

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

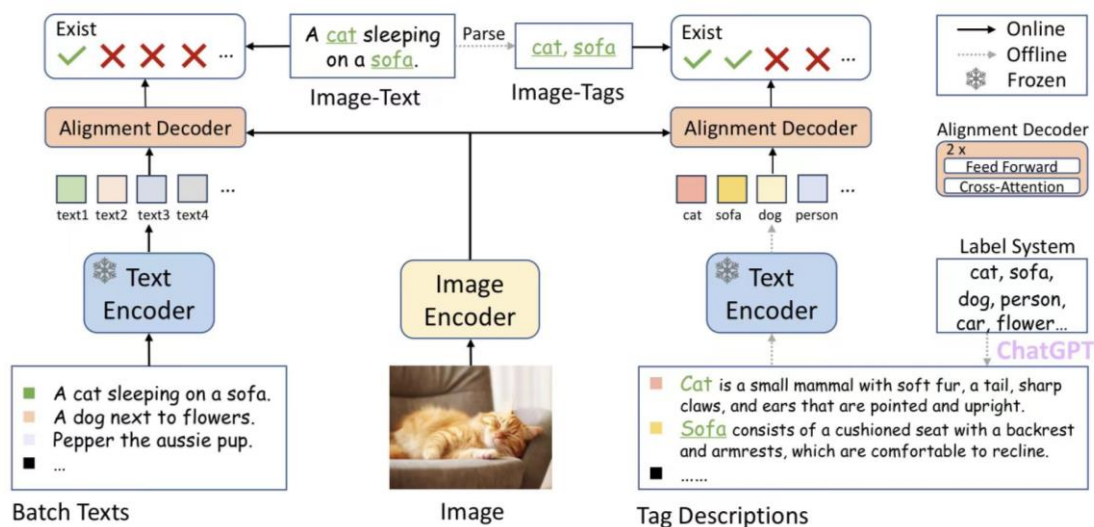


图 1 RAM++训练框架解释说明

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 .

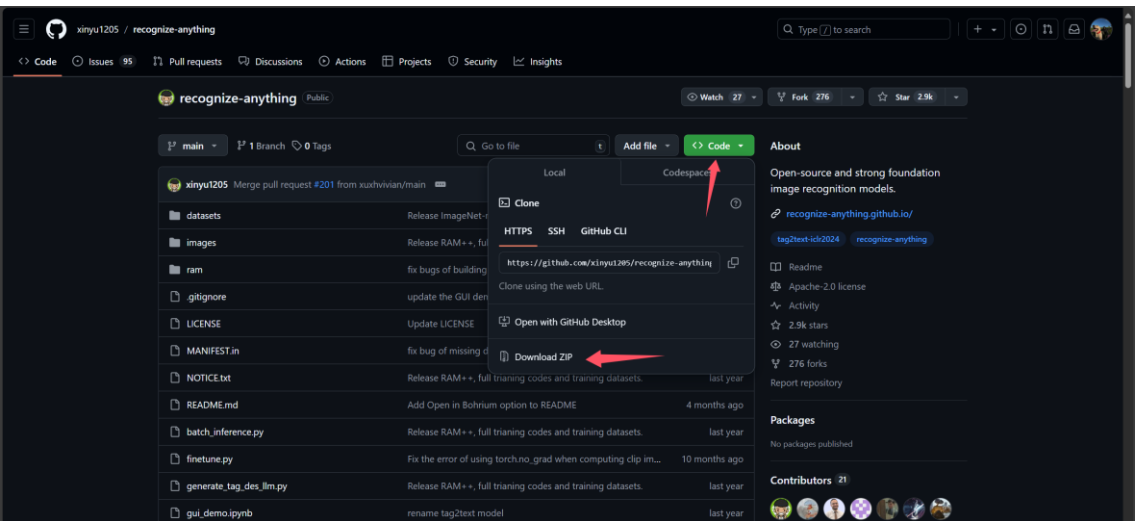


图 2 下载 recognize-anything-main 代码

下载预训练模型：https://huggingface.co/xinyu1205/recognize-anything-plus-model/blob/main/ram_plus_swin_large_14m.pth，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.pth 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）：

