计算机视觉作业2: 利用聚类技术实现纹理图像分割

唐麒 21120299
qitang@bjtu.edu.cn

摘要—纹理作为图像的一个重要属性, 在计算机视觉和图像处理中占有举足轻重的地位。对于纹理的分析可以用于图像合成、图像匹配等任务。纹理作为一种区域特性，与区域的大小和形状有关。两种纹理模式之间的边界，可以通过观察纹理度量是否发生显著改变来确定。因此分析纹理可以得到图像中物体的重要信息，是特征提取、图像分割的重要手段。纹理分析主要包括纹理表征和纹理分割两个任务，本文利用灰度共生矩阵和 Gabor 滤波器两种方法对合成纹理图像的纹理进行描述，从而形成特征向量，并进一步利用无监督聚类方法对特征向量空间中的点进行聚类，最终完成对纹理图像的分割。纹理特征的提取及其分割结果将在本文的实验部分给出，不同纹理表征方法和部分参数设置对实验结果的影响也会进行简单探讨。

关键词—计算机视觉、纹理图像、图像分割、灰度共生矩阵、Gabor 变换、聚类

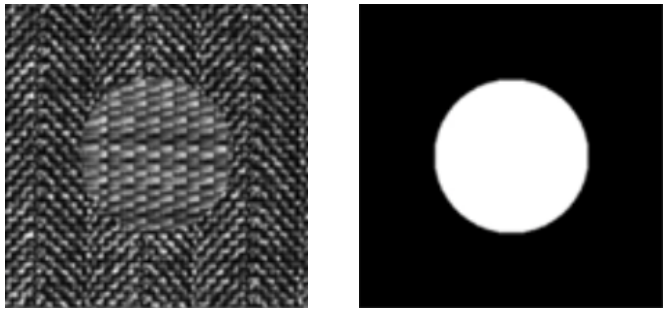
1 简介

纹理在日常生活中无处不在，它们对于区分图像的不同部分非常重要。纹理是由于物体表面的物理属性的多样性而造成的, 物理属性不同表示某个特定表面特征的灰度或者颜色信息不同, 不同的物理表面会产生不同的纹理图像，例如布纹、草地、砖砌地面的等。然而, 对于自然纹理图像而言这种重复模式往往是近似的和复杂的, 难以用语言描述, 而人类对纹理的感受多是与心理效果相结合的，因此，对纹理很难下一个确切的定义，但它对于描述图像非常有用。

一般而言，纹理图像中的灰度分布具有某种周期性，即使灰度变化是随机的，它也具有一定的统计特性。J.K.Hawkins 对纹理给出了一个比较详细的描述, 他认为纹理有三个主要的标志:1) 某种局部的序列性在比该序列更大的区域内不断重复；2) 序列是由基本元素非随机排列组成的；3) 各部分大致是均匀的统一体, 在纹理区域内的任何地方都有大致相同的结构尺寸。当然，这些也只是从感觉上看来是合理的，并不能的出定量的纹理测度。正因如此，对纹理特征的研究方法也是各种各样的。

作为提取信息的一种手段，纹理分析使用各种数学方法来表征图像中灰度分布的变化。纹理分析有主要两种不同的方式: 统计方法和句法结构方法。纹理分析的一个重要技术是构建图像的特征，这些特征通常以特征向量的形式表示，这些特征向量可以用于与纹理相关的下游任务，如纹理分割等。

纹理分割是图像处理中的一个重要问题，旨在对具有相同纹理的区域进行识别和划分。在某一区域内，图像的纹理是一致的，即该区域的一些统计量在某种程度上是恒定的或周期性的。首先通过对图像进行纹理分析来构建一组特征向量，然后根据是否有各种纹理的先验知识，以有监督或无监督的方式对其分类，最终将图像分割为不同的区域。如图 1 所示，原始图像可以看作由两种不同的纹理所构成，两种纹理近似分布于一个圆形的内部和外部，在图像进行纹理分析和纹理分割后，图像被按照纹理边界划分为两个区域，同一区域内的纹理是相同的周期性的。



(a) (b)

图 1: 合成纹理图像及其基准分割图像示例：(a) 合成纹理图像, (b) 基准分割图像.

