

# 《数字图像处理与分析》学习报告

2024-2025 学年秋季学期

班级：软英 2101

学号：20216573

姓名：黄子恒

1. 请你分别介绍基于深度学习、自适应的图像增强算法，并比较这两种算法的优缺点。 (40 分)

基于深度学习的图像增强算法涵盖了多种模型和技术，如 UNet、GAN 和 Segment Anything 等。这些算法通常利用卷积神经网络 (CNN) 来自动学习图像的特征，能够高效处理复杂的图像任务。例如，UNet 特别适合医学图像分割，而 GAN 则通过生成对抗的方式改善图像质量。在深度学习方法中，网络可以根据具体任务进行训练，从而实现超分辨率、去噪和风格迁移等功能。尽管这些算法在处理复杂图像方面表现出色，但它们的缺点在于需要大量的标注数据和高计算资源，训练过程也可能较为繁琐，容易受到过拟合的影响。

相对而言，自适应的图像增强算法通常依赖于传统的图像处理技术，如直方图均衡化和自适应滤波。这些方法通过调整图像的亮度、对比度和色彩等属性，快速提高图像的视觉效果。自适应算法的优势在于实现简单、计算效率高，且不需要大量的数据支持。然而，它们在处理复杂场景时的效果相对有限，无法充分利用图像中的高层次特征，也缺乏灵活性。因此，尽管自适应算法在简单场景中表现良好，但在面对复杂的图像时，深度学习方法通常能够提供更高质量的增强效果。

特性	基于深度学习的算法	自适应图像增强算法
特征提取能力	强（自动学习复杂特征）	弱（依赖于手动设计的规则）
数据需求	大（需要大量标注数据）	小（无需训练数据）
计算资源消耗	高（需要 GPU 等高性能计算设备）	低（计算量小，快速处理）
灵活性	高（可针对特定应用进行训练）	低（规则固定，不易调整）
适用场景	适合复杂图像和高要求的应用	适合简单场景和快速处理需求

2. 请你分别介绍基于大津阈值化、区域生长、深度学习的图像分割方法的原理，设计与编码实现包括这三种图像分割方法的可视化软件，并给出利用上述三种方法进行图像分割的结果比较与定量比较评价。（60 分）

### 大津阈值化

大津阈值化是一种自动选择图像分割阈值的方法，其核心目标是将图像分成前景和背景。该方法的基本思路是通过分析图像的灰度直方图，寻找一个最佳阈值，使得分割后的前景和背景的类内方差最小，类间方差最大。

Otsu 方法会计算图像中每个可能阈值（通常在 0 到 255 的范围内）下的前景和背景的灰度分布。对于每个阈值，算法会将图像分为两个部分：低于阈值的像素（背景）和高于阈值的像素（前景）。然后，计算这两部分的灰度平均值和各自的方差。

然后，算法会通过加权的方式来计算类间方差。这是因为每个部分在图像中的占比不同，因此需要考虑其对整体方差的贡献。类间方差的计算旨在衡量前景和背景之间的分离度，分离度越大，说明阈值选择得越好。

Otsu 方法会选取最大化类间方差的阈值，作为最佳的分割阈值。这种方法的优势在于不需要用户手动选择阈值，能够有效应对不同光照条件和对比度的图像。通过这种方式，Otsu 阈值化能够自动化地实现高效的图像分割。

## 区域生长

区域生长是一种经典的图像分割方法，主要通过逐步扩展预定义的“种子点”来将相似的像素聚集成一个区域。该方法的核心思想是，从一个或多个初始像素（种子点）开始，通过逐步检查其邻近像素，根据某种相似性准则（例如颜色、灰度值、纹理等），将相似的邻近像素加入到同一个区域中，直到无法再扩展为止。

区域生长的第一步是选择种子点。这些种子点通常是图像中某些特定的像素，可能由用户手动选定，或通过其他算法自动选定。这些种子点是区域生长的起点，意味着每个种子点所代表的像素是该区域的初始像素。

区域生长的核心在于如何判断一个像素是否应当被包含到现有的区域中。这个判断依据被称为生长准则（growth criteria），通常根据像素的相似性来决定。最常见的相似性准则是像素之间的灰度差或颜色差，若邻近像素与区域内的种子点或区域内其他像素的灰度值差异在一定阈值范围内，则认为它们相似，可以归为同一区域。

### 常见的相似性标准包括：

- 如果邻近像素的灰度值与区域内部像素的灰度值差异小于某个阈值，则将其归入该区域。

- 在某些应用中，可以用纹理特征来衡量像素的相似性，比如局部图像的结构是否相似。
- 为了避免区域生长过度扩展，通常只考虑种子点周围一定距离内的像素，来控制生长范围。

一旦种子点和生长准则确定，生长过程就开始了。具体过程如下：

- 从种子点出发，检查种子点四周的邻近像素（通常是上下左右或更广泛的邻居），判断这些像素是否满足生长准则。
- 如果某个邻近像素满足准则，它会被添加到当前的区域中，并且该像素也会成为新的种子点。
- 重复这个过程，对新的种子点的邻近像素进行同样的判断和扩展，直到没有更多的像素满足生长条件为止。

这个过程一直持续到所有满足准则的像素都被加入到区域中，形成最终的分割区域。

区域生长的停止条件通常有两种：

- 当某个区域的所有邻近像素都不满足生长条件时，生长停止。
- 有时可以根据图像的先验信息设定一个区域的最大大小，当区域达到一定的像素数量时，即停止生长。

在处理复杂图像时，可能需要从多个种子点同时进行区域生长。每个种子点会独立启动一个区域，互不干扰。随着区域的扩展，可能会出现不同区域相遇的情况，这时可以根据某些策略来决定是否合并两个区域，或者将它们保持分离。

深度学习

深度学习的图像分割方法近年来取得了显著的进展，特别是在医学影像、自动驾驶等领域。与传统的基于阈值或区域生长的方法不同，深度学习利用了神经网络的强大建模能力，能够自动学习图像的特征并执行复杂的分割任务。

深度学习图像分割的核心思想是通过训练一个神经网络，将输入图像映射为一个“分割图”，即为每个像素分配一个类别标签。这个任务被称为语义分割，它要求网络不仅要识别出图像中的物体，还要在像素级别上标注这些物体的具体位置。根据任务的不同，图像分割可以分为以下几种类型：

- **语义分割**: 为图像中的每个像素赋予一个类别标签，如道路、车辆、行人等。
- **实例分割**: 不仅要分割出物体类别，还要区分同类中的不同实例（例如区分多辆车）。
- **全景分割**: 结合语义分割和实例分割，处理复杂场景中的所有物体。

在深度学习图像分割中，**卷积神经网络** 是最常用的模型之一。CNN 能够自动学习图像中的局部特征，这使得它特别适合处理图像数据。CNN中的卷积层负责提取图像的局部特征，而池化层则通过减少特征图的分辨率来降低计算成本。

