论文复现
SLIC Superpixels Compared to State-of-the-Art Superpixel Methods
数字图像处理作业三

07111805 1120181319 崔晨曦

February 15, 2021

Figure 0.1: 论文信息

1 论文概述

1.1 背景与动机

超像素是由多个像素构成的一块区域，其在感知上具有意义，换言之，构成同一个超像素的像素点在位置、亮度、纹理、颜色等相关特征上都较为相似。

超像素生成常作为各种图像处理任务中的预处理步骤，为下游任务提供了方便，可以有效地降低算法的计算复杂度，提升内存效率，对于图像分割任务来说，还可以提高分割结果的质量。

在该论文发表时，已有多种生成超像素的算法，但效果相对来说并不理想。鉴于此，本论文提出了简单线性迭代聚类 (SLIC) 超像素生成算法，其在性能上要优于现有的其他方法。

1.2 优势与特色

SLIC 算法与 k-means 非常相似，实现较为简单，但其能够更好地贴合图像本身的边界，并在 Berkeley 数据集上达到了 SOTA 水平。同时，SLIC 算法速度更快，内存效率更高，生成的超像素紧密度更高，且具有更强的拓展性。

2 算法原理

整体算法的伪码描述如下所示：

Algorithm 1 SLIC superpixel segementation

Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regelar grid steps S
Move cluster centers to the lowest gradient position in a 3*3 neighborhood
Set label $l(i) = -1$ for each pixel i
Set distance $d(i) = \infty$ for each pixel i
repeat
  for each cluster center C_k in image **do**
    for each cluster pixel i in a $2S * 2S$ region around C_k **do**
      if $D \leq d(i)$ **then**
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  Compute new cluster centers
  Computer residual error E
until $E \leq threshold$

对于图像中的每个像素 i，其有关信息均可以用一个向量 $C_i = [l_i, a_i, b_i, x_i, y_i]^T$ 来表示，前三维代表颜色信息，后两维代表位置信息。最开始将聚类中心初始化在距离为 S 的网格上，并在 3*3 的邻域内进行微调，使聚类中心位于 3*3 区域中梯度最小的像素点所在位置，这是为了避免聚类中心被初始化在图像边缘或噪声点上，为后续操作带来不便。之后将所有像素所属的类别置为-1，到聚类中心的距离置为无穷，完成初始化。

然后进行迭代，对于每一个聚类中心，考虑其周围 2S*2S 的区域，计算其中所有像素到中心的度量距离。如距离变小，则更新距离最小值与该像素所属的聚类中心。每轮迭代结束

后计算新的聚类中心和残差，即本次迭代的超像素中心位置与上次之差。

经分析可知，对于一般的 k-means 算法，其时间复杂度为 $O(kNI)$，其中 k 为聚类中心数，N 为像素个数，I 为迭代次数。而对于 SLIC 算法，其时间复杂度为 $O(NI)$，因为只需要在 2S*2S 的临域中进行操作，据论文所述，对于大多数图片而言，SLIC 迭代不超过 10 次即可收敛，这将算法的时间复杂度进一步降低为 $O(N)$，与 k-means 相比在性能上有着显著的提升。

下面对算法中的一些细节进行进一步的阐释：

2.1 转换颜色空间

SLIC 算法处理的是 CIELAB 颜色空间中的图片。但 RGB 颜色空间无法直接转化为 LAB 颜色空间，需要借助 XYZ 颜色空间作为中介。
从 RGB 到 XYZ 的转换如下式所示：

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357380 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

从 XYZ 到 LAB 的转换如下所示：

$$\begin{aligned} L &= 116 f(Y/Y_n) - 16 \\ a &= 500\left[f(X/X_n) - f(Y/Y_n)\right] \\ b &= 200\left[f(Y/Y_n) - f(Z/Z_n)\right] \end{aligned}$$

其中：

$$f(t) = \begin{cases} t^{1/3} & \text{if } t > \left(\frac{6}{29}\right)^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 t + \frac{4}{29} & \text{otherwise} \end{cases}$$

