Project-1 Pacman 实验报告

林子开 21307110161

2023 年 10 月 19 日

目录

1 Restate the Basic Global Thresholding using the histogram

Given the histogram of an image, we assume $P(i)$ as the proportion of pixel(s) with intensity i in the image. We can implement the BGT algorithm more efficiently based on this histogram.

The algorithm is as follows:

1. Select the initial threshold T_0 as the mean intensity

$$T = \mu = \sum_{i=0}^{255} iP(i)$$

2. Calculate respectively the mean intensity values for Foreground and Background, which are

$$\mu_{FG} = \frac{\sum_{i=0}^{T_k-1} iP(i)}{\sum_{i=0}^{T_k-1} P(i)}, \mu_{BK} = \frac{\sum_{i=T_k}^{255} iP(i)}{\sum_{i=T_k}^{255} P(i)}$$

3. Set a new threshold

$$T_{k+1} = \frac{\mu_{FG} + \mu_{Bk}}{2}$$

4. Repeat steps 2-3, until $|T_{k+1} - T_k| < \varepsilon$, where ε is a given small number.

2 Locally adaptive thresholding

2.1 算法实现简介

在本题中，我首先调用了在第一次作业中实现的 local-list 函数，该函数能够根据所输入的 patchSize，返回对应于图像中所有像素点的局部灰度值直方图。此外，我也提前生成了对应于 patchSize=$25, 50, 100, 150$ 的局部直方图文件，也可以直接导入，节省时间。

对于每个像素点以及该像素点所对应的局部直方图，我使用了 **OTSU 算法**，找出该局部直方图的最优阈值 T，再根据 T 与该像素点灰度值的大小关系，返回 0 或 255。

具体的 Python 代码如下一小节所示。

2.2 Python 代码

Listing 1: Python codes for locally adaptive thresholding based on OTSU

```python
from math import *
import numpy as np
from PIL import Image
from hw1_exercise2_2 import local_hist # 从第一次的作业中导入高效计算Local histogram的函数

def local_adaptive_thresholding(pixel,hist):
    """
    功能：使用OTSU，基于局部histogram，判断pixel应该变成0还是255
    input：pixel是当前像素的灰度值，hist是局部直方图
    output：0或者255
    """
    S =  int( sum(hist))
    one_hist = hist/S # 归一化
    mu_global =  np.inner(one_hist, np.array( range(256)))
    T = 127 # 为了防止出现纯色的图片（比如灰度全都为255）而找不出T，这里先预设一个，防止报错
    sigma = - float('inf')
    flg = 0
    for t in  range(1,255):
        if  int(hist[t]) == 0:
            if flg == 1:
                continue  # 连续为0的区间只需要计算第一个作为阈值的情况，节省运算量
            else: # flg == 0
                flg == 1
        else: # int(hist[t]) != 0:
            flg == 0

        omega_bk =  int( sum(hist[0:t]))
        if omega_bk == 0:
            continue
            # 当累积概率为0时，可以跳过
            # 为了避免浮点运算，则使用整数运算判断
        if omega_bk == S: # 当累积概率为1时，肯定不是分界值，可以直接结束迭代，节省时间
            if pixel > T:
                return 255
            else:
                return 0
```

```
37
38          # 以下是OTSU算法的核心部分，该算法具有高效的更新方式
39          omega_bk =  sum(one_hist[0:t]) # 累积概率
