计算机视觉实验

期末大作业

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 问题描述

1.1 Seam Carving

结合 Lecture 6 Resizing 的 Seam Carving 算法，设计并实现前景保持的图像缩放，前景由 gt 文件夹中对应的标注给定。要求使用“Forward Seam Removing ”机制 X Y 方向均要进行压缩。压缩比例视图像内容自行决定（接近 1-前景区域面积/2* 图像面积）即可。每一位同学从各自的测试子集中任选两张代表图，将每一步的 seam removing 的删除过程记录，做成 gif 动画格式提交测试子集的其余图像展示压缩后的图像结果。

1.2 Graph-Based Image Segmentation

结合 Lecture 7 Segmentation 内容及参考文献，实现基于 Graph-based image segmentation 方法（可以参考开源代码，建议自己实现），通过设定恰当的阈值将每张图分割为 50 ∼ 70 个区域，同时修改算法要求任一分割区域的像素个数不能少于 50 个（即面积太小的区域需与周围相近区域合并）。结合 GT 中给定的前景 mask，将每一个分割区域标记为前景（区域 50% 以上的像素在 GT 中标为 255）或背景 50% 以上的像素被标为 0 。区域标记的意思为将该区域内所有像素置为 0 或 255。要求对测试图像子集生成相应处理图像的前景标注并计算生成的前景 mask 和 GT 前景 mask 的 IOU 比例。假设生成的前景区域为 $R1$ 该图像的 GT 前景区域为 $R2$，则 IOU $= \frac{R1 \cap R2}{R1 \cup R2}$。

1.3 Visual Bag of Words

从训练集中随机选择 200 张图用以训练，对每一张图提取归一化 RGB 颜色直方图（$8*8*8 = 512$ 维），同时执行问题 2 对其进行图像分割，（分割为 50 ∼ 70 个区域），对得到的每一个分割区域提取归一化 RGB 颜色直方图特征（维度为 $8*8*8 = 512$），将每一个区域的颜色对比度特征定义为区域颜色直方图和全图颜色直方图的拼接，因此区域颜色区域对比度特征的维度为 $2*512 = 1024$ 维，采用 PCA 算法对特征进行降维取前 20 维。利用选择的 200 张图的所有区域（每个区域 20 维特征）构建 visual bag of words dictionary （参考 Lecture 12. Visual Bag of Words 内容），单词数（聚类数）设置为 50 个，visual word 的特征设置为聚簇样本的平均特征，每个区域降维后颜色对比度特征 20 维）和各个 visual word 的特征算点积相似性得到 50 个相似性值形成 50 维。将得到的 50 维特征和前面的 20 维颜色对比度特征拼接得到每个区域的 70 维特征表示。根据问题 2 ，每个区域可以被标注为类别 1 （前景：该区域 50% 以上像素为前景）或 0 （背景：该区域 50% 以上像素为背景），选用任意分类算法（SVM，Softmax，随机森林，KNN 等）进行学习得到分类模型。最后在测试集上对每一张图的每个区域进行测试（将图像分割为 50 ∼ 70 个区域，对每个区域提取同样特征并分类）根据测试图像的 GT 分析测试集区域预测的准确率。

2 实验原理

2.1 Seam Carving

Seam Carving 是一种内容感知的图像重定位算法。与传统图像缩放算法（如剪裁、几何缩放等）相比，Seam Carving 可以在保留图像主要内容不被破坏干扰的同时，通过删除（或添加）不重要的内容像

素来改变图像的尺寸大小。

一般来说，图像重定位算法包括两个部分，分别是重要特征的检测和图像大小的调整。Seam Carving 算法的流程类似，首先检测图像中的重要部分、标注每个像素的重要性；然后，算法试图从图像中获取一条从上到下（或从左到右）的包含最不重要像素的八连通路径，通过删除（或复制）路径来实现图像尺寸的调整。

2.1.1 能量函数和 seam 的定义

我们首先介绍基于反向能量的传统 Seam Carving 算法。前面我们提到，算法的第一步是要判断图片中每个像素的重要性。我们有很多种方法来判断一个像素是否重要，而比较通用、易于实现的方法就是定义一个能量函数，其输入是一幅图像，输出是图像中每个像素的能量值（重要性）。当一个像素位于图像主要内容的组成部分时，其应该具有较大的能量值，反之则能量值较小。能量函数有很多种形式，最简单的形式之一就是用每个像素在图像的梯度绝对值来代表能量值。这是因为图像中的重要部分通常有丰富的细节，从而梯度绝对值较大；而不重要的部分（一般为背景）缺乏变化，梯度绝对值较小。因此，假设像素矩阵为 $\boldsymbol{I}$，我们可以通过能量函数 $e()$ 得到对应的能量图 $\boldsymbol{E}$：

$$\boldsymbol{E} = e(\boldsymbol{I}) = |\frac{\partial \boldsymbol{I}}{\partial x}| + |\frac{\partial \boldsymbol{I}}{\partial y}| \tag{1}$$

其中，$\boldsymbol{E}(i,j)$ 即为像素 $\boldsymbol{I}(i,j)$ 的能量值。

得到能量图后，我们就可以获取图像的 seam。seam 是一条从左到右（从上到下）的、横（纵）穿整个图片的八连通像素路径。假设 $\boldsymbol{I}$ 是一个大小为 $m \times n$ 的图片，左上角的像素为原点 $(0,0)$、垂直向下为 x 轴方向、水平向右为 y 轴方向，那么一条水平 seam 的定义如式2所示：

$$\begin{aligned} &\boldsymbol{S}^{\mathrm{V}} = \{s_i^{\mathrm{V}}\}_{i=0}^{m-1} = \{(i, y(i))\}_{i=0}^{m-1} \\ &texts.t. \forall i, |y(i) - y(i-1)| \leq l \end{aligned} \tag{2}$$

