCE})$$

（1）定位损失 Loss_{reg}，判断框的位置：

YOLOv5 不再使用简单的均方误差（MSE），而是采用了 CIoU Loss。CIoU 不仅考虑预测框与真实框的重合面积（IoU），还考虑了中心点距离和宽高比的一致性。这使得边框的回归更加准确和快速，是 YOLOv5 精度高的关键原因之一。

（2）置信度损失 Loss_{obj}，判断“框内是否有物体”，是一个二分类问题（物体/背景）。

（3）分类损失 Loss_{cls}，判断“框内的物体具体是哪一类”，是一个多分类问题。

两者都使用二元交叉熵损失（BCEWithLogitsLoss）。BCE 损失在处理概率预测时比平方误差更有效。

5.4 目标检测案例实战

通过前面的学习，读者已经掌握了目标检测与 YOLO 的核心原理。本节将学习如何使用一个预训练的 YOLO 模型来完成检测任务，在此基础上，探索如何为其注入新的“知识”，使其适应我们自己的特定场景。

5.4.1 使用预训练模型进行快速推理

前面已经简单接触了 YOLO 快速推理的步骤与实现方式，这里将其适当完善并简单讲解。

如今，YOLO 的官方代码库已经提供了极其完善的封装，最主流和推荐的方式是使用 ultralytics 库。一般情况下可以直接调用预训练模型完成推理。

首次使用 ultralytics 库加载某个预训练模型（如 YOLO('yolov5s.pt')）时，如果本地没有该模型，ultralytics 库会自动从网络下载并将其保存到特定目录。

这些预训练模型通常默认保存在用户目录下的 .ultralytics 文件夹中。具体路径根据操作系统有所不同：

macOS/Linux：/home/<你的用户名>/.ultralytics/

Windows：`C:\Users\<用户名>\.ultralytics`

在某些情况下，预训练权重也可能被缓存到`C:\Users <用户名>\.cache\torch\hub\checkpoints`目录下。

第一步：模型加载与推理

```
from ultralytics import YOLO
    # 1. 加载一个预训练模型
    # 'yolov5s.pt' 是最轻量、最快的版本，适合快速验证和部署在算力受限的设备上
    # 其他可选模型：'yolov5m.pt', 'yolov5l.pt', 'yolov5x.pt'（精度更高，速度更慢）
    model = YOLO('yolov5s.pt')      # 自动从官方仓库下载权重
```

```
# 2. 进行推理
# `source` 可以是指定图片/视频路径、摄像头 ID(如 0)、URL 或图片目录
results = model(source = 'path/to/your/image.jpg',
                save = True,    # 保存绘制了结果框的图像
                conf = 0.5,     # 置信度阈值,只显示大于此值的预测框
                iou = 0.45,     # NMS 操作的 IoU 阈值,值越大,允许的重叠越小
                show = False,   # 如果为 True,会尝试显示结果(适合 Jupyter Notebook)
                device = 'cpu') # 使用设备,'cpu' 或 '0'(第 0 块 GPU)
```

上述代码中的 model()调用是核心,其常用参数详解如下:

- source:指定待检测目标的路径,可以对图像、视频、摄像头等进行推理。
- conf:置信度阈值。模型会输出大量预测框,此参数用于过滤掉低置信度的预测,值越高,剩下的框越少、越可信。
- iou:非极大值抑制(NMS)的阈值,用于合并重叠的预测框。值越小,NMS 越严格,保留的框越少。
- device:指定计算设备,通常 GPU(device=0)比 CPU 快一个数量级。
- save/ show:控制结果的输出形式。

第二步:解析与使用结果

推理返回的 results 是一个列表,其中包含了所有输入图像的检测结果(如果输入是视频或批量图片)。可以方便地遍历和提取信息:

```
# 假设我们只处理一张图片
result = results[0]

# 1. 获取检测到的目标信息
# result.boxes 包含了所有检测框的信息
boxes = result.boxes
print(f"检测到 {len(boxes)} 个目标")

# 遍历每个框
for box in boxes:
    # 获取坐标 (xyxy 格式: 左上 x, 左上 y, 右下 x, 右下 y)
    coord = box.xyxy[0].cpu().numpy()
    # 获取置信度
    confidence = box.conf[0].cpu().numpy()
    # 获取类别 ID 和名称
    class_id = int(box.cls[0].cpu().numpy())
    class_name = result.names[class_id]

    print(f"检测到 '{class_name}', 置信度: {confidence:.2f}, 位置: {coord}")

# 2. 可视化结果
# 带标注的结果图像已自动保存(如果设置了 save = True)
# 你也可以获取绘制好的图像数组,用于进一步显示或处理
annotated_img = result.plot()      # 返回一个 BGR 格式的 NumPy 数组

# 在 OpenCV 中显示
import cv2
cv2.imshow('YOLOv5 Detection', annotated_img)
cv2.waitKey(0)
```