统计方法的纹理分析寻找刻划纹理的数字特征，用这些特征或同时结合其他非纹理特征对图像中的区域进行分类。图像局部区域的自相关函数、灰度共生矩阵、灰度游程以及灰度分布的各种统计量，是常用的数字纹理特征。如灰度共生矩阵用灰度的空间分布表征纹理。由于粗纹理的灰度分布随距离的变化比细纹理缓慢得多，因此二者有完全不同的灰度共生矩阵。此外，由于纹理图像具有微观不规则但宏观存在某种统计规律性的特点，因此人们越来越关注纹理图像的多尺度特征，从不同的尺度层次来捕捉纹理的微观和宏观特性。而 Gabor 滤波器可以在频域上不同尺度、不同方向上提取相关的特征，且 Gabor 函数与人眼的作用相仿，所以 Gabor 滤波器也被于纹理分析，并取得了较好的效果。

本文剩余部分将对基于灰度共生矩阵和 Gabor 滤波器的纹理表示及基于聚类技术的纹理分割的原理、实现及实验进行介绍，所有实验涉及的方法、函数实现均基于 Python 语言。

2 方法

这一章节将对统计方法的纹理分析和基于聚类技术的纹理分割进行介绍，其中本文对统计方法的纹理分析采用了灰度共生矩阵和 Gabor 滤波器两种纹理特征表示方法，其对分割结果将在实验部分给出。

2.1 纹理表示

2.1.1 灰度共生矩阵

灰度共生矩阵 (GLCM, Gray Level Coocurrence Matrices) 是描述具有某种空间位置关系两个像素灰度的联合分布概率，不仅反映灰度的分布特性，也反映具有同样灰度或相近灰度的像素之间的位置分布特性，是有关图象灰度变化的二阶统计特征。这是由于纹理是由灰度分布在空间位置上反复出现而形成的，因而在图像空间中相隔某距离的两像素之间会存在一定的灰度关系，即图像中灰度的空间相关特性。灰度共生矩阵描述的从来不是单个像素，而是成对的像素之间的关系，对应的，灰度直方图则可以看做是对单个像素的统计与描述，并不涉及灰度间的关联关系。灰度共生矩阵能反映图像灰度关于方向、相邻间隔、变化幅度等综合信息，它是分析图像的局部模式和它们排列规则的基础。其计算公式定义为：

$$
\begin{aligned}
&P(i,j \mid \Delta x, \Delta y) = W \cdot Q(i,j \mid \Delta x, \Delta y) \\
&W = \frac{1}{(M-\Delta x)(N-\Delta y)}, Q(i,j \mid \Delta x, \Delta y) = \sum_{n=1}^{N-\Delta y} \sum_{m=1}^{M-\Delta x} A \\
&A = \begin{cases} 1 & \text{if} \quad f(m,n) = i \text{ and } f(m+\Delta x, y+\Delta y) = j \\ & 0 \text{ else} \end{cases}
\end{aligned} \tag{1}
$$

其中，$P(i,j)$ 为灰度共生矩阵中的元素，表示对于一个有着 G 个灰度级、大小为 $M \times N$ 的图像而言，沿着图像的某一方向移动距离 d，像素的灰度级从 i 变为 j 的概率，$P(i,j \mid \Delta x, \Delta y)$ 也可以用符号 $P(i,j \mid d,\theta)$ 来表示；$f(m,n)$ 表示图像中坐标为 (m,n) 的像素的灰度值。

如图 2 所示，x 方向是图像的列，y 方向是图像的行，$f(x,y)=i$，为坐标 (x,y) 处的灰度值，灰度共生矩阵需要统计的是距离 (dx,dy) 处 $f(x+dx,y+dy)=j$ 的频数 (概率)，由于 (dx,dy) 的选择不同，则会导致角度的不同，通常选择 0°、45°、90° 和 135°。

灰度共生矩阵的计算量是由图像的灰度级和图像的大小共同决定的。例如，假定图像 I 有 L 个灰度级，其大小为 $R \times C$，则运算量大约为 $L^2 \times R \times C$。实际生活中，L 一般为 256, 若令 $R=512, C=512$，则至少要 1.7×10^{10} 次运算，约需耗费 30 分钟，这显然不太切合实际的。因此在计算灰度共生矩阵时，在不影响纹理特征的前提下往往先对灰度图像的灰度值进行量化，压缩到一个较小的范围，一般取 16 级。代码实现如下：