X_n, Y_n, Z_n 为修正系数，其值为 RGB 到 XYZ 转换矩阵的行之和，即 0.950456、1.0、1.088754。

2.2 初始化

初始化步骤中的距离 S 通过以下方式确定：设图中包含的像素总数为 N，则一个超像素约由 N/k 个像素构成，若假定其为正方形，则超像素间距约为 $S = \sqrt{\frac{N}{k}}$，因此我们在图像中以 S 为间距初始化聚类中心。

2.3 距离度量

对于任意两个像素之间的距离，不可直接用五维欧氏距离来定义，其原因在于，当超像素的大小变化时，像素间的空间距离在总体距离中所占有的比重会发生改变，这与我们定义距离的初衷相矛盾。

为了寻找一种能够平衡空间距离和色彩距离的度量方法，我们给出如下的距离度量方式:

$$d_c = \sqrt{(l_j - l_i)^2 + (a_j - a_i)^2 + (b_j - b_i)^2}$$
$$d_s = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$
$$D' = \sqrt{\left(\frac{d_c}{N_c}\right)^2 + \left(\frac{d_s}{N_s}\right)^2}$$

其中 N_c 和 N_s 分别用于标准化色彩距离和空间距离。可以取 N_s 为上文计算过的超像素平均间距 S。而由于不同簇之间色彩距离可能较大，该算法将 N_c 设为一个可变的参数。为方便计算，像素间的距离可用如下变换后的公式来表达：

$$D = \sqrt{d_c^2 + \left(\frac{d_s}{S}\right)^2 m^2}$$

m 越大，空间相似性所占的比重越大；m 越小，颜色相似性所占的比重越大。

2.4 强制连续性

在迭代结束时，算法尚不能保证超像素区域的连通性，也就是说，可能存在某个超像素由多个不连续的孤立区域组成。为了弥补这个缺陷，我们还需要进行保证连通性的后处理操作，即将面积小于阈值的孤立区域与其相邻的区域合并。

3 代码实现

对于算法的实现，作业附件中的源代码 cpp 文件附带注释，在此就不做过多赘述，仅挑选论文中“所述不详”的强制连续性步骤结合代码进行详细分析。在我的实现中，强制连续性由函数 EnforceConnectivity 完成。

首先我们初始化一个二维数组 newcluster 用于记录修正后的像素的类别归属。

```
554 //强制连续性
555 void EnforceConnectivity(vector<vector<labColor>>& img) {
556
557     int rows = img.size();
558     int cols = img[0].size();
559
560     int adjlabel = 0, label = 0;
561
562     int threshold = rows * cols / centers.size();
563     vector<vector<int>> newcluster;
564
565     for (int i = 0; i < rows; i++) {
566         vector<int> newrows;
567         for (int j = 0; j < cols; j++) {
568             newrows.push_back(-1);
569         }
570         newcluster.push_back(newrows);
571     }
```

然后我们在图像上进行广度优先搜索，对于每一个未访问过的像素，先在其四邻域中寻找到一个已知的聚类中心 adjlabel。

```
572
573     for (int i = 0; i < rows; i++) {
574         for (int j = 0; j < cols; j++) {
575             if (newcluster[i][j] == -1) {
576                 newcluster[i][j] = label;
577                 //BFS
578                 queue<pair<int, int>> q;   //BFS队列
579                 vector<pair<int, int>> connectiveregion;   //连通区域
580 
581                 q.push(pair<int, int>(i, j));
582                 connectiveregion.push_back(pair<int, int>(i, j));
583 
584                 //在四邻域中寻找新聚类中心
585                 for (int t = 0; t < 4; t++) {
586                     int y = i + step4nbr[t][0];
587                     int x = j + step4nbr[t][1];
588                     if (y >= 0 && y < rows && x >= 0 && x < cols) {
589                         if (newcluster[y][x] >= 0) {
590                             adjlabel = newcluster[y][x];
591                         }
592                     }
593                 }
```