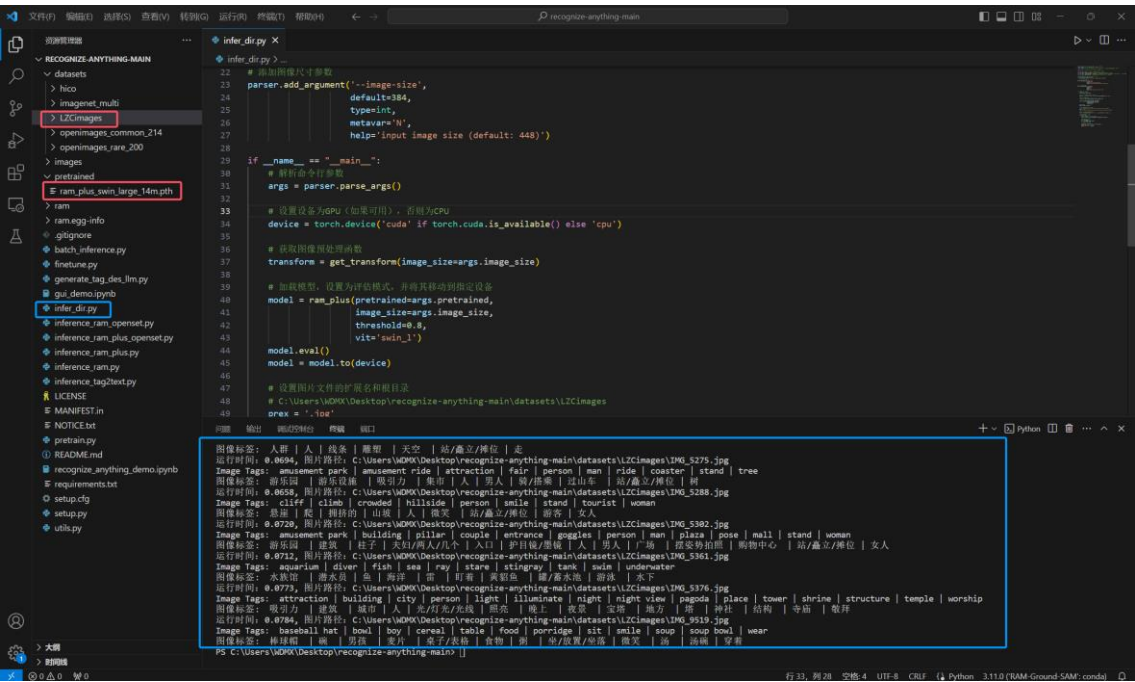


图 3 RAM++运行测试

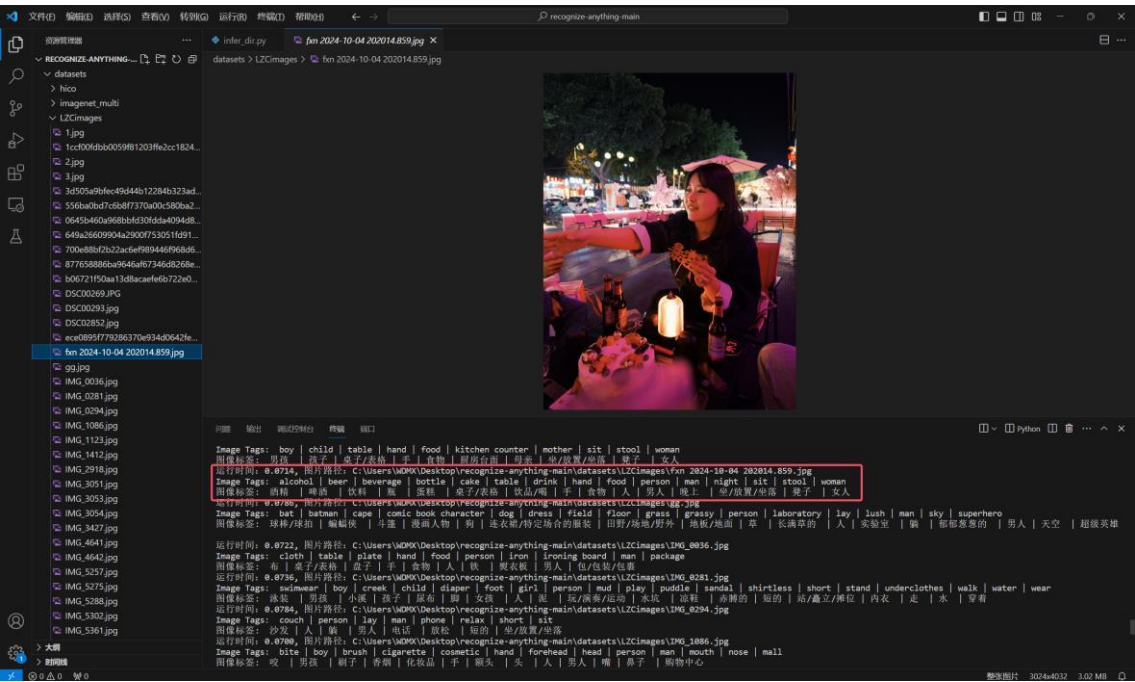


图 4 RAM++结果检查

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:

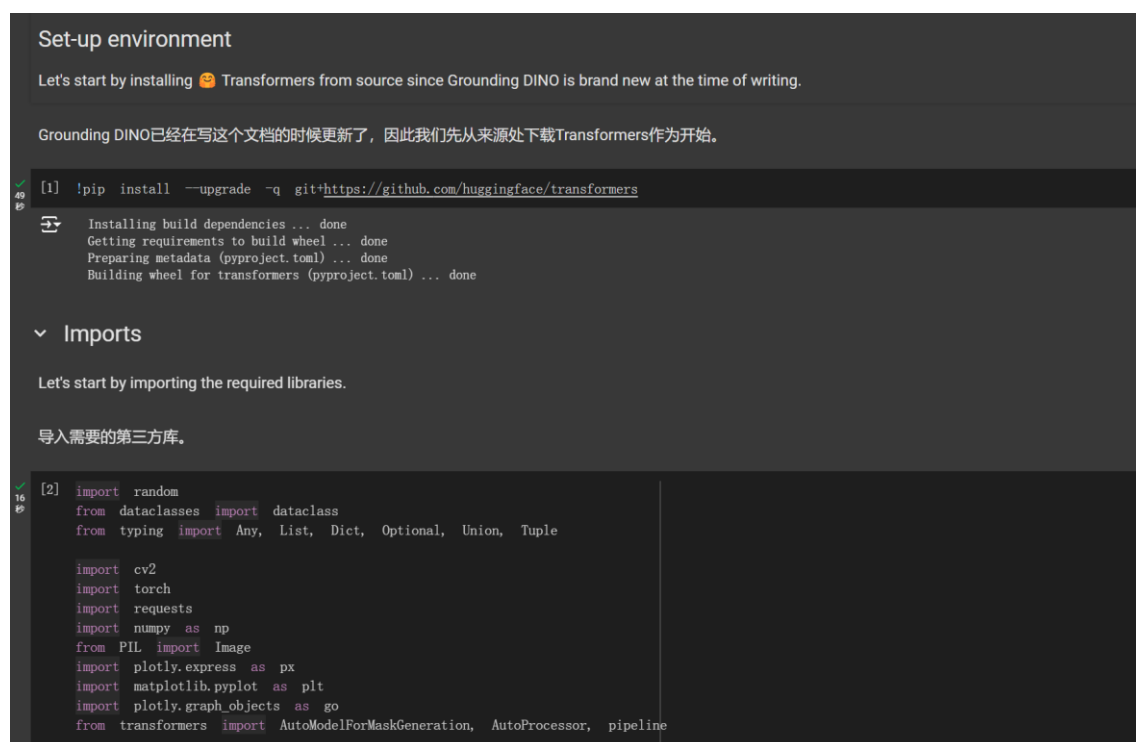
- 1、使用 Grounding DINO 根据文本提示生成边界框。
- 2、提示 Segment-Anything-Model 为其生成相应的分割蒙版。

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%20with%20Segment%20Anything.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.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)
```

```

# 定义一个函数 plot_detections, 用于绘制标注后的图像
def plot_detections(
    image: Union[Image, Image, np.ndarray], # 输入图像, 可以是 PIL Image 或 numpy 数组
    detections: List[DetectionResult], # 检测结果列表
    save_name: Optional[str] = None # 保存图像的文件名 (可选)
) -> None:
    annotated_image = annotate(image, detections) # 调用 annotate 函数, 获取标注后的图像
    plt.imshow(annotated_image) # 使用 matplotlib 显示标注后的图像
    plt.axis('off') # 关闭坐标轴
    # 如果提供了保存文件名, 则保存图像
    if save_name:
        plt.savefig(save_name, bbox_inches='tight')
    # 显示图像
    plt.show()

def random_named_css_colors(num_colors: int) -> List[str]:
    """
    返回随机选择的命名CSS颜色列表。