图像分割的关键挑战在于，最终的输出需要保持与输入图像相同的空间分辨率。这与传统的图像分类任务不同，分类任务的输出只是一个简单的类别标签，而分割任务要求输出的每个像素都需要有一个对应的类别。因此，图像分割网络不仅要在卷积过程中提取特征，还需要通过某种方式恢复高分辨率的输出。

为了处理上述挑战，深度学习中专门设计了几种图像分割网络架构，其中最经典的是 **U-Net**。**U-Net** 是一种编码器-解码器结构的网络，其工作原理如下：

- **编码器部分**: 类似于经典的卷积神经网络，编码器部分通过一系列卷积层和池化层提取图像的多层次特征。在这个过程中，图像的分辨率会逐渐下降，而特征的深度逐渐增加。
- **解码器部分**: 为了恢复与输入图像相同大小的分割图，**U-Net** 设计了一个解码器部分，通过上采样操作逐步恢复图像的空间分辨率。上采样操作的目的是将低分辨率的特征图恢复到高分辨率，从而生成细粒度的分割结果。

- **跳跃连接：**U-Net 中的一个重要设计是跳跃连接，即直接将编码器中对应分辨率的特征图与解码器中的特征图进行拼接。这一设计可以帮助网络在恢复空间分辨率时保留更多的低层次细节信息，从而提高分割的精度。

除了 U-Net，另一种常用的图像分割架构是 **全卷积网络**。FCN 的核心思想是将传统 CNN 中的全连接层替换为卷积层，从而能够处理任意大小的输入图像，并生成相同大小的分割输出。通过在网络的末端引入上采样层，FCN 可以将低分辨率的特征图恢复到原始分辨率，生成像素级的分割图。

FCN 是最早提出的端到端深度学习图像分割网络之一，它为后续的许多架构（如 U-Net 和 DeepLab）奠定了基础。

DeepLab 系列模型是另一类广泛应用的深度学习图像分割架构，尤其是在自动驾驶和自然场景分割任务中表现出色。DeepLab 通过引入**空洞卷积**，能够在不增加参数的前提下扩大感受野。这意味着网络可以在较大范围内捕捉全局信息，而不会显著增加计算成本。

此外，DeepLab 还引入了**条件随机场**作为后处理步骤，以增强分割结果的边界细节，防止分割区域过于模糊。这使得 DeepLab 在处理复杂场景（如路面和行人）时能取得非常精细的分割效果。

在深度学习图像分割中，损失函数的设计非常关键。与分类任务不同，分割任务是一个像素级的分类问题。因此，最常用的损失函数是**交叉熵损失**，它用于衡量每个像素的分类误差。

另外，对于一些不平衡的数据集，常常使用**Dice系数损失**或**IoU损失**，这些损失函数能够更好地衡量分割区域的重叠程度，从而提高小区域的分割精度。

深度学习的图像分割通常需要大量的标注数据，而标注像素级的分割图像非常耗时。因此，**数据增强技术**非常重要，通过旋转、翻转、缩放等操作，生成更多的训练样本，增强模型的泛化能力。

迁移学习也是常用策略。通常可以使用在大规模数据集上（如ImageNet）预训练好的网络作为基础，然后在特定的分割任务上进行微调。这种方法可以显著减少训练时间，并提高分割精度，尤其是当训练数据较少时。

编码实现：

## 大津阈值化

大津方法（Otsu's Method）是一种用于自动确定阈值将灰度图像进行二值化处理的算法。它的核心思想是通过最大化类间方差来找到最佳的阈值，从而将图像分割为前景和背景。

首先，输入的图像必须是灰度图，即每个像素的取值范围在 0 到 255 之间。大津方法首先会计算图像的灰度直方图，这个直方图描述了图像中每个灰度值出现的频率。比如，如果某个灰度值出现的次数很多，则该值在直方图中对应的柱状高度就会比较高。

接下来需要获取图像的总像素数。图像的总像素数等于图像的宽度乘以高度，这个值后续会用于计算前景和背景区域的权重。为了找到最优的阈值，大津方法会遍历所有可能的灰度值（从 0 到 255），并将其分别作为潜在的分割阈值。

在进行遍历之前，还需要初始化一些变量。首先是一个名为 `current_max` 的变量，用来存储当前发现的最大类间方差值。其次是 `threshold`，用于记录对应于最大类间方差的阈值。还有两个重要的变量是 `sum_total` 和 `sum_background`。`sum_total` 表示图像中所有像素灰度值的总和，它通过遍历整个直方图乘以对应的灰度值来计算。这个值用于后续的前景和背景灰度均值计算。而 `sum_background` 用于累加被分类为背景像素的灰度值总和。

算法会从灰度值 0 开始逐步测试每个灰度值作为可能的阈值。在每次循环中，计算该灰度值下背景和前景的权重。背景权重（`weight_background`）可以通过累加从 0 到当前灰度值的像素数量来得到，而前景权重（`weight_foreground`）则是总像素数减去背景权重。前景和背景的权重分别代表被分类为背景和前景的像素数量占总像素的比例。

接下来，大津方法会分别计算背景和前景的灰度均值。背景的灰度均值可以通过将当前灰度范围内的像素灰度总和除以背景的权重得到。同理，前景的灰度均值则是总的灰度和减去背景的灰度和，再除以前景的权重得到。

关键步骤是计算类间方差，这是衡量前景和背景分离效果的标准。类间方差的计算公式是背景权重乘以前景权重再乘以背景灰度均值和前景灰度均值的平方差。通过这种方式，类间方差越大，表示前景和背景的差异越显著，分割效果越好。

大津方法的目标是找到能使类间方差最大的那个灰度值作为最佳阈值。因此，算法会在每次迭代中比较当前计算出的类间方差和之前记录的最大值 `current_max`，如果当前的类间方差更大，则更新 `current_max` 并记录当前的灰度值作为新的阈值。

当遍历完所有的灰度值后，最终找到的那个最大类间方差对应的灰度值就是最佳的二值化阈值。最后，使用这个阈值对原始图像进行二值化处理，将所有灰度值高于阈值的像素设为前景（通常为白色，值为 255），低于阈值的像素设为背景（通常为黑色，值为 0）。最终输出的结果就是一幅经过阈值分割后的二值图像。

```
import cv2
import numpy as np

def otsu_thresholding(image):
    # 计算直方图
    hist = cv2.calcHist([image], [0], None, [256], [0, 256])
    total_pixels = image.size

    current_max, threshold = 0, 0
    sum_total, sum_background = 0, 0
    weight_background, weight_foreground = 0, 0

    for i in range(256):
        sum_total += i * hist[i]
```

```

for i in range(256):
    weight_background += hist[i]
    if weight_background == 0:
        continue

    weight_foreground = total_pixels - weight_background
    if weight_foreground == 0:
        break

    sum_background += i * hist[i]
    mean_background = sum_background / weight_background
    mean_foreground = (sum_total - sum_background) / weight_foreground

    # 计算类间方差
    between_class_variance = weight_background * weight_foreground * (mean_background -
mean_foreground)**2

    if between_class_variance > current_max:
        current_max = between_class_variance
        threshold = i

    _, thresh_image = cv2.threshold(image, threshold, 255, cv2.THRESH_BINARY)

# 确保输出为 uint8 格式
thresh_image = thresh_image.astype(np.uint8)

return thresh_image