40          mean_bk = np.inner(one_hist[0:t], np.array( range(t))) # accumulated mean
41          sigma_new = (mu_global*omega_bk-mean_bk)**2 / (omega_bk*(1-omega_bk)) # 前景与背景的方
                差
42          if sigma_new>sigma:
43              sigma = sigma_new
44              T = t
45      if pixel > T:
46          return 255
47      else:
48          return 0
49
50
51  def main():
52      myPicture = '肝脏.png'
53      im_raw = Image. open(myPicture)
54      im = im_raw.convert('L')
55      pixels = np.array(im) # 记得转置
56      pixels = pixels.transpose()
57      pixels2 = im.load()
58
59      # 【重要说明】
60      # 在此处，我提供两种生成局部直方图的方式（默认使用第二种！）
61      # 第一种方式直接调用第一次作业的代码，高效地求出当前图片的灰度值的所有局部直方图
62      # 如需使用第一种方式，请将以下两行代码解除注释，然后将第二种方法注释掉。
63      # patch_size = int(input('请输入patch size: '))
64      # hist_list = local_hist(pixels, patch_size)
65
66      # 第二种方式，则导入我已经提前准备好的不同patchSize对应的局部直方图文件
67      # 只需要设置patch_size参数即可，更加方便
68      # patch_size可以选择: 25, 50, 100, 150
69      patch_size =  int( input('请输入patch size（可选25, 50, 100, 150）: '))
70      hist_list = np.load('some local histograms/'+myPicture+'hist_list_patchSize='+
            str(patch_size)+'.npy')
71
72
73      # 以下为局部自适应二值化过程，需要比较久的时间，需耐心等待
74      for i in  range(hist_list.shape[0]//2):
75          row = hist_list[2*i,:]
76          row_parl = hist_list[2*i+1,:] # 引入并行的方法，提高运算速度，不然会等得更久
77          x,y =  int(row[0]),  int(row[1])
78          x_parl, y_parl =  int(row_parl[0]),
                int(row_parl[1]) # 一次循环计算两个像素，利用空间局部性
79          # print(x,y)
80          # print(x_parl,y_parl)
81          pixels2[x,y] = local_adaptive_thresholding(pixels[x,y],row[2:])
82          pixels2[x_parl,y_parl] = local_adaptive_thresholding(pixels[x_parl,y_parl],row_parl
                [2:])
83
84      if hist_list.shape[0]%2 == 1: # 对hist的最后一个元素进行二值化
```

```
85          # print('像素数量为奇数！')
86          row = hist_list[-1,:]
87          x,y =  int(row[0]),  int(row[1])
88          # print(x,y)
89          pixels2[x,y] = local_adaptive_thresholding(pixels[x,y],row[2:])
90
91
92      # 解除下面这行代码的注释后，会将二值化后的灰度值矩阵，自动命名并存储到本地
93      # 以后就可以直接导入灰度值矩阵生成二值化后的图像，不用等那么久了
94      # np.save('pixels after thresholding/'+'local_Thresholding_patchSize='+str(patch_size),np.
            array(im).transpose())
95
96      im.save('exercise2_'+myPicture+'_patchSize='+ str(patch_size)+'.jpg')
97
98
99
100 if __name__ == '__main__':
101     main()
```