然后计算与该像素联通且同属于一个超像素的像素点数量。

```
594 
595                 int numclusterpixel = 1;
596                 while (!q.empty()) {
597                     pair<int, int> now = q.front();
598                     q.pop();
599                     for (int k = 0; k < 4; k++) {
600                         int y = now.first + step4nbr[k][0];
601                         int x = now.second + step4nbr[k][1];
602                         if (y >= 0 && y < rows && x >= 0 && x < cols
603                             && newcluster[y][x] == -1 && cluster[y][x] == cluster[i][j]) {
604                             numclusterpixel++;
605                             q.push(pair<int, int>(y, x));
606                             connectiveregion.push_back(pair<int, int>(y, x));
607                             newcluster[y][x] = label;
608                         }
609                     }
610                 }
611 
```

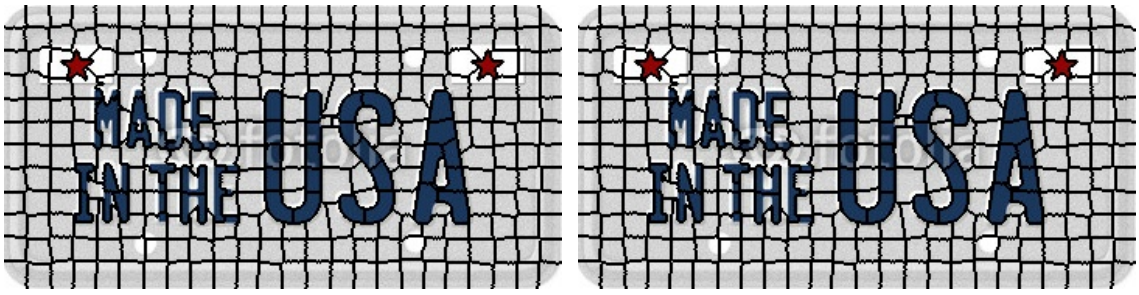
若数量小于阈值，则将这些像素全部合并到之前找到的相邻的聚类中心 adjlabel 中。

```
612                 //区域面积小于阈值，进行合并
613                 if (numclusterpixel <= threshold / 2) {
614                     for (int k = 0; k < connectiveregion.size(); k++) {
615                         newcluster[connectiveregion[k].first][connectiveregion[k].second] = adjlabel;
616                     }
617                     label--;
618                 }
619                 label++;
620             }
621         }
622     }
623
624     cluster = newcluster;
625 }
```

4 实验结果与分析

4.1 算法的收敛性

残差由相邻两次迭代之间聚类中心位置的变化所决定，当 k=256，m=40 时，迭代 200 次左右残差趋近于 0。我们不妨将迭代 200 次的结果于迭代 10 次的结果进行比较，如下所示：



