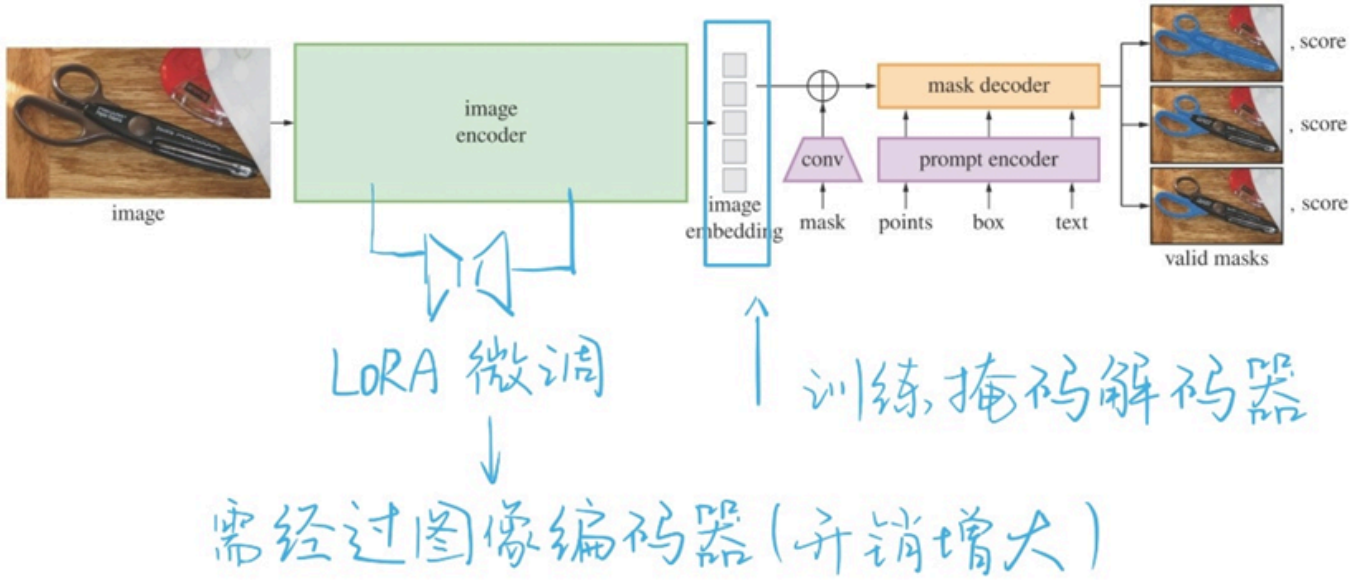
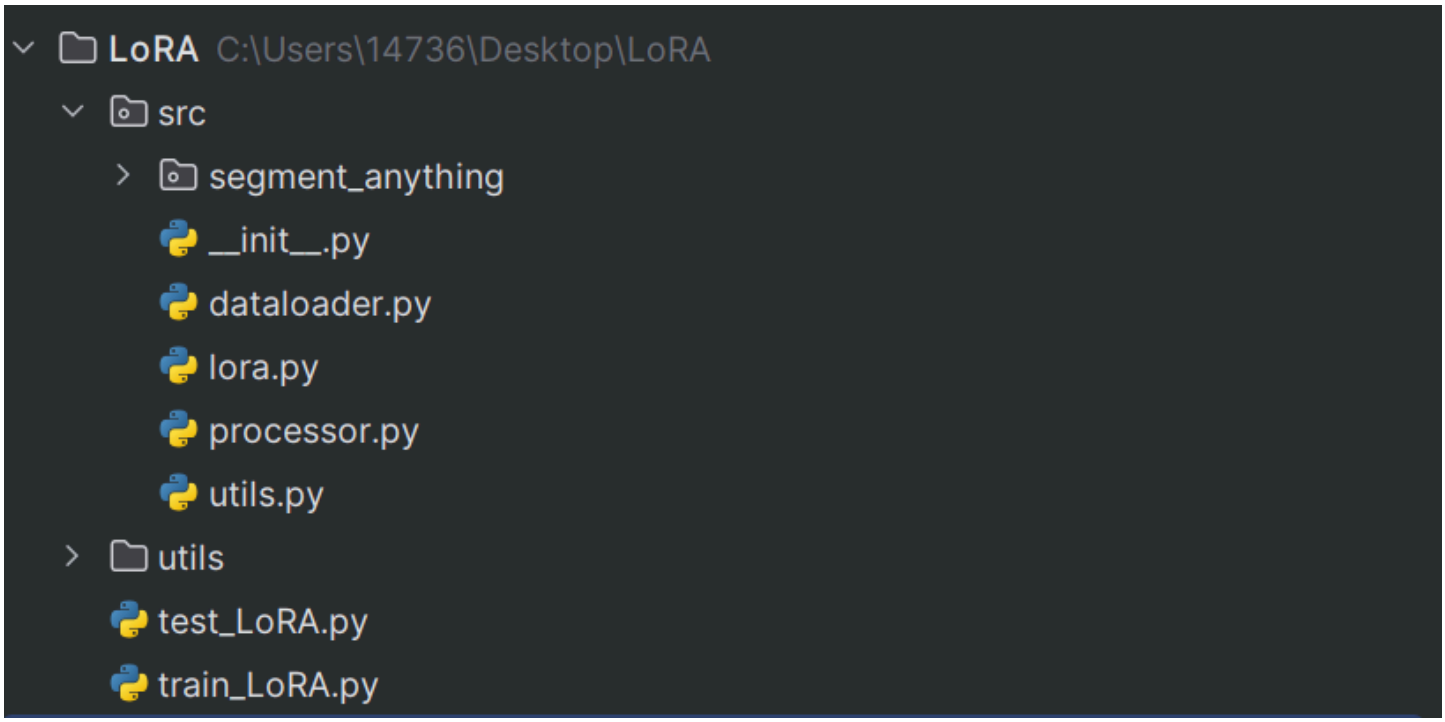
SAM模型lora微调代码解释

