山东大学

崇新学堂

2023－2024 学年第一学期

实 验 报 告

课程名称：信息基础 II

专 业 班 级：崇新学堂 21 级

学 生 姓 名：刘浩

个 人 学 号：202120120312

实验四：Selective search

一、 实验要求

选择性搜索是区域规划的方法，常用于目标检测，本实验要求自己编程实现 Selective Search 算法；

二、 实验原理

其实算法的主要步骤在实验四的 PPT 中已经说明：

Algorithm 1: Hierarchical Grouping Algorithm

DontPrintSemicolon **Input**: (colour) image
Output: Set of object location hypotheses L

Obtain initial regions $R = \{r_1, \cdots, r_n\}$ using Felzenszwalb and Huttenlocher (2004) Initialise similarity set $S = \emptyset$;
foreach *Neighbouring region pair* (r_i, r_j) **do**
  Calculate similarity $s(r_i, r_j)$;
  $S = S \cup s(r_i, r_j)$;

while $S \neq \emptyset$ **do**
  Get highest similarity $s(r_i, r_j) = \max(S)$;
  Merge corresponding regions $r_t = r_i \cup r_j$;
  Remove similarities regarding r_i : $S = S \setminus s(r_i, r_*)$;
  Remove similarities regarding r_j : $S = S \setminus s(r_*, r_j)$;
  Calculate similarity set S_t between r_t and its neighbours;
  $S = S \cup S_t$;

算法 1：层次分组算法

Input：（彩色）图像
Output：目标定位假设 L 的集合（区域集合）
使用 Fel&Hut (2004) 得到初始区域 $R = \{r_1, \ldots, r_n\}$
初始化相似度集 $S = \Phi$
For each 相邻的区域对 (r_i, r_j) do
  计算(r_i, r_j)的相似度$s(r_i, r_j)$
  $S = S \cup s(r_i, r_j)$
End
While $S \neq \Phi$ do
  得到最高的相似度值：$s(r_i, r_j) = \max(S)$
  对相应区域进行合并：$r_t = r_i \cup r_j$
  从S里面移除所有关于区域 r_i 的相似度：$S = S \setminus s(r_i, r_*)$
  从S里面移除所有关于区域 r_j 的相似度：$S = S \setminus s(r_j, r_*)$
  计算r_t与它相邻区域的相似度得到相似度集S_t
  更新相似度集：$S = S \cup S_t$
  更新区域集：$R = R \cup r_t$

图 1 算法步骤

在原论文中考虑了四个方面的相似度，分别是空间，纹理，尺度，空间交叠，并将这四个相似度以线性组合的方式综合在一起，作为最终被使用的相似度，即：

$$S(r_i, r_j) = \alpha_1 s_{colour}(r_i, r_j) + \alpha_2 s_{texture}(r_i, r_j) + \alpha_3 s_{size}(r_i, r_j) + \alpha_4 s_{fill}(r_i, r_j) \quad (1)$$

其中(c_i, c_j)是某个区域的颜色直方图向量，颜色相似度：

$$s_{colour}(r_i, r_j) = \sum_{k=1}^{n} \min(c_i^k, c_j^k) \quad (2)$$

(t_i, t_j)是某个区域的纹理直方图向量，纹理相似度：

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} \min(t_i^k, t_j^k) \quad (3)$$

尺度相似度(合并比较小的区域)：

$$s_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(im)} \quad (4)$$

空间交叠相似度(用于优先合并被包含进其他区域的区域):

$$s_{fill}(r_i, r_j) = 1 - \frac{\text{size}(BB_{ij}) - \text{size}(r_i) - \text{size}(r_j)}{\text{size}(im)} \quad (5)$$

将相似度代入上述算法步骤中即可

三、 实验步骤

对于 Selective search 的实现过程实际上是比较复杂的，此处我主要结合了实验原理和库函数的定义进行代码的编写。

首先是生成原始区域集的函数，用 Felzenszwalb 图像分割算法：

```
# 生成原始区域集的函数，用 Felzenszwalb 图像分割算法，每个区域都有一个编号
def generate_segments(image, scale, sigma, min_size):
    im_mask = felzenszwalb(img_as_float(image), scale=scale, sigma=sigma, min_size=min_si
ze)
    im_orig = np.append(image, np.zeros(image.shape[:2])[:, :, np.newaxis], axis=2)
    im_orig[:, :, 3] = im_mask
    return im_orig
```