    参数:
    - num_colors (int): 要生成的随机颜色数量。

    返回:
    - list: 随机选择的命名CSS颜色列表。
    """
    # 命名CSS颜色列表
    named_css_colors = [
        'aliceblue', 'antiquewhite', 'aqua', 'aquamarine', 'azure', 'beige', 'bisque', 'black', 'blanchedalmond',
        'blue', 'blueviolet', 'brown', 'burlywood', 'cadetblue', 'chartreuse', 'chocolate', 'coral', 'cornflowerblue',
        'cornsilk', 'crimson', 'cyan', 'darkblue', 'darkcyan', 'darkgoldenrod', 'darkgray', 'darkgreen', 'darkgrey',
        'darkkhaki', 'darkmagenta', 'darkolivegreen', 'darkorange', 'darkorchid', 'darkred', 'darksalmon', 'darkseagreen',
        'darkslateblue', 'darkslategray', 'darkslategrey', 'darkturquoise', 'darkviolet', 'deeppink', 'deepskyblue',
        'dimgrey', 'dimgrey', 'dodgerblue', 'firebrick', 'floralwhite', 'forestgreen', 'fuchsia', 'gainsboro', 'ghostwhite',
        'gold', 'goldenrod', 'gray', 'green', 'greenyellow', 'grey', 'honeydew', 'hotpink', 'indianred', 'indigo', 'ivory',
        'khaki', 'lavender', 'lavenderblush', 'lawngreen', 'lemonchiffon', 'lightblue', 'lightcoral', 'lightcyan', 'lightgoldenrodyellow',
        'lightgray', 'lightgreen', 'lightgrey', 'lightpink', 'lightsalmon', 'lightseagreen', 'lightskyblue', 'lightslategray',
        'lightslategrey', 'lightsteelblue', 'lightyellow', 'lime', 'limegreen', 'linen', 'magenta', 'maroon', 'mediumaquamarine',
        'mediumblue', 'mediumorchid', 'mediumpurple', 'mediumseagreen', 'mediumslateblue', 'mediumspringgreen', 'mediumturquoise',
        'mediumvioletred', 'midnightblue', 'mintcream', 'mistyrose', 'moccasin', 'navajowhite', 'navy', 'oldlace', 'olive',
        'olivedrab', 'orange', 'orangered', 'orchid', 'palegoldenrod', 'palegreen', 'paleturquoise', 'palevioletred', 'papayawhip',
        'peachpuff', 'peru', 'pink', 'plum', 'powderblue', 'purple', 'rebeccapurple', 'red', 'rosybrown', 'royalblue', 'saddlebrown',
        'salmon', 'sandybrown', 'seagreen', 'seashell', 'sienna', 'silver', 'skyblue', 'slateblue', 'slategray', 'slategrey',
        'snow', 'springgreen', 'steelblue', 'tan', 'teal', 'thistle', 'tomato', 'turquoise', 'violet', 'wheat', 'white',
        'whitesmoke', 'yellow', 'yellowgreen'
    ]

    # 随机抽取命名CSS颜色
    return random.sample(named_css_colors, min(num_colors, len(named_css_colors)))

def plot_detections_plotly(
    image: np.ndarray,
    detections: List[DetectionResult],
    class_colors: Optional[Dict[str, str]] = None
) -> None:
    """
    使用 Plotly 绘制图像检测结果, 包括边界框和检测对象的类别与得分。

    参数:
    - image: 待绘制的图像, numpy 数组格式。
    - detections: 检测结果列表, 每个元素为 DetectionResult 类型。
    - class_colors: 可选参数, 类别颜色字典, 如果未提供, 则为每个类别生成随机颜色。

    返回值:
    - None
    """

```

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。

```

Utlis

def mask_to_polygon(mask: np.ndarray) -> List[List[int]]:
    """
    将二值掩码转换为多边形顶点列表。