相比于原先的训练掩码解码器的代码，lora微调是对图像编码器的Transformer块的Q和V添加低秩矩阵，然后微调这些个低秩矩阵。这里微调的是sam_vit_b模型，Transformer层一共12个，把秩（rank）设置为4的时候，训练的参数数量一共是768×4×2×2×12=147456个。
因为图像编码器的参数量非常多，计算梯度的时候会占用大量显存，所以我在4090的服务器上，单卡训练的时候，batch_size只能到2，但是在A40机器上，单卡可以跑到btach_size=4。
这里放一张lora微调的原理示意图



代码解释

1、lora结构部分



文件里面，src目录下的lora.py,就是lora的结构，实现了添加低秩矩阵的功能。

2、数据读取+提示生成部分

然后是dataloader.py

```
class DatasetSegmentation(Dataset):

    def __init__(self, data_root, processor: Samprocessor):
        self.data_root = data_root
        self.npz_files = sorted(os.listdir(self.data_root))
        self.npz_data = [np.load(join(data_root, f)) for f in self.npz_files]

        self.imgs = np.vstack([d['imgs'] for d in self.npz_data])
        self.ori_gts = np.vstack([d['gts'] for d in self.npz_data])

        self.processor = processor

    def __len__(self):
        # 返回数据集中图像的总数量
        return self.ori_gts.shape[0]

    def __getitem__(self, index: int) -> list:
        # 获取指定索引对应的图像和掩膜
        img = self.imgs[index]  # 获取图像
        mask = self.ori_gts[index]  # 获取对应掩码
        mask = (mask > 0).astype(np.uint8)  # 将掩膜转换为二进制格式（0/1）

        gt2D = np.array(mask)  # 将掩码转换为 NumPy 数组
        original_size = tuple(img.shape)[:2]  # 获取图像的原始大小（高度，宽度）

        box = utils.get_bounding_box(gt2D)

        point = utils.get_point(gt2D)

        inputs = self.processor(img, original_size, box, point)
        inputs["ground_truth_mask"] = torch.from_numpy(gt2D)
        return inputs

def collate_fn(batch: torch.utils.data) -> list:
    """
    自定义 collate 函数，用于处理来自 DataLoader 的一批数据。
    参数:
        batch: 批处理数据集（字典列表）
    返回:
        (list): 包含批处理数据的字典列表。
    """
    return list(batch)  # 将批处理数据转换为列表并返回
```

我们的数据集经过预处理，会变成.npz格式的文件，这里就是读取训练数据集目录下的所有.npz文件，然后将图像和真实掩码分开读取，并且将真实掩码传入函数，得到随机打点的点提示和扰动为5的框提

示。

这个代码实现了一个图像分割数据集类，能够加载图像和掩码数据，并进行预处理，最终生成模型所需的输入格式，包括原始图，图像尺寸，真实掩码，点提示，框提示。不过这里的输入input会在后面重新提到。