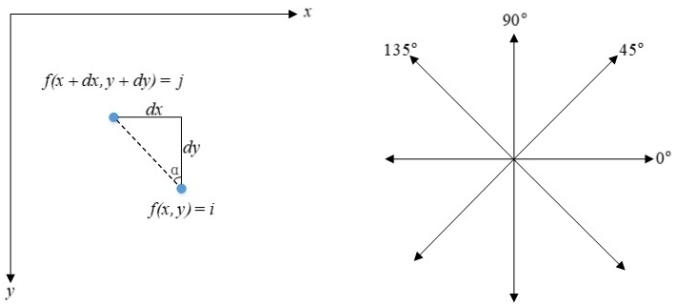


图 2: 灰度共生矩阵的几何表示

```python
# 图像灰度值量化
def image_requantize(image, gray_level):
    bins = np.linspace(0, 255, gray_level + 1)
    image = np.digitize(image, bins)
    image[image > gray_level] = gray_level
    image = image - 1
    return image
```

main.py

除此之外，通过直方图均衡化对图像进行预处理也是计算灰度共生矩阵的常用技巧，从而可以消除绝对灰度值的影响。通常，更多地是使用各向同性的灰度共生矩阵来提取具有旋转不变性的纹理特征，即分别计算 0°、45°、90° 和 135° 四个方向的灰度共生矩阵，如图 3 所示，然后求均值得到各向同性的灰度共生矩阵。

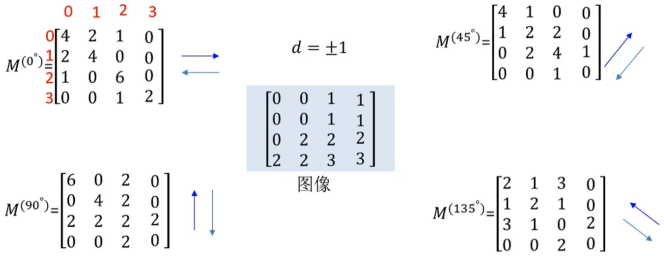


图 3: 简单的图像矩阵及其灰度共生矩阵的示例

统计方法的纹理分析都是通过统计向量 (特征向量) 来描述区域中的纹理，即在一个局部窗口中计算局部的纹理表征，然后在图像中滑动窗口，最终获得整个图像的纹理表征。灰度共生矩阵也是在这样一个 $w \times w$ 大小的滑动窗口中进行计算的。对于有 G 个灰度级的图像而言，其灰度共生矩阵的维度为 $G \times G$。以图 3 为例，灰度共生矩阵的计算过程如下：

1) 如图 3 所示，图像的灰度级为 4，因此首先创建一个大小为 4×4 的空灰度共生矩阵；
2) 分别按照 $(d=1 \ \ =0°)$、$(d=1 \ \ =45°)$、$(d=1 \ \ =90°)$ 和 $(d=1 \ \ =135°)$ 进行计算；
3) 以 $(d=1 \ \ =0°)$ 为例，如第一个值，代表的是图像中灰度值为 0，且其水平方向上距离一个像素的点灰度值也为 0，即灰度值对 $(0,0)$ 出现的次数；

4) 滑动窗口遍历整个图像，并求均值即可完成灰度共生矩阵的计算。

代码实现如下[1]：

```
# 计算灰度共生矩阵
def calculate_glcm(input_win, step_x, step_y,
    gray_level):
    w, h = input_win.shape
    ret = np.zeros([gray_level, gray_level])
    for index_h in range(h):
        for index_w in range(w):
            try:
                row = int(input_win[index_h][
                    index_w])
                col = int(input_win[index_h +
                    step_y][index_w + step_x])
                ret[row, col] += 1
            except:
                continue
    return ret
...
mean_glcm = (calculate_glcm(input_win=
    windowed_image[w][h], step_x=1, step_y=0,
    gray_level=gray_level) +
             calculate_glcm(input_win=
                 windowed_image[w][h], step_x=0,
                 step_y=1, gray_level=gray_level)
                 +
             calculate_glcm(input_win=
                 windowed_image[w][h], step_x=1,
                 step_y=1, gray_level=gray_level)
                 +
            calculate_glcm(input_win=
                windowed_image[w][h], step_x=-1,
                step_y=1,gray_level=gray_level)) /
                 4
```

main.py

灰度共生矩阵主要用于纹理特征的提取，但是，一般不直接采用灰度共生矩阵来进行对纹理的统计特性进行度量，而是进一步基于灰度共生矩阵构建统计量。Haralick 提出了 14 种基于灰度共生矩阵计算出来的统计量，即能量、熵、对比度、均匀性、相关性、方差、和平均、和方差、和熵、差方差、差平均、差熵、相关信息测度以及最大相关系数[2]。这些特征彼此之间具有较强的相关性，一般采用对比度、角度方向二阶矩、熵和平均值来描述纹理的特性。本文采用了如表 1 所示的统计量，在将统计量可视化的基础上针对不同的图像进行选择，尽可能建立最有助于纹理分割的纹理特征表示。部分代码实现如下[3]。