计算颜色直方图的函数：

```
# 计算颜色直方图
def calculate_color_histogram(image):
    BINS = 25
    hist = np.array([])
    for color_channel in (0, 1, 2):
        c = image[:, color_channel]
        hist = np.concatenate([hist] + [np.histogram(c, BINS, (0.0, 255.0))[0]])
    hist = hist / len(image)
    return hist
```

计算纹理直方图的函数：

```
# 计算纹理直方图
def calculate_texture_histogram(image):
    BINS = 10
    hist = np.array([])
    for color_channel in (0, 1, 2):
        fd = image[:, color_channel]
        hist = np.concatenate([hist] + [np.histogram(fd, BINS, (0.0, 1.0))[0]])
    hist = hist / len(image)
    return hist
```

然后是纹理梯度的计算：

```
# 计算纹理梯度
def calculate_texture_gradient(image):
    ret = np.zeros((image.shape[0], image.shape[1], image.shape[2]))
    for color_channel in (0, 1, 2):
        ret[:, :, color_channel] = local_binary_pattern(image[:, :, color_channel], 8, 1.
0)
    return ret
```

下面是区域的尺寸，颜色和纹理特征的提取，返回的是包含区域信息的字典：

```
# 提取区域的尺寸，颜色和纹理特征
def extract_regions(image):
    R = {}
    hsv = rgb2hsv(image[:, :, :3])
    for y, row in enumerate(image):
        for x, (r, g, b, l) in enumerate(row):
            if l not in R:
                R[l] = {"min_x": float('inf'), "min_y": float('inf'), "max_x": 0, "max_y"
: 0, "labels": [l]}
            if R[l]["min_x"] > x:
                R[l]["min_x"] = x
            if R[l]["min_y"] > y:
                R[l]["min_y"] = y
            if R[l]["max_x"] < x:
                R[l]["max_x"] = x
            if R[l]["max_y"] < y:
                R[l]["max_y"] = y
    tex_grad = calculate_texture_gradient(image)
    for k, v in list(R.items()):
        masked_pixels = hsv[:, :, :][image[:, :, 3] == k]
        R[k]["size"] = len(masked_pixels) // 4
        R[k]["hist_c"] = calculate_color_histogram(masked_pixels)
        R[k]["hist_t"] = calculate_texture_histogram(tex_grad[:, :][image[:, :, 3] == k])

    return R
```

下面是寻找邻居的函数，通过计算每个区域与其余的所有区域是否有相交来判断是否邻居，返回所有的邻居列表

```
# 找邻居的函数
def extract_neighbours(regions):
    def intersect(a, b):
        return (a["min_x"] < b["min_x"] < a["max_x"] and a["min_y"] < b["min_y"] < a["max
_y"]) or \
               (a["min_x"] < b["max_x"] < a["max_x"] and a["min_y"] < b["max_y"] < a["max
_y"]) or \
               (a["min_x"] < b["min_x"] < a["max_x"] and a["min_y"] < b["max_y"] < a["max
_y"]) or \
               (a["min_x"] < b["max_x"] < a["max_x"] and a["min_y"] < b["min_y"] < a["max
_y"])
    R = list(regions.items())
    neighbours = []
    for cur, a in enumerate(R[:-1]):
        for b in R[cur + 1:]:
            if intersect(a[1], b[1]):
                neighbours.append((a, b))
    return neighbours
```

