

# 大模型驱动开放世界目标感知研究报告

## 实验目标:

- 1、研究大模型驱动的开放世界目标感知，融合使用多个大模型的多模态感知能力，实现开放世界零样本条件下，自动目标检测、识别、分割功能。

## 实验内容 1 - RAM++ 的理解和实现:

### 现有的技术方案的缺陷:

- 在细类别上存在不足，无法展现出图像中的二级类别信息，导致监督信号存在偏移
- 使用既定的词汇进行训练，限制了其在开放词汇下的检索能力。

### 论文主要贡献可以总结如下:

- 将图像-标签-文本三联体集成在一个统一的对齐框架中，在预定义的标签类别上实现了优越的性能，并增强了对开放集类别的识别能力。
- 第一次将 LLM 的知识纳入图像标记训练阶段，允许模型集成视觉描述概念，以便在推理过程中进行开放集类别识别。
- 对 OPENImage、ImageNet、HICO 基准测试的评估表明，RAM++在大多方面都超过了现有的 SOTA 开放集图像标记模型，综合实验为强调多粒度文本监督的有效性提供了证据。

### 相关工作（提高标记模型的开放集能力）:

Tag Supervision-标签监督：为一个图像分配多个标签，多标签训练（图片多分类）

Text Supervision-文本监督：使用统一的对齐框架内对齐多个文本和单个标签（图文对齐训练）

Description Supervision-描述监督：Tag 对应的 LLM 知识集成到训练过程中

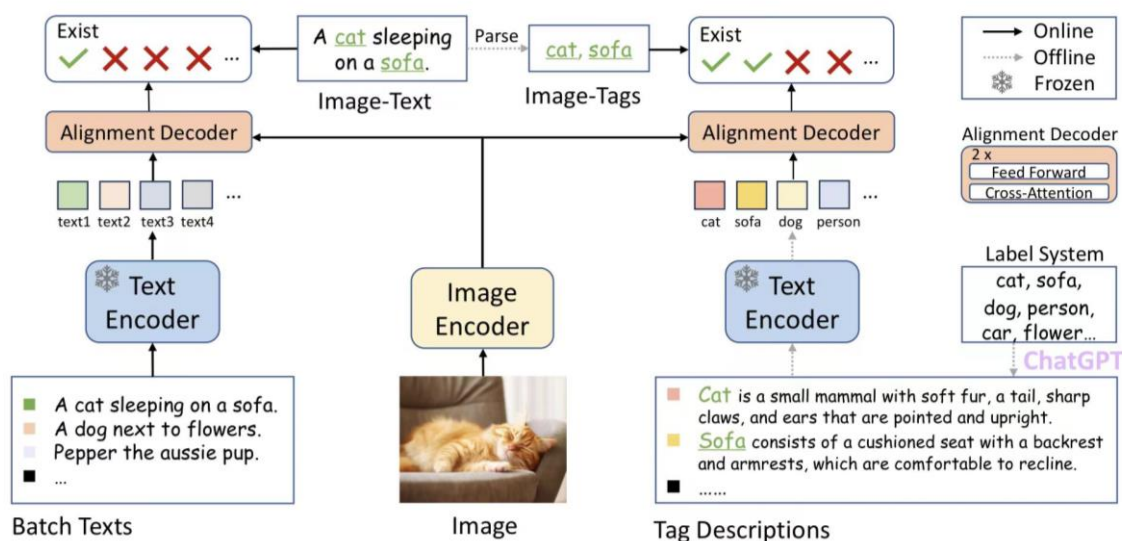


图 1 RAM++ 训练框架解释说明

RAM++, 是一个基于多粒度文本监督的开放集图像标记模型, 包括细节描述文本监督和 tag 描述监督。如图 1 所示, RAM++的体系结构包括图像编码器、文本编码器和对齐解码器。训练数据是图像-标签-文本三联体, 包括图像-文本对和从文本中解析出的 image 标记。在训练过程中, 模型的输入由包含 batch 间可变的文本和固定标签描述的图像组成。然后模型输出对应于每个图像标签/文本对的对齐概率分数, 通过对齐损失进行优化。

【框架图再解释】—— 对于图像-标签-文本三联体, RAM++采用了一个共享的对齐解码器来同时对齐图像-文本和图像标签。为了清晰起见, 图 1 将这个框架分成了两个部分。左边的部分说明了图像-文本对齐的过程, 其中来自当前训练批处理的文本通过文本编码器来提取全局文本嵌入。这些文本嵌入随后通过对齐解码器中的交叉注意层与图像特征对齐, 其中文本嵌入作为查询, 图像特征作为键和值。相反, 右边的部分强调图像标记的过程, 其中图像特征使用相同的文本编码器和对齐解码器交互。

### 创新方法的理解:

- RAM++的图像-标签-文本对齐 (ITTA) 架构通过合并全局文本监督 (句子监督, 类似 BLIP) 和单个标签监督 (tag 监督, 类似 CLIP) 来解决精度与性能的问题
- RAM++基于 LLM 对标签进行描述 (“GPT-3.5-turbo”模型), 补充了 tag 的额外知识, 增强了模型对未知标签的泛化能力
- 结合了针对不同步骤的在线/离线设计, 确保了图像文本对齐和图像标签过程的无缝集成。

本文介绍了一种具有鲁棒泛化能力的开放集图像标记模型 RAM++。通过利用多粒度的文本监督, RAM++在各种开放集类别中实现了卓越的性能。综合评估表明, RAM++在大多数方面都超过了现有的 SOTA 模型。鉴于 LLM 在自然语言过程中的革命, RAM++强调, 整合自然语言知识可以显著增强视觉模型。我们希望我们的努力能为其他作品提供一些灵感。

【Summary】RAM++: 输入图片获得图像包含的标签 (Pictures -> Image Tags)

### RAM++项目使用

下载代码 <https://github.com/xinyu1205/recognize-anything>, 并解压进入命令行窗口, cd 切换目录至文件夹, 执行 pip install -e .

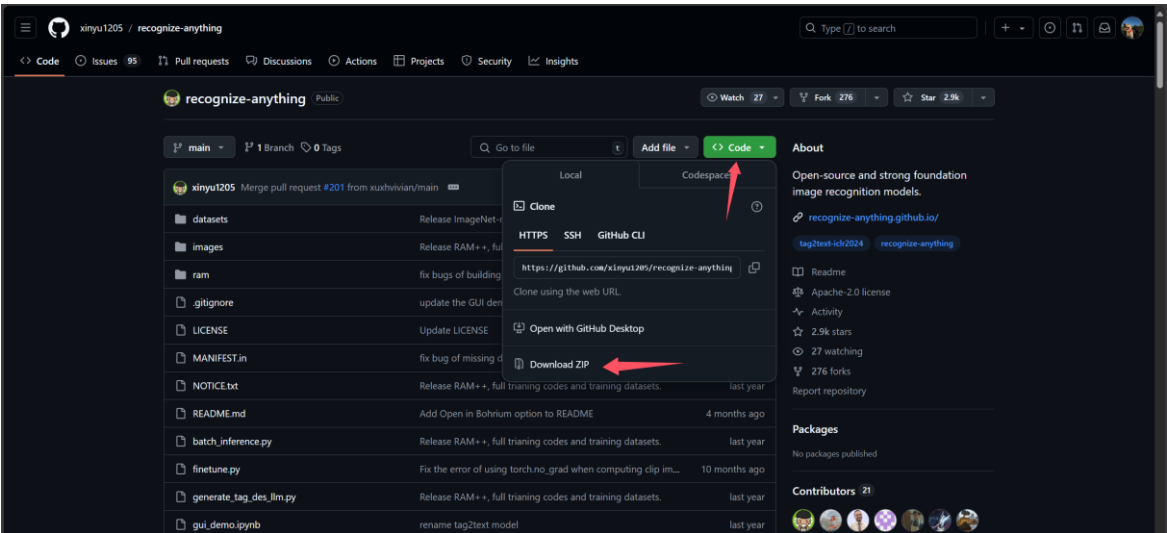


图 2 下载 recognize-anything-main 代码

下载预训练模型: [https://huggingface.co/xinyu1205/recognize-anything-plus-model/blob/main/ram\\_plus\\_swin\\_large\\_14m.pth](https://huggingface.co/xinyu1205/recognize-anything-plus-model/blob/main/ram_plus_swin_large_14m.pth), 并在 Anaconda Prompt 中新建 RAM-Ground-SAM 虚拟环境, 配置 GPU 和相关第三方库 (timm、scipy 等等)