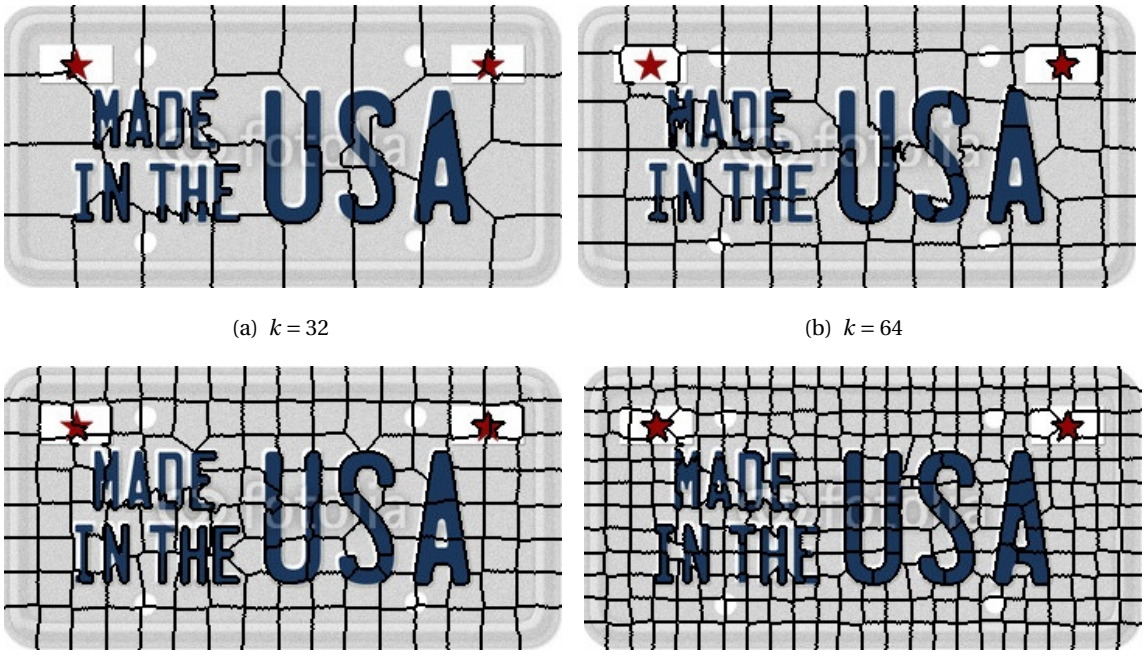
(a) 迭代 200 次　　(b) 迭代 10 次

Figure 4.1: pics

可见二者差距几乎难以分辨。虽然迭代 10 次时残差有 500 左右，但由于共有 256 个聚类中心，平均算下来聚类中心平均只变化了一两个像素的距离，故影响不大。因而 10 次迭代是足够的，这进一步证明了 SLIC 算法的高效。

4.2 超像素数量 k

我们固定 m=40，分别取 k=32,64,128,256。效果如下所示

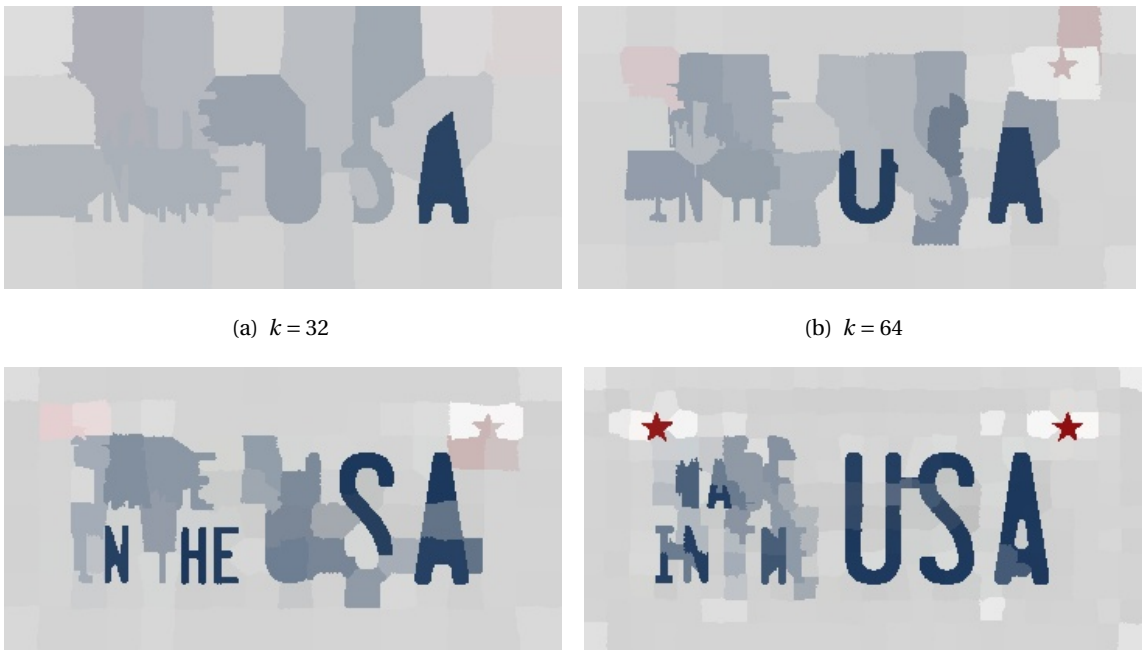


(a) $k = 32$ (b) $k = 64$

(c) $k = 128$ (d) $k = 256$

Figure 4.2: pics

可以看到，超像素的边缘基本与图像本身的边缘所吻合，譬如图像左上角和右上角的五角星基本得到了完整的保留，这证明了复现的正确性。
我们将超像素包含的所有像素的颜色用它们的均值来代替，结果如下所示：