下面是合并两个区域的函数，返回的是合并之后的区域：

```
# 合并两个区域的函数
def merge_regions(r1, r2):
    new_size = r1["size"] + r2["size"]
    rt = {
        "min_x": min(r1["min_x"], r2["min_x"]),
        "min_y": min(r1["min_y"], r2["min_y"]),
        "max_x": max(r1["max_x"], r2["max_x"]),
        "max_y": max(r1["max_y"], r2["max_y"]),
        "size": new_size,
        "hist_c": (r1["hist_c"] * r1["size"] + r2["hist_c"] * r2["size"]) / new_size,
        "hist_t": (r1["hist_t"] * r1["size"] + r2["hist_t"] * r2["size"]) / new_size,
        "labels": r1["labels"] + r2["labels"]
    }
    return rt
```

下面就需要计算颜色相似度，纹理相似度，尺寸相似度和填充相似度了：

```
# 计算颜色相似度
def sim_color(r1, r2):
    return sum([min(a, b) for a, b in zip(r1["hist_c"], r2["hist_c"])])

# 计算纹理相似度
def sim_texture(r1, r2):
    return sum([min(a, b) for a, b in zip(r1["hist_t"], r2["hist_t"])])

# 计算尺寸相似度
def sim_size(r1, r2, imsize):
    return 1.0 - (r1["size"] + r2["size"]) / imsize

# 计算填充相似度
def sim_fill(r1, r2, imsize):
    bbsize = (max(r1["max_x"], r2["max_x"]) - min(r1["min_x"], r2["min_x"])) * \
             (max(r1["max_y"], r2["max_y"]) - min(r1["min_y"], r2["min_y"]))
    return 1.0 - (bbsize - r1["size"] - r2["size"]) / imsize
```

下面我们就可以完成两个区域相似度的计算了：

```
# 计算两个区域的相似度
def calc_similarity(r1, r2, imsize):
    return (sim_color(r1, r2) + sim_texture(r1, r2)
            + sim_size(r1, r2, imsize) + sim_fill(r1, r2, imsize))
```

最后是 selective search 主函数的构造，与 pip install 的 selective search 类似，我们同样传入三个参数，image: 输入图像，scale: 分割的集群程度，sigma: 高斯核的大小 min_size: 最小区域像素点个数：

```
# selective_search 主函数
def selective_search(image, scale=1.0, sigma=0.8, min_size=50):
    assert image.shape[2] == 3, "判读输入是不是三通道的图片，如果不是的话不可"
    img = generate_segments(image, scale, sigma, min_size)
    if img is None:
        return None, {}

    imsize = img.shape[0] * img.shape[1]
    R = extract_regions(img)
    neighbours = extract_neighbours(R)
    S = {}
    for (ai, ar), (bi, br) in neighbours:
        S[(ai, bi)] = calc_similarity(ar, br, imsize)

    while S != {}:
        i, j = sorted(S.items(), key=lambda x: x[1])[-1][0]
        t = max(R.keys()) + 1.0
        R[t] = merge_regions(R[i], R[j])
        key_to_delete = [k for k, v in S.items() if i in k or j in k]
        for k in key_to_delete:
            del S[k]
        for k in [a for a in key_to_delete if a != (i, j)]:
            n = k[1] if k[0] in (i, j) else k[0]
            S[(t, n)] = calc_similarity(R[t], R[n], imsize)

    regions = []
    for k, r in R.items():
        regions.append({
            'rect': (r['min_x'], r['min_y'], r['max_x'] - r['min_x'], r['max_y'] - r['min
_y']),
            'size': r['size'],
            'labels': r['labels']
        })

    return img, regions
```

下面我们就可以导入图片进行分割了，我采用 opencv 库导入我们需要分割的图片，由于 cv 库导入图片的时候读取 RGB 格式而存储会采用 BGR 因此我需要进行格式的转化：

```
#     cv2 默认读取的 RGB 图像为 BGR 存储格式，此处需要转化为 RGB
    img = cv2.imread('test.jpg')
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

下面可以进行 Selective search 的操作，返回的是一个字典，包含利用实验原理寻找到的矩形坐标以及长和宽(x,y,w,h),还有 size 和 label，根据我们的需要筛选指定尺寸的矩形框即可：