3、数据处理和提示生成的具体操作部分

接着是processor.py，上一个代码基本都是调用各种函数什么的，这个代码和下一个就是具体功能的实现了

```
class Samprocessor:
    def __init__(self, sam_model: LoRA_sam):

        super().__init__()  # 初始化父类
        self.model = sam_model  # 保存传入的 SAM 模型实例
        self.transform = ResizeLongestSide(sam_model.image_encoder.img_size)  # 初始化图像变换器
        self.reset_image()  # 重置图像状态

    def __call__(self, image: np, original_size: tuple, box: list, points: list) -> dict:
            # 处理图像和边界框提示
        image_torch = self.process_image(image, original_size)  # 处理图像

            # 转换输入提示
        box_torch = self.process_prompt(box, original_size)  # 处理边界框提示

            # 处理点提示（如果存在）
        point_torch = self.process_points(points, original_size) if points is not None else Nor

            # 构建输入字典
        inputs = {
                "image": image_torch,  # 添加处理后的图像
                "original_size": original_size,  # 添加原始图像大小
                "boxes": box_torch,  # 添加处理后的边界框
                "points": point_torch  # 添加点提示
        }

        return inputs  # 返回输入字典

    def process_image(self, image: np, original_size: tuple) -> torch.tensor:

        nd_image = np.array(image)  # 将图像转换为 NumPy 数组
        input_image = self.transform.apply_image(nd_image)  # 应用图像变换
        input_image_torch = torch.as_tensor(input_image, device=self.device)  # 转换为张量并移动
        input_image_torch = input_image_torch.permute(2, 0, 1).contiguous()[None, :, :, :]  # i
        return input_image_torch  # 返回处理后的图像张量

    def process_prompt(self, box: list, original_size: tuple) -> torch.tensor:

        box_torch = None  # 初始化边界框张量
        # 设置框提示
        box_np = torch.tensor(box).float().numpy()
        sam_trans = ResizeLongestSide(self.model.image_encoder.img_size)
        box = sam_trans.apply_boxes(box_np, original_size=original_size)
        box_torch = torch.as_tensor(box, dtype=torch.float, device=self.device)
        if len(box_torch.shape) == 2:
            box_torch = box_torch[:, None, :]  # (B, 1, 4)
```

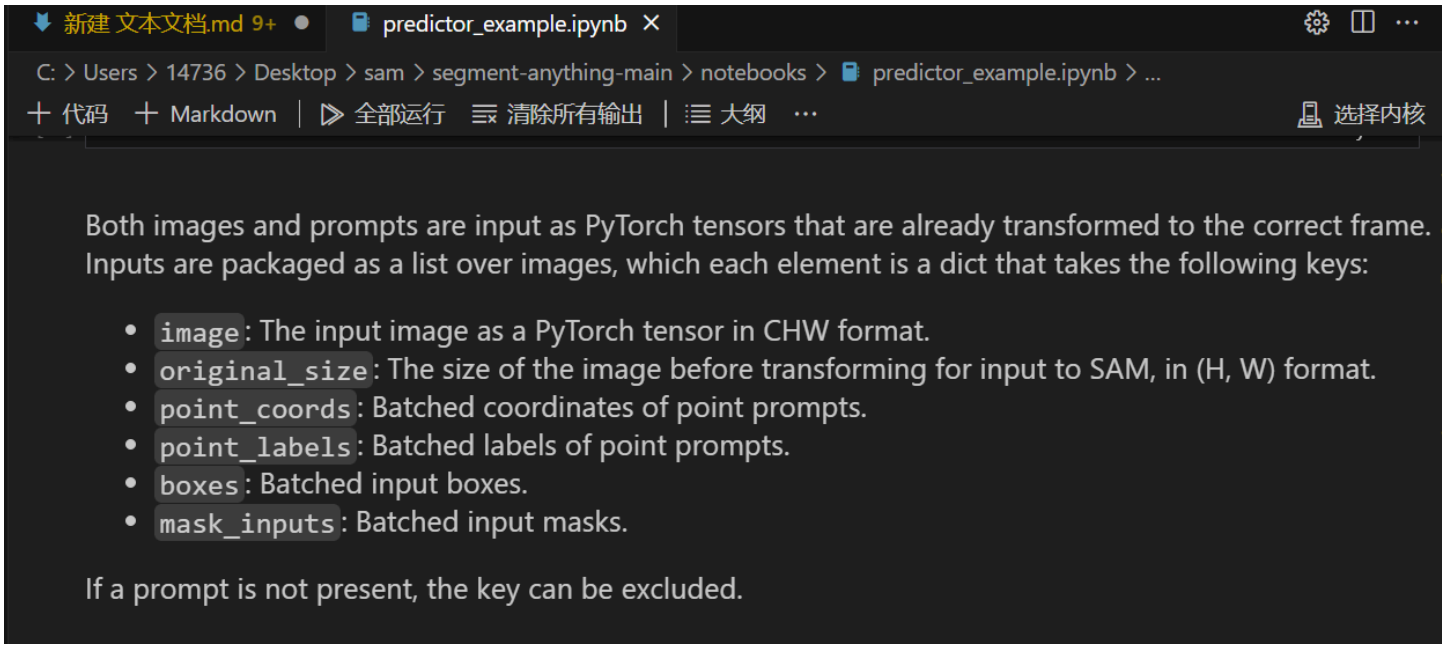
```
        return box_torch  # 返回处理后的边界框张量

    def process_points(self, points: list, original_size: tuple) -> torch.tensor:

        points_torch = None  # 初始化点提示张量

        points = torch.tensor(points).to(self.device)
        original_image_size = (256, 256)  # 手动设置每个切片的size(不太智能)
        transform = ResizeLongestSide(self.model.image_encoder.img_size)

        point_coords = transform.apply_coords_torch(points, original_image_size)
        num_points = point_coords.shape[0]
        label = torch.full((num_points, 5), 1, dtype=torch.float, device=self.device)
        point_torch = (point_coords, label)

        return point_torch  # 返回处理后的点提示张量

    @property
    def device(self) -> torch.device:
            # 返回模型所在的设备（CPU 或 GPU）
        return self.model.device

    def reset_image(self) -> None:
        """重置当前设置的图像。"""
        self.is_image_set = False  # 标记图像未设置
        self.features = None  # 重置特征
        self.orig_h = None  # 重置原始高度
        self.orig_w = None  # 重置原始宽度
        self.input_h = None  # 重置输入高度
        self.input_w = None  # 重置输入宽度
```