(a) $k = 32$ (b) $k = 64$

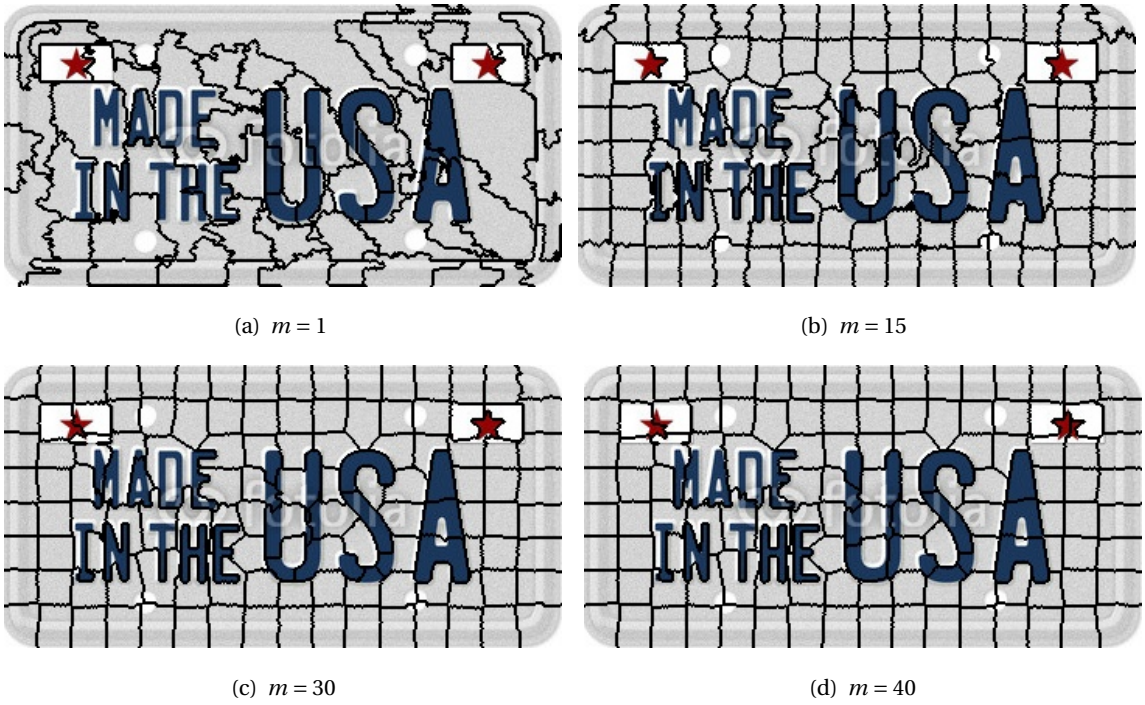
(c) $k = 128$ (d) $k = 256$

Figure 4.3: pics

可见随着超像素数目 k 的增加，超像素图片的感知特征逐渐增加，与原图像愈发接近。

4.3 距离参数 m

我们固定 k=128，分别取 m=1,15,30,40。效果如下所示：



(a) $m = 1$　(b) $m = 15$

(c) $m = 30$　(d) $m = 40$

Figure 4.4: pics

可以看到，m 较小时，超像素的轮廓极为不规则，m 增大后，超像素变得愈发规则，符合 m 越大，空间相似性占比约高的距离定义。

5 总结

SLIC 作为一种生成超像素的算法，其性能与表现超越了当时现有的所有同类算法，达到了 SOTA 水平，得益于其较快的速度和更高的内存效率，为各种图像处理领域的下游任务带来了极大的遍历。本项目使用 C++ 成功复现了 SLIC 算法，使我深刻体会到了该算法的精妙所在。