```
img_lbl, regions = selectivesearch.selective_search(img, scale=500, sigma=0.8, min_size=1
00)
# 从 regions 中筛选出具有足够大小的候选区域
region_rect = [r['rect'] for r in regions if r['size'] >= 5000]
```

另外，为了避免一些扭曲的框的影响，我对筛选出来的矩形再次进行了过滤：

```
def filter_regions(regions):
    candidates = set()
    for r in regions:
        # 排除重复的候选区
        if r['rect'] in candidates:
            continue
        # 排除小于 2000 pixels 的候选区域(并不是 bounding box 中的区域大小)
        if r['size'] < 2000:
            continue
        # 排除扭曲的候选区域边框  即只保留近似正方形的
        x, y, w, h = r['rect']
        if w / h > 1.2 or h / w > 1.2:
```

```
            continue
        candidates.add(r['rect'])
    return candidates
```

然后我定义了一个可视化函数，用来绘制标注过矩形框的图片和标注当前使用的参数值：

```
def plot(image, rectangles, scale, sigma, min_size):
    fig, ax = plt.subplots(figsize=(8, 8))
    img_draw = image.copy()

    for x, y, w, h in rectangles:
        rect = mpatches.Rectangle(
            (x, y), w, h, fill=False, edgecolor='red', linewidth=2)
        ax.add_patch(rect)

    text = f"Scale: {scale}, Sigma: {sigma}, Min Size: {min_size}"
    text_x = (img_draw.shape[1] - len(text) * 7) / 2  # 计算使文本居中的 x 坐标
    ax.text(text_x, 10, text, fontsize=12, color='black', bbox=dict(facecolor='white', al
pha=0.7), va='bottom', ha='left')