其中 y 是一个映射函数，$y : [0, \cdots, m-1] \to [0, \cdots, n-1]$。同样，垂直 seam 的定义如3所示：

$$\begin{aligned} &\boldsymbol{S}^{\mathrm{H}} = \{s_j^{\mathrm{H}}\}_{j=0}^{n-1} = \{(x(j), j)\}_{j=0}^{n-1} \\ &\text{s.t. } \forall j, |x(j) - x(j-1)| \leq l \end{aligned} \tag{3}$$

其中 $x : [0, \cdots, n-1] \to [0, \cdots, m-1]$。此外，式2和式3中的 l 用于控制 seam 中两个相邻像素之间的最大距离，$l = 0$ 时 $\boldsymbol{S}$ 为一条直线，而 $l = 1$ 时 $\boldsymbol{S}$ 即为八连通路径。后续内容我们默认 $l = 1$。

给定 seam $\boldsymbol{S}$，我们就可以得到它的 cost：

$$C(\boldsymbol{S}) = \sum_{s \in \boldsymbol{S}} e(\boldsymbol{I}(s)) \tag{4}$$

因此，我们需要通过最小化 $C(\boldsymbol{S})$ 来获得 optimal seam：

$$\boldsymbol{S}^* = \underset{\boldsymbol{S}}{\operatorname{argmin}} C(\boldsymbol{S}) \tag{5}$$

为了获得 $\boldsymbol{S}^*$，我们可以使用动态规划。以垂直 seam 为例，我们可以设置状态：

$$M(i,j) = \boldsymbol{E}(i,j) + \min\{M(i-1, j-1), M(i-1, j), M(i-1, j+1)\} \tag{6}$$

从上到下按顺序推导后，最终我们可以得到 $C(\boldsymbol{S}^{\mathrm{V}*}) = \min\limits_{j \in [0, n-1]} M(m-1, j)$。我们可以倒推获得这个 seam 的组成像素，从而进行下一步操作。水平 optimal seam 的获取方法类似。

2.1.2 前向能量和反向能量

2.1.1介绍的是传统的 Seam Carving 算法，使用的是反向能量。然而，这种方法存在缺陷，其只考虑了当前图像的最小能量像素，却没有考虑更新后图像的变化情况，例如去除一些像素会导致图像出现伪影。因此，学者们提出了前向能量的概念。

与反向能量不同，前向能量来源于相邻像素之间的邻边、而不是像素本身，其取决于这两个像素的值。当两个像素之间差异越大时，我们说这个邻边的能量越大。当我们从图像中删除一条 seam 时，图像中的边会发生变化，旧像素的删除会丢失一些旧边、原先不相邻的像素相邻后会产生新的边。一般来说，我们希望新产生的邻边的能量越小越好，因为当两个像素相邻后如果差异过大（即邻边的能量太大），会产生明显的伪影；此外，丢失的旧邻边能量也越小越好，因为原先相邻的像素差异大时、说明这个区域的细节丰富、可能是重要部分。为了进一步简化，我们只考虑新加入的边、不考虑丢失的边，这也是“前向”的由来。

现在，seam $\boldsymbol{S}$ 的 cost$C(\boldsymbol{S})$ 定义为从图像中删除 $\boldsymbol{S}$ 的像素后，新产生的邻边的能量和。我们通过一个简单的公式 $|\boldsymbol{I}(i,j)-\boldsymbol{I}(x,y)|$ 来计算像素 (i,j) 和 (x,y) 之间邻边的能量。我们同样可以通过动态规划的方式求得 $\boldsymbol{S}^*$。以垂直 seam 为例，我们为每个像素定义状态 $M(i,j)$，表示当前 seam 终点为 (i,j) 时的最小能量。如图1所示，对于像素 (i,j)，其上一个像素有三种不同情况：

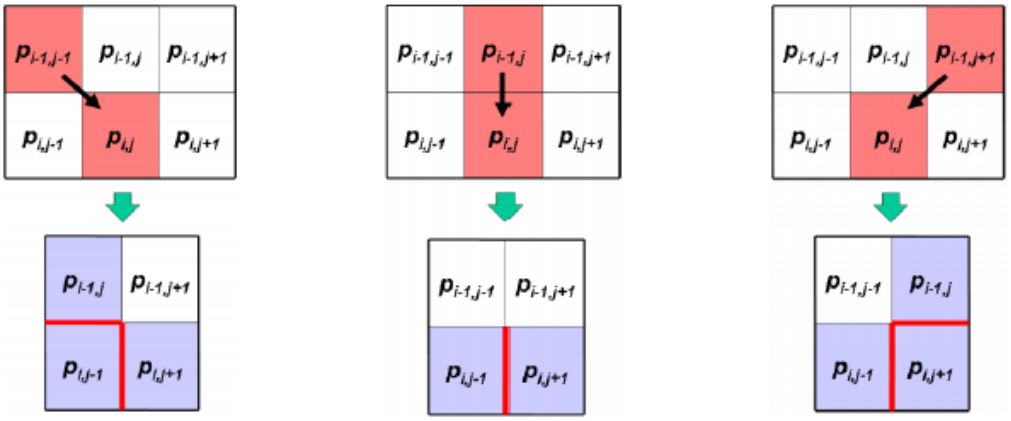

图 1: 计算 $M(i,j)$ 的三种情况

对应的新邻边能量如下所示：

$$\begin{aligned} F_L(i,j) &= |\boldsymbol{I}(i,j-1)-\boldsymbol{I}(i-1,j)| + |\boldsymbol{I}(i,j+1)-\boldsymbol{I}(i,j-1)| \\ F_U(i,j) &= |\boldsymbol{I}(i,j+1)-\boldsymbol{I}(i,j-1)| \\ F_R(i,j) &= |\boldsymbol{I}(i,j+1)-\boldsymbol{I}(i,j-1)| + |\boldsymbol{I}(i,j+1)-\boldsymbol{I}(i-1,j)| \end{aligned} \tag{7}$$