    参数:
    - mask: 二值掩码, numpy 数组格式。

    返回值:
    - List[List[int]]: 多边形顶点列表, 每个顶点为 [x, y] 格式。
    """
    # 在二值掩码中寻找轮廓
    contours, _ = cv2.findContours(mask.astype(np.uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # 寻找面积最大的轮廓
    largest_contour = max(contours, key=cv2.contourArea)

    # 提取轮廓的顶点
    polygon = largest_contour.reshape(-1, 2).tolist()

    return polygon

def polygon_to_mask(polygon: List[Tuple[int, int]], image_shape: Tuple[int, int]) -> np.ndarray:
    """
    将多边形转换为分割掩码。

    参数:
    - polygon: 多边形顶点列表, 每个顶点为 (x, y) 格式。
    - image_shape: 掩码的图像形状, (height, width) 格式。
    """

```

```

    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
        xxyy = result.box.xxyy # 提取边界框坐标
        boxes.append(xxyy)

    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(images=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"
)

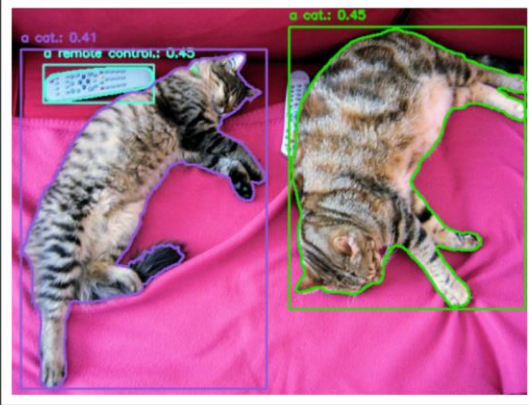

```

图 11 可视化模型绘制结果

交互式图表可视化结果展示。

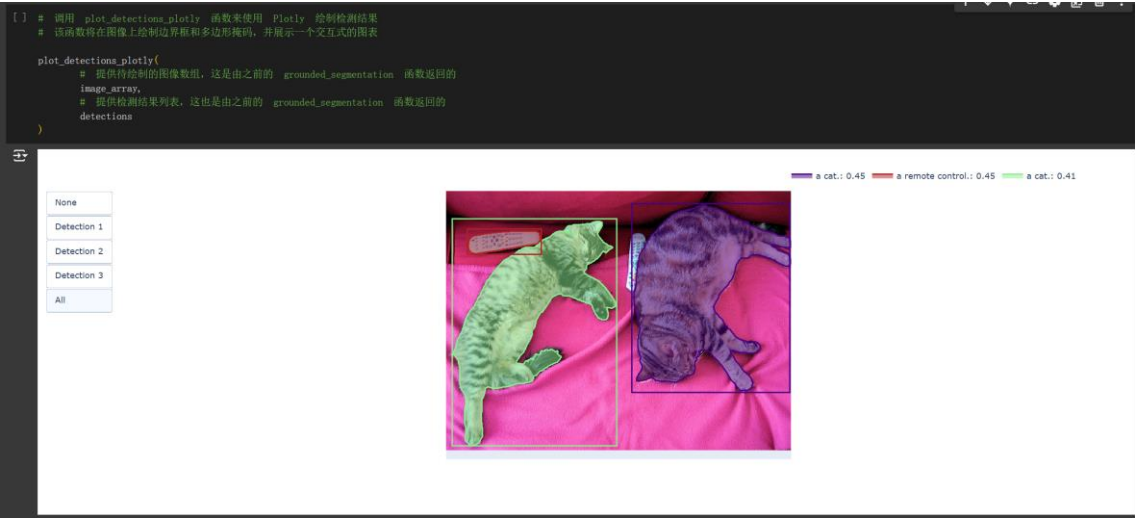


图 12 交互式图表可视化结果展示

实验内容 3 - 集成 RAM++ 和 Grounded-SAM:

问题 1: 由于 Google Colab 对 GPU 资源的限制，需要将模型在 VS Code 或者 Notebook 里面集成，但是 “!pip install --upgrade -q git+https://github.com/huggingface/transformers” 这行代码只能在 Google Colab 里面运行，在 VS Code 或者 Notebook 里面会因为网络或者其他问题一直报错，无从下手。

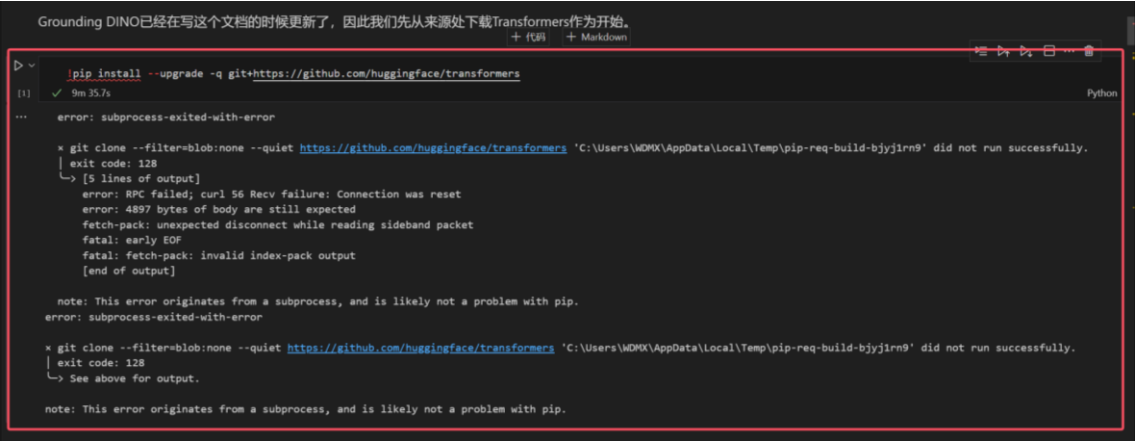
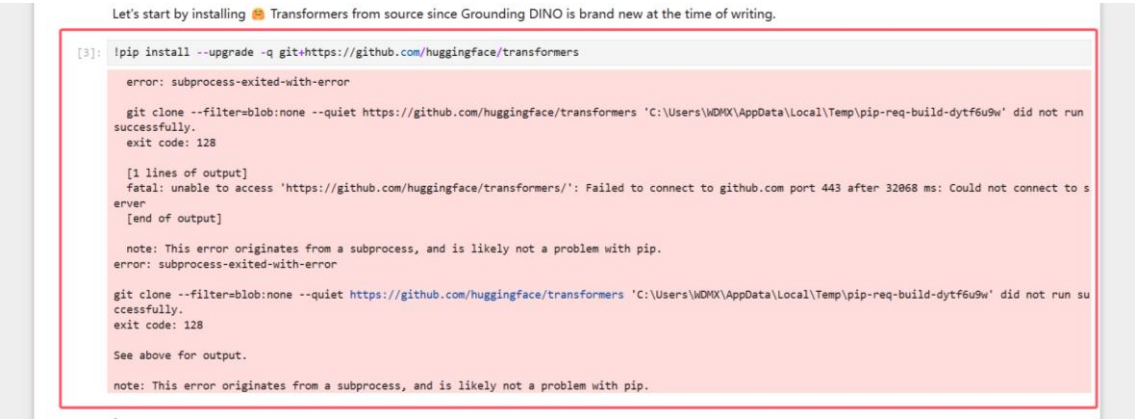
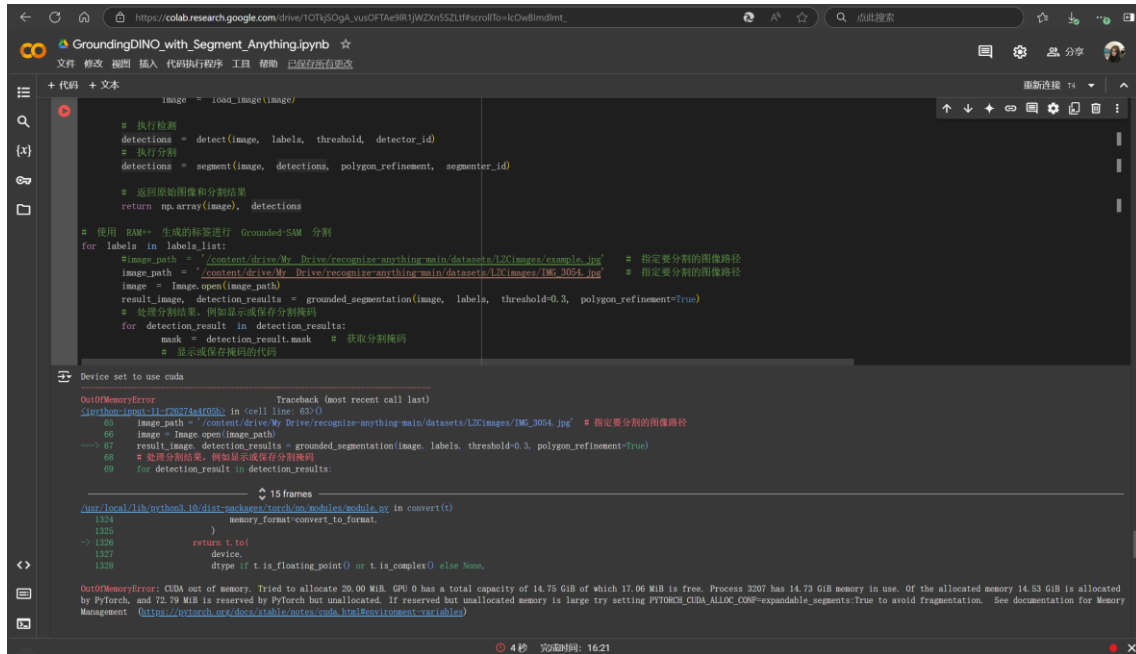


图 13 上为 VS Code 报错，下为 Notebook 报错



问题 2: 思路是“将 RAM++对图片数据集运行生成标签，保留标签结果并传给 Grounded-SAM 模型替换图像标签运行，然后使用本地数据集测试后，再使用 OCID 数据集检验”目前是这个思路不知道对不对。然后把 RAM++和 Grounded-SAM 在 Google Colab 合在一起了，但是 GPU 跑了一会儿容量不够报错，CPU 跑了 1 个多 2 个小时 RAM 满了，都跑不出来。