```

## 区域生长

区域生长是一种经典的图像分割方法，它通过从一个或多个“种子点”出发，根据灰度值的相似性逐步扩展来确定图像中的区域。这个过程模拟了真实世界中某种区域的增长，逐渐将相邻的像素纳入到同一个区域，直到不满足相似性条件为止。下面对该方法进行详细的解释。

首先，输入的是一幅灰度图像，通常以二维的 numpy 数组形式表示。每个像素都有一个对应的灰度值，范围从 0 到 255。与此同时，还需要提供一个“种子点”，即区域生长的起始

点，通常以坐标形式给出（例如， $(x, y)$ ），并且还可以设定一个“阈值”，它控制区域扩展的灰度值相似程度。阈值越小，区域扩展的要求越严格，区域就会相对较小；阈值较大时，更多的相邻像素将被包含进来。

在算法开始时，首先获取图像的尺寸（高度和宽度），并创建一个与输入图像大小相同的二值化图像 `seg_image`，这个图像将用于存储分割后的结果。初始时，所有像素的值都设置为 0，这意味着图像中还没有被分割的区域。接下来，会创建一个 `visited` 数组，来标记每个像素是否已经被访问，防止重复处理。刚开始时，所有像素都未被访问，因此这个数组中的所有值都设置为 `False`。

然后，算法将种子点加入一个待处理的列表 `seed_list` 中，表示该点是区域生长的起始点。种子点的像素值作为初始的区域均值 `region_mean`，并且将该种子点标记为已访问，并将它在分割图像 `seg_image` 中对应的像素值设置为 1，表示它属于目标区域。

为了实现区域的扩展，定义了一个四邻域规则，即每个像素只与它上下左右的四个像素相邻。这些邻居像素将用于检查是否可以纳入当前区域。之后，算法进入一个循环，从 `seed_list` 中取出待处理的种子点，检查其四邻域内的像素是否符合扩展条件。

对于每个邻居像素，首先会检查它是否在图像的边界内，确保不超出图像范围。然后，检查这个像素是否已经被访问过，若没有访问过，再比较它的灰度值与当前区域的平均灰度值 `region_mean` 的差异。如果该差异小于或等于预设的阈值，说明该像素与当前区域的灰度值足够相似，可以归入同一分割区域。于是，将这个像素的值在 `seg_image` 中标记为 1，并将它加入到 `seed_list` 中，等待进一步处理。同时，区域的平均灰度值 `region_mean` 会随着新加入像素的灰度值动态更新，以便更准确地描述当前区域的特性。

这种“区域生长”的过程会一直持续，直到待处理的种子点列表 `seed_list` 为空，说明没有更多的像素可以被纳入当前区域。最终，算法返回的是一个二值图像，图像中值为 1 的像素表示属于生长区域，值为 0 的像素表示背景区域。

区域生长算法的优点在于它能够根据种子点的选取和阈值设置，灵活地分割具有相似灰度值的连通区域。它特别适用于灰度值变化较小、对比度不太明显的图像。在某些应用场景中，例如医学影像分析或纹理检测，区域生长可以用于提取特定的结构或目标。通过适当调整阈值和种子点的位置，区域生长可以精确控制分割结果，适应不同的图像特性。

```
# region_growing.py

import numpy as np

def region_growing(image, seed_point, threshold=10):
    """
    对图像进行区域生长分割。
    
```

参数：

image: 输入的灰度图像 (numpy 数组)。

seed\_point: 种子点坐标，格式为(x, y)。

threshold: 灰度值差异的阈值。

返回：

seg\_image: 分割后的二值图像，区域为 1，其余为 0。

```
"""
height, width = image.shape
seg_image = np.zeros((height, width), np.uint8)
visited = np.zeros((height, width), np.bool_)

# 初始化种子点列表
seed_list = [seed_point]
visited[seed_point[1], seed_point[0]] = True
seg_image[seed_point[1], seed_point[0]] = 1
region_mean = float(image[seed_point[1], seed_point[0]])

# 定义四邻域
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

```
while seed_list:  
    x, y = seed_list.pop(0)  
    for dx, dy in neighbors:  
        nx, ny = x + dx, y + dy  
        if 0 <= nx < width and 0 <= ny < height and not visited[ny, nx]:  
            pixel_value = float(image[ny, nx])  
            if abs(pixel_value - region_mean) <= threshold:  
                seg_image[ny, nx] = 1  
                seed_list.append((nx, ny))  
                # 更新区域均值  
                region_mean = (region_mean * np.sum(seg_image) + pixel_value) / (np.sum(seg_image) + 1)  
                visited[ny, nx] = True  
  
return seg_image
```

## 深度学习

这一部分我分为两个部分：第一部分是写软件界面直接调用 SOTA 通用模型 meta 的 **segmentate-anything**；

第二部分我对 **segment-anything2** 在特定数据集上进行微调，并且和前者进行比较。

在这段代码中，Segment-Anything 2 (SAM 2) 模型被微调用于特定的 LabPics V1 数据集。

这个微调过程旨在让 SAM 2 更好地适应该数据集中的图像，进而提升在这些特定场景下的表现。下面从整体流程上详细解释这个微调过程，并与原始的 SAM 进行比较。

首先，SAM 2 的核心思想是对任意对象或场景进行图像分割，它能够处理复杂的场景、物体以及边缘区域。然而，尽管预训练的 SAM 模型具备强大的泛化能力，但在处理特定领域的数据时，直接应用预训练的模型可能并不能取得最佳效果。为此，通过在特定数据集上微调模型，可以让它更加适应这些特定的目标和场景，例如 LabPics V1 中的实验室器皿和材料。

该代码的第一步是从 LabPics V1 数据集中读取图像。LabPics V1 包含了实验室中的各种物体以及其对应的实例分割标注。为了实现微调，代码将数据加载为图像和对应的实例分割掩码。对于每张图像，将根据需要缩放至 1024x1024 的标准尺寸，以确保输入到模型中的图像

尺寸一致。此外，数据中的器皿和材料注释被组合成一个单一的掩码，用于后续的分割处理。

在训练过程中，模型会随机采样图像和对应的实例分割掩码。代码通过 `read_single` 函数从数据集中随机选取一张图像及其对应的掩码，并根据掩码生成单个二值化的分割区域。这个过程可以看作是模型的“输入提示”，即给定某个区域（例如实验室器皿的边缘），模型需要预测出该区域的完整掩码。

为了更高效地训练，`read_batch` 函数将多个样本组合成一个批次，这样在每次迭代时，模型可以同时处理多个图像，有助于加快训练并提升模型的泛化能力。

接下来，代码加载了 SAM 2 模型的预训练权重（`sam2_hiera_small.pt`），以及模型的结构配置文件（`sam2_hiera_s.yaml`）。SAM 2 的架构与之前的 SAM 模型类似，但在某些方面进行了优化，例如其图像编码器和掩码解码器。通过微调这些组件（图像编码器、提示编码器、掩码解码器），模型可以更好地适应 LabPics V1 数据集中的特定场景。

在代码中，显式启用了这些模块的训练模式，这表明它们将参与反向传播和权重更新。这一点与原始 SAM 的使用有区别，原始 SAM 模型通常不进行微调，而是直接用于推理；在这里，通过微调，这些模型模块会根据 LabPics V1 的数据进行调整。

训练循环的核心是通过批次图像来微调模型。每个训练步骤中，模型会首先将输入图像传递给图像编码器，生成图像的特征嵌入。接着，提示编码器根据输入的分割提示（如点或掩码）生成提示嵌入。随后，掩码解码器会根据图像嵌入和提示嵌入，生成一个高分辨率的分割掩码。

损失函数的设计在微调过程中至关重要。代码中采用了两种损失：分割损失和评分损失。分割损失基于二值交叉熵（cross entropy），用于衡量预测掩码与真实掩码之间的误差。评分损失则通过计算交并比（IOU，Intersection over Union），评估模型预测掩码与真实掩码的重叠程度。二者的结合帮助模型在分割精度和评分预测上进行全面优化。

## 微调的效果和对比

与直接使用预训练的 SAM 模型相比，SAM 2 经过微调后可以更好地适应 LabPics V1 数据集中的复杂场景。例如，实验室图像中常常存在器皿和材料的边缘模糊或重叠现象，预训练模型可能无法精准分割这些区域。而通过微调后的 SAM 2，模型能够更好地学习这些细节特征，从而提升分割精度。

原始 SAM 模型的优势在于其广泛的泛化能力，适用于不同领域的图像分割任务，无需特定数据集的微调。然而，它的缺点在于对特定领域图像（如 LabPics V1 中的实验室器皿）的分割效果可能不如微调后的模型精准。相比之下，SAM 2 在经过微调后，能更好地处理这些特定领域的数据。