在获得基于灰度共生矩阵的统计量后，对其进行均值滤波和标准化处理可以获得更好的分割效果。

```
class glcm_features:
    def __init__(self, glcm, gray_level):
        self.glcm = glcm
        self.gray_level = gray_level

    def calculate_glcm_mean(self):
        mean = np.zeros((self.glcm.shape[2], self.
            glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                mean += self.glcm[i, j] * i / self
                    .gray_level ** 2
        return mean

    def calculate_glcm_variance(self):
        mean = self.calculate_glcm_mean()
        variance = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                variance += self.glcm[i, j] * (i -
                    mean) ** 2
        return variance

    def calculate_glcm_inertia(self):
        inertia = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                inertia += 1 / (1 + (i - j) ** 2)
                    * self.glcm[i, j]
        return inertia

    def calculate_glcm_contrast(self):
        contrast = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                contrast += self.glcm[i, j] * (i -
                    j) ** 2
        return contrast

    def calculate_glcm_dissimilarity(self):
        dissimilarity = np.zeros((self.glcm.shape
            [2], self.glcm.shape[3]), dtype=np.
            float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                dissimilarity += self.glcm[i, j] *
                    np.abs(i - j)
        return dissimilarity

    . . .
```

glcm_features.py

1. 由于篇幅原因，部分中间过程代码在此省略
2. 相关统计量的计算方法将在附录 A 给出
3. 完整代码见附录 B

2.1.2 Gabor 滤波器

对于纹理分析而言，采用 Gabor 滤波具有较好的时-频局部话特性，它能够同时表示和捕捉二维信号在空间位置、空间频率、方向选择性和相位、频率带宽等方面的信息。用 Gabor 滤波器对图像信号滤波，相当于将其按不同方向、频段进行分解，如果计算经不同方向、频段滤波后的纹理特征，这些特征将具有明显的差异，根据这些差异就可以区分不同的纹理。在计算机视觉和图像处理中，多通道 Gabor 方法被认为是一种非常有用的工具，特别是在纹理分析方面。自从一维 Gabor 函数提出之后，很多学者对 Gabor 分析方法进行了研究，研究表明在哺乳类生物视觉系统中，特别是涉及到纹理方面，Gabor 线性空间滤波器有着非常重要的作用。本文采用的基于 Gabor 滤波器的纹理分析方法[4]如图 4 所示。

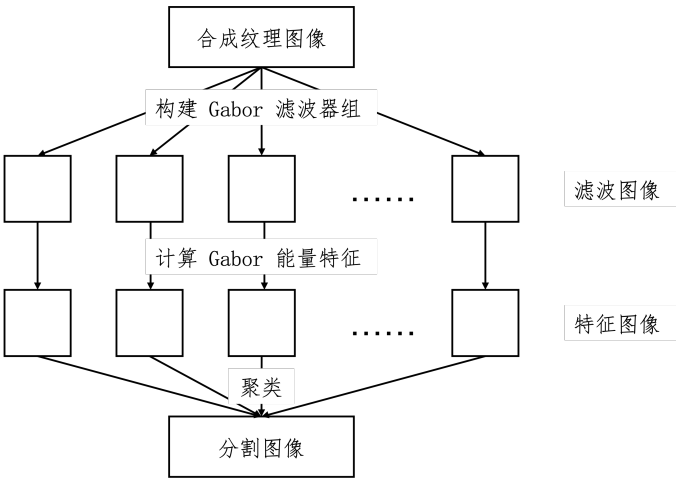


图 4: 基于 Gabor 滤波器的纹理分析及纹理分割算法流程图

Gabor 滤波的方法实际上是多分辨率分解的过程，该方法类似小分析方法。用 Gabor 滤波器进行纹理分析有几个步骤：

1) 设计一个对应于不同空间频率和方向的滤波器组；
2) 把原图像分解为多个滤波图像；
3) 利用 Gabor 能量特征从滤波图像中提取纹理特征；
4) 在特征空间中聚类，生成分割图像。