    ax.imshow(img_draw)
    ax.axis('off')
    plt.show()
```

至此实验可以实现的基本的图像分割效果，此处 selective search 的参数解释如下：

参数	含义
img	输入的图像,RGB 格式
scale	指定图像分割时用于生成不同大小区域的尺度参数
sigma	控制高斯滤波的参数。高斯滤波用于图像分割前的预处理，它有助于平滑图像以减小噪声
min_size	最小区域像素点个数

四、 实验结果

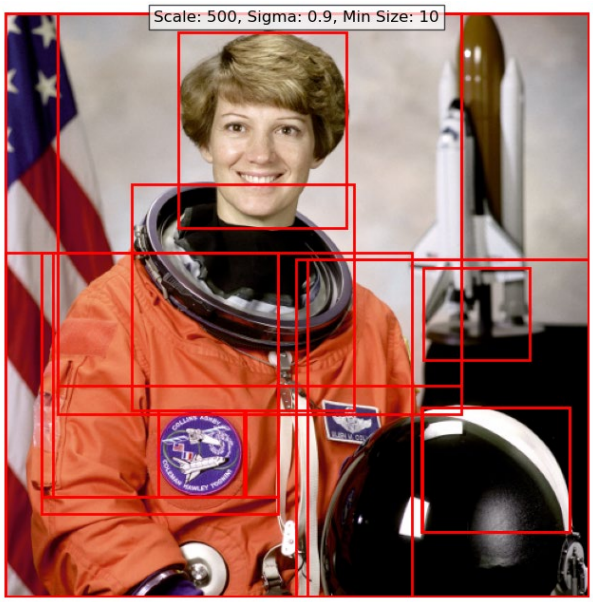
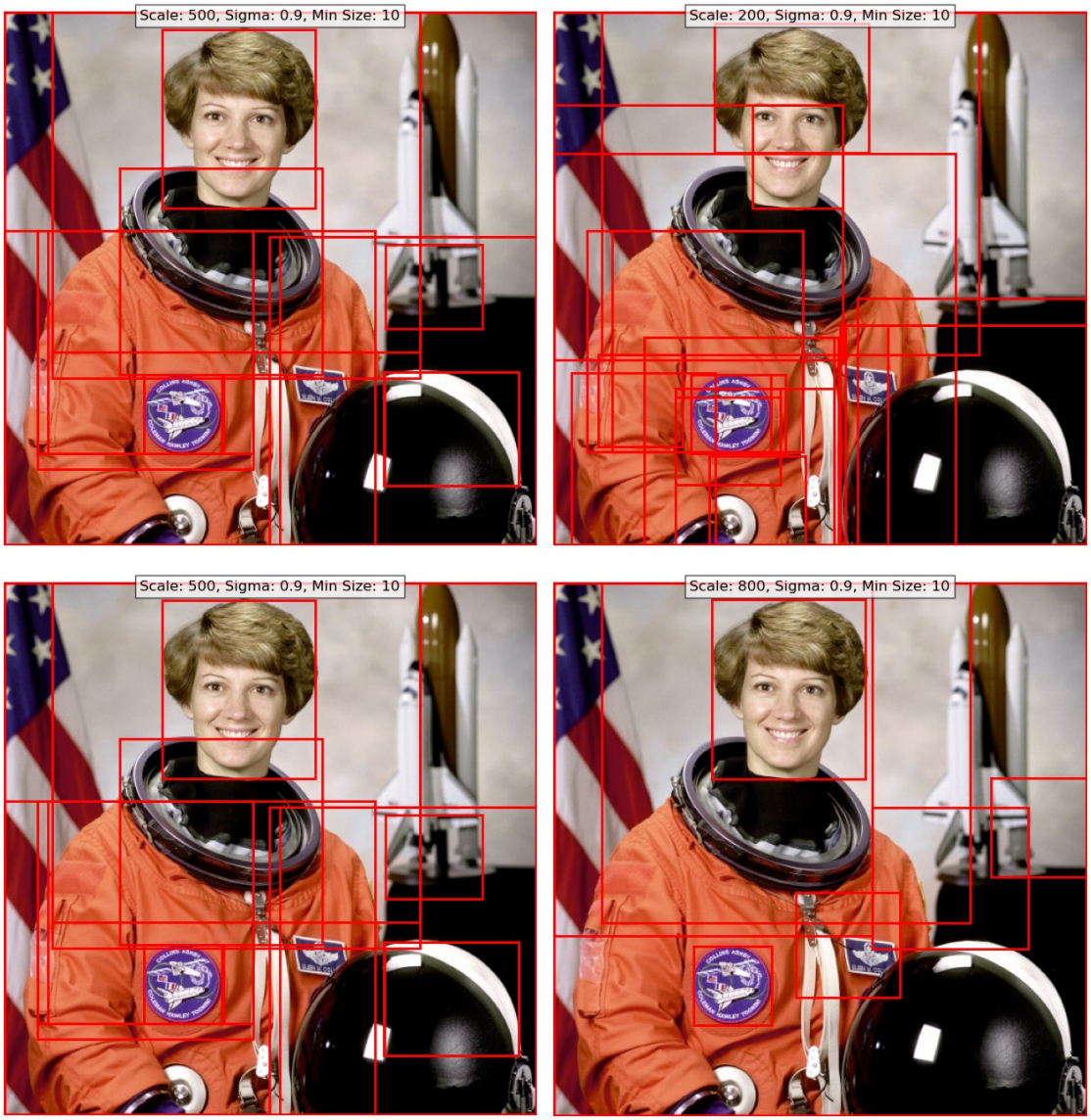