2.3 测试案例：肝脏核磁共振影像

下面，我们使用如下的肝脏核磁共振影像作为测试案例：

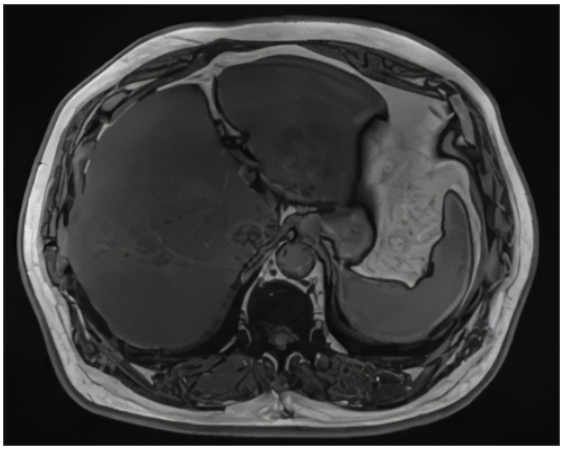
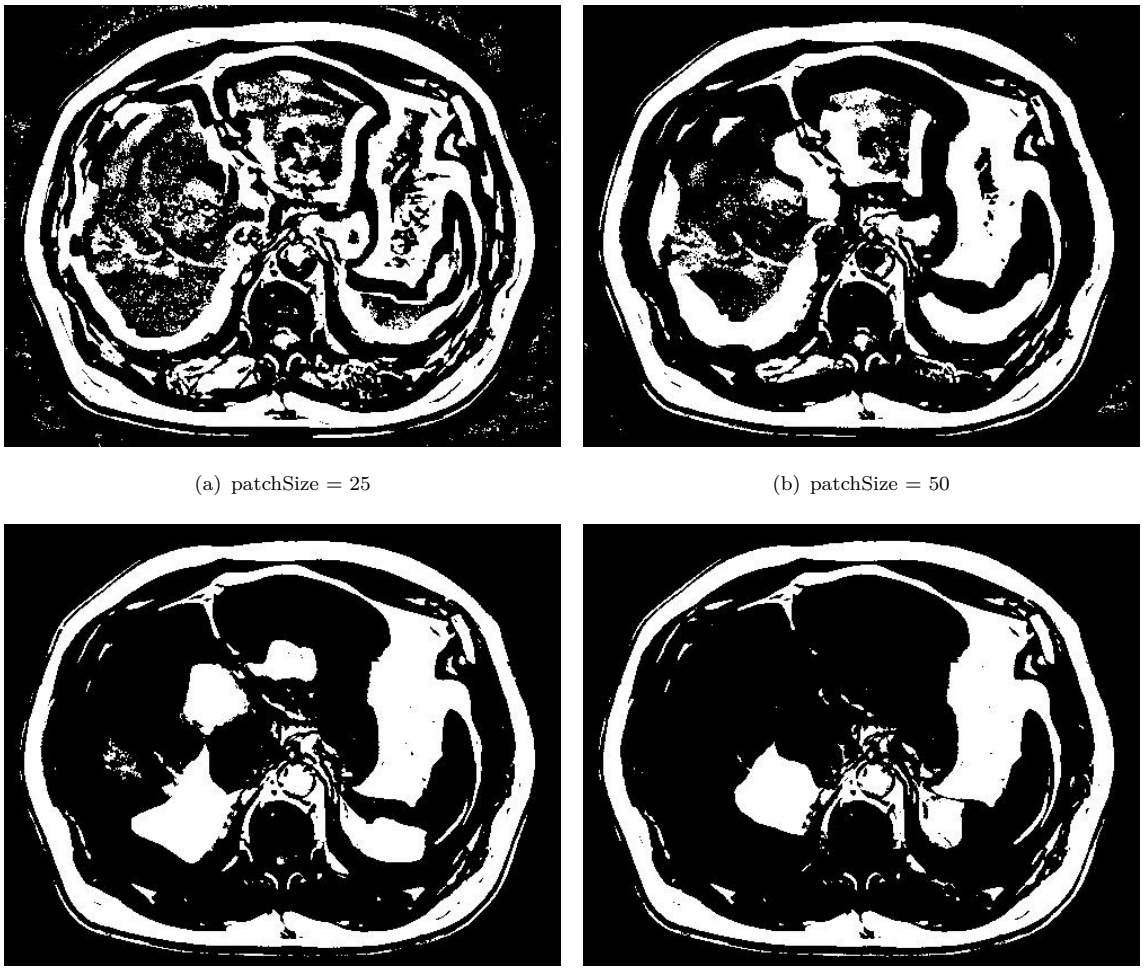


图 1: 肝脏核磁共振影像

注意到，该影像的对比度不是特别高。此外，该肝脏（目测）有一些密度发生改变的地方，有可能代表着某些器质性病变，需要将这些可能病变的部位更加明显地展示出来。

我们使用局部自适应二值化的算法，分别取局部直方图的 patchSize=$25, 50, 100, 150$ 的情况进行测试，测试结果如下：



(a) patchSize = 25

(b) patchSize = 50

(c) patchSize = 100

(d) patchSize = 150

图 2: locally adaptive thresholding with different patch size

观察以上四个图像可以发现，patchSize 选取为 25 时，图像非常混乱，展示了过多无效的细节；patchSize 选取为 100 和 150 时，则又出现大面积的黑色与白色，提供的关于可能病变区域的信息不足。相较而言，patchSize 选取为 50 时的图像中，肝脏的整体轮廓仍然是清晰的，且展示了肝脏各个区域在局部的密度变化的情况，更有助于发现可能病变的位置。因此，在本次实验所选取的 4 个 patchSize 中，我推荐使用 patchSize=50。

3 线性插值算法：放大图片分辨率

3.1 算法实现简介

我们定义：将图片的分辨率放大 N 倍，就是向原本相邻的两个像素点之间插入 $N-1$ 个像素点，也即将总的像素点数量扩大到原来的 N^2 倍。

我实现线性插值的方法是：先在第一个（横向的）维度上进行线性插值，再在第二个（纵向的）维度上进行线性插值，最终完成对整张图片的分辨率扩大。以 $N=3$ 为例，示意图如下：