由于网上没有 RAM++的 Demo, 因此自己设计完成, 在 VS Code 打开 recognize-anything-main 项目文件夹, 在项目根目录创建 pretrained 文件, 并放置下载好的 ram\_plus\_swin\_large\_14m.pt h 模型, 加入 infer\_dir.py 文件用以推理。

先使用本地数据集尝试运行 (jpg 格式), 得到结果如下 (Google Colab 也能运行):

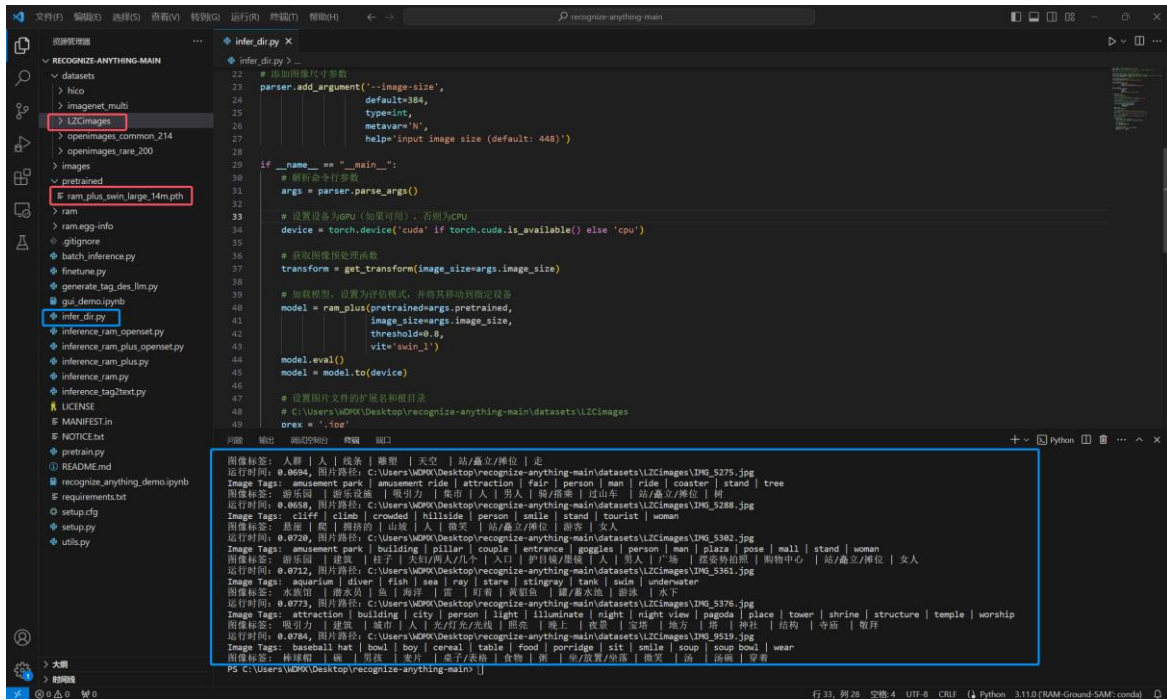


图 3 RAM++运行测试

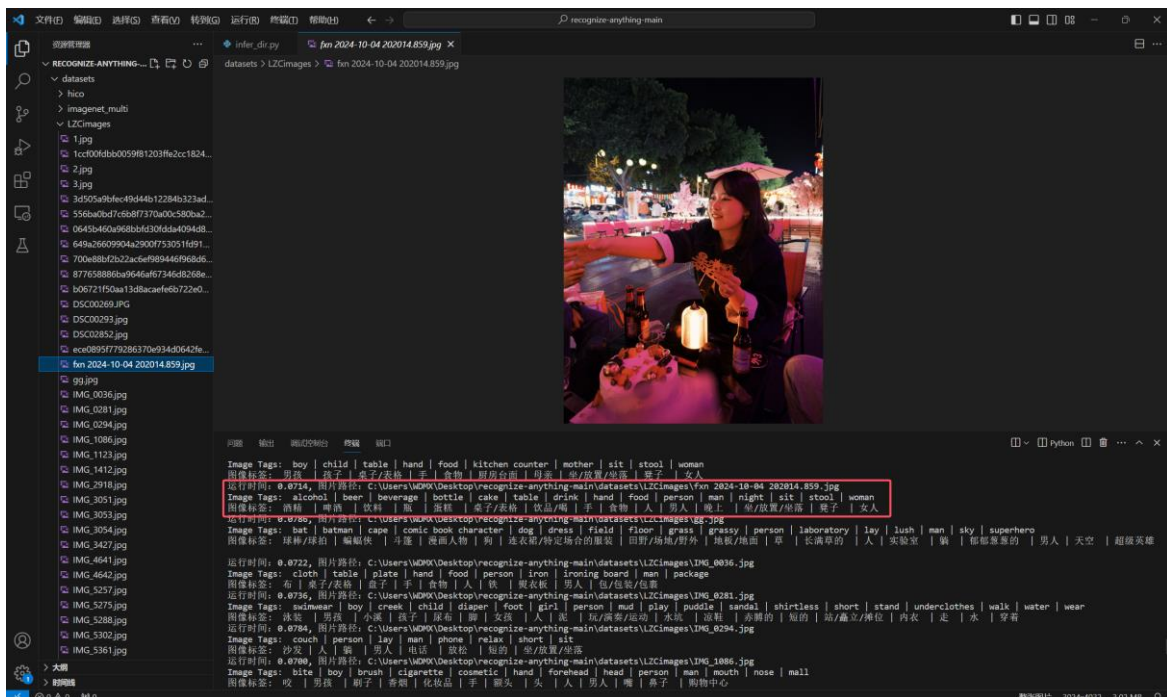


图 4 RAM++结果检查

## 实验内容 2 - 结合 DINO 和 SAM 实现 Grounded-SAM:

了解 Grounded-Segment-Anything 模型时，主要了解全自动标注集成项目(Grounded-SAM)和由文本提示检测图像任意目标(Grounding DINO)，其中 Grounded-SAM 使用 Grounding DINO 作为开放集对象检测器，并与任何分割模型（SAM）相结合。这种整合可以根据任意文本输入检测和分割任何区域。【不过多赘述】

【Summary】Grounded-SAM: 根据标签等输入信息，在图像中生成框和掩码

### 总体实现流程:

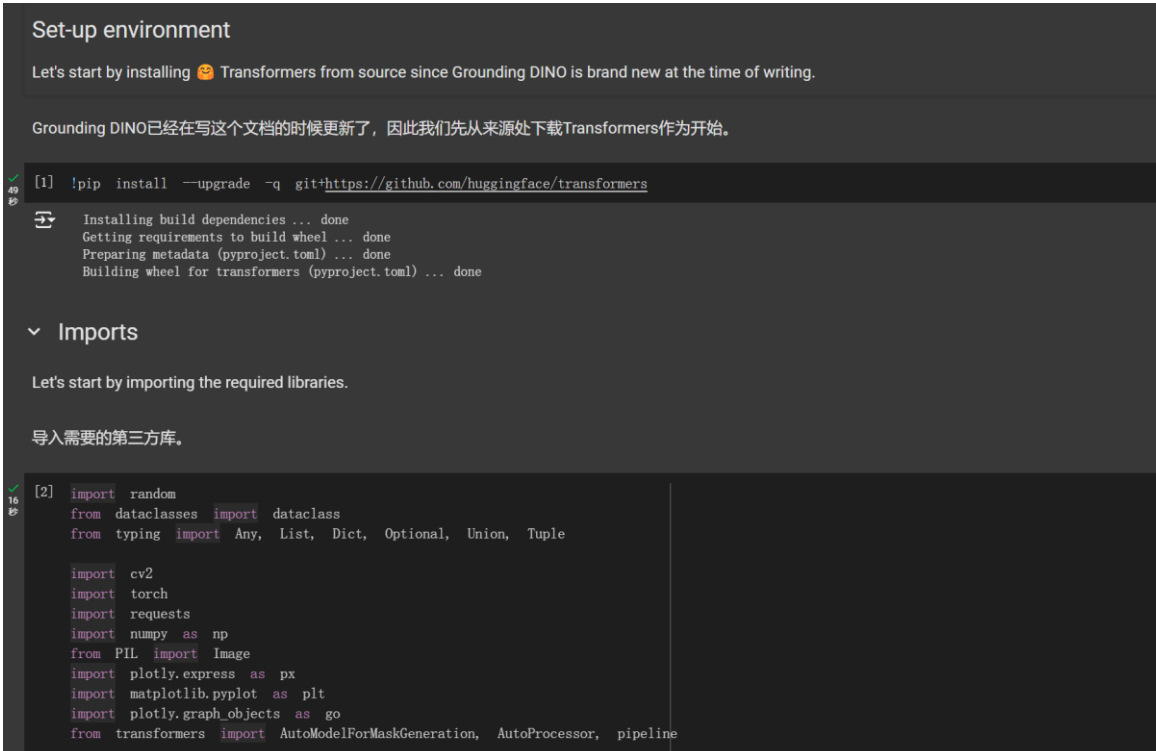
- 1、使用 Grounding DINO 根据文本提示生成边界框。
- 2、提示 Segment-Anything-Model 为其生成相应的分割蒙版。