因此我们可以得到转移公式：

$$M(i,j) = \min \begin{cases} M(i-1,j-1)+F_L(i,j) \\ M(i-1,j)+F_U(i,j) \\ M(i-1,j+1)+F_R(i,j) \end{cases} \tag{8}$$

从上到下按顺序推导后，最终我们可以得到 $C((\boldsymbol{S}^{\mathrm{V}})^*) = \min\limits_{j\in[0,n-1]} M(m-1,j)$。我们同样可以倒推得到组成 $()\boldsymbol{S}^{\mathrm{V}})^*$ 的各个像素。水平 optimal seam 的获取方法类似。

2.1.3 基于动态规划的最优顺序

在上述介绍中，我们只介绍了寻找并删除一条 seam 的方法。事实上，当我们想要缩小图片的宽或高时，我们就是执行以下操作：

(1) 从当前图像 $\boldsymbol{I}$ 获得 $\boldsymbol{S}^*$；

(2) $\boldsymbol{I} \leftarrow \boldsymbol{I} - \boldsymbol{S}^*$;

(3) 重复 (1) 和 (2)，直到 $\boldsymbol{I}$ 的宽（高）缩小到规定值；

然而，当我们想要将一个大小为 $m \times n$ 的图片缩小至 $m' \times n'$（$m > m', n > n'$）时，我们需要确定一个最优的删除顺序，使得 cost 最小。

这个问题同样可以通过动态规划来解决。我们定义 $\boldsymbol{I}_{i,j}$ 表示删除 i 条垂直 seam 和 j 条水平 seam 后的图片，而状态 $T(i,j)$ 表示得到 $\boldsymbol{I}(i,j)$ 的 cost；那么，我们首先从 $\boldsymbol{I}_{i-1,j}$ 获取最优垂直 seam $(\boldsymbol{S}^{\mathrm{H}})^*$、从 $\boldsymbol{I}_{i,j-1}$ 获取最优水平 seam $(\boldsymbol{S}^{\mathrm{V}})^*$。然后我们得到如下更新公式：

$$T(i,j) = \min \begin{cases} T(i-1,j) + C((\boldsymbol{S}^{\mathrm{H}})^*) \\ T(i,j-1) + C((\boldsymbol{S}^{\mathrm{V}})^*) \end{cases} \tag{9}$$

若 $T(i,j) = T(i-1,j) + C((\boldsymbol{S}^{\mathrm{H}})^*)$，那么 $\boldsymbol{I}_{i,j} \leftarrow \boldsymbol{I}_{i-1,j} - (\boldsymbol{S}^{\mathrm{H}})^*$；反之 $\boldsymbol{I}_{i,j} \leftarrow \boldsymbol{I}_{i,j-1} - (\boldsymbol{S}^{\mathrm{V}})^*$。最终 $\boldsymbol{I}_{m-m',n-n'}$ 即为所需图片。

2.2 Graph-Based Image Segmentation

Graph-Based Image Segmentation 是一种经典的图像分割算法，本质上是一种基于图的贪心聚类算法。对于一个图像，我们可以构建图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，其中顶点 $v \in \boldsymbol{V}$ 即为图像中的单个像素点，连接一对相邻顶点的边 $(v_i, v_j) \in \boldsymbol{E}$ 是八连通边。其具有权重 $w(v_i, v_j)$，是 v_i 和 v_j 之间的不相似度。假设 v 具有 RGB 值 (r, g, b)，那么我们可以简单用欧氏距离定义这个不相似度：

$$w(v_i, v_j) = \sqrt{(r_i - r_j)^2 + (g_i - g_j)^2 + (b_i - b_j)^2} \tag{10}$$

基于图 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$，我们的目标是寻找一种分割，将 $\boldsymbol{G}$ 分割成若干个区域，每个区域都是一棵最小生成树（MST），区域与区域之间不存在边连接；单个区域内的顶点之间的不相似度较小，不同区域的顶点差异较大。起始状态下，顶点之间彼此不相连，各自构成一个区域。我们的目标是连接顶点之间的边、不断合并相似的区域，直到剩下的区域不再相似为止。

假设现在存在区域 C_i 和 C_j，我们通过如下的方法判断是否需要合并。首先，假设区域 C 的 MST 为 $(\boldsymbol{V}_C, \boldsymbol{E}_C)$，那么类内差异为：

$$Int(C) = \max_{(v_1,v_2) \in \boldsymbol{E}_C} w(v_1, v_2) \tag{11}$$

讲人话，类内差异就是 C 中存在边的最大不相似度。然后，我们定义两个区域 C_i 和 C_j 的类间差异为：

$$Diff(C_i, C_j) = \min_{v_1 \in \boldsymbol{V}_{C_i}, v_2 \in \boldsymbol{V}_{C_j}, (v_1,v_2) \in \boldsymbol{E}} w(v_1, v_2) \tag{12}$$

显然，类间差异是指连接两个区域的所有边中的最小不相似度。那么，我们就可以得到一个允许合并的标准：

$$Diff(C_i, C_j) \leq \min(Int(C_i), Int(C_j)) \tag{13}$$

需要注意的是，当 C 只有一个顶点时，$Int(C) = 0$。此时根据式13，除非存在完全一致的像素点，否则没有任何区域能与其相连。所以我们添加一个修正项 $\frac{k}{|\boldsymbol{V}_C|}$，其中 k 是个常数，$|\boldsymbol{V}_C|$ 是区域 C 的顶点数量。修正后的合并标准为：

$$Diff(C_i, C_j) \leq \min(Int(C_i) + \frac{k}{|\boldsymbol{V}_{C_i}|}, Int(C_j) + \frac{k}{|\boldsymbol{V}_{C_j}|}) \tag{14}$$