图 2 对宇航员图片进行分割效果

五、 实验探究

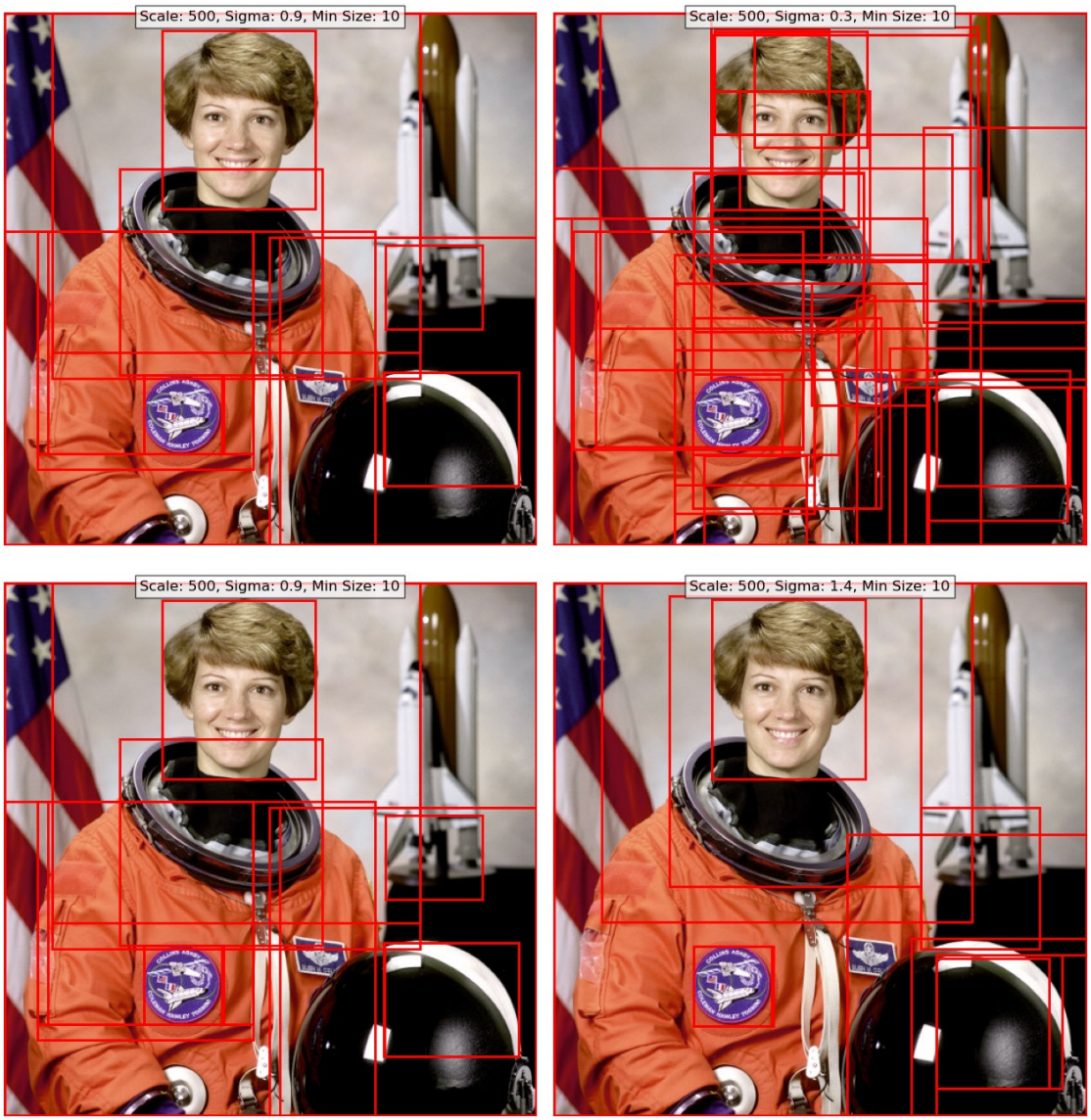
本实验中我主要探究了这几个参数对实验结果的影响:

- 改变 scale



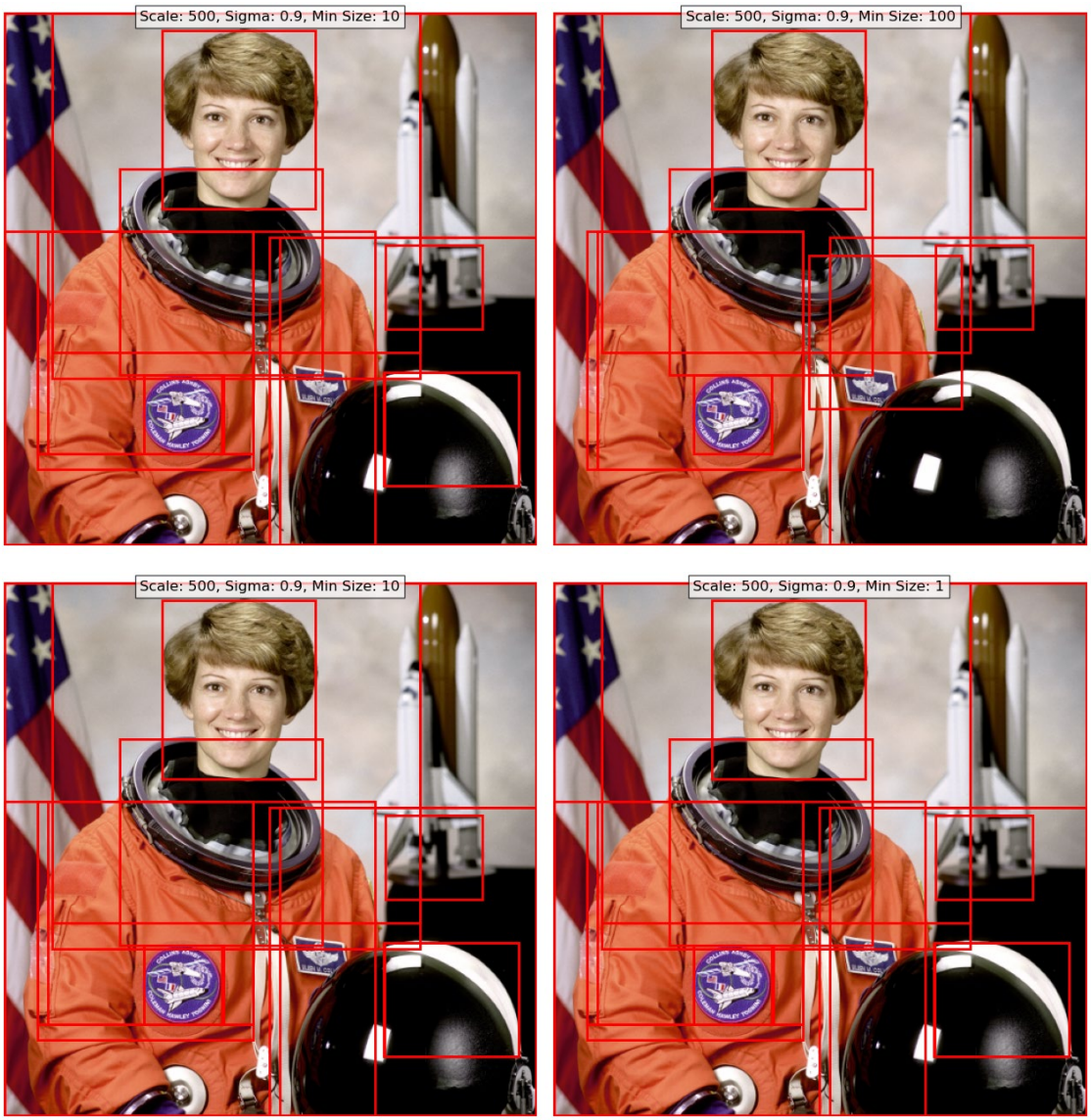
结论：scale 增大会导致初始的框减小，整体保留下来的框大小变大，个数变少，scale 减小同理，保留下来的框尺寸变小，个数增多

- 改变 Sigma



结论：Sigma 增大会增强图像的滤波效果，保留的矩形框个数会减少；Sigma 减小会降低图像的滤波效果，保留的矩形框的个数会增多

- 改变 Min_size



结论:min_size 是最小像素点个数的判断，很直观的会影响计算的时间，将 Min_size 改成 1 后，程序运行时间由 3s 上升至 7s

六、 实验感想

Selective search 的原理其实是比较清晰的，但是动手实现确实不是一件容易的事情。在本实验中我参考了[2]中源代码的详细注释，对原理才更加清晰明了，才得以实现，在实验中我也探究了这几个参数各自的作用，学会了如何正确使用 Selective search。

七、 参考资料

[1]. http://t.csdnimg.cn/23hQF
[2]. http://t.csdnimg.cn/Nb1IY