此处根据 Github 教程: [https://github.com/NielsRogge/Transformers-Tutorials/blob/master/Grounding%20DINO/GroundingDINO\\_with\\_Segment\\_Anything.ipynb](https://github.com/NielsRogge/Transformers-Tutorials/blob/master/Grounding%20DINO/GroundingDINO_with_Segment_Anything.ipynb), 先实现将 Combining Grounding DINO 和 Segment Anything (SAM)结合来生成基于文本的掩码。

由于本地 GPU 环境的 Jupyter Notebook 始终报错且 VPN 连接存在一定网速障碍，在多次尝试后最后选择使用 Google Colab, 运行 ipynb 的结合示例代码并对内容进行解释说明，理解输入和输出，并对最终结合的代码集成进行思考和准备。

【在本案例中，我们将结合 2 个非常酷的模型--[Grounding DINO](#) 和 [SAM](#)，将使用 Grounding DINO 根据文本提示（后续换为 RAM++）生成边界框，再提示 SAM 为其生成相应的分割蒙版。】

我们首先构建环境和导入第三方库，见图 5 所示。



```
Set-up environment

Let's start by installing 🧡 Transformers from source since Grounding DINO is brand new at the time of writing.

Grounding DINO已经在写这个文档的时候更新了，因此我们先从来源处下载Transformers作为开始。

[1] !pip install --upgrade -q git+https://github.com/huggingface/transformers

Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
Building wheel for transformers (pyproject.toml) ... done

Imports

Let's start by importing the required libraries.

导入需要的第三方库。

[2] import random
    from dataclasses import dataclass
    from typing import Any, List, Dict, Optional, Union, Tuple

    import cv2
    import torch
    import requests
    import numpy as np
    from PIL import Image
    import plotly.express as px
    import matplotlib.pyplot as plt
    import plotly.graph_objects as go
    from transformers import AutoModelForMaskGeneration, AutoProcessor, pipeline
```

图 5 环境准备



设计一个专用的 Python 数据类——存储 Grounding DINO 的检测结果，并编写多个工具类。

#### Result Utils

We'll store the detection results of Grounding DINO in a dedicated Python dataclass.

我们在一个专用的Python数据类中存储Grounding DINO的检测结果。

```
from dataclasses import dataclass
from typing import List, Optional, Dict
import numpy as np

@dataclass # 使用 @dataclass 装饰器定义一个名为 BoundingBox 的数据类
class BoundingBox:
    # 定义四个整数类型的字段，分别表示边界框的左上角和右下角的坐标
    xmin: int
    ymin: int
    xmax: int
    ymax: int

    # 定义一个名为 xyxy 的属性方法，返回一个包含边界框坐标的列表
    @property
    def xyxy(self) -> List[float]:
        # 返回一个包含 xmin, ymin, xmax, ymax 的列表，类型为 float
        return [self.xmin, self.ymin, self.xmax, self.ymax]

@dataclass # 使用 @dataclass 装饰器定义一个名为 DetectionResult 的数据类
class DetectionResult:
    # 定义三个字段: score (浮点数), label (字符串), box (BoundingBox 对象)
    score: float
    label: str
    box: BoundingBox
    # 定义一个可选字段 mask, 默认值为 None
    mask: Optional[np.array] = None

    # 定义一个类方法 from_dict, 用于从字典创建 DetectionResult 实例
    @classmethod
    def from_dict(cls, detection_dict: Dict) -> 'DetectionResult':
        # 从字典中提取 score 和 label
        score = detection_dict['score']
        label = detection_dict['label']
        # 从字典中提取 box 信息, 并创建 BoundingBox 实例
        box = BoundingBox(
            xmin=detection_dict['box']['xmin'],
            ymin=detection_dict['box']['ymin'],
            xmax=detection_dict['box']['xmax'],
            ymax=detection_dict['box']['ymax']
        )
        # 返回一个新的 DetectionResult 实例
        return cls(score=score, label=label, box=box)
```

图 6 定义检测结果存储数据类

#### Plot Utils

Below, some utility functions are defined as we'll draw the detection results of Grounding DINO on top of the image.

定义一些工具函数用以在图像的顶部写下Grounding DINO的检测结果。

```
from typing import Union, List, Optional, Dict
from PIL import Image
import numpy as np
import cv2
import matplotlib.pyplot as plt

# 定义一个函数 annotate, 用于在图像上标注检测结果
def annotate(image: Union[Image, np.ndarray], detection_results: List[DetectionResult]) -> np.ndarray:
    # 如果输入图像是 PIL Image 类型, 将其转换为 OpenCV 格式 (numpy 数组)
    image_cv2 = np.array(image) if isinstance(image, Image.Image) else image
    # 将图像从 RGB 格式转换为 BGR 格式 (OpenCV 默认格式)
    image_cv2 = cv2.cvtColor(image_cv2, cv2.COLOR_RGB2BGR)

    # 遍历所有的检测结果, 并在图像上添加边界框和掩码
    for detection in detection_results:
        label = detection.label # 获取检测结果的标签
        score = detection.score # 获取检测结果的得分
        box = detection.box # 获取检测结果的边界框
        mask = detection.mask # 获取检测结果的掩码 (如果有)

        # 为每个检测结果随机生成一个颜色
        color = np.random.randint(0, 256, size=3)
        # 在图像上绘制边界框
        cv2.rectangle(image_cv2, (box.xmin, box.ymin), (box.xmax, box.ymax), color.tolist(), 2)
        # 在图像上添加标签和得分文本
        cv2.putText(image_cv2, f'{label}: {score:.2f}', (box.xmin, box.ymin - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color.tolist(), 2)
        # 如果掩码存在, 则应用掩码
        if mask is not None:
            # 将掩码转换为 uint8 类型
            mask_uint8 = (mask * 255).astype(np.uint8)
            # 查找掩码的轮廓
            contours, _ = cv2.findContours(mask_uint8, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
            # 在图像上绘制掩码的轮廓
            cv2.drawContours(image_cv2, contours, -1, color.tolist(), 2)

    # 将图像从 BGR 格式转换回 RGB 格式, 并返回
    return cv2.cvtColor(image_cv2, cv2.COLOR_BGR2RGB)
```