YOLO 模型本身是一个活跃的、快速迭代的系列。ultralytics 库会持续更新，以支持包括 YOLOv8、YOLOv11 乃至未来版本在内的新一代模型。因此，在实际项目中，YOLOv5 并非唯一的选择。新一代模型通常在精度、速度或架构效率上有所提升。对于新项目，建议访问 ultralytics 官方文档或 GitHub 仓库，了解当前推荐的最新稳定版本。

ultralytics 为每个版本的 YOLO（如 v5、v8）都提供了多个不同规模的子版本，形成了从极致轻快到超高精度的模型家族。选择的核心是在速度、精度和模型大小之间找到符合项目需求的平衡点。以下是一个通用的选型策略：

极致速度与轻量化（YOLOv5n，YOLOv8n...）：适合边缘计算设备（如 Jetson Nano、树莓派）、移动端或需要极高帧率的实时应用（如无人机、嵌入式监控），这是资源受限环境下的首选。

最佳平衡（YOLOv5s，YOLOv8s...）：在速度和精度之间取得了良好平衡，是大多数桌面级应用和服务器端应用的理想起点，适合初学者快速验证想法和部署原型。

高精度（YOLOv5m/l/x，YOLOv8m/l/x...）：拥有更大的参数量，能够捕捉更复杂的特征，从而获得更高的检测精度，适合对准确率要求严格的场景（如工业质检、医学影像分析），但需要更强的计算资源（如 GPU 服务器）。

对于大多数新项目，可以从 YOLOv8m 或当前最新的中等规模模型开始进行基准测试，它通常能在性能和资源消耗之间提供一个不错的平衡点。最终选择都应基于你在自己的数据集和硬件环境上进行实际测试的结果。

5.4.2 超越调库——自定义数据训练与微调

直接调用预训练模型虽然方便，但有其局限性：

（1）类别限制：预训练模型（如在 COCO 上训练的）只能检测 80 个固定类别。如果目标是“安全帽”“显微镜”“熊猫”，它将无法识别。

（2）场景适应性：预训练模型在通用数据上表现良好，但在特定领域（如医疗影像、遥感图像、工业质检），背景、目标形态、光照条件可能差异巨大，导致精度下降。

（3）精度要求：某些应用对误检、漏检有极严格要求，通用模型可能无法满足。

这时需要对模型进行微调（Fine-tuning），即在预训练模型的基础上，使用自己的数据集进行继续训练微调。以下是微调的详细步骤。

1. 数据准备：制作自定义数据集

收集并标注自己的图像。标注工具推荐使用 LabelImg 或 CVAT。标注完成后，需要将数据集组织成 YOLO 格式：

```
custom_dataset/
├── images/
│   ├── train/                # 训练集图片
│   └── val/                  # 验证集图片
└── labels/
    ├── train/                # 训练集标签（.txt 文件）
    └── val/                  # 验证集标签（.txt 文件）
```

每个标签.txt 文件的内容格式为：< class_id > < center_x > < center_y > < width > < height >，其中坐标是归一化后的（0～1 之间）。

2. 创建数据集配置文件

按照 3.1 节的形式创建一个.yaml 文件(如 my_data.yaml)。

3. 启动微调训练

使用 ultralytics 库提供的命令行接口或 Python API 进行训练是最高效的方式。

命令行方式：

```
yolo task = detect mode = train model = yolov5s.pt data = my_data.yaml epochs = 100 imgsz = 640
batch = 16
```

其中：

- model＝yolov5s.pt：指定预训练权重作为起点，这是微调的核心，能加速收敛并提升性能。
- data＝my_data.yaml：指定你的数据集配置文件。
- epochs＝100：训练轮数 100 轮。
- imgsz＝640：指定输入图像尺寸为 640×640。
- batch＝16：批大小为 16，根据 GPU 显存调整。

Python API 方式(如 3.1 节调用数据集配置.yaml 文件相似)：

```
from ultralytics import YOLO
    model = YOLO('yolov5s.pt')        # 加载预训练模型
    results = model.train(data = 'my_data.yaml', epochs = 100, imgsz = 640, batch = 16)
```

4. 使用微调后的模型

训练完成后，最佳模型会保存在 runs/train/exp/weights/best.pt。使用它与使用预训练模型毫无区别：

```
from ultralytics import YOLO
    # 加载我们自定义训练的模型
    custom_model = YOLO('runs/train/exp/weights/best.pt')
    # 用它进行推理，它会识别我们自定义的类别
    results = custom_model('path/to/new_image.jpg', save = True)
```

通过微调，不需要从零开始训练一个需要数百万张图片的模型，而是利用预训练模型已经学到的通用特征提取能力(如边缘、形状、纹理)，快速适应新的特定任务，通常只需几百到几千张标注图像即可获得优异的效果。