**数字图像处理仿真实验报告**

**姓名 孔睿志 ­­­­ 学号 28210202 ； 2022 级 192201 班**

**成 绩\_\_\_\_\_\_\_\_\_\_\_\_**

**课程名称：数字图像处理 教师签字\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
| **实验内容** | **数字图像处理期末大作业** | **实验时间** | **12.15** |
| **实验目的** | 1. 用 Matlab 设计实现图形化界面，调用后台函数完成设计，函数尽量使用自己编写的函数。设计完成后，点击 GUI图形界面上的菜单或者按钮，进行必要的交互式操作后，最终能显示运行结果。 2. 熟练运用所学知识，在综合实践中培养技能。 3. 运用所学知识，对图像进行处理，用传统机器学习或者深度学习方法，训练智能模型实现图像的分类。 | | |
| **实验工具** | **Matlab R2023b** | | |
| 1. **背景介绍**   全球生物多样性现状，生物多样性是指地球上各种生命形式，包括动植物、微生物及其基因和生态系统的多样性。它对维持地球生态平衡、提供生态服务和促进人类福祉至关重要。然而，全球生物多样性正面临前所未有的威胁，包括栖息地丧失、气候变化、非法狩猎和过度开发等因素。据《全球生物多样性展望报告》指出，鸟类作为生态系统中重要的指示物种之一，正遭遇着显著的生存压力。全球约有 13% 的鸟类物种处于濒危状态，其中许多物种仅存在于特定区域，保护这些物种迫在眉睫。  鸟类作为生态系统的关键物种，在种子传播、害虫控制和生态系统健康维持方面扮演着重要角色。它们的存在不仅影响植物群落的结构和功能，也对其他动物种群的稳定起到至关重要的作用。因此，鸟类的多样性和分布状况可以反映出生态系统的整体健康状况。  随着全球生物多样性丧失的加剧，多国合作和国际组织的努力变得尤为重要。例如超级生物多样性同盟（Super Biodiversity Alliance）等国际组织，旨在通过合作保护全球范围内的关键物种及其栖息地，推动生物多样性保护项目。该联盟包括多个具有生物多样性关键地区的国家，倡导通过数据共享、跨境保护区建设和科学研究，提升全球生态保护水平。  在此背景下，保护鸟类及其栖息地，特别是在“关键生物多样性国家”中，对于实现全球生物多样性保护目标具有重要意义。通过技术手段，如数字图像处理和人工智能的结合，能有效提升鸟类物种的监测和识别效率，为保护工作提供精准的数据支持。  数字图像处理与鸟类物种识别 随着计算机视觉和深度学习技术的快速发展，数字图像处理在生物多样性监测中展现出巨大的潜力。通过高分辨率图像数据和高效的物种识别算法，能够快速、准确地识别鸟类物种并跟踪其栖息地变化。特别是在野外环境中，通过无人机、监控摄像头等设备采集鸟类图像，再结合深度学习模型进行自动分类，能够极大提升鸟类物种识别的效率和准确性。  本实验旨在利用数字图像处理技术，建立一个鸟类种类识别系统。通过识别鸟类的种类，为鸟类保护工作提供科学依据。同时，研究也将为全球范围内鸟类保护项目，特别是超级生物多样性同盟成员国提供技术支持，促进跨国合作和全球鸟类多样性保护的实现。   1. **程序流程**   1) 对于打开的图像可以显示其灰度直方图，并实现直方图均衡化和直方图匹配(规定化)。  2) 读取图像并灰度化，然后实现该灰度图像的对比度增强，要求实现线性变换和非线性变换(包括对数变换和指数变换)。  3) 实现图像的缩放变换、旋转变换等。  4) 图像加噪(用输入参数控制不同噪声)，然后分别使用空域和频域进行滤波处理。  5) 采用 roberts算子，prewitt 算子，sobel 算子，拉普拉斯算子对图像进行边缘提取。  6) 读入图像，应用所学的知识从图像中提取出目标。  7) 对原始图像和所提取的目标分别进行特征提取(LBP和HOG)  8) 应用传统机器学习或者深度学习方法，训练智能模型实现图像的分类。  **结果预览**     1. **程序模块**   **1. 直方图**  **显示图像灰度直方图，并实现直方图均衡化和直方图匹配(规定化)**   1. **真彩图转灰度图：rgb\_to\_gray**   **算法原理：**  灰度图像的每个像素只有一个值（表示亮度），而 RGB 图像的每个像素由红色 (R)、绿色 (G)、蓝色 (B) 三个分量的值组合表示。将 RGB 图像转化为灰度图像的过程，就是将 R、G、B 三个通道的像素值合成为一个单通道的亮度值。  亮度值通常是根据 R、G、B 通道的贡献比例进行加权计算。一般情况下，绿色对亮度的贡献最大，其次是红色，蓝色最小。  **标准加权公式：**  **code：**   1. function gray\_img = rgb\_to\_gray(img) 2. % 检查输入图像是否是 RGB 图像 3. **if** size(img, 3) ~= 3 4. error('输入图像必须是RGB图像'); 5. end 7. % 灰度 = 0.299 \* R + 0.587 \* G + 0.114 \* B 8. gray\_img = 0.299 \* **double**(img(:,:,1)) + 0.587 \* **double**(img(:,:,2)) + 0.114 \* **double**(img(:,:,3)); 10. % 转换为 uint8 类型 11. gray\_img = uint8(gray\_img); 12. end 13. **计算灰度直方图：computeHistogram**   **算法原理：**  计算灰度图像的直方图，即统计每个灰度值（0-255）在图像中出现的次数。创建一个长度为 256 的数组 hist，每个元素初始值为 0，用于记录每个灰度值的频率。对图像的每个像素进行循环，根据像素的灰度值增加对应索引的计数。  **code：**   1. % 计算灰度直方图 2. function hist = computeHistogram(gray\_img) 3. **if** size(gray\_img, 3) > 1 4. error('输入必须是灰度图！'); 5. end 7. hist = zeros(1, 256); 8. **for** i = 1:numel(gray\_img) 9. hist(gray\_img(i) + 1) = hist(gray\_img(i) + 1) + 1; 10. end 11. end 12. **直方图均衡化：histogramEqualization**   **算法原理：**  通过重新分布灰度值，提高图像对比度，使暗部和亮部细节更加明显。主要步骤如下：  计算灰度直方图：统计每个灰度值（0-255）的像素数量。  计算累计分布函数 (CDF)：累加直方图，计算每个灰度值及以下像素的累计数量。  归一化 CDF：将CDF归一化到 [0, 255] 的范围，生成灰度值的映射关系。  映射灰度值：遍历图像的每个像素，用归一化的 CDF 映射原始灰度值到新的均衡化灰度值。  输出均衡化图像：生成增强对比度的图像。  **code：**   1. % 直方图均衡化 2. function equalized\_img = histogramEqualization(gray\_img) 3. % 检查输入是否为灰度图 4. **if** size(gray\_img, 3) > 1 5. error('输入必须是灰度图！'); 6. end 8. % 获取图像大小和总像素数 9. [rows, cols] = size(gray\_img); 10. num\_pixels = rows \* cols; 12. % 计算灰度直方图 13. hist = computeHistogram(gray\_img); 15. % 计算累计分布函数 (CDF) 16. cdf = zeros(1, 256); 17. cdf(1) = hist(1); % 第一个值等于直方图的第一个值 18. **for** i = 2:256 19. cdf(i) = cdf(i-1) + hist(i); 20. end 22. % 归一化累计分布函数，使其映射到 [0, 255] 23. cdf\_min = min(cdf(cdf > 0)); % 最小的非零值 24. cdf\_normalized = round((cdf - cdf\_min) / (num\_pixels - cdf\_min) \* 255); 26. % 应用直方图均衡化映射规则 27. equalized\_img = zeros(rows, cols, 'uint8'); % 初始化均衡化后的图像 28. **for** i = 1:rows 29. **for** j = 1:cols 30. equalized\_img(i, j) = cdf\_normalized(gray\_img(i, j) + 1); 31. end 32. end 33. end 34. **直方图匹配：histogramMatching**   **算法原理：**  直方图匹配是一种将一幅图像的灰度分布调整为与目标图像相似的技术。其核心思想是，通过调整源图像的灰度分布，使得它的累积分布函数 (CDF) 尽可能接近目标图像的 CDF。  首先计算源图像和目标图像的 CDF，其次构建灰度值映射表，最后根据映射表调整源图像的灰度值。  **code：**   1. % --- 直方图匹配函数 --- 2. function matchedImage = histogramMatching(sourceImage, targetImage) 3. % 将图像转换为灰度图（如需要） 4. **if** size(sourceImage, 3) == 3 5. sourceImage = rgb2gray(sourceImage); 6. end 7. **if** size(targetImage, 3) == 3 8. targetImage = rgb2gray(targetImage); 9. end 11. % 计算源图像和目标图像的灰度直方图及其累积分布函数 (CDF) 12. sourceHist = computeHistogram(sourceImage); 13. targetHist = computeHistogram(targetImage); 15. sourceCdf = cumsum(sourceHist) / numel(sourceImage); 16. targetCdf = cumsum(targetHist) / numel(targetImage); 18. % 构建映射表，将源图像的灰度值映射到目标图像的灰度值 19. mapping = zeros(256, 1, 'uint8'); 20. targetIndex = 1; 21. **for** sourceIndex = 1:256 22. **while** targetIndex < 256 && targetCdf(targetIndex) < sourceCdf(sourceIndex) 23. targetIndex = targetIndex + 1; 24. end 25. mapping(sourceIndex) = targetIndex - 1; 26. end 28. % 根据映射表对源图像进行像素值替换 29. matchedImage = mapping(**double**(sourceImage) + 1); 30. end   **运行结果：**    **均衡化后的图像整体亮度分布更均匀，图像细节更加清晰。直方图显示为更平滑的分布，消除了原始图像中的亮度偏差。**  **匹配后的图像在灰度分布上更接近目标图像。** | | | |

|  |
| --- |
| **2.增强图像对比度**  **读取图像并灰度化，然后实现该灰度图像的对比度增强，要求实现线性变换和非线性变换(包括对数变换和指数变换)。**   1. **线性变换：linearTransform**   **算法原理：**  线性变换是一种用于增强图像对比度的简单方法，通过减去最小值和按比例缩放，将像素值范围调整到 [0, 255]，从而调整图像的亮度和对比度，使图像的细节更加清晰。  **线性变换公式：**  **code：**   1. % 线性变换函数 2. function linearImg = linearTransform(grayImg) 3. minVal = **double**(min(grayImg(:))); 4. maxVal = **double**(max(grayImg(:))); 5. % 线性变换公式：标准化到 [0, 255] 6. linearImg = uint8(255 \* (**double**(grayImg) - minVal) / (maxVal - minVal)); 7. end   **运行结果：**     1. **分段线性变换：linearTransform**   **算法原理：**  通过将灰度值划分为若干区间，并对每个区间内的灰度值应用线性插值，实现对图像亮度和对比度的分段线性调整。  **分段的线性插值公式：**  **code：**  **三段式：**   1. function output\_value = threeSegmentLinearTransform(input\_value, a, b, c, d) 2. % 三段线性灰度级变换 3. % 输入: 4. %   input\_value: 输入像素值 (0-1 或 0-255) 5. %   a, b: 分段阈值 (输入灰度值范围中的两个分段点) 6. %   c, d: 输出灰度值范围对应的映射点 7. % 输出: 8. %   output\_value: 经过三段线性变换后的像素值 9. a=30/256; b=100/256; c=75/256; d=250/256;  %参数设置 10. **if** input\_value < a 11. % 第一段 [0, a] 12. output\_value = input\_value \* c / a; 13. elseif input\_value < b 14. % 第二段 [a, b] 15. output\_value = (input\_value - a) \* (d - c) / (b - a) + c; 16. **else** 17. % 第三段 [b, 1] 18. output\_value = (input\_value - b) \* (1 - d) / (1 - b) + d; 19. end 20. end   **多段式：**   1. function output\_img = piecewiseLinearTransform(input\_img) 2. % 输入： 3. %   input\_img: 输入图像（灰度或彩色） 4. % 输出： 5. %   output\_img: 经过分段线性变换后的灰度图像 7. % 检查是否为彩色图像，如果是则转换为灰度图 8. **if** size(input\_img, 3) == 3 