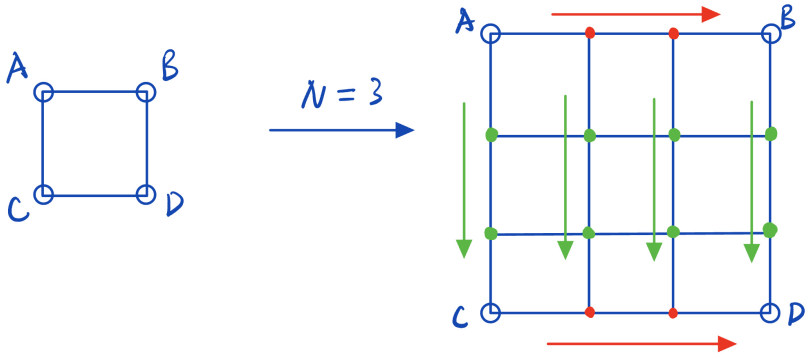


图 3: 线性插值算法示意图

假设 A, B, C, D 是原图片中相邻的 4 个像素点，在新图片中，原本相邻的像素点之间被插入了 $N-1=2$ 个新像素点。我们首先沿着第一个方向（横向），对 AB 边上的点以及 CD 边上的点（图中红点）进行线性插值，AB 边上的点的灰度值是 A 点和 B 点的灰度值的线性组合，CD 边同理。

然后，沿着第二个方向（纵向），对内部的其他的点（绿点）进行线性插值，这些内部点的灰度值都是其上方的红点（或者原图像素点）及其下方的红点（或者原图像素点）的灰度值的线性组合。

实现线性插值算法的 Python 代码如下一小节所示。

3.2 Python 代码

Listing 2: Python codes for linear interpolation

```
"""
实现线性插值算法,读出一幅图像,
利用线性插值把图片空间分辨率放大N倍，然后保存图片
"""

from math import *
import numpy as np
from PIL import Image


def main():
    myPicture = '86版西游记模糊剧照.png'
    im_raw = Image. open(myPicture)
    im = im_raw.convert('L')
    pixels = np.array(im)
    N =  int( input('请输入图片空间放大倍数N（要求为大于1的正整数）：'))
    pixels2 = np.empty((N*pixels.shape[0],N*pixels.shape[1]))

    for i in  range(pixels.shape[0]): # 先放入顶点位置的值
        for j in  range(pixels.shape[1]):
            pixels2[N*i,N*j] = pixels[i,j]

    for i in  range(pixels.shape[0]): # 在第一个方向上进行线性插值
```

```
24          for j in  range(pixels.shape[1] - 1):
25              Ni = N*i
26              Nj = N*j
27              for k in  range(1,N):
28                  pixels2[Ni,Nj+k] = ((N-k)/N)*pixels[i,j] + (k/N)*pixels[i,j+1]
29
30      for i in  range(pixels.shape[0]-1): # 在第二个方向上进行插值
31          Ni = N*i
32          for j in  range(pixels.shape[1]-1):
33              Nj = N*j
34              for p in  range(1,N):
35                  for q in  range(N+1):
36                      pixels2[Ni+p,Nj+q] = ((N-p)/N)*pixels2[Ni,Nj+q] + (p/N)*pixels2[Ni+N,Nj+q]
37
38      pixels2 = pixels2.astype(np.uint8)
39      new_im = Image.fromarray(pixels2)
40      new_im.save('exercise_3_线性插值_'+ str(myPicture)+'_N='+ str(N)+'.png')
41
42
43  if __name__ == '__main__':
44      main()
```

3.3 测试案例：86 版西游记剧照

该剧照的原始分辨率为 329×259，通过上面的算法，将其分辨率分别扩大到 $N = 2, 3, 4$，原图以及分别率扩大后的效果如下所示：

可以看出，经过线性插值之后，分辨率放大后的图片仍然保持较好的清晰度，将这些图片放大后没有出现“棋盘状”的纹路，说明线性插值后图片的分辨率仍然较好。

(a) 原图

(b) 放大倍数 N = 2

(c) 放大倍数 N = 3

(d) 放大倍数 N = 4

图 4: locally adaptive thresholding with different patch size