在滤波图像的基础上提取纹理特征，生成特征图像，相同特征区域中的像素具有相同的纹理特性，在特征空间中它们彼此靠近，无监督纹理分割方法即将图像像素聚类到几个簇中，并表现出缘由图像纹理区域，然后给每个簇定义新的标签实现纹理图像分割的目的。一个好的“分类器”在特征空间要实现类内距离尽量小，而类间距离尽量大。下一小节将对基于聚类技术的纹理分割进行介绍。

2.2 纹理分割

纹理分割本质上是一个图像分割问题，即采用聚类的方法对图像进行分割，而聚类的特征空间则是由图像的纹理特征构建的。常用的聚类方法，包括 K-Means、Mean-Shift 和 DBScan 等。相较于后者而言，K-Means 更适合本文的实验数据且收敛速度更快，因此，本文将主要对 K-Means 算法进行介绍。K-Means 算法伪代码如下：

Algorithm 1: K-Means

Input: 输入样本集 $D = \{x_1, x_2, \cdots, x_N\}$, 分簇数 $K = n$, 最大迭代次数为 M, 从分簇样本中随机选取 n 点 $\{u_1, u_2, \cdots, u_n\}$ 作为初始质心

Output: 输出各样本所在簇 $\{C_1, C_2, \cdots, C_n\}$

1 for $m = 1 \to M$ do // m 表示迭代次数
2 $C_1 \Leftarrow \emptyset, C_2 \Leftarrow \emptyset, \cdots, C_n \Leftarrow \emptyset$ // 初始化各簇
3 for $i = 1, 2, ..., N$ do // i 表示样本集编号
4 $d_{i1} \Leftarrow \|x_i - u_1\|^2, d_{i2} \Leftarrow \|x_i - u_2\|^2, \cdots, d_{in} \Leftarrow \|x_i - u_n\|^2$ // 计算 x_i 到各质心的欧式距离
5 if $argmin(d_{in}) == n$ then
6 $C_n \Leftarrow C_n \cup \{x_i\}$ // 将 x_i 划分到相应的簇
7 $\tilde{u_1} \Leftarrow \frac{1}{|C_1|} \sum_{x \in C_1} x, \tilde{u_2} \Leftarrow \frac{1}{|C_2|} \sum_{x \in C_2} x, \cdots, \tilde{u_n} \Leftarrow \frac{1}{|C_n|} \sum_{x \in C_n} x$ // 重新计算各簇质心
8 if $\forall \tilde{u_n}, \tilde{u_n} == u_n$ then // 各簇质心未改变，跳出循环
9 break from line 3 else
10 $u_1 \Leftarrow \tilde{u_1}, u_2 \Leftarrow \tilde{u_2}, \cdots, u_n \Leftarrow \tilde{u_n}$ // 更新各簇质心
11 return $C_1, C_2, \cdots, C_n$ // 输出结果

代码实现如下[5]：

```
# K 均值聚类
def k_means(matrix, k):
    w, h, c = matrix.shape
    centers = np.array(
        [matrix[np.random.randint(low=0, high=w)
            ][[np.random.randint(low=0, high=h)
            ]][0] for i in range(k)])
    origin_centers = np.zeros_like(centers)
    clusters = np.zeros((w, h))
    while cluster_center_change(centers,
        origin_centers) != 0:
        for i in range(w):
            for j in range(h):
                distance = calculate_distance([i,
                    j], centers, matrix)
                cluster = np.argmin(distance)
                clusters[i][j] = cluster
        origin_centers = copy.deepcopy(centers)
        for k in range(k):
```