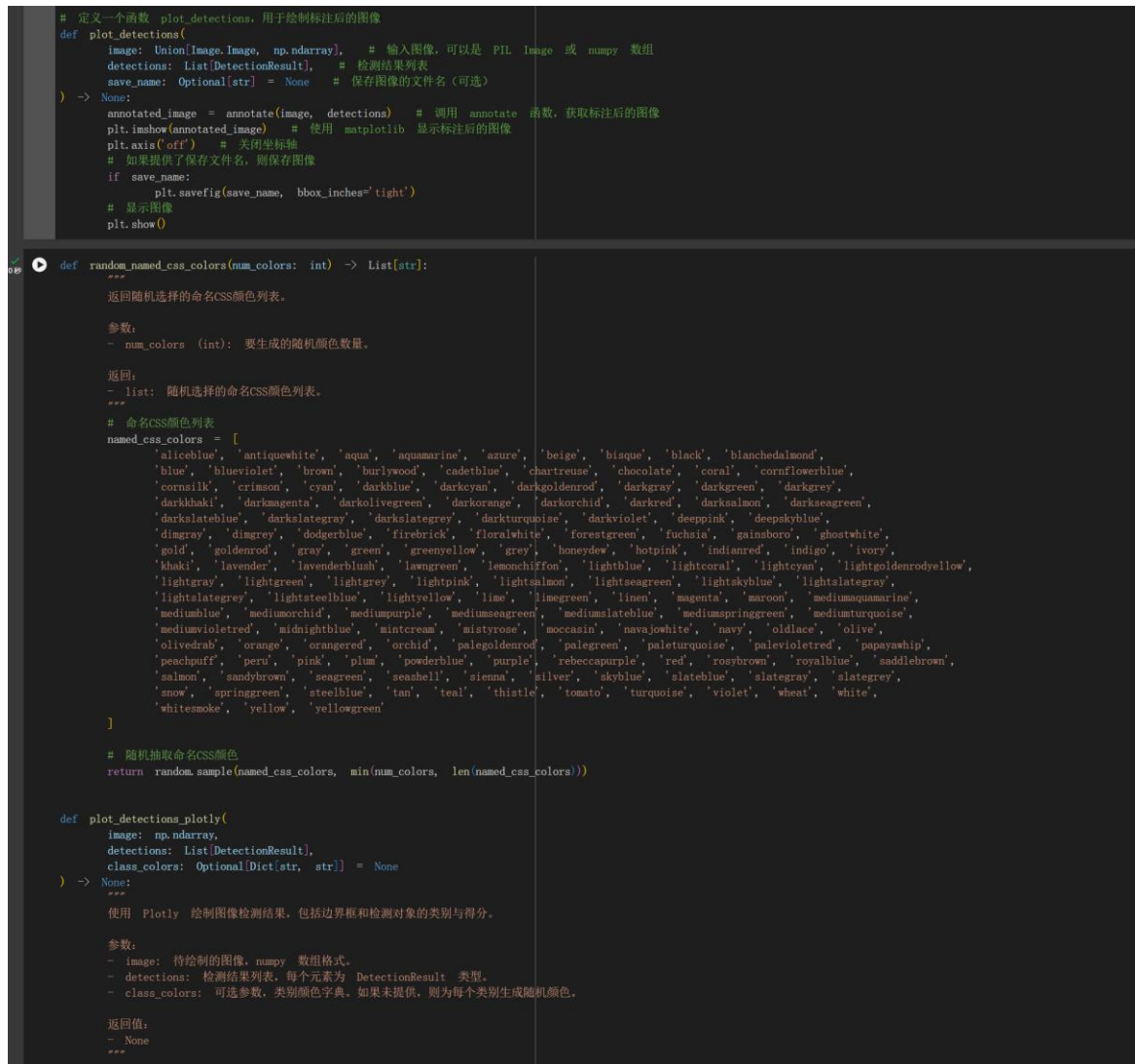
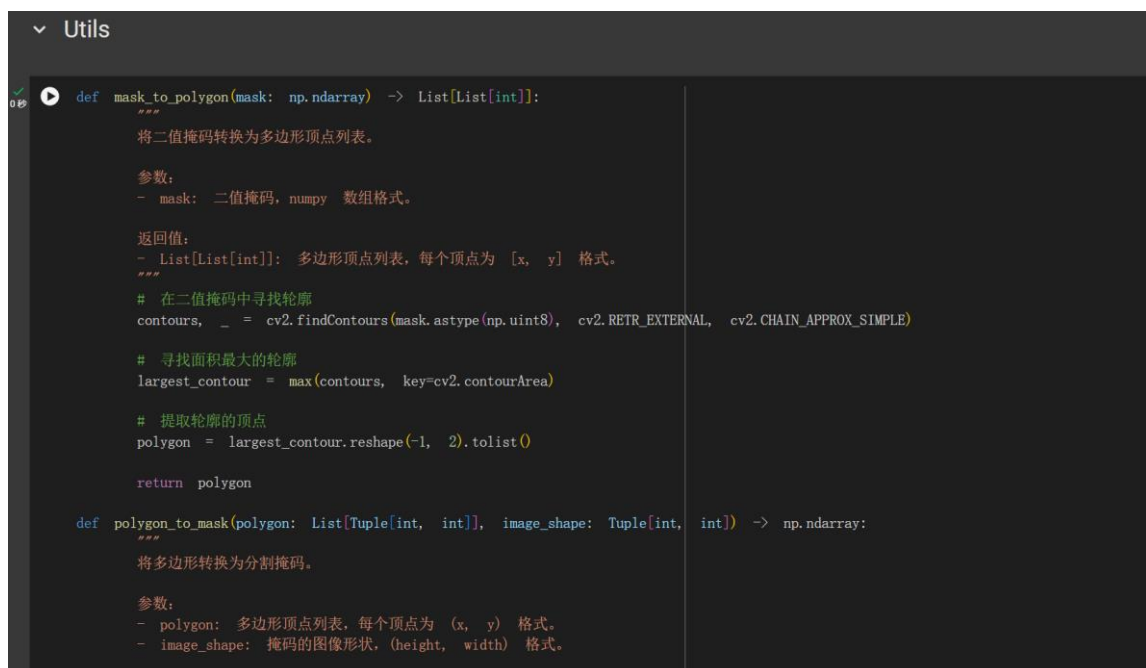


图 7 定义工具函数类

定义用于转化、加载、提取和优化等用途的工具函数。



```

    返回值:
    - np.ndarray: 填充了多边形的分割掩码。
    """
    # 创建一个空掩码
    mask = np.zeros(image_shape, dtype=np.uint8)

    # 将多边形顶点转换为数组
    pts = np.array(polygon, dtype=np.int32)

    # 用白色 (255) 填充多边形
    cv2.fillPoly(mask, [pts], color=(255,))

    return mask

def load_image(image_str: str) -> Image.Image:
    """
    加载图像。

    参数:
    - image_str: 图像的路径或URL。

    返回值:
    - Image.Image: 加载的图像。
    """
    if image_str.startswith("http"): # 如果是URL, 则通过HTTP请求获取图像
        image = Image.open(requests.get(image_str, stream=True).raw).convert("RGB")
    else: # 否则直接从文件路径加载图像
        image = Image.open(image_str).convert("RGB")

    return image

def get_boxes(results: DetectionResult) -> List[List[List[float]]]:
    """
    从检测结果中提取边界框。

    参数:
    - results: 检测结果。

    返回值:
    - List[List[List[float]]]: 边界框列表, 每个边界框为 [x1, y1, x2, y2] 格式。
    """
    boxes = []
    for result in results: # 遍历每个检测结果
        xyxy = result.box.xyxy # 提取边界框坐标
        boxes.append(xyxy)

    return [boxes] # 返回边界框列表

def refine_masks(masks: torch.BoolTensor, polygon_refinement: bool = False) -> List[np.ndarray]:
    """
    对掩码进行优化处理, 可选择是否进行多边形细化。

    参数:
    - masks: 原始掩码, torch.BoolTensor 格式。
    - polygon_refinement: 是否进行多边形细化, 默认为 False。

    返回值:
    - List[np.ndarray]: 优化后的掩码列表。
    """
    masks = masks.cpu().float() # 将掩码转移到CPU并转换为浮点数
    masks = masks.permute(0, 2, 3, 1) # 调整掩码的维度顺序
    masks = masks.mean(axis=-1) # 在最后一个维度上取平均值
    masks = (masks > 0).int() # 二值化处理
    masks = masks.numpy().astype(np.uint8) # 转换为numpy数组并指定数据类型
    masks = list(masks) # 将掩码转换为列表

    if polygon_refinement: # 如果需要进行多边形细化
        for idx, mask in enumerate(masks): # 遍历每个掩码
            shape = mask.shape # 获取掩码的形状
            polygon = mask_to_polygon(mask) # 将掩码转换为多边形
            mask = polygon_to_mask(polygon, shape) # 将多边形转换回掩码
            masks[idx] = mask # 更新掩码列表

    return masks # 返回优化后的掩码列表