显然，当顶点个数为 1 时，k 值的存在可以允许较大的类间差异；而当数量越来越大时，修正项逐渐趋于 0、不发挥作用。

综合上述内容，整个算法的流程如下所示：

(1) 计算每一个像素点与其 8 邻域或 4 邻域的不相似度；

(2) 将边按照不相似度从小到大排序，得到 $e_1, e_2, \cdots, e_N$；

(3) $n \leftarrow 1$；

(4) 对 e_n 进行合并判断。设其所连接的顶点为 (v_1, v_2)，那么当 v_1 和 v_2 不属于同个区域，且满足公式14（若 v_1 和 v_2 不属于同个区域，那么 $Diff(C_{v_1}, C_{v_2}) = w(v_1, v_2)$）时，合并 C_{v_i} 和 C_{v_j}；

(5) $n \leftarrow n + 1$；

(6) 重复步骤 (4)(5)，直到遍历完所有边；

其中，步骤 (4) 的合并操作包括两个部分：

(1) 更新标号：将 C_{v_i} 和 C_{v_j} 的标号统一为 C_{v_i} 的标号；

(2) 更新类内差异：此时 C_{v_i} 的类内差异为 $w(v_i, v_j) + \frac{k}{|C_{v_i}|}$。需要注意 $|C_{v_i}|$ 是合并后的数量；

2.3 Visual Bag of Words

许多应用需要快速的图像特征匹配，需要实时判断当前图像帧是否在之前的图像数据库中出现。一些 SIFT 等特征提取算法在执行特征匹配时往往比较耗时，而视觉词袋可以加速特征匹配的速度。

视觉词袋是一种利用视觉词汇表将图像转换为稀疏分层向量的技术手段，从而可以操作数量较大的图像集合。当我们得到图像的不同特征向量后，我们可以使用聚类算法对这些特征进行聚类，然后用聚类中的一个簇来代表视觉词袋中的一个视觉词。在得到 k 个视觉词、构建完成码本后，我们可以提取图像中的特征点，计算每个特征点与各个视觉词之间的距离（相似度）、并映射到距离最近的视觉词上。通过统计码本中每个视觉词在图像中出现的次数，我们就可以将图像转换成一个与视觉词序列相对应的向量表示。

在本次实验中，我们使用视觉词袋的方法与上述传统方法有所不同。在得到码本后，我们计算的是每个区域与各个视觉词的相似度，从而生成一个由相似度构成的向量表示。某种意义上，这个向量表示是各个视觉词构成这个区域的比例组成。最后，我们就可以通过处理这个向量得到这个区域的相关信息。

3 关键代码展示与说明

3.1 Seam Carving

第一题的相关代码位于 seam_carving.py。我将相关功能封装成一个类 SeamCarving，结构如下所示：

```
class SeamCaving():
    def __getDynamicEnergyMap(self, img, mask):
        '''
        获取动态规划能量图和路径图
        '''
    def __getSeam(self, routeMap, index):
        '''
        通过index获取该像素对应的seam
        '''
    def __removeSeam(self, img, seamIndex):
        '''
        去除img中的seam
        '''
    def __markSeam(self, rgbImg, seamIndex):
        '''
        标记rgbImg中的seam为黑色
        '''
    def changeShape(self, img, new_shape, mask=None, file_path="process", id=0):
        '''
        改变img的尺寸
        '''
```

其中，__getDynamicEnergyMap() 函数就是用于实现式8的过程，从上到下计算得到最小 cost。对于 RGB 图像而言，由于存在三个颜色通道，我们很难给出计算相邻像素之间邻边能量的合理公式。因此基于人眼对亮度的细节更加敏感的思想，我们将图像转换成灰度图，然后在灰度图上计算能量。此外，在实验中我们需要保持图像前景，为此我读取并处理了图像对应的蒙版 mask，mask 中位于前景的像素位置值为 1、位于背景的像素位置值为 0.2。通过将 img 和 mask 进行对应元素相乘，我们可以在后续计算能量时使得背景像素生成的 cost 小于前景像素生成的 cost，从而引导 seam 尽量包含背景像素。此外，算法描述缺乏对图像中特殊位置像素的处理方法。在具体实现过程中，我们在原始灰度图的边缘外添加了一圈值为 0 的虚拟像素，用于辅助计算删除边缘像素的 cost。当然，对于左右边缘的像素，其计算 $M(i,j)$ 时只有两种情况。