4. 本方法利用网络资源进行实现，由于篇幅原因其实现不再具体展示

5. 由于篇幅原因，部分中间过程代码在此省略

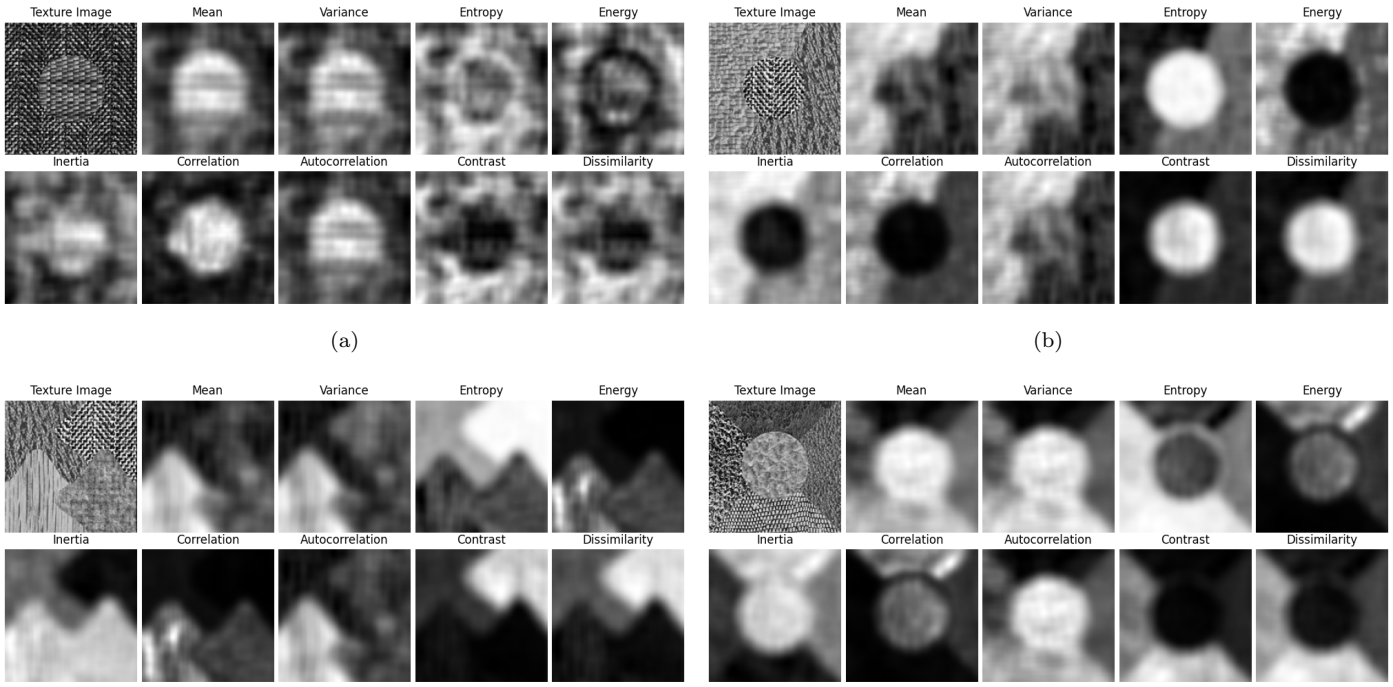


图 5: 灰度共生矩阵的统计特征可视化结果：(a) Texture_mosaic_1, (b) Texture_mosaic_2, (c)Texture_mosaic_3, (d) Texture_mosaic_4; 第一行：纹理图像、Mean、Variance、Entropy、Energy; 第二行：Inertia、Correlation、Autocorrelation、Contrast、Dissimilarity.

表 1: 基于灰度共生矩阵的统计量及不同合成纹理图像指标选取示意表

	Mean	Variance	Entropy	Energy	Inertia	Correlation	Autocorrelation	Contrast	Dissimilarity
1	✓	✓	✓	✓		✓		✓	
2	✓	✓	✓	✓	✓	✓			
3		✓	✓		✓		✓	✓	✓
4	✓	✓			✓		✓	✓	✓

```
            points = []
            for i in range(w):
                for j in range(h):
                    if clusters[i][j] == k:
                        points.append(matrix[i][j
                            ])
            centers[k] = np.mean(np.array(points),
                axis=0)
    return clusters
```

main.py

3 实验

3.1 灰度共生矩阵法的纹理分割实验

如前文所述，在采用灰度共生矩阵法对纹理图像进行分割时，拟在表 1 中列出的统计量中进行选择用以对纹理进行表示和度量。因此本文首先先对测试图像的统计量结果进行了可视化处理，如图 5 所示。在此基础上，根据不同统计量的可视化结果分别对测试图像使用的统计量进行选择，选取结果如表 1 所示。

在固定窗口大小为 19，滑动步长为 1，灰度等级为 16 的前提下，按照选取的统计量逐步计算各图像的灰度共生矩阵及相关统计量，并采用 K-Means 聚类方法对纹理图像进行分割，分割结果如图 6 所示。

分割结果受到窗口大小、滑动步长及聚类初始中心点的选择等因素的影响，文本将会在对比实验部分选择其中的一些因素对分割结果的影响进行探讨。

3.2 Gabor 滤波器的纹理分割实验

固定参数 Gabor 滤波器核的大小为 19，高斯窗口大小为 31，高斯函数的纵横比为 0.5，高斯窗口函数的标准差为 7，用于聚类的行和列的空间权重为 2，在这一参数条件下，Gabor 滤波器的纹理分割结果如图 7 所示。

对比图 6 来看，与灰度共生矩阵的纹理分割结果相比，如果忽略聚类初始中心点选择的影响，Gabor 滤波器的纹理

(a) (b) (c) (d)

图 6: 灰度共生矩阵的纹理分割结果：(a) Texture_mosaic_1, (b) Texture_mosaic_2, (c)Texture_mosaic_3, (d) Texture_mosaic_4.