```

图 8 定义多用途工具函数

现在是时候定义 Grounded SAM 方法了！这个方法非常简单：

- 1、用 Grounding DINO 去检测含一套文本（后用 RAM++给出）的图片，输出是一套的边界框。
- 2、根据边界框调用 Segment Anything（SAM），对模型输出每个图片的切割掩码。

```

def detect(
    image: Image.Image,
    labels: List[str],
    threshold: float = 0.3,
    detector_id: Optional[str] = None
) -> List[Dict[str, Any]]:
    """
    使用 Grounding DINO 以零样本的方式在图像中检测一组标签。

    参数:
    - image: 待检测的图像, PIL Image 对象。
    - labels: 需要检测的标签列表。
    - threshold: 检测阈值, 默认为 0.3。
    - detector_id: 检测模型的标识符, 如果未指定, 则使用默认模型。

    返回值:
    - List[Dict[str, Any]]: 检测结果列表, 每个结果为一个字典。
    """
    device = "cuda" if torch.cuda.is_available() else "cpu" # 判断是否有可用的CUDA设备
    detector_id = detector_id if detector_id is not None else "IDEA-Research/grounding-dino-tiny" # 如果未指定detector_id, 则使用默认模型
    object_detector = pipeline(model=detector_id, task="zero-shot-object-detection", device=device) # 初始化检测管道

    # 确保所有标签以点号结尾
    labels = [label if label.endswith(".") else label+"." for label in labels]

    # 使用检测管道进行检测
    results = object_detector(image, candidate_labels=labels, threshold=threshold)
    # 将检测结果转换为 DetectionResult 对象列表
    results = [DetectionResult.from_dict(result) for result in results]

    return results # 返回检测结果列表

def segment(
    image: Image.Image,
    detection_results: List[Dict[str, Any]],
    polygon_refinement: bool = False,
    segmenter_id: Optional[str] = None
) -> List[DetectionResult]:
    """
    使用 Segment Anything (SAM) 模型根据图像和一组边界框生成掩码。

    参数:
    - image: 待分割的图像, PIL Image 对象。
    - detection_results: 检测结果列表。
    - polygon_refinement: 是否进行多边形细化, 默认为 False。
    - segmenter_id: 分割模型的标识符, 如果未指定, 则使用默认模型。

    返回值:
    - List[DetectionResult]: 分割结果列表, 每个结果包含掩码信息。
    """
    device = "cuda" if torch.cuda.is_available() else "cpu" # 判断是否有可用的CUDA设备
    segmenter_id = segmenter_id if segmenter_id is not None else "facebook/sam-vit-base" # 如果未指定segmenter_id, 则使用默认模型

    # 初始化分割模型和处理器
    segmentator = AutoModelForMaskGeneration.from_pretrained(segmenter_id).to(device)
    processor = AutoProcessor.from_pretrained(segmenter_id)

    # 从检测结果中提取边界框
    boxes = get_boxes(detection_results)
    # 将图像和边界框输入处理器, 获取模型输入
    inputs = processor(image=image, input_boxes=boxes, return_tensors="pt").to(device)

    # 运行分割模型
    outputs = segmentator(**inputs)
    # 后处理分割结果, 获取掩码
    masks = processor.post_process_masks(
        masks=outputs.pred_masks,
        original_sizes=inputs.original_sizes,
        reshaped_input_sizes=inputs.reshaped_input_sizes
    )[0]

    # 对掩码进行细化处理
    masks = refine_masks(masks, polygon_refinement)

    # 将掩码信息添加到检测结果中
    for detection_result, mask in zip(detection_results, masks):
        detection_result.mask = mask

    return detection_results # 返回包含掩码信息的检测结果列表

def grounded_segmentation(
    image: Union[Image.Image, str],
    labels: List[str],
    threshold: float = 0.3,
    polygon_refinement: bool = False,
    detector_id: Optional[str] = None,
    segmenter_id: Optional[str] = None
) -> Tuple[np.ndarray, List[DetectionResult]]:
    """
    执行基于标签的图像分割。

    参数:
    - image: 待分割的图像, 可以是 PIL Image 对象或图像路径。
    - labels: 需要检测的标签列表。
    - threshold: 检测阈值, 默认为 0.3。
    - polygon_refinement: 是否进行多边形细化, 默认为 False。
    - detector_id: 检测模型的标识符。
    - segmenter_id: 分割模型的标识符。

    返回值:
    - Tuple[np.ndarray, List[DetectionResult]]: 包含原始图像和分割结果的元组。
    """
    if isinstance(image, str): # 如果图像是路径字符串, 则加载图像
        image = load_image(image)

    # 执行检测
    detections = detect(image, labels, threshold, detector_id)
    # 执行分割
    detections = segment(image, detections, polygon_refinement, segmenter_id)

    # 返回原始图像和分割结果
    return np.array(image), detections

```

图 9 定义 Grounded SAM 方法



在 COCO 数据集的猫的图片上展示 Grounded SAM。

```
在COCO数据集的猫的图片上展示Grounded SAM

[ ] # 定义图像的URL地址
image_url = "http://images.cocodataset.org/val2017/000000039769.jpg"
# 定义需要检测的标签列表, 这里检测 "一只猫" 和 "一个遥控器"
labels = ["a cat.", "a remote control."]
# 设置检测阈值, 即检测结果的置信度门限
threshold = 0.3

# 指定检测模型的标识符, 这里使用IDEA-Research提供的Grounding DINO tiny模型
detector_id = "IDEA-Research/grounding-dino-tiny"
# 指定分割模型的标识符, 这里使用facebook提供的SAM-vit-base模型
segmenter_id = "facebook/sam-vit-base"

[ ] # 调用 grounded_segmentation 函数执行基于标签的图像分割
# 该函数将下载图像、检测标签中的对象, 并生成对应的分割掩码
image_array, detections = grounded_segmentation(
    # 指定待分割的图像URL
    image=image_url,
    # 指定需要检测的标签列表
    labels=labels,
    # 设置检测阈值, 即检测结果的置信度门限
    threshold=threshold,
    # 设置是否进行多边形细化, True表示进行细化, 可以提高掩码质量
    polygon_refinement=True,
    # 指定检测模型的标识符, 这里使用IDEA-Research提供的Grounding DINO tiny模型
    detector_id=detector_id,
    # 指定分割模型的标识符, 这里使用facebook提供的SAM-vit-base模型
    segmenter_id=segmenter_id
)

Device set to use cuda
```

图 10 COCO 数据集结果展示

可视化 Grounded-SAM 模型绘制结果。

```
Let's visualize the results:

# 调用 plot_detections 函数来绘制检测结果
# 该函数将在图像上绘制边界框、多边形掩码, 并保存到指定的文件中

# 函数参数解释:
# image_array: 待绘制的图像, numpy 数组格式。
# detections: 检测结果列表, 包含每个检测对象的详细信息。
# "cute_cats.png": 绘制结果的保存文件名。

plot_detections(
    # 提供待绘制的图像数组, 这是由之前的 grounded_segmentation 函数返回的
    image_array,
    # 提供检测结果列表, 这也是由之前的 grounded_segmentation 函数返回的
    detections,
    # 指定绘制结果的保存文件名
    "cute_cats.png"
)

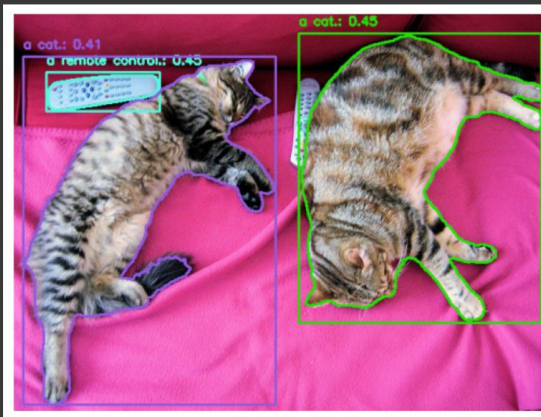

```

图 11 可视化模型绘制结果

交互式图表可视化结果展示。

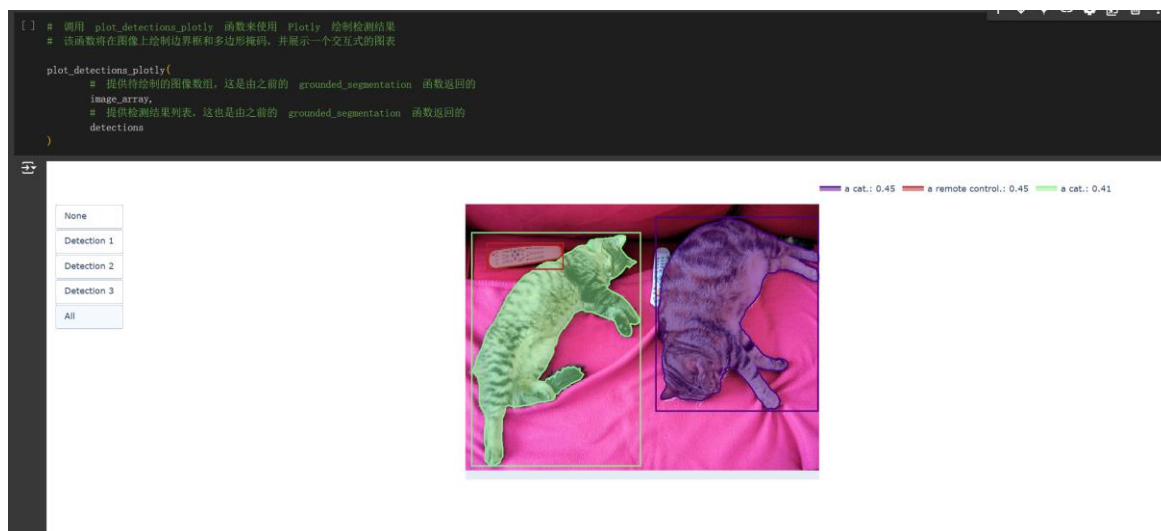


图 12 交互式图表可视化结果展示

### 实验内容 3 - 集成 RAM++ 和 Grounded-SAM:

由于 Google Colab 对 GPU 资源的限制，需要将模型在本地集成，对于 Google Colab 中的“!pip install --upgrade -q git+https://github.com/huggingface/transformers”这行代码只能在 Google Colab 里面运行，在本机运行会因为网络报错。因此选择在 GitHub 中直接下载 transformers 文件夹，再在本机终端打开，cd 修改文件路径，然后 pip install -e .，通过文件夹中的 setup 文件安装，完成上述操作后即可对 Grounded-SAM 的运行。