还需要说明的是，函数还生成一个用于后续 seam 索引的路径图 routeMap。当 routeMap$[i,j]=k$、其中 $k\in\{-1,0,1\}$ 时，像素 (i,j) 的上一个像素是 $(i-1,j+k)$。我们仅实现了计算垂直 seam 的过程。若要获取水平 seam，我们可以把图像转置后再输入。

```
    def __getDynamicEnergyMap(self, img, mask):
        height, width = img.shape
        # 用蒙版处理图像
        img = img*mask
        # 扩充0，便于后续操作
```

```
        paddingImg = np.pad(img, ((1,1),(1,1)), 'constant', constant_values=0)
        dynamicEnergyMap = np.zeros((height+2, width+2))
        routeMap = np.ones((height, width), dtype=int)*-1
        for hIndex in range(height):
            M = np.zeros((width, 3))
            M[:, 0] = dynamicEnergyMap[hIndex, :-2]+np.abs(paddingImg[hIndex][1:-1]-
                paddingImg[hIndex+1][:-2])
            M[:, 1] = dynamicEnergyMap[hIndex, 1:-1]
            M[:, 2] = dynamicEnergyMap[hIndex, 2:]+np.abs(paddingImg[hIndex][1:-1]-
                paddingImg[hIndex+1][2:])
            colEnergy = np.abs(paddingImg[hIndex+1][2:]-paddingImg[hIndex+1][:-2])
            dynamicEnergyMap[hIndex+1, 1:-1] = np.min(M, axis=1)
            routeMap[hIndex] = np.argmin(M, axis=1)-1
            # 像素(i,0)只能选择(i-1,0)和(i-1,1)
            if M[0,1]<=M[0,2]:
                routeMap[hIndex,0] = 0
                dynamicEnergyMap[hIndex+1,1] = M[0,1]
            else:
                routeMap[hIndex,0] = 1
                dynamicEnergyMap[hIndex+1,1] = M[0,2]
            # 像素(i,-1)只能选择(i-1,-1)和(i-1,-2)
            if M[-1,0]<=M[-1,1]:
                routeMap[hIndex,-1] = -1
                dynamicEnergyMap[hIndex+1,-1] = M[-1,0]
            else:
                routeMap[hIndex,-1] = 0
                dynamicEnergyMap[hIndex+1,-1] = M[-1,1]
            dynamicEnergyMap[hIndex+1,1:-1] += colEnergy

        return dynamicEnergyMap[1:-1,1:-1], routeMap
```

在得到 dynamicEnergyMap 后，我们就可以获得最后一行中 cost 最小的像素对应的 index。然后，我们调用 ___getSeam() 来获取相应的整条 seam：

```
    def __getSeam(self, routeMap, index):
        seamIndex = np.zeros(routeMap.shape[0], dtype=int)
        seamIndex[seamIndex.size-1] = index
        for i in range(routeMap.shape[0]-2,-1,-1):
            seamIndex[i] = seamIndex[i+1]+routeMap[i+1, seamIndex[i+1]]
        return seamIndex
```

上述函数只是针对单条 seam，现在我们需要实现图像尺寸调整的动态规划。相关函数 changeShape() 如下所示。我们仅展示了关键语句、省略了大量细节。相关代码和相关注释详见 seam_carving.py。

```
    def changeShape(self, img, new_shape, mask=None, file_path="process", id=0):
        # 确保新尺寸小于原尺寸
        assert len(new_shape)==2
        assert img.shape[0]>=new_shape[0] and img.shape[1]>=new_shape[1]
        # 修改蒙版
```

```
        if mask is None:
            newMask = np.ones((img.shape[0], img.shape[1]))
        else:
            newMask = mask.copy()
            newMask[newMask<0.5] = 0.2
        # 计算缩小的高度和宽度
        reducedHeight = img.shape[0]-new_shape[0]
        reducedWidth = img.shape[1]-new_shape[1]
        # 获取灰度图
        grayImg = np.asarray(Image.fromarray(np.uint8(img)).convert('L')).copy().astype(
            int)
        ......
        grayImgLists = list()# 灰度图状态
        maskLists = list()# mask状态
        imgLists = list()# 图像状态
        indexLists = list()# 图像索引矩阵状态
        seamLists = list()# 每个状态操作的seam
        costLists = list()# 每个状态的最小cost
        chosenLists = list()# 当前状态的上一个状态(对于[i,j], 0:[i,j-1]; 1:[i-1,j])
        # 动态规划
        for hIndex in tqdm(range(reducedHeight+1)):
            grayImgLists.append(list())
            ......
            for wIndex in range(reducedWidth+1):
                if wIndex!=0:
                    # 获取垂直seam
                    dynamicEnergyMap, routeMap = self.__getDynamicEnergyMap(grayImgLists
                        [-1][wIndex-1], maskLists[-1][wIndex-1])
                    colSeamIndex = self.__getSeam(routeMap, np.argmin(dynamicEnergyMap
                        [-1,:]))
                    colCost = costLists[-1][wIndex-1]+np.min(dynamicEnergyMap[-1,:])
                if hIndex!=0:
                    # 获取水平seam时, 将相关矩阵转置后、通过获取垂直seam的方式获取
                    tranGrayImg = np.transpose(grayImgLists[-2][wIndex], (1,0))
                    tranImg = np.transpose(imgLists[-2][wIndex], (1,0,2))
                    tranMask = np.transpose(maskLists[-2][wIndex], (1,0))
                    tranIndexMap = np.transpose(indexLists[-2][wIndex], (1,0,2))
                    dynamicEnergyMap, routeMap = self.__getDynamicEnergyMap(tranGrayImg,
                        tranMask)
                    rowSeamIndex = self.__getSeam(routeMap, np.argmin(dynamicEnergyMap
                        [-1,:]))
                    rowCost = costLists[-2][wIndex]+np.min(dynamicEnergyMap[-1,:])
                if wIndex==0 and hIndex==0:
                    # 起始写入原始数据
                    ......
                elif hIndex==0:
                    # 只有垂直seam
                    ......
```

```
                elif wIndex==0:
                    # 只有水平seam
                    ......
                else:
                    # 比较cost
                    if rowCost>colCost:
                        ......
                    else:
                        ......
        # 输出最终图像和gif图
        ......
```

3.2 Graph-Based Image Segmentation

第二题的相关代码位于 segmentation.py。同 Seam Carving 类似，我同样把相关操作封装成一个类：

```
class Segmentation():
    def __init__(self, k, min_size = 50, gaussian_radius = 1, gaussian_sigma=1.5):
    def __getEdgeValue(self, img):
        '''
        获取八连通边的权重值字典
        '''
    def __getPixelDiff(self, pixel1, pixel2):
        '''
        计算两个像素的不相似度
        '''
    def getCluster(self, img):
        '''
        获取图像的图分割
        '''
    def getColorImg(self, img, clusterIndexMap, clusterIndexList):
        '''
        获取标记分割的图像，不同区域用不同颜色表示
        '''
    def getSuperPixelImg(self, img, clusterIndexMap, clusterIndexList):
        '''
        获取标记分割的图像，不同区域用红色边包围
        '''
    def getMaskedImg(self, img, mask, clusterIndexMap, clusterIndexList):
        '''
        获取img新的mask
        '''
    def calcIoU(self, img, mask, clusterIndexMap, clusterIndexList):
        '''
        计算IoU值
        '''
```