在上面dataloader.py文件里，有一句inputs = self.processor(img, original_size, box, point)，这里是将原图，图像尺寸，我得到的点和框，转换为符合SAM模型输入格式的形式。

```
        # 构建输入字典
        inputs = {
                "image": image_torch,  # 添加处理后的图像
                "original_size": original_size,  # 添加原始图像大小
                "boxes": box_torch,  # 添加处理后的边界框
                "points": point_torch  # 添加点提示
        }
```

这个输入字典并不完整，因为在dataloader.py的代码里可以看到，后面还是添加了一个真实掩码的。也就是在dataloader.py中对于processor的调用是这样的，第二局，就是往里面又传了一个真实掩码，

这是为了之后计算损失的。




```
    inputs = self.processor(img, original_size, box, point)
    inputs["ground_truth_mask"] = torch.from_numpy(gt2D)
```

4、utils.py文件

这个文件实际上是点提示、框提示、计算损失的相关函数。前面一半是原先的参考代码里的，我们这里用不到有用的只有这部分

```python
def get_bounding_box(gt2D: np.array) -> list:
  # 获取框
  y_indices, x_indices = np.where(gt2D > 0)
  x_min, x_max = np.min(x_indices), np.max(x_indices)
  y_min, y_max = np.min(y_indices), np.max(y_indices)
  # add perturbation to bounding box coordinates
  H, W = gt2D.shape
  x_min = max(0, x_min - np.random.randint(0, 5))
  x_max = min(W, x_max + np.random.randint(0, 5))
  y_min = max(0, y_min - np.random.randint(0, 5))
  y_max = min(H, y_max + np.random.randint(0, 5))
  bboxes = np.array([x_min, y_min, x_max, y_max])

  return bboxes

def get_point(gt2D: np.array) -> list:
    if gt2D.sum() == 0:

        n = 5  # 你想要[0, 0]的个数
        selected_points = [[0, 0] for _ in range(n)]
        print(selected_points)

    else:
        ########### 随机打点 #################
        y_indices, x_indices = np.where(gt2D == 1)
        points = np.column_stack((x_indices, y_indices))

        selected_indices = np.random.choice(len(points), size=5, replace=True)
        selected_points = points[selected_indices]
        selected_points = selected_points.tolist()

    return selected_points

def stacking_batch(batch, outputs):
    stk_gt = torch.stack([b["ground_truth_mask"] for b in batch], dim=0)
    stk_out = torch.stack([out["low_res_logits"] for out in outputs], dim=0)
    return stk_gt, stk_out
```

这部分在最开始的训练代码里也有。但是最后一个stacking_batch是为计算损失做铺垫。
这个函数的作用是将一个批次（batch）中的真实标签掩码（ground_truth_mask）和模型输出的低分辨率 logits 分别堆叠起来，返回两个张量。
然后在训练代码中

```
        stk_gt, stk_out = utils.stacking_batch(batch, outputs)
        stk_out = stk_out.squeeze(1)
        stk_gt = stk_gt.unsqueeze(1)  # We need to get the [B, C, H, W] starting from [H, W]

        loss = seg_loss(stk_out, stk_gt.float().to(device))
```

训练代码中计算损失时，则是将我的batch（这里是我训练代码中输入的训练数据，本质上就是input形式的）和output传入函数，然后该函数将output中的结果和我的[b["ground_truth_mask"] for b in batch]这一真实掩码都堆叠起来，返回给两个变量，再计算损失。

```
[out["low_res_logits"] for out in outputs]
```

其中的“low_res_logits”，就是SAM模型输出的一种