集成 RAM++ 和 Grounded-SAM 的思路是：先运行 RAM++ 对图片数据集运行生成标签，把所有图片的标签结果保存到硬盘上，体现出图片路径和标签的对应关系；然后释放本机 GPU 资源，然后对标签做一些数据格式的处理，再传递给 Grounded-SAM 运行结果（先使用本地数据集测试后，再使用 OCID 数据集评估检验）。

受本机专用 GPU 内存（8.0 GB）和共享 GPU 内存（7.8 GB）的限制，尝试运行之后发现对于 RAM++ 生成标签数在 10 个左右能运行 Grounded-SAM 且不会撑满 GPU 导致运行失败，因此对本地测试数据集筛选能够使 RAM++ 生成标签小于等于 10 个的图片，并完成运行集成的基于 RAM++ 的 Grounded-SAM 模型，以下为一些运行成功的案例。

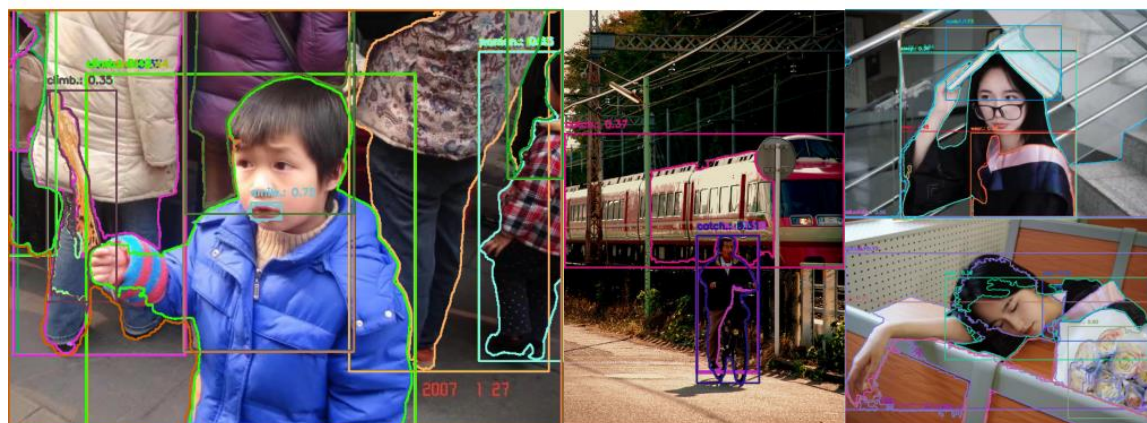


图 13 集成 RAM++ 和 Grounded-SAM 测试结果

OCID 数据集包含 96 个完全构建的杂乱场景。每个场景都是一系列带标签的点云，这些点云是通过逐步构建一个越来越杂乱的场景并一个接一个地添加对象来创建的。序列中的第一项不包含任何对象，第二项包含 **object**，直到添加的对象的最终计数。

该数据集使用了 89 个不同的对象，这些对象是从[自主机器人室内数据集](#)（ARID）[1] 类和[YCB 对象和模型集](#)（YCB）[2] 数据集对象中选择的代表。

ARID20 子集包含的场景最多包括 ARID 中的 20 个对象。ARID10 和 YCB10 子集包括杂乱的场景，分别包含来自 ARID 和 YCB 对象的多达 10 个对象。每个子集中的场景一次仅由来自一组对象组成，以保持数据集之间的分离。场景变化包括不同的地板（塑料、木材、地毯）和桌子纹理（木材、橙色条纹板、绿色图案板）。完整的数据集提供 2346 个标记的点云。

OCID 子集的结构允许单独评估特定的实际因素，详细描述如下

#### **ARID20-structure**

- 位置：地板、桌子
- 视图：底部、顶部
- 场景：序列 ID
- 自由：明确分离（对象 1-9 按相应顺序排列）
- 触摸：物理接触（对象 10-16 按相应顺序排列）
- 堆叠：彼此重叠（对象 17-20 按相应顺序排列）

#### **ARID10-structure**

- 位置：地板、桌子
- 视图：底部、顶部
- 盒子：具有锋利边缘的物体（如谷物盒）
- 弯曲：具有光滑曲面的物体（如球）
- 混合：来自盒子和弯曲的物体
- 水果：水果和蔬菜
- 非水果：没有水果的混合物体
- 场景：序列 ID

#### **YCB10-structure**

- 位置：地板、桌子
- 视图：底部、顶部
- 盒子：具有锋利边缘的物体（例如谷物盒）
- 弯曲：具有光滑曲面的物体（例如球）
- 混合：来自盒子和弯曲的物体
- 场景：序列 ID

用于实例识别/分类的 OCID 数据：对于 ARID10 和 ARID20，有可用于对象识别和分类任务的额外数据。它包含从 OCID 数据集中提取的语义注释 RGB 和深度图像裁剪。

结构如下：

- **type:** depth, RGB
- **类名:** eg. 香蕉、kleenex .....
- **类实例:** 例如 banana\_1、banana\_2、kleenex\_1 kleenex\_2,...

数据由 Mohammad Reza Loghmani 提供。



由于本地 GPU 资源限制和目标检测评估的需求，决定使用 ARID10-structure 数据集，并从中随机抽取数十张图片（同一 seq 里面只取一张，从而保证每张图片的场景不同）来完成基于 RAM++ 的 Grounded-SAM 模型检测，设置检测结果的置信度门限 threshold 为 0.3，结果不显著的再修改为 0.2，指定检测模型的标识符为 IDEA-Research 提供的 Grounding DINO tiny 模型，指定分割模型的标识符为 facebook 提供的 SAM-vit-base 模型（此处标签数量不受限制）。