其中核心部分是函数 getCluster()，代码如下所示。首先我们调用函数 __getEdgeValue() 函数，得

到每个边的权重值字典。需要注意的是，在计算权重值时，我们将先对图片进行一次高斯模糊操作，目的是去除伪影（这是原论文的说法，个人观点是为了让边权重值更小、更方便合并）。

然后后续代码按照2.2中的算法流程实现。需要注意的是，我们有两轮遍历过程，第一轮是常规合并，而第二轮旨在把过小的区域按照最小类间差异优先的顺序强制合并。最后，我们输出包括一个标记了各个像素所属类标号的矩阵，以及类标号列表。

```
    def getCluster(self, img):
        edgeDict = self.__getEdgeValue(img)
        clusterIndexMap = np.ones((img.shape[0],img.shape[1]),dtype=int)*-1# 标号矩阵
        clusterDict = dict() # 存储各个区域的信息
        clusterId = 1
        # 初始情况下每个像素就是一个区域
        for i in range(img.shape[0]):
            for j in range(img.shape[1]):
                clusterDict[clusterId] = dict()
                clusterDict[clusterId]['list'] = [(i,j)] # 区域像素坐标列表
                clusterDict[clusterId]['diff'] = self.k/len(clusterDict[clusterId]['list'
                    ]) # 类内差异
                clusterIndexMap[i,j] = clusterId
                clusterId += 1
        # 排序
        edgeSortList = sorted(edgeDict.items(), key = lambda ed:(ed[1],ed[0]))
        # 常规合并
        for ((pixelInx1,pixelInx2),edgeValue) in edgeSortList:
            # 获取所属类标号
            clusterId1 = clusterIndexMap[pixelInx1]
            clusterId2 = clusterIndexMap[pixelInx2]
            if clusterId1==clusterId2:
                continue
            if edgeValue>min(clusterDict[clusterId1]['diff'], clusterDict[clusterId2]['
                diff']):
                continue
            # 合并两个区域
            clusterDict[clusterId1]['list'].extend(clusterDict[clusterId2]['list'])
            num = len(clusterDict[clusterId1]['list'])
            clusterDict[clusterId1]['diff'] = edgeValue + self.k/num
            clusterDict.pop(clusterId2)
            clusterIndexMap[clusterIndexMap==clusterId2] = clusterId1
        # 合并过小的区域
        for ((pixelInx1,pixelInx2),edgeValue) in edgeSortList:
            clusterId1 = clusterIndexMap[pixelInx1]
            clusterId2 = clusterIndexMap[pixelInx2]
            if clusterId1==clusterId2:
                continue
            if len(clusterDict[clusterId1]['list'])>=self.__minSize and len(clusterDict[
                clusterId2]['list'])>=self.__minSize:
                continue
            clusterDict[clusterId1]['list'].extend(clusterDict[clusterId2]['list'])
```

```
            num = len(clusterDict[clusterId1]['list'])
            clusterDict[clusterId1]['diff'] = edgeValue + self.k/num
            clusterDict.pop(clusterId2)
            clusterIndexMap[clusterIndexMap==clusterId2] = clusterId1
        return clusterIndexMap, list(clusterDict.keys())
```

3.3 Visual Bag of Words

第三题的代码主要位于 synthesize.py，此外还引用了3.2中的 Segmentation 类。整个算法实现被封装成类 ForePredict：

```
class ForePredict():
    def __init__(self):
    def __transform(self, imgList, maskList):
        '''
        预处理数据，对图像进行分割、获得区域的向量表示和对应的标签
        '''
    def train(self, imgList, maskList, new_features = 20, bag_size=50):
        '''
        训练函数
        '''
    def test(self, imgList, maskList):
        '''
        测试函数
        '''
```

假设我们获得了一个图像集，我们首先要对其提取直方图数据，并调用3.2的 Segmentation 类完成分割、获取区域标签。最终我们得到图像区域的向量表示和对应标签，实现代码 ___transform() 如下所示：

```
    def __transform(self, imgList, maskList):
        '''
        预处理数据，对图像进行分割、获得区域的向量表示和对应的标签
        '''
        model = Segmentation(k=10)
        data_x = list()
        data_y = list()
        for imgIndex in range(len(imgList)):
            print("{} begins.".format(imgIndex))
            # 获取图片的全局直方图
            totalHint = np.zeros((512))
            ......
            # 归一化
            totalHint /= imgList[imgIndex].shape[0]*imgList[imgIndex].shape[1]
            # 获取分割区域，需要自动调整k值
            model.k = 150
            clusterIndexMap, clusterIdList = model.getCluster(imgList[imgIndex])
            while len(clusterIdList)<50 or len(clusterIdList)>70:
```

```
            ......
            # 获取每个区域的标签(0: 背景; 1: 前景)
            newMask, yList = model.getMaskedImg(imgList[imgIndex], maskList[imgIndex],
                clusterIndexMap, clusterIdList)
            for cId in clusterIdList:
                # 获取区域的局部直方图
                localHint = np.zeros((512))
                ......
                # 归一化
                localHint /= localIndex.shape[0]
                # 区域的向量表示
                data_x.append(np.append(totalHint,localHint))
            # 区域标签
            data_y += yList
        return np.array(data_x), np.array(data_y)
```