(a) (b) (c) (d)

图 7: Gabor 滤波器的纹理分割结果：(a) Texture_mosaic_1, (b) Texture_mosaic_2, (c)Texture_mosaic_3, (d) Texture_mosaic_4.

分割结果在不同纹理的交界处的分割效果更好，这是由于其提取目标的局部空间和频率域信息方面具有良好的特性，且 Gabor 小波对于图像的边缘敏感，能够提供良好的方向选择和尺度选择特性，对噪声更加鲁棒。

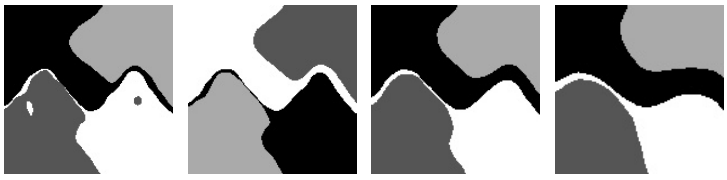
3.3 对比实验

这一章节将在测试图片 Texture_mosaic_3 上对不同的参数进行调整，进行对比实验。为消除聚类初始中心点选择对分割结果的影响，根据图像特性，选择图像的四个四分位点作为聚类的初始中心。下面分别对滑动窗口大小和步长对分割结果的影响进行讨论分析。

3.3.1 滑动窗口大小的影响

在探究滑动窗口大小对纹理分割结果的影响时，固定其他参数不变，窗口大小分别选取 13、19、30 和 50，分割结果如图 8 所示。

从图 8 可以看出，当滑动窗口的选择偏小时，在区域内部会有被错分的较小区域，其原因可能是窗口的选择过小，导致对纹理特征的估计发生了错误。而随着窗口的增大，区域内部被错分的情况消失，但随之出现的是区域的交界处易被划分错误，其原因可能是窗口的选择过大，不能很好地捕捉到纹理的局部特性。



(a) (b) (c) (d)

图 8: 不同大小滑动窗口得到纹理分割结果：(a) $win\_size = 13$, (b) $win\_size = 19$, (c) $win\_size = 30$, (d) $win\_size = 50$.

3.3.2 滑动步长大小的影响

在探究滑动步长大小对纹理分割结果的影响时，固定其他参数不变，步长大小分别选取 1、2 和 3，分割结果如图 9 所示。

从图 9 可以看出，随着滑动步长的逐渐增大，分割边界逐渐变得不再平滑，呈现出锯齿状。

(a) (b) (c)

图 9: 不同大小滑动步长得到纹理分割结果：(a) $stride = 1$, (b) $stride = 2$, (c) $stride = 3$.

4 总结

本文主要的对与图像纹理相关的两个任务进行了探讨和实验，即纹理表征和纹理分割。相比较基于阈值的方法将图像分割为目标和背景两部分，基于聚类技术的纹理图像分割任务更加具有挑战性，其难点在于前期的纹理分析，如何较好地构建纹理特征，将在很大程度上对后续的纹理分割有影响。本文分别采用了灰度共生矩阵和 Gabor 滤波器两种方法分析图像的纹理特征。灰度共生矩阵方法简单，易于实现，但无法利用全局信息，与人类视觉模型不匹配，且计算复杂度较高，计算耗时。而 Gabor 滤波器具有较好的时-频局部话特性，其频率和方向与人类的视觉系统类似，特别适合于纹理表征与分割。

附录 A
灰度共生矩阵统计量计算公式

1) 能量

$$f_1 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} p(i,j)^2 \tag{2}$$

2) 对比度

$$f_2 = \sum_{k=0}^{N_g-1} k^2 p_{x-y}(k) \tag{3}$$

3) 相关性

$$f_3 = \frac{\sum_{i=1}^{N_g}\sum_{j=1}^{N_g}(ij)p(i,j) - \mu_x\mu_y}{\sigma_x\sigma_y} \tag{4}$$