共展示随机取得的 6 张图像数据以及运行可视化结果分别如下：

OCID-dataset\ARID10\table\bottom\curved\seq08\rgb\result\_2018-08-23-11-06-53



图 14-1 原图

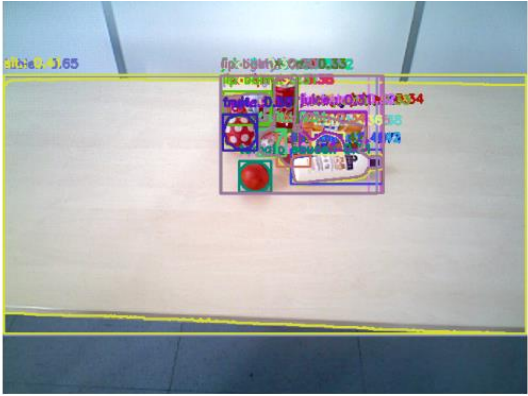


图 14-2 结果图

OCID-dataset\ARID10\table\top\non-fruits\seq12\rgb\result\_2018-08-23-11-39-36



图 15-1 原图

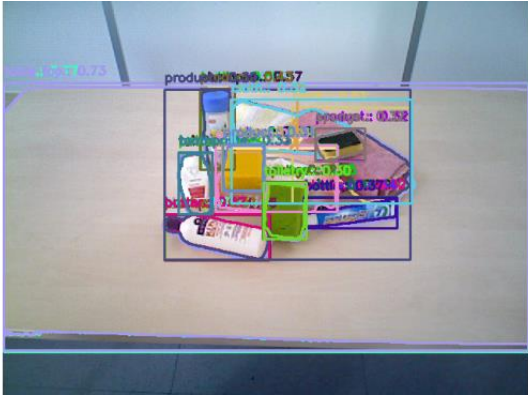


图 15-2 结果图

OCID-dataset\ARID10\table\bottom\mixed\seq13\rgb\result\_2018-08-23-12-06-01



图 16-1 原图

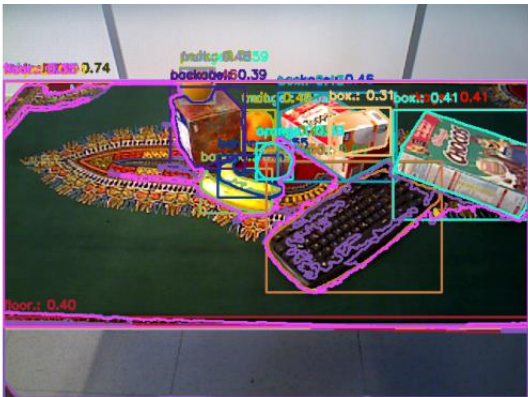
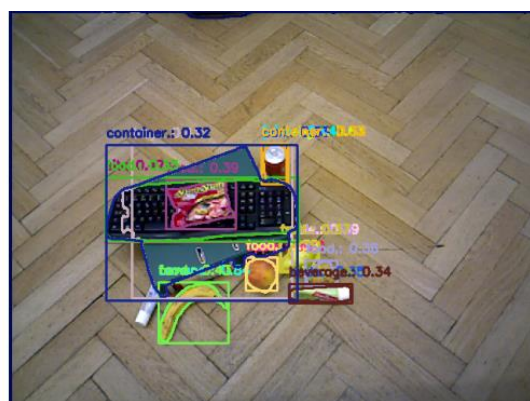
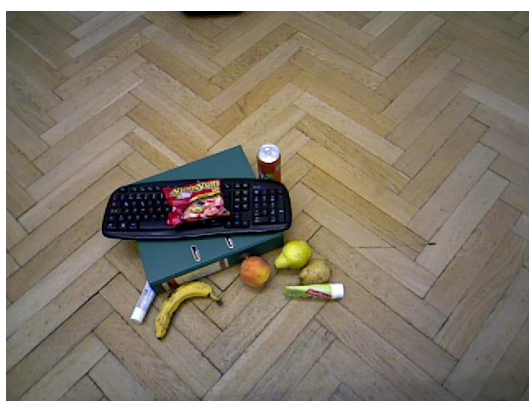


图 16-2 结果图

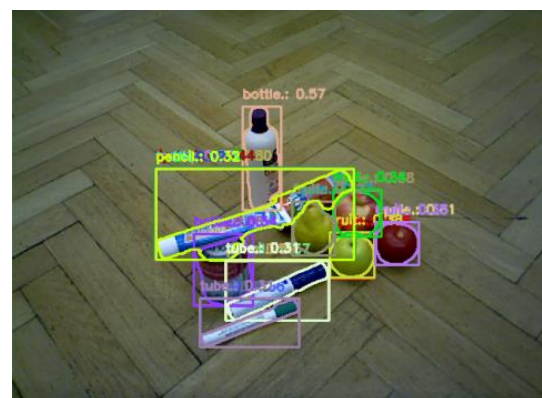
OCID-dataset\ARID10\floor\top\non-fruits\seq40\rgb\result\_2018-08-23-18-06-05



OCID-dataset\ARID10\floor\top\mixed\seq01\rgb\result\_2018-08-24-16-59-06



OCID-dataset\ARID10\floor\bottom\curved\seq05\rgb\result\_2018-08-24-17-17-53



由于是使用 OCID 数据集上进行测试评估,虽然 `table\bottom` 和 `floor\top` 的背景较为杂乱,但也依旧较为不错的完成了 Seg 任务,接下来通过模型评估代码对比检测识别结果和数据集的 `groundtruth`, 评估检测识别性能。

将目标图片数据的真实物体的单独分割图筛选出来，存放在 `real` 文件夹，再将集成模型预测出来的图片存为 `predict` 文件。通过代码将预测掩码和每张真实分割图读入为 `NumPy` 数组，再对于每张真实分割图，计算其与预测掩码的 `IoU` 和平均值，并基于阈值计算精度和召回率。



下载 OCID semantic crops 数据集【从 OCID 数据集的 ARID20 和 ARID10 子集裁剪的对象，其中包含根据对象的实例和类别组织的 RGB 和深度数据】，对展示的 6 张图片查找搜索它们图片包含的所有对象实例，从而完成对 OCID 中部分数据的评估。

由于集成模型和对象实例两种图片的形状不匹配，因此在计算掩码的交集和并集时无法进行元素级别的操作，我们使用 `.convert('L')` 将所有掩码转换为灰度图像，确保它们都是二维的。然后，我们调整每个真实掩码的尺寸以匹配预测掩码的尺寸，保证所有掩码都具有相同的形状，从而可以进行 IoU 计算。

```
import numpy as np
from PIL import Image
import os
import matplotlib.pyplot as plt

def iou(mask_pred, mask_true):
    # 确保掩码是二值的
    mask_pred = (mask_pred > 0.5).astype(bool)
    mask_true = (mask_true > 0.5).astype(bool)

    # 计算交集和并集
    intersection = np.logical_and(mask_pred, mask_true).sum()
    union = np.logical_or(mask_pred, mask_true).sum()

    # 计算 IoU
    if union == 0:
        return 0
    return intersection / union

# 读取预测掩码
mask_pred_path = r'C:\Users\WDMX\Desktop\recognize-anything-main\OCID-evaluate\seg-result_2018-08-23-12-06-01\predict.png'
mask_pred = np.array(Image.open(mask_pred_path).convert('L')) # 转换为灰度图像

# 读取所有真实分割图
true_mask_dir = r'C:\Users\WDMX\Desktop\recognize-anything-main\OCID-evaluate\seg-result_2018-08-23-12-06-01\real'
true_mask_paths = [os.path.join(true_mask_dir, f) for f in os.listdir(true_mask_dir)]
true_masks = [np.array(Image.open(path).convert('L')) for path in true_mask_paths] # 转换为灰度图像

# 调整真实掩码的尺寸以匹配预测掩码
true_masks = [np.array(Image.fromarray(mask).resize(mask_pred.shape[1::-1])) for mask in true_masks]

# 计算每个真实分割图与预测掩码的 IoU
ious = [iou(mask_pred, mask_true) for mask_true in true_masks]

# 计算平均 IoU
mean_iou = np.mean(ious)

# 根据 IoU 阈值确定预测是否正确
iou_threshold = 0.5
correct_predictions = [iou > iou_threshold for iou in ious]

# 计算精度和召回率
precision = sum(correct_predictions) / len(ious)
recall = sum(correct_predictions) / len(true_masks)

# 打印结果
print(f'Mean IoU: {mean_iou}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')

# 绘制预测掩码和真实掩码的对比图
plt.figure(figsize=(10, 5))

# 显示预测掩码
plt.subplot(1, 2, 1)
plt.imshow(mask_pred, cmap='gray')
plt.title('Predicted Mask')
plt.axis('off')

# 显示第一个真实掩码
plt.subplot(1, 2, 2)
plt.imshow(true_masks[0], cmap='gray')
plt.title('True Mask')
plt.axis('off')

plt.show()

[11] ✓ 0.1s
```

图 20 模型评估代码

针对上述展示的 6 张图片展示其检测性能（均抽取第一张实例作为对比图）

OCID-dataset\ARID10\table\bottom\curved\seq08\rgb\result\_2018-08-23-11-06-53  
性能指标: Mean IoU (0.999648939588689) ; Precision (1.0) ; Recall (1.0)

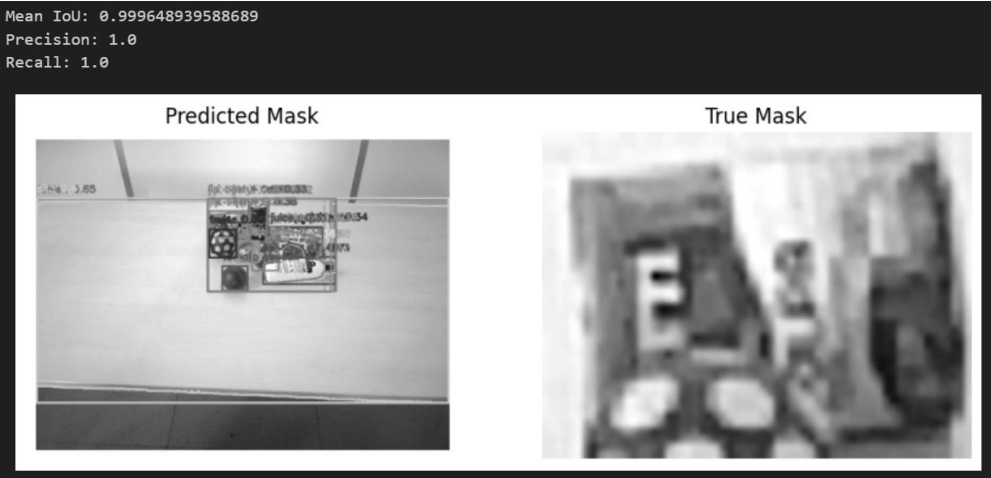


图 21 集成模型对数据文件 1 的性能评估

OCID-dataset\ARID10\table\top\non-fruits\seq12\rgb\result\_2018-08-23-11-39-36  
性能指标: Mean IoU (0.9995292918541131) ; Precision (1.0) ; Recall (1.0)