训练函数 train() 代码如下所示，我们首先使用 PCA 降维，然后构建词袋。在得到新的训练集数据后，我们使用逻辑回归模型来训练，最终计算得到模型在训练集上的准确率和召回率：

```
    def train(self, imgList, maskList, new_features = 20, bag_size=50):
        '''
        训练函数
        '''
        # 处理数据
        train_x, train_y = self.__transform(imgList, maskList)
        # pca降维
        self.__pcaModel = PCA(n_components=new_features).fit(train_x)
        train_x = self.__pcaModel.transform(train_x)
        # 构建visual bag
        self.__kmeansModel = KMeans(n_clusters=bag_size, random_state=0).fit(train_x)
        # 计算相似度，拼接得到新训练数据
        similarity = np.dot(train_x, self.__kmeansModel.cluster_centers_.T)
        similarity /= np.sum(self.__kmeansModel.cluster_centers_**2, axis=1)
        similarity /= np.sum(train_x**2, axis=1).reshape((-1,1))
        train_x = np.concatenate((train_x, similarity),axis=1)
        # 使用逻辑回归
        self.__predictModel = LogisticRegression(random_state=0).fit(train_x, train_y)
        # 计算准确率和召回率
        pred_y = self.__predictModel.predict(train_x)
        acc = np.sum(pred_y==train_y)/train_y.size
        recall = np.sum(pred_y & train_y)/np.sum(pred_y)
        return acc, recall
```

测试函数与训练函数类似，不同的是测试函数不会拟合 PCA 模型和逻辑回归模型，而是直接套用模型计算输出结果。

4 实验过程与结果分析

4.1 第一题实验过程与结果分析

我们首先测试 Seam Carving 实现代码。在调整图像比例上，假设原始图片的高度为 H、宽度为 W，而背景占整张图片的面积比例为 P，那么调整后的图片高度 H' 和宽度 W' 分别为：

$$H' = H \times \sqrt{1 - \frac{P}{2}} \tag{15}$$

$$W' = W \times \sqrt{1 - \frac{P}{2}} \tag{16}$$

通过这样操作，我们可以大致删除图片中一半的背景面积。

根据题目要求，我挑选了名称末两位为 “13” 的图片进行测试。部分实验结果如图2-9所示。受限于实验报告篇幅，更多实验结果和结果图像详见 results/question1 文件夹。此外各个图像的变化 gif 动图也在该文件夹中。

图 2: 原始图像 13

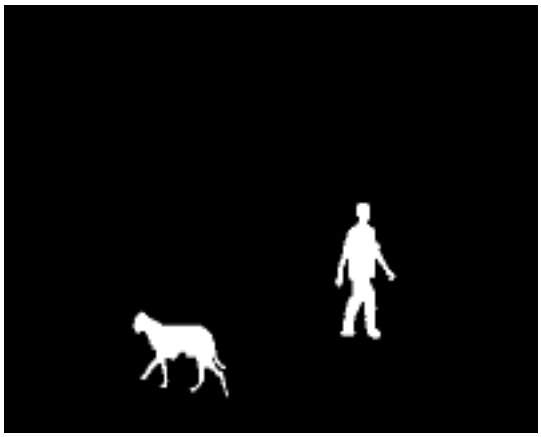

图 3: 原始图像 13 前景

图 4: 原始图像 13 中被删除的 seam

图 5: 压缩后的图像 13

图 6: 原始图像 513

图 7: 原始图像 513 前景

图 8: 原始图像 513 中被删除的 seam

图 9: 压缩后的图像 513

从上述结果中我们可以发现，原始图像中的前景部分得到了很好的保留，没有被破坏。而被删除的 seam 基本穿过的是背景部分，在删除与周围差异较少的像素同时、没有生成较明显的伪影，处理后的结果图像十分自然。这说明我们的算法实现得比较成功。results/question1 文件夹中保存的动图可以清晰地展示我们算法选择和删除 seam 的过程。

4.2 第二题实验过程与结果分析

与第一题相似，我们同样使用文件名末尾为 “13” 的图片对分割算法进行测试，其中高斯模糊的参数 radius 设为 1，σ 设为 0.8。题目要求最终分割区域数量为 $50 \sim 70$ 个，且一个区域的像素数量不得少于 50。为了满足区域最低像素数量要求，我在算法中设计了两次遍历，第一次遍历是按照正常流程合并相似区域，第二次是强制让不满足最低像素数量要求的区域与类间差异最小的区域合并。而分割区域的数量主要通过调整参数 k 来控制。经过实验，我发现 $k = 150$ 是一个比较通用的值，k 越大、区域数量越少，反之越大。我设计了一个自动调整 k 值的过程，从而保证不同图片都可以得到符合要求的分割区域数。