```
import numpy as np
import torch
import cv2
import os

from torch.onnx.symbolic_opset11 import hstack

from sam2.build_sam import build_sam2
from sam2.sam2_image_predictor import SAM2ImagePredictor

# Read data

data_dir=r"LabPicsV1//" # Path to dataset (LabPics 1)
data=[] # list of files in dataset
for ff, name in enumerate(os.listdir(data_dir+"Simple/Train/Image/")): # go over all folder annotation
    data.append({"image":data_dir+"Simple/Train/Image/"+name,"annotation":data_dir+"Simple/Train/Instance/"+name[:-4]+".png"})
def read_single(data): # read random image and single mask from the dataset (LabPics)

    # select image

        ent = data[np.random.randint(len(data))] # choose random entry
        Img = cv2.imread(ent["image"])[...,::1] # read image
        ann_map = cv2.imread(ent["annotation"]) # read annotation

    # resize image
```

```

r = np.min([1024 / Img.shape[1], 1024 / Img.shape[0]]) # scaling factor
Img = cv2.resize(Img, (int(Img.shape[1] * r), int(Img.shape[0] * r)))
ann_map = cv2.resize(ann_map, (int(ann_map.shape[1] * r), int(ann_map.shape[0] * r)), interpolation=cv2.INTER_NEAREST)
if Img.shape[0]<1024:
    Img = np.concatenate([Img,np.zeros([1024 - Img.shape[0], Img.shape[1],3],dtype=np.uint8)],axis=0)
    ann_map = np.concatenate([ann_map, np.zeros([1024 - ann_map.shape[0], ann_map.shape[1],3],dtype=np.uint8)],axis=0)
if Img.shape[1]<1024:
    Img = np.concatenate([Img, np.zeros([Img.shape[0] , 1024 - Img.shape[1], 3],dtype=np.uint8)],axis=1)
    ann_map = np.concatenate([ann_map, np.zeros([ann_map.shape[0] , 1024 - ann_map.shape[1] , 3],dtype=np.uint8)],axis=1)

# merge vessels and materials annotations

mat_map = ann_map[:, :, 0] # material annotation map
ves_map = ann_map[:, :, 2] # vessel annotation map
mat_map[mat_map==0] = ves_map[mat_map==0]*(mat_map.max() + 1) # merge maps

# Get binary masks and points

inds = np.unique(mat_map)[1:] # load all indices
if inds._len_()>0:
    ind = inds[np.random.randint(inds._len_())] # pick single segment
else:
    return read_single(data)

#for ind in inds:
mask=(mat_map == ind).astype(np.uint8) # make binary mask corresponding to index ind
coords = np.argwhere(mask > 0) # get all coordinates in mask
yx = np.array(coords[np.random.randint(len(coords))]) # choose random point/coordinate
return Img,mask,[yx[1], yx[0]]]

def read_batch(data,batch_size=4):
    limage = []
    lmask = []
    linput_point = []
    for i in range(batch_size):
        image,mask,input_point = read_single(data)
        limage.append(image)
        lmask.append(mask)
        linput_point.append(input_point)

    return limage, np.array(lmask), np.array(linput_point), np.ones([batch_size,1])

# Load model

```

```

sam2_checkpoint = "sam2_hiera_small.pt" # path to model weight
model_cfg = "sam2_hiera_s.yaml" # model config
sam2_model = build_sam2(model_cfg, sam2_checkpoint, device="cuda") # load model
predictor = SAM2ImagePredictor(sam2_model)

# Set training parameters

predictor.model.sam_mask_decoder.train(True) # enable training of mask decoder
predictor.model.sam_prompt_encoder.train(True) # enable training of prompt encoder
predictor.model.image_encoder.train(True) # enable training of image encoder: For this to work you need to
scan the code for "no_grad" and remove them all
optimizer=torch.optim.AdamW(params=predictor.model.parameters(),lr=1e-5,weight_decay=4e-5)
scaler = torch.cuda.amp.GradScaler() # mixed precision

# Training loop

for itr in range(100000):
    with torch.cuda.amp.autocast(): # cast to mix precision
        image,mask,input_point, input_label = read_batch(data,batch_size=4) # load data batch
        if mask.shape[0]==0: continue # ignore empty batches
        predictor.set_image_batch(image) # apply SAM image encoder to the image
        # predictor.get_image_embedding()
        # prompt encoding

        mask_input, unnorm_coords, labels, unnorm_box = predictor._prep_prompts(input_point,
input_label, box=None, mask_logits=None, normalize_coords=True)
        sparse_embeddings, dense_embeddings =
predictor.model.sam_prompt_encoder(points=(unnorm_coords, labels), boxes=None, masks=None,)

        # mask decoder

        high_res_features = [feat_level[-1].unsqueeze(0) for feat_level in
predictor.features["high_res_feats"]]
        low_res_masks, prd_scores, _ _ =
predictor.model.sam_mask_decoder(image_embeddings=predictor.features["image_embed"],image_pe=predictor.model.sam_prompt_encoder.get_dense_pe(),sparse_prompt_embeddings=sparse_embeddings,dense_prompt_embeddings=dense_embeddings,multimask_output=True,repeat_image=False,high_res_features=high_res_features,)

        prd_masks = predictor.transforms.postprocess_masks(low_res_masks, predictor.orig_hw[-1])#
Upscale the masks to the original image resolution