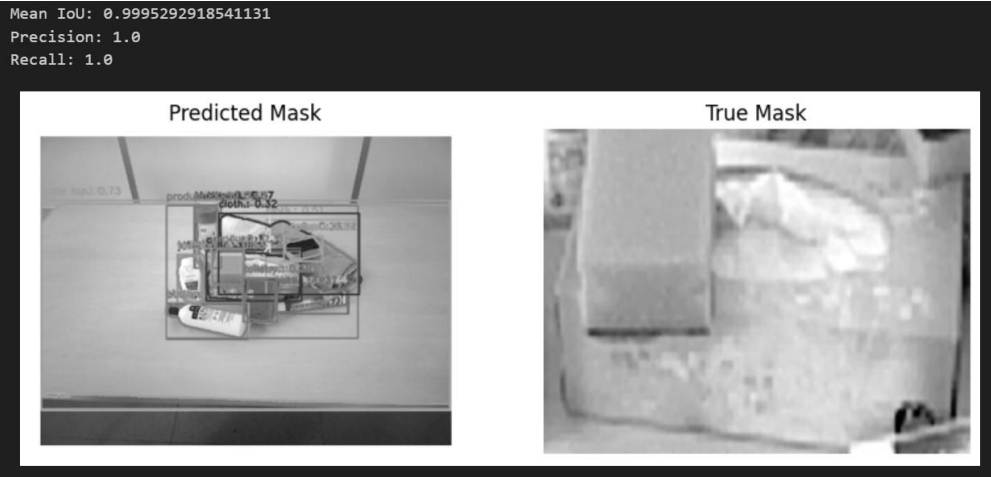


图 22 集成模型对数据文件 2 的性能评估

OCID-dataset\ARID10\table\bottom\mixed\seq13\rgb\result\_2018-08-23-12-06-01  
性能指标: Mean IoU (0.9902119176352364) ; Precision (1.0) ; Recall (1.0)

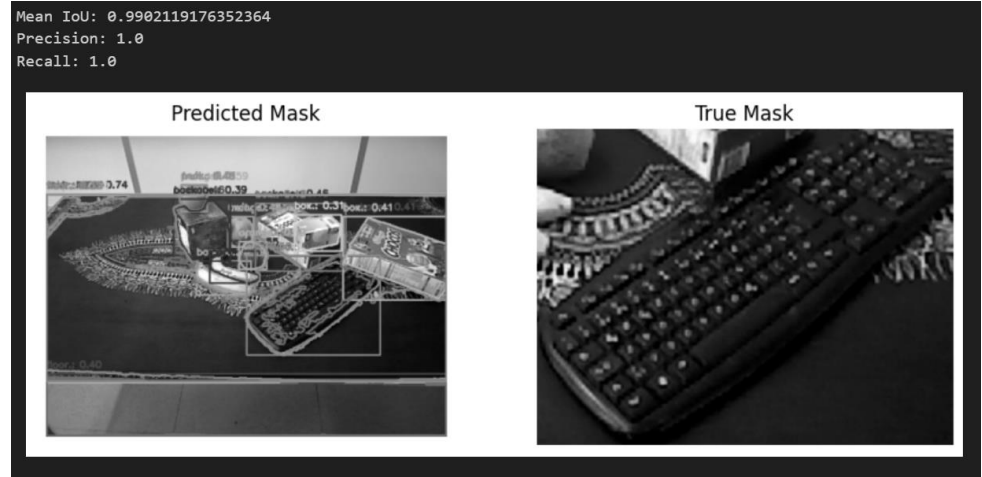


图 23 集成模型对数据文件 3 的性能评估

OCID-dataset\ARID10\floor\top\non-fruits\seq40\rgb\result\_2018-08-23-18-06-05  
性能指标: Mean IoU (0.98586349744265) ; Precision (1.0) ; Recall (1.0)



图 24 集成模型对数据文件 4 的性能评估

OCID-dataset\ARID10\floor\top\mixed\seq01\rgb\result\_2018-08-24-16-59-06  
性能指标: Mean IoU (0.9749060850611163) ; Precision (1.0) ; Recall (1.0)

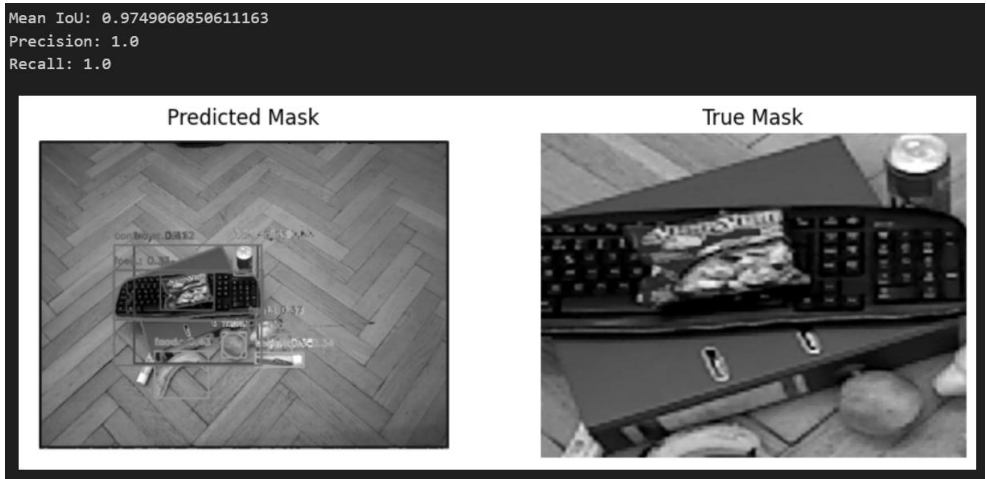


图 25 集成模型对数据文件 5 的性能评估

OCID-dataset\ARID10\floor\bottom\curved\seq05\rgb\result\_2018-08-24-17-17-53  
性能指标: Mean IoU (0.9964648590309599) ; Precision (1.0) ; Recall (1.0)

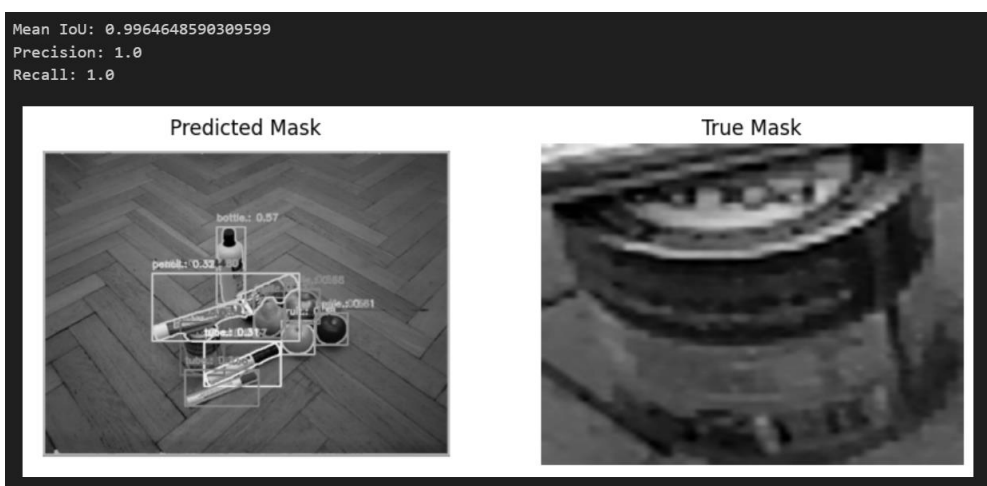


图 25 集成模型对数据文件 6 的性能评估

评估结果的解释说明:

### 1、Mean IoU (平均交并比)

IoU 是一个衡量预测掩码与真实掩码之间重叠程度的指标。它的值范围从 0 到 1，其中 1 表示完美的重叠。此处的 Mean IoU 值均在 0.9749 以上，几乎接近于 1，这表明预测掩码与真实掩码之间有极高的重叠度，几乎完美匹配。

### 2、Precision (精确率):

精确率是衡量预测为正的样本中实际为正确的比例。它反映了模型预测的准确性。此处的 Precision 值均为 1.0 表示所有预测为正的样本都是正确的，没有误报。

### 3、Recall (召回率):

召回率是衡量所有实际为正的样本中被正确预测为正确的比例。它反映了模型捕捉所有正样本的能力。此处的 Recall 值均为 1.0 表示所有实际为正的样本都被正确预测，没有漏报。

综合来看，这些指标的结果表明模型在目标检测任务上几乎达到了完美的性能，既能准确地识别出所有的目标对象，也能确保没有误报和漏报。这在实际应用中是非常理想的结果，但仍然需要在实际应用中进一步验证和调整。