4) 方差

$$f_4 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g}(i-\mu)^2 p(i,j) \tag{5}$$

5) 均匀性

$$f_5 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g}\frac{1}{1+(i-j)^2}p(i,j) \tag{6}$$

6) 和平均

$$f_6 = \sum_{i=2}^{2N_g} i p_{x+y}(i) \tag{7}$$

7) 和方差

$$f_7 = \sum_{i=2}^{2N_g}(i-f_8)^2\, p_{x+y}(i) \tag{8}$$

8) 和熵

$$f_8 = -\sum_{i=2}^{2N_g} p_{x+y}(i)\log\left(p_{x+y}(i)\right) \tag{9}$$

9) 熵

$$f_9 = -\sum_{i=1}^{N_g}\sum_{j=1}^{N_g} p(i,j)\log(p(i,j)) \tag{10}$$

10) 差方差

$$f_{10} = \text{ variance of } p_{x-y} \tag{11}$$

11) 差熵

$$f_{11} = -\sum_{i=0}^{N_g-1} p_{x-y}(i)\log\left(p_{x-y}(i)\right) \tag{12}$$

12) 相关信息测度 1

$$f_{12} = \frac{f_9 - HXY1}{\max(HX, HY)} \tag{13}$$

13) 相关信息测度 2

$$f_{13} = \sqrt{1-\exp^{-2(HXY2-f_9)}} \tag{14}$$

$$HXY1 = -\sum_{i=1}^{N_g}\sum_{j=1}^{N_g} p(i,j)\log\left(p_x(i)p_y(j)\right) \tag{15}$$

$$HXY2 = -\sum_{i=1}^{N_g}\sum_{j=1}^{N_g} p_x(i)p_y(j)\log\left(p_x(i)p_y(j)\right) \tag{16}$$

14) 最大相关系数

Q 的第二大特征值的平方根

$$\mathrm{Q}(i,j) = \sum_k \frac{p(i,k)p(j,k)}{p_x(i)p_y(k)} \tag{17}$$

附录 B
灰度共生矩阵统计量代码实现

```python
class glcm_features:
    def __init__(self, glcm, gray_level):
        self.glcm = glcm
        self.gray_level = gray_level

    def calculate_glcm_mean(self):
        mean = np.zeros((self.glcm.shape[2], self.
            glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                mean += self.glcm[i, j] * i / self
                    .gray_level ** 2
        return mean

    def calculate_glcm_variance(self):
        mean = self.calculate_glcm_mean()
        variance = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
```

```
            for j in range(self.gray_level):
                variance += self.glcm[i, j] * (i -
                    mean) ** 2
        return variance

    def calculate_glcm_inertia(self):
        inertia = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                inertia += 1 / (1 + (i - j) ** 2)
                    * self.glcm[i, j]
        return inertia

    def calculate_glcm_contrast(self):
        contrast = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                contrast += self.glcm[i, j] * (i -
                    j) ** 2
        return contrast

    def calculate_glcm_dissimilarity(self):
        dissimilarity = np.zeros((self.glcm.shape
            [2], self.glcm.shape[3]), dtype=np.
            float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                dissimilarity += self.glcm[i, j] *
                    np.abs(i - j)
        return dissimilarity

    def calculate_glcm_entropy(self):
        eps = 1e-10
        entropy = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                entropy -= self.glcm[i, j] * np.
                    log10(self.glcm[i, j] + eps)
        return entropy

    def calculate_glcm_energy(self):
        energy = np.zeros((self.glcm.shape[2],
            self.glcm.shape[3]), dtype=np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                energy += self.glcm[i, j] ** 2
        return energy

    def calculate_glcm_correlation(self):
        mean = self.calculate_glcm_mean()
        variance = self.calculate_glcm_variance()
        correlation = np.zeros((self.glcm.shape
            [2], self.glcm.shape[3]), dtype=np.
            float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                correlation += ((i - mean) * (j -
                    mean) * (self.glcm[i, j] ** 2)
                    ) / variance
        return correlation

    def calculate_glcm_auto_correlation(self):
        auto_correlation = np.zeros((self.glcm.
            shape[2], self.glcm.shape[3]), dtype=
            np.float32)
        for i in range(self.gray_level):
            for j in range(self.gray_level):
                auto_correlation += self.glcm[i, j
                    ] * i * j
        return auto_correlation
```

glcm_features_full.py