        # Segmentation Loss calculation

        gt_mask = torch.tensor(mask.astype(np.float32)).cuda()
        prd_mask = torch.sigmoid(prd_masks[:, 0])# Turn logit map to probability map
        seg_loss = (-gt_mask * torch.log(prd_mask + 0.00001) - (1 - gt_mask) * torch.log((1 - prd_mask) +
0.00001)).mean() # cross entropy loss

        # Score loss calculation (intersection over union) IOU

```

```

inter = (gt_mask * (prd_mask > 0.5)).sum(1).sum(1)
iou = inter / (gt_mask.sum(1).sum(1) + (prd_mask > 0.5).sum(1).sum(1) - inter)
score_loss = torch.abs(prd_scores[:, 0] - iou).mean()
loss=seg_loss+score_loss*0.05 # mix losses

# apply back propagation

predictor.model.zero_grad() # empty gradient
scaler.scale(loss).backward() # Backpropagate
scaler.step(optimizer)
scaler.update() # Mix precision

if itr%1000==0: torch.save(predictor.model.state_dict(), "model.torch") # save model

# Display results

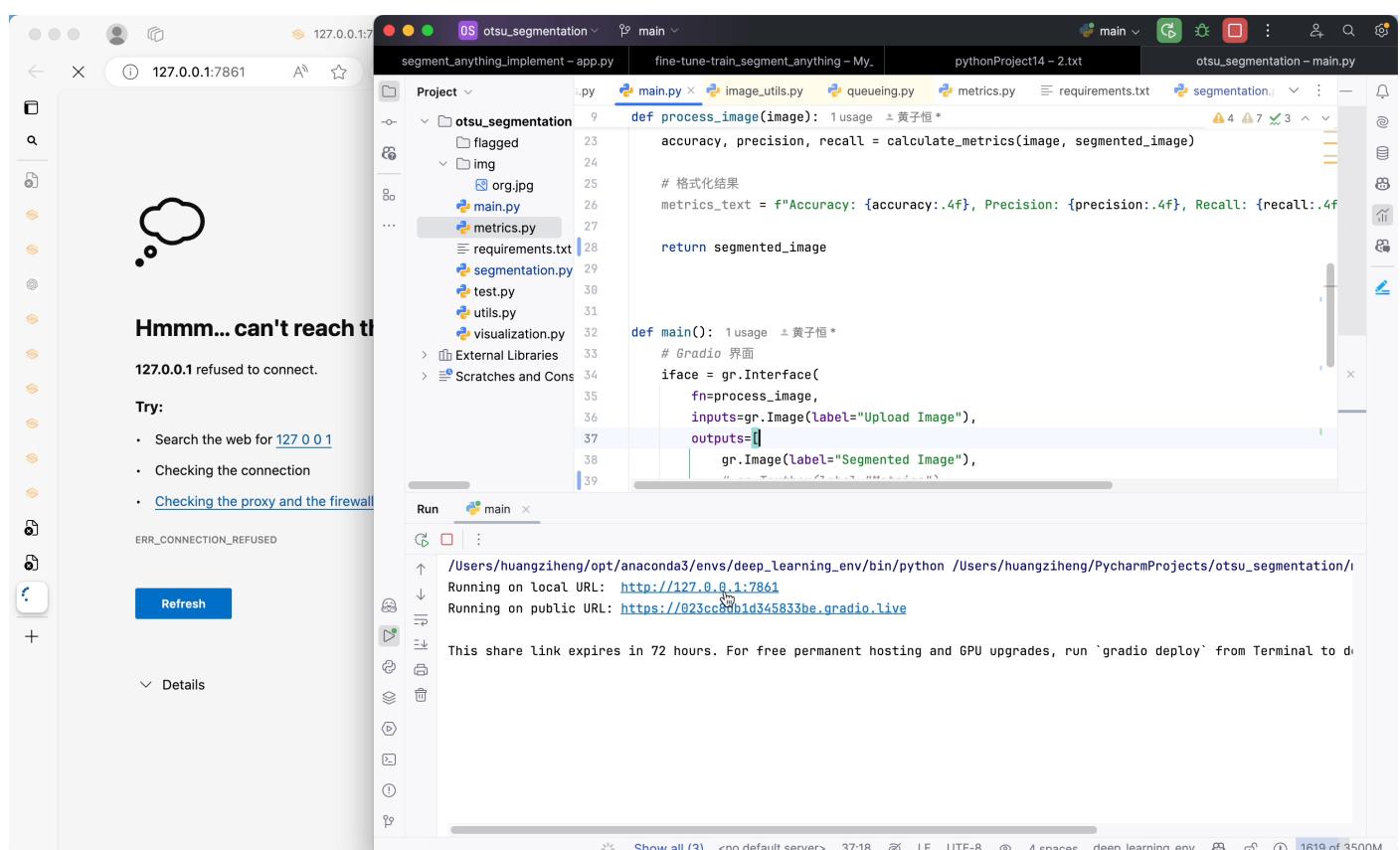
if itr==0: mean_iou=0
mean_iou = mean_iou * 0.99 + 0.01 * np.mean(iou.cpu().detach().numpy())
print("step)",itr, "Accuracy(IOU)=",mean_iou)

```

## 编码实现结果

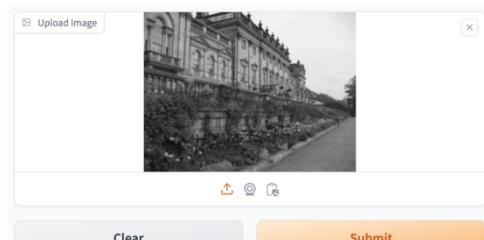
### 大津阈值化

结果动图（在 Microsoft Word 中查看或者在文件夹中查看）



### Otsu Thresholding Segmentation

Upload an image to apply Otsu's thresholding method and view the segmented result along with evaluation metrics.

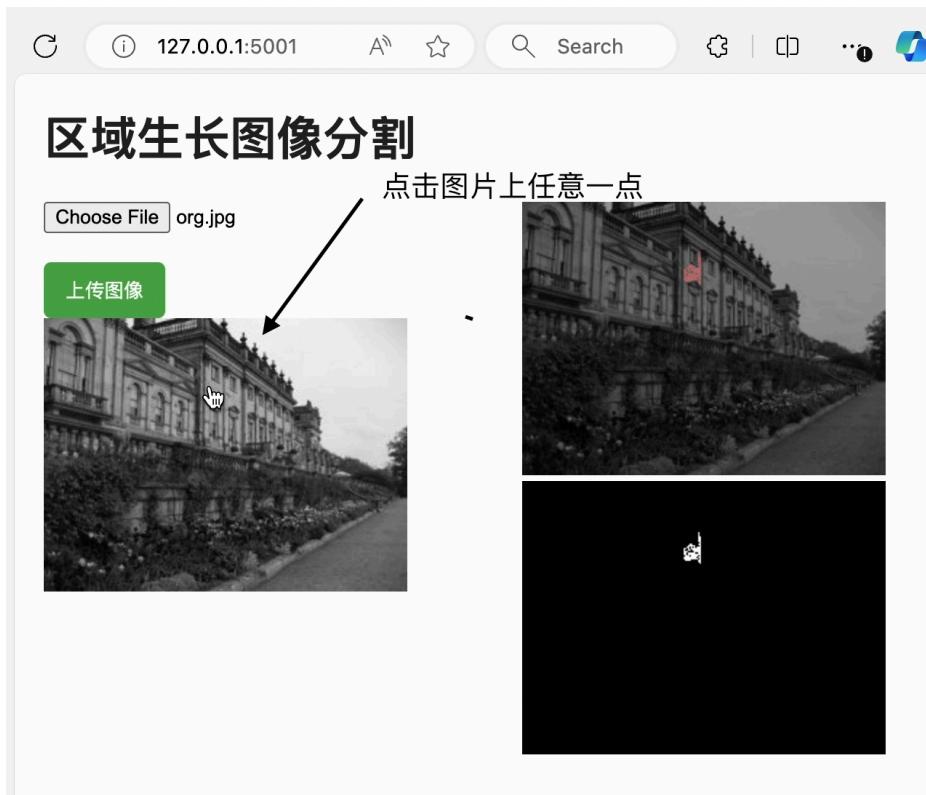


可以看出来轮廓清晰，效果直观。缺点是不够精细。