9. input\_img = rgb\_to\_gray(input\_img); 10. end 12. % 将输入图像转换为双精度数组 13. input\_img = **double**(input\_img); 15. % 动态生成控制点：5个点 16. num\_points = 5;  % 控制点数量 17. x\_points = linspace(0, 255, num\_points);  % 灰度值等间隔分布 18. y\_points = [0, 50, 180, 220, 255];        % 自定义输出灰度值映射关系 20. % 初始化输出图像 21. [rows, cols] = size(input\_img); 22. output\_img = zeros(rows, cols); 24. % 对每个像素进行分段线性插值 25. **for** i = 1:rows 26. **for** j = 1:cols 27. pixel\_value = input\_img(i, j);  % 当前像素值 28. % 找到当前像素所在的分段 29. **for** k = 1:(num\_points - 1) 30. **if** pixel\_value >= x\_points(k) && pixel\_value <= x\_points(k + 1) 31. % 线性插值计算 32. x1 = x\_points(k); 33. y1 = y\_points(k); 34. x2 = x\_points(k + 1); 35. y2 = y\_points(k + 1); 36. output\_img(i, j) = y1 + (pixel\_value - x1) \* (y2 - y1) / (x2 - x1); 37. **break**; 38. end 39. end 40. end 41. end 43. % 转换为 uint8 格式输出 44. output\_img = uint8(output\_img); 45. end   **运行结果：**     1. **对数变换：logTransform**   **算法原理：**  增强暗部细节：对数函数的快速增长特性使得较小的灰度值（暗部）拉伸幅度较大，从而增强暗部的对比度和细节。  压缩亮部灰度值：对数函数的增长逐渐趋缓，使得较大的灰度值（亮部）变化幅度减小，从而避免亮部过度曝光。  **对数变换公式：**  **code：**   1. % 对数变换 2. function logImg = logTransform(grayImg) 3. grayImg = **double**(grayImg); 4. c = 255 / log(1 + max(grayImg(:))); % 对数常数 5. logImg = uint8(c \* log(1 + grayImg)); 6. end   **运行结果：**     1. **指数变换：logTransform**   **算法原理：**  指数变换是一种非线性灰度变换方法，与对数变换相反，它可以增强图像中高灰度值的细节，同时压缩低灰度值的范围。其效果是突出图像中亮部区域的细节。  对于较大的指数值，变化幅度较大，从而增强图像中亮部区域的细节。  对于较小的指数值，变化幅度较小，从而压缩图像中暗部区域的灰度值。  **对数变换公式：**  **code：**   1. % 指数变换 2. function expImg = expTransform(grayImg) 3. grayImg = **double**(grayImg); 4. c = 255 / (exp(max(grayImg(:)) / 255) - 1); % 指数常数 5. expImg = uint8(c \* (exp(grayImg / 255) - 1)); 6. end   **运行结果：**    **3.图像几何变换**   1. **图像缩放：**   **算法原理：**  通过双线性插值实现对灰度图像和多通道彩色图像的缩放。核心是基于目标像素的反向映射，利用邻近像素的加权平均估算目标像素值。  双线性插值通过利用目标像素周围的四个邻近像素值，结合水平和垂直方向的距离，对目标像素值进行插值估算。  **公式如下：**  **code：**   1. function resizedImg = resizeColorImage(img, kx, ky) 2. % 判断输入图像是否为彩色图像 3. **if** size(img, 3) == 3 4. % 获取图像的三个颜色通道 5. redChannel = img(:,:,1); 6. greenChannel = img(:,:,2); 7. blueChannel = img(:,:,3); 9. % 对每个通道分别进行缩放 10. resizedRed = bilinearResize(redChannel, kx, ky); 11. resizedGreen = bilinearResize(greenChannel, kx, ky); 12. resizedBlue = bilinearResize(blueChannel, kx, ky); 14. % 确保每个通道的像素值在合理范围 15. resizedRed = mat2gray(resizedRed);   % 对 **double** 类型进行归一化，范围 [0, 1] 16. resizedGreen = mat2gray(resizedGreen); 17. resizedBlue = mat2gray(resizedBlue); 19. % 合并三个缩放后的通道 20. resizedImg = cat(3, resizedRed, resizedGreen, resizedBlue); 21. **else** 22. % 对灰度图像进行缩放 23. resizedImg = bilinearResize(img, kx, ky); 25. % 确保图像像素值在合理范围 26. resizedImg = mat2gray(resizedImg);   % 对 **double** 类型进行归一化，范围 [0, 1] 27. end 28. end 29. % 双线性插值函数 30. function g = bilinearResize(f, kx, ky) 31. [srcH, srcW] = size(f); 32. dstH = ceil(srcH \* ky); 33. dstW = ceil(srcW \* kx); 34. g = zeros(dstH, dstW); 36. **for** dstX = 1:dstW 37. **for** dstY = 1:dstH 38. srcX = (dstX-1)/kx; 39. srcY = (dstY-1)/ky; 40. x = floor(srcX)+1; 41. y = floor(srcY)+1; 42. a = srcX - floor(srcX); 43. b = srcY - floor(srcY); 45. **if** x<srcW && y<srcH 46. g(dstY,dstX) = f(y,x) + b\*(f(y+1,x)-f(y,x)) + a\*(f(y,x+1)-f(y,x)) + a\*b\*(f(y+1,x+1)+f(y,x) - f(y,x+1)-f(y+1,x)); 47. elseif x>= srcW && y<srcH % 左上角的点在右边界，只能在垂直方向插值 48. g(dstY,dstX) = f(y,x) + b\*(f(y+1,x)-f(y,x)); 49. elseif x<srcW && y>= srcH % 左上角的点在下边界，只能在水平方向插值 50. g(dstY,dstX) = f(y,x) + a\*(f(y,x+1)-f(y,x)); 51. **else** 52. g(dstY,dstX) = f(y,x); 53. end 54. end 55. end 56. end   **运行结果：**     1. **图像旋转：**   **算法原理：**  通过反向映射和双线性插值实现了对图像的高质量旋转  **公式：**  **绕图像原点逆时针旋转**  **绕原点旋转**  **绕原点旋转逆变换**  旋转变换过程—后向映射法：  1.确定旋转后新图像尺寸：计算原图像四个角在旋转后的坐标，求方向的最大最小值,确定新图像的分辨率  2.坐标变换  3.旋转逆变换  4.给新图像赋值  **code：**   1. function g = rotateImage(f, angle) 2. % 获取输入图像的尺寸 3. [srcH, srcW, channels] = size(f); 5. % 如果是彩色图像（有3个通道） 6. **if** channels == 3 7. % 分别提取RGB三个通道 8. R = f(:,:,1); 9. G = f(:,:,2); 10. B = f(:,:,3); 12. % 对每个通道进行旋转 13. R = rotateSingleChannel(R, angle); 14. G = rotateSingleChannel(G, angle); 15. B = rotateSingleChannel(B, angle); 17. % 合成新的RGB图像 18. g = cat(3, R, G, B); 19. **else** 20. % 如果是灰度图像 21. g = rotateSingleChannel(f, angle); 22. end 23. end 24. % 旋转单通道图像的函数（包括优化的双线性插值） 25. function g = rotateSingleChannel(f, angle) 26. [srcH, srcW] = size(f);  % 获取输入图像的尺寸 27. theta = deg2rad(angle);   % 角度转换为弧度 29. % 计算旋转后的图像尺寸 30. cornerx = [0, srcW-1, srcW-1, 0]; 31. cornery = [0, 0, srcH-1, srcH-1]; 33. newcornerx = cornerx \* cos(theta) + cornery \* sin(theta); % 旋转后的四个角的x坐标 34. newcornery = -cornerx \* sin(theta) + cornery \* cos(theta); % 旋转后的四个角的y坐标 36. minx = min(newcornerx); % 新图像的最小x坐标 37. miny = min(newcornery); % 新图像的最小y坐标 39. dstH = ceil(max(newcornery) - miny + 1); % 目标图像的高度 40. dstW = ceil(max(newcornerx) - minx + 1); % 目标图像的宽度 42. % 创建新的空白图像 43. g = zeros(dstH, dstW); 45. % 旋转图像并应用双线性插值 46. **for** newx = 1:dstW 47. **for** newy = 1:dstH 48. % 将新图像坐标转换为原图像坐标 49. oldx = (newx - 1 + minx) \* cos(theta) - (newy - 1 + miny) \* sin(theta); 50. oldy = (newx - 1 + minx) \* sin(theta) + (newy - 1 + miny) \* cos(theta); 52. **if** oldx < 1 || oldy < 1 || oldx > srcW || oldy > srcH 53. g(newy, newx) = 255;  % 超出原图范围，设为白色（255） 54. **else** 55. % 双线性插值 56. x1 = floor(oldx); x2 = ceil(oldx); 57. y1 = floor(oldy); y2 = ceil(oldy); 59. % 确保不超出图像边界 60. x1 = max(1, x1); x2 = min(srcW, x2); 61. y1 = max(1, y1); y2 = min(srcH, y2); 63. % 获取插值所需的四个像素 64. Q11 = f(y1, x1); 65. Q12 = f(y1, x2); 66. Q21 = f(y2, x1); 67. Q22 = f(y2, x2); 69. % 双线性插值 70. a = oldx - x1; 71. b = oldy - y1; 72. g(newy, newx) = (1-a)\*(1-b)\*Q11 + a\*(1-b)\*Q12 + (1-a)\*b\*Q21 + a\*b\*Q22; 73. end 74. end 75. end 77. % 确保图像像素值在有效范围内 78. g = uint8(min(max(g, 0), 255));  % 裁剪到[0, 255]之间 79. end   **运行结果：**     1. **图像错切：shearImageRGB**   **算法原理：**  沿水平方向（X 方向）或垂直方向（Y 方向）拉伸或压缩图像，同时保持图像的一个维度不变。这种变换使图像看起来发生了“倾斜”的效果。  通过逆变换法计算目标图像中每个像素的原始坐标，可以避免像素空洞问题，再结合双线性插值估算像素值，通过周围四个像素的加权平均值，保证插值结果平滑且高质量。分通道计算，确保颜色信息不受干扰。  **错切变换矩阵表达式：**  **code：**   1. function output\_img = shearImageRGB(img, k\_x, k\_y) 2. % 输入：img - 原始图像 (**double**类型) 3. %        k\_x, k\_y - 错切因子 4. % 输出：output\_img - 错切后的图像 (uint8类型) 6. % 获取图像尺寸 7. [rows, cols, channels] = size(img); 9. % 计算新的图像尺寸 10. new\_cols = round(cols + abs(k\_x) \* rows); 11. new\_rows = round(rows + abs(k\_y) \* cols); 13. % 初始化输出图像 14. output\_img =zeros(new\_rows, new\_cols, channels); 16. % 错切变换公式 17. **for** x = 1:new\_cols 18. **for** y = 1:new\_rows 19. % 逆变换计算原始坐标 20. x\_src = (x - 1 - k\_x \* (y - 1)) / (1 - k\_x \* k\_y); 21. y\_src = (y - 1 - k\_y \* (x - 1)) / (1 - k\_x \* k\_y); 23. % 