```
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

# 使用 RAM++ 生成的标签进行 Grounded-SAM 分割
for labels in labels_list:
    #image_path = '/content/drive/My_Drive/recognize-anything-main/datasets/L2C/images/example.jpg' # 指定要分割的图像路径
    image_path = '/content/drive/My_Drive/recognize-anything-main/datasets/L2C/images/IMG_3054.jpg' # 指定要分割的图像路径
    image = Image.open(image_path)
    result_image, detection_results = grounded_segmentation(image, labels, threshold=0.3, polygon_refinement=True)
    # 处理分割结果，例如显示或保存分割掩码
    for detection_result in detection_results:
        mask = detection_result.mask # 获取分割掩码
        # 显示或保存掩码的代码

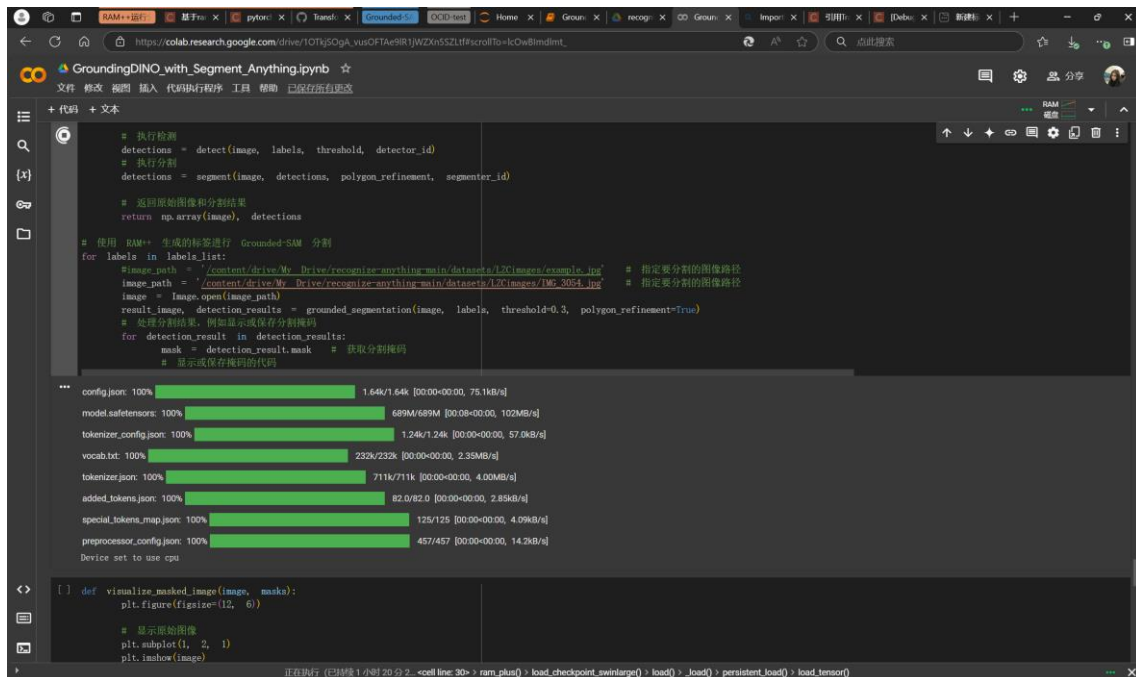
Device set to use cuda

Traceback (most recent call last):
  OutOfMemoryError                                Traceback (most recent call last)
  [python] input_11_726734f40b3b in <cell line: 63>()
    65 image_path = '/content/drive/My_Drive/recognize-anything-main/datasets/L2C/images/IMG_3054.jpg' # 指定要分割的图像路径
    66 image = Image.open(image_path)
--> 67 result_image, detection_results = grounded_segmentation(image, labels, threshold=0.3, polygon_refinement=True)
    68 # 处理分割结果，例如显示或保存分割掩码
    69 for detection_result in detection_results:

      15 frames
    /usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py in convert(t)
    1324         memory_format=convert_to_format,
    1325     )
    1326     return t.to(
    1327         device,
    1328         dtype if t.is_floating_point() or t.is_complex() else None,

  OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacity of 14.75 GiB of which 17.06 MiB is free. Process 3307 has 14.53 GiB memory in use. Of the allocated memory 14.53 GiB is allocated by PyTorch, and 72.59 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PTDETECT_CUDA_ALLOC_CONC_EXPANDABLE_SEGMENTS=True to avoid fragmentation. See documentation for Memory Management (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```

图 14 上为 GPU 不足，下为 CPU 运行（RAM 撑满了）



```
def visualize_masked_image(image, masks):
    plt.figure(figsize=(12, 6))

    # 显示原始图像
    plt.subplot(1, 2, 1)
    plt.imshow(image)
```

【结合问题 1 和问题 2，目前想把这个模型转移到 Notebook 去跑，但被限制】

问题 3: 然后 OCID 的数据集的测试不是很懂，下载下来了 OCID-dataset.tar.gz，是没见过后的后缀名，大概 6.35G，那个使用教程是按照红框来的话感觉不知道怎么输入进去，蓝框不知道需不需要在红框做之前去做，有点不太理解

← → ↺ 🔍

https://github.com/NVlabs/UnseenObjectClustering?tab=readme-ov-file

🔍 点此搜索

📄 自述文件 📄 许可证

在桌面对象数据集 (TOD) 上进行训练和测试

1. 从[此处](#)下载桌面对象数据集 (TOD) (34G)。

2. 为 TOD 数据集创建符号链接

```
cd $ROOT/data
ln -s $TOD_DATA tabletop
```

3. 在 TOD 数据集上进行训练和测试

```
cd $ROOT

# multi-gpu training, we used 4 GPUs
./experiments/scripts/seg_resnet34_8s_embedding_cosine_rgbdd_add_train_tabletop.sh

# testing, $GPU_ID can be 0, 1, etc.
./experiments/scripts/seg_resnet34_8s_embedding_cosine_rgbdd_add_test_tabletop.sh $GPU_ID $EPOCH
```

在 OCID 数据集和 OSD 数据集上进行测试

1. 从[此处](#)下载 OCID 数据集, 并创建符号链接:

```
cd $ROOT/data
ln -s $OCID_dataset OCID
```

2. 从[此处](#)下载 OSD 数据集, 并创建元件链接:

```
cd $ROOT/data
ln -s $OSD_dataset OSD
```

3. 检查 experiments/名称为 test_ocid 或 test_ocr 的脚本中的脚本。确保已训练的 checkpoint 的路径存在。

```
experiments/scripts/seg_resnet34_8s_embedding_cosine_rgbdd_add_test_ocid.sh
experiments/scripts/seg_resnet34_8s_embedding_cosine_rgbdd_add_test_osd.sh
```