### 区域生长

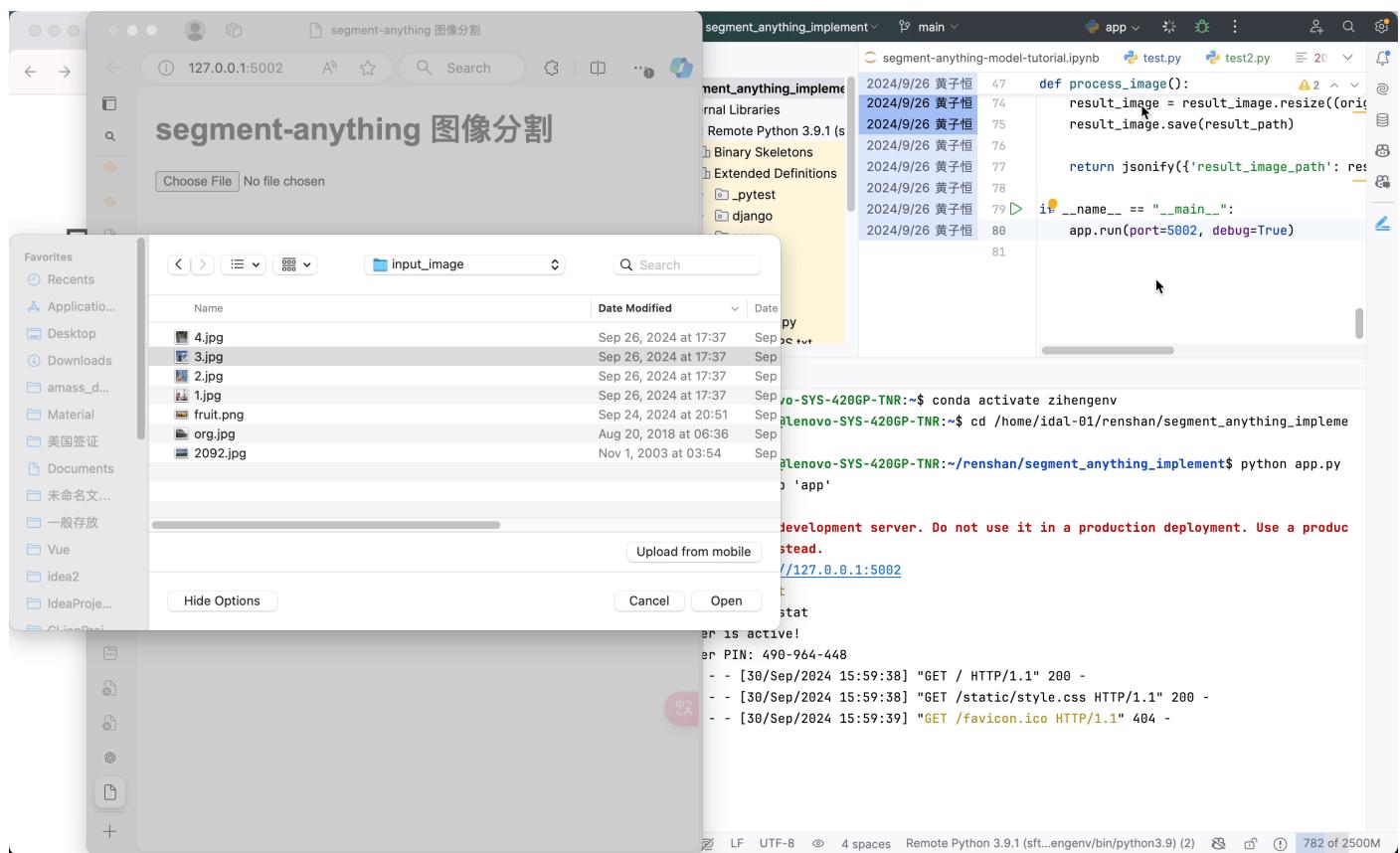
### 结果动图（在 Microsoft Word 中查看或者在文件夹中查看）

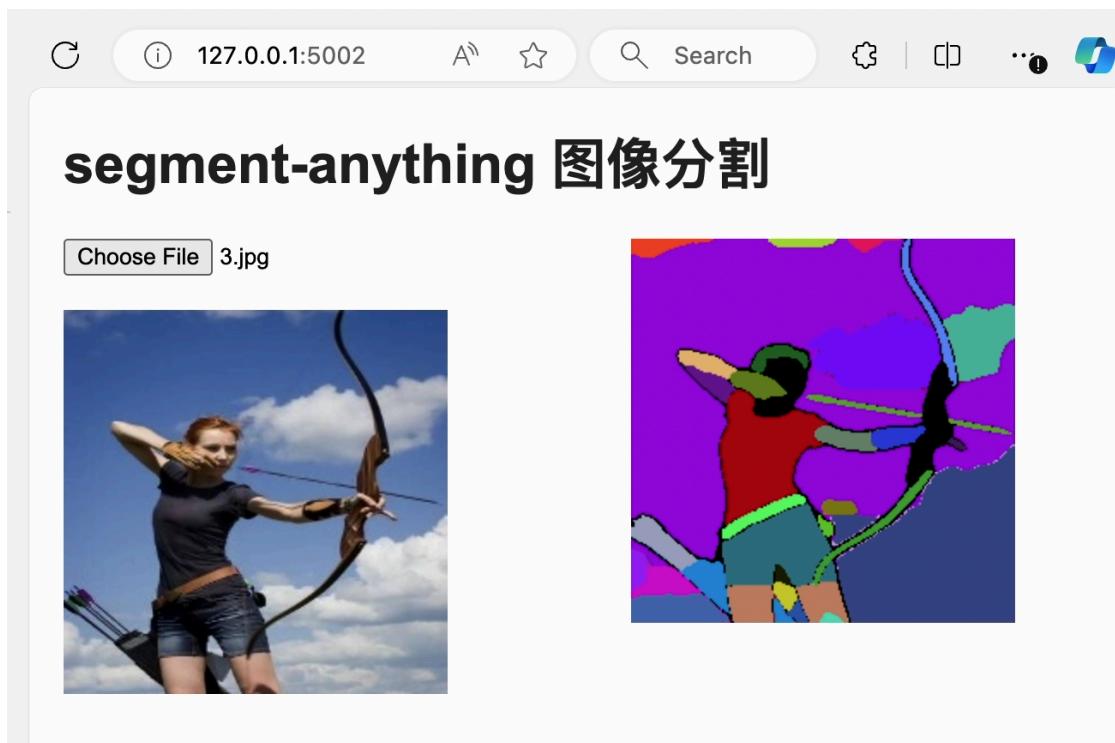
A screenshot of the PyCharm IDE. The project structure on the left shows a "region\_growing" folder containing "images", "templates", and "app.py". The "app.py" file is open in the editor, displaying code for a Flask application. The terminal at the bottom shows the application running on port 5001. The status bar indicates the file is 1527 of 3500M.



## 深度学习 sement-anything 原模型

结果动图（在 Microsoft Word 中查看或者在文件夹中查看）





可以看出来适用于各种任务，泛化性很强；而且分割非常精细（甚至是弓箭都能精准刻画）  
深度学习 **sement-anything** 特定材料数据集微调模型（额外附加代码，因为不在作业范畴内  
所以没有制作可视化软件）

由于 **sement-anything** 原来的模型预训练需要两三百个 A100 训练两三天，所以只有微调任务  
才是可能做出的。本人在特定数据集上微调得到以下效果：



## 定量分析：

由于 segment-anything 和我在特定数据集微调后的 segment-anything2 模型远超前两种 image segmentation 技术，而且属于多物体分割，甚至和二值化分割都不是同一种任务。所以我的定量分析分为：

### 大津阈值化 & 区域生长定量分析和定性分析比较

### segment-anything & 在特定数据集微调后的 segment-anything2 定量分析和定性分析比较。

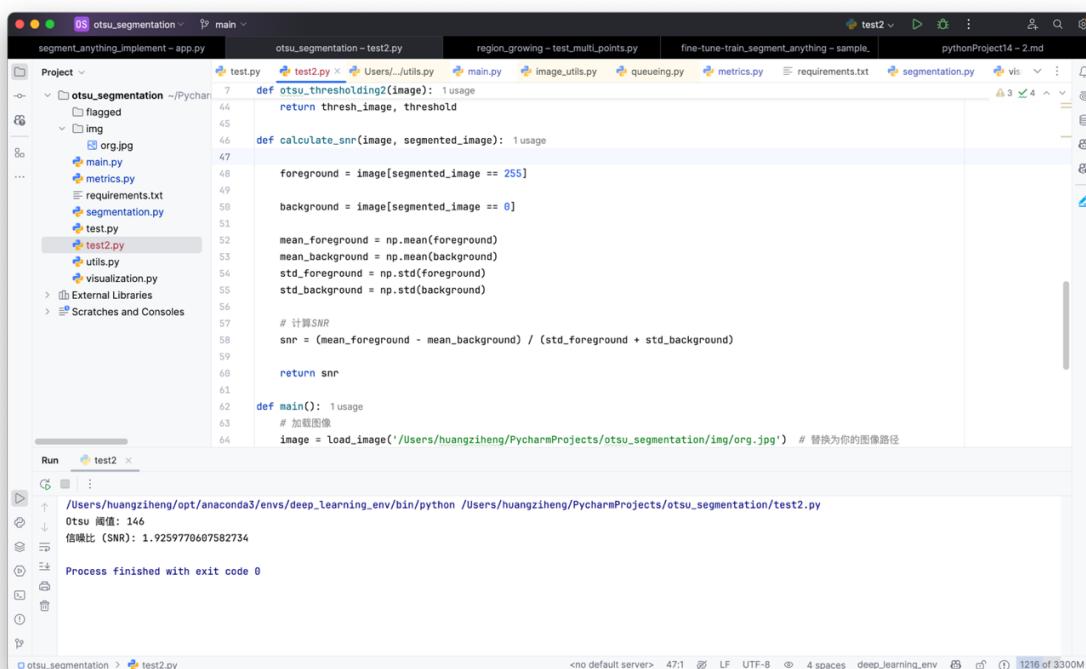
### 大津阈值化 & 区域生长定量分析和定性分析比较

#### 大津阈值化定量分析

大津阈值化结果可以通过信噪比来定量分析，表示前景与背景信号的差异。

$$\text{SNR} = \frac{\mu_F - \mu_B}{\sigma_F + \sigma_B}$$

其中  $\mu_F$ 、 $\mu_B$  分别是前景和背景的平均灰度值， $\sigma_F$ 、 $\sigma_B$  分别是前景和背景的标准差。



The screenshot shows the PyCharm IDE interface with the project 'otsu\_segmentation' open. The code editor displays a Python script named 'test2.py' containing the following functions:

```
def otsu_thresholding2(image): 1usage
    return thresh,image, threshold

def calculate_snr(image, segmented_image): 1usage
    foreground = image[segmented_image == 255]
    background = image[segmented_image == 0]

    mean_foreground = np.mean(foreground)
    mean_background = np.mean(background)
    std_foreground = np.std(foreground)
    std_background = np.std(background)

    # 计算SNR
    snr = (mean_foreground - mean_background) / (std_foreground + std_background)

    return snr

def main(): 1 usage
    # 加载图像
    image = load_image('/Users/huangziheng/PycharmProjects/otsu_segmentation/img/org.jpg') # 替换为你的图像路径
```

The run tab shows the output of running the script:

```
/Users/huangziheng/opt/anaconda3/envs/deep_learning_env/bin/python /Users/huangziheng/PycharmProjects/otsu_segmentation/test2.py
Otsu 阈值: 146
信噪比 (SNR): 1.9259770607582734
Process finished with exit code 0
```

```

segment_anything_implement - app.py          otsu_segmentation - test2.py          region_growing - evaluation2.py          fine-tune-train_segmentAnything - sample          pythonProject14 - 2.md
Project ~Pycharm 46 def calculate_snr(image, seg 3 4 foreground = image[segmented_image == 1]
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75 if __name__ == "__main__":
76     main()
77
    background = image[segmented_image == 0]
    mean_foreground = np.mean(foreground)
    mean_background = np.mean(background)
    std_foreground = np.std(foreground)
    std_background = np.std(background)

    # 计算SNR
    snr = (mean_foreground - mean_background) / (std_foreground + std_background)

    return snr

def main():  usage
    # 加载图像
    image = load_image('/Users/huangziheng/Desktop/建筑.jpg')