部分实验结果如图10-图17所示。对于分割后的区域，我们设置边界为红色便于区分。

图 10: 原始图像 13

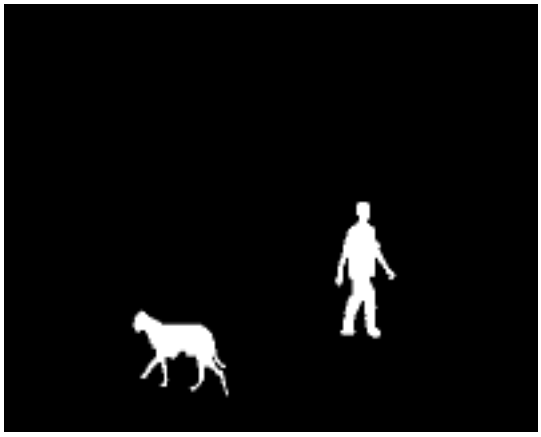

图 11: 原始图像 13 前景

图 12: 原始图像 13 分割区域

图 13: 图像 13 分割后生成的新前景

图 14: 原始图像 513

图 15: 原始图像 513 前景

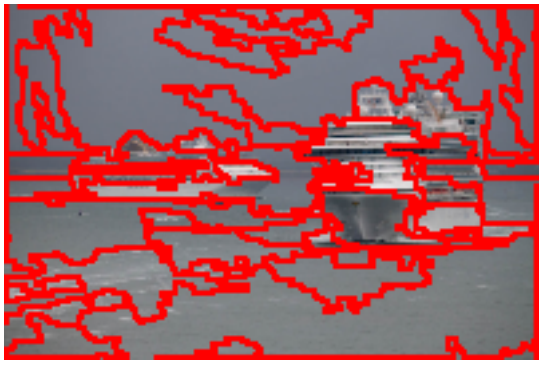

图 16: 原始图像 513 分割区域

图 17: 图像 513 分割后生成的新前景

可以看到，我们得到的图像分割区域内部的像素比较相似，基本上大部分红色边界线与原始图片的

纹理相切。然而，部分图片重新生成的前景与原始前景差异较大，尤其是当原始前景面积过小、含有大量细节时。如图11所示，原始图片的人物本身面积较小、并且含有大量细节、之间的差异较大，因此人物的区域会被强制和周围背景区域融合，从而无法生成人物前景。而对于图17而言，图片右侧邮轮的部分舷窗与海岸线颜色相似，故被合并在一起、没有显示在邮轮前景。不过大体而言，图像分割的结果和生成的新前景达到了我们的预期要求。更多实验生成结果图像见 results/question2 文件夹。

此外，我们还计算了测试图片生成前景和原始前景的 IoU 值，结果如表1所示。可以看到，图像分割算法在大多数图像上表现良好，平均 IoU 值达到 **65.47%**。

	13.png	**113.png**	**213.png**	**313.png**	**413.png**
IoU	0.3324	0.8676	0.7680	0.4786	0.6364
	513.png	**613.png**	**713.png**	**813.png**	**913.png**
IoU	0.7298	0.8001	0.5686	0.6369	0.7287

表 1: 不同图像的 IoU 值

4.3 第三题实验过程与结果分析

相比前两道题，第三题没有太多可以展示的内容。我的程序完全按照题目的算法顺序实现。需要说明的是，在计算降维后的颜色对比度特征和各个 visual word 的特征的点积相似性时，我们需要对特征进行归一化。这是因为如果只是单纯使用点积来得到相似性的话，我们可以轻易得到向量 $(0.5, 0.5)$ 和 $(1, 1)$ 之间的相似度比 $(0.5, 0.5)$ 和 $(0.5, 0.5)$ 更高的错误结论。由于是对特征进行归一化后再得到点积值，因此本质上我们计算的就是两个向量之间的余弦相似度：

$$similarity(\boldsymbol{X}, \boldsymbol{Y}) = \frac{\boldsymbol{X} \cdot \boldsymbol{Y}}{\|\boldsymbol{X}\|\|\boldsymbol{Y}\|} \tag{17}$$

此外，在分类算法上，我选择使用 sklearn 的逻辑回归模型进行学习分类，所有参数均为默认值。我选用文件名末尾为 “13” 的图片作为测试集，然后设置随机种子为 0、从数据集中随机抽取 200 张非测试集图片作为训练集。最终，我们得到的结果如表2所示。可以看到，无论是训练集还是测试集，准确率都在 82% 以上。考虑到大多数区域都是背景（标签为 0），而这种数据的不平衡性可能会导致模型总是预测区域为背景，我们额外计算了模型在训练集上的召回率。可以看到，模型在训练集上的召回率达到 54.72%，测试集上达到 75%，说明我们的模型的预测是基于输入区域的，数据不平衡的影响较小。

	train	**test**
accuracy	0.8204	0.8286
recall	0.5472	0.7500

表 2: 输入未标准化时的模型表现

在训练模型时我们发现，在 sklearn 的默认最大迭代次数下模型没有收敛完毕、没有到达局部最优点。我们认为这可能是输入数据的不同维度的尺度不一致导致的。因此我们尝试对最后的输入数据进行标准化，再用来训练模型、预测结果。对应的实验结果见3。这次模型可以在默认最大迭代次数下收敛完毕。然而，模型在训练集上的性能有所提升的同时，在测试集上的性能有所下降，说明模型有轻微的过拟合现象。

	train	test
accuracy	0.8211	0.8252
recall	0.5726	0.5000

表 3: 输入标准化后的模型表现

5 实验总结

本次期末大作业是对这学期计算机视觉课程的一次知识汇总，难度不算很大。对于第一题，由于我期中报告的主题就是 Seam Carving，对相关内容较为熟悉、也自行实现了相关算法，因此完成起来比较轻松。对于第二题，在了解了算法的主要思想后，我在具体的实现方法上没有头绪，一直受限于要为图像构建一个图、然后基于图进行搜索的想法。后来，在阅读了原始论文后，我才意识到可以不用建图、只通过简单排序即可按照最优顺序合并区域。此外，也是在原始论文中，我才得知需要事先对图片高斯模糊这种没有被介绍的实现细节。而第三题是在第二题的基础上完成的，通过视觉词袋和机器学习来完成分类任务，是多种算法的大杂烩，互相依赖彼此的结果。在生成区域的向量表示时，我一开始在直方图统计上写错了获取索引的代码，导致后续结果十分糟糕。所幸后来排查出了 bug，最终顺利完成实验。

通过这次实验，我重温了计算机视觉中的相关算法和有关知识点，也体会到了前深度学习时代的各种计算机视觉算法的智慧与魅力，简单朴素、却巧妙有效。希望这次的作业能给我以后的学习与生活带来帮助。