    # 应用大津阈值化分割
    segmented_image, threshold = otsu_threshold(image)

    # 计算并输出定量分析结果 (SNR)
    snr = calculate_snr(image, segmented_image)
    print(f"Otsu 阈值: {threshold}")
    print(f"信噪比 (SNR): {snr}")

if __name__ == "__main__":
    main()

```

- **Otsu 阈值:** 146。这意味着在图像中，像素值为146是自动选择的最佳阈值，低于这个值的像素被认为是背景，高于这个值的像素被认为是前景。阈值为146表明在灰度级范围内，这个值能较好地区分图像中的前景和背景。
- **信噪比 (SNR):** 1. 93。这个值反映了前景和背景之间的区分度。通常，SNR越高，分割的效果越好，SNR值超过3–5会表示非常明显的分割效果。SNR值为1. 93表示图像前景与背景的区别还算明显，但不是非常高。如果SNR值过低，可能需要进一步优化分割算法或图像预处理。

## 区域生长定量分析

取不同点数量可以得到不同效果和信噪比。

PyCharm Screenshot showing the execution of `evaluation2.py`. The terminal output shows:

```
/Users/huangziheng/opt/anaconda3/envs/deep_learning_env/bin/python /Users/huangziheng/PycharmProjects/region_growing/evaluation2.py
信噪比 (SNR): 2.9414976884755117
Process finished with exit code 0
```

PyCharm Screenshot showing the execution of `evaluation2.py`. The terminal output shows:

```
/Users/huangziheng/opt/anaconda3/envs/deep_learning_env/bin/python /Users/huangziheng/PycharmProjects/region_growing/evaluation2.py
信噪比 (SNR): 2.961409952346673
Process finished with exit code 0
```

以种子数量 6 为例子，信噪比 (SNR)：2.9414976884755117

以种子数量 200 为例子，信噪比 (SNR)：2.961409952346673

## 总结

从信噪比的角度区域生长定量分析可能占轻微优势，但是从图片的直观角度，大津阈值化更加符合直观体验，效果更好更稳定。对于噪声较多的图像或灰度值变化较大的区域，区域生长可能无法得到很好的分割效果，因此通常需要配合其他预处理技术来提高其鲁棒性。

## segment-anything & 在特定数据集微调后的 segment-anything2 定量分析和定性分析比较

```
itr 99942 Accuracy(Intersection over Union, IoU)= 0.49299125
itr 99943 Accuracy(Intersection over Union, IoU)= 0.4955804
itr 99944 Accuracy(Intersection over Union, IoU)= 0.49217247
itr 99945 Accuracy(Intersection over Union, IoU)= 0.49453534
itr 99946 Accuracy(Intersection over Union, IoU)= 0.49242644
itr 99947 Accuracy(Intersection over Union, IoU)= 0.48880881
itr 99948 Accuracy(Intersection over Union, IoU)= 0.4911947
itr 99949 Accuracy(Intersection over Union, IoU)= 0.4913865
itr 99950 Accuracy(Intersection over Union, IoU)= 0.49839613
itr 99951 Accuracy(Intersection over Union, IoU)= 0.4922897
itr 99952 Accuracy(Intersection over Union, IoU)= 0.4931862
itr 99953 Accuracy(Intersection over Union, IoU)= 0.4942159
itr 99954 Accuracy(Intersection over Union, IoU)= 0.49311166
itr 99955 Accuracy(Intersection over Union, IoU)= 0.4929886
itr 99956 Accuracy(Intersection over Union, IoU)= 0.4945991
itr 99957 Accuracy(Intersection over Union, IoU)= 0.4924145
itr 99958 Accuracy(Intersection over Union, IoU)= 0.48991953
itr 99959 Accuracy(Intersection over Union, IoU)= 0.4915287
itr 99960 Accuracy(Intersection over Union, IoU)= 0.4911688
itr 99961 Accuracy(Intersection over Union, IoU)= 0.49565405
itr 99962 Accuracy(Intersection over Union, IoU)= 0.49139345
itr 99963 Accuracy(Intersection over Union, IoU)= 0.4888689
itr 99964 Accuracy(Intersection over Union, IoU)= 0.4866996
itr 99965 Accuracy(Intersection over Union, IoU)= 0.48465247
itr 99966 Accuracy(Intersection over Union, IoU)= 0.48590635
itr 99967 Accuracy(Intersection over Union, IoU)= 0.4883072
itr 99968 Accuracy(Intersection over Union, IoU)= 0.49513393
itr 99969 Accuracy(Intersection over Union, IoU)= 0.48193112
itr 99970 Accuracy(Intersection over Union, IoU)= 0.48255642
itr 99971 Accuracy(Intersection over Union, IoU)= 0.48184215
itr 99972 Accuracy(Intersection over Union, IoU)= 0.48378496
itr 99973 Accuracy(Intersection over Union, IoU)= 0.48184894
itr 99974 Accuracy(Intersection over Union, IoU)= 0.48194879
itr 99975 Accuracy(Intersection over Union, IoU)= 0.48186112
itr 99976 Accuracy(Intersection over Union, IoU)= 0.48089742
itr 99977 Accuracy(Intersection over Union, IoU)= 0.47952328
itr 99978 Accuracy(Intersection over Union, IoU)= 0.47677948
itr 99979 Accuracy(Intersection over Union, IoU)= 0.47916864
```

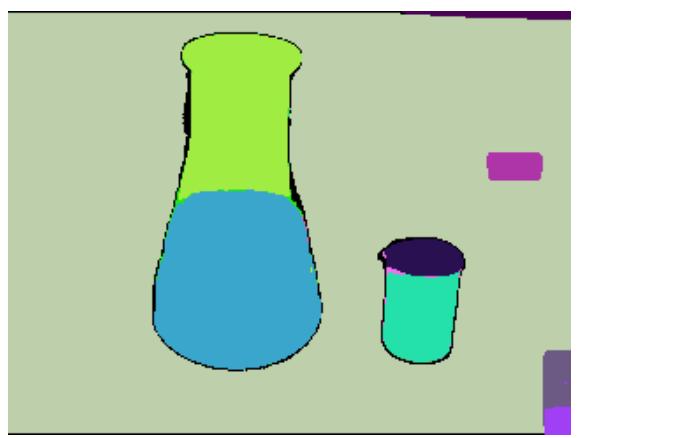
## 定量比较：

segment-anything2 微调后 Accuracy Intersection over Union. IoU= 0.67916864 远高于微调前的 0.1

备注：因为是微调任务 itr epoch 设置在 1000 以内就能快速收敛到 0.6，没有必要使用如此多的 itr epoch. (为简单起见使用的是一张图片一次 epoch)。没有过拟合迹象。

## 定性比较：

微调前 segment-anything	微调后 segment-anything2
----------------------	-----------------------

	
基本完成任务。	可以看出来更精准地分割不同材料和物体。

结论：segment-anything 模型泛化性特别强，但是在专业领域，微调后的 segment-anything 模型效果更佳。