如果计算得到的坐标在原图像范围内，则进行插值 24. **if** x\_src >= 1 && x\_src <= cols && y\_src >= 1 && y\_src <= rows 25. **for** c = 1:channels 26. % 双线性插值 27. x1 = floor(x\_src); x2 = ceil(x\_src); 28. y1 = floor(y\_src); y2 = ceil(y\_src); 30. % 边界检查 31. x1 = max(1, x1); x2 = min(cols, x2); 32. y1 = max(1, y1); y2 = min(rows, y2); 34. % 获取插值所需的四个像素 35. Q11 = img(y1, x1, c); 36. Q12 = img(y1, x2, c); 37. Q21 = img(y2, x1, c); 38. Q22 = img(y2, x2, c); 40. % 计算插值权重 41. a = x\_src - x1; 42. b = y\_src - y1; 44. % 双线性插值公式 45. value = (1 - a) \* (1 - b) \* Q11 + a \* (1 - b) \* Q12 + (1 - a) \* b \* Q21 + a \* b \* Q22; 46. output\_img(y, x, c) = value; 47. end 48. **else** 49. % 如果坐标超出原图像范围，填充为白色 50. output\_img(y, x, :) = 255; 51. end 52. end 53. end 55. % 将输出图像转换为 uint8 类型 56. output\_img = uint8(output\_img); 57. end   **运行结果：**     1. **图像镜像：**   **horizontal\_flip 水平镜像**  **vertical\_flip 垂直镜像**  **算法原理：**  水平镜像是一种几何变换，将图像的每一行按照列的顺序进行翻转。  垂直镜像是一种几何变换，将图像的每一列按照行的顺序进行翻转。  矩阵索引操作：  end:-1:1，从最后一列/行（end）到第一列/行（1）按步长 −1倒序排列。  逐通道镜像：对于多通道（如 RGB 图像），代码会同时对每个通道执行列倒序操作，确保颜色不被打乱。  **code：**   1. function result = horizontal\_flip(img) 2. % 水平镜像：交换列 3. result = img(:, end:-1:1, :); 4. end 5. function result = vertical\_flip(img) 6. % 垂直镜像：交换行 7. result = img(end:-1:1, :, :); 8. end   **运行结果：**    **4.图像加噪与滤波**  **图像加噪(用输入参数控制不同噪声)，然后分别使用空域和频域进行滤波处理。**   1. **高斯噪声：linearTransform**   **算法原理：**  使用正态分布生成随机噪声，根据用户指定的均值和方差调整噪声的分布特性。  叠加到图像：将噪声矩阵逐像素叠加到原图像，模拟现实中的随机干扰。  裁剪像素值范围：确保加噪后的图像像素值合法，避免出现无效像素。  转换数据类型：输出图像保持与输入图像一致的格式和范围。  **高斯噪声的概率密度函数：**  **code：**   1. function noisy\_img = addGaussianNoise(img, mean, variance) 2. % 添加高斯噪声 3. % 输入： 4. %   img - 原始图像（灰度或彩色图像） 5. %   mean - 高斯噪声的均值 6. %   variance - 高斯噪声的方差 7. % 输出： 8. %   noisy\_img - 加入高斯噪声后的图像 10. % 将图像转换为 **double** 类型，便于计算 11. img = im2double(img); 12. % 生成高斯噪声 13. noise = sqrt(variance) \* randn(size(img)) + mean; 14. % 将噪声叠加到原图像 15. noisy\_img = img + noise; 17. % 限制噪声图像的范围到 [0, 1] 18. noisy\_img = min(max(noisy\_img, 0), 1); 20. % 转换回 uint8 类型输出 21. noisy\_img = im2uint8(noisy\_img); 22. end   **运行结果：**     1. **椒盐噪声：addSaltAndPepperNoise**   **算法原理：**  随机生成噪声位置：根据输入的噪声密度，随机选择一部分像素的位置用于添加噪声。一半像素变为白点（盐噪声），另一半像素变为黑点（椒噪声）。  逐像素修改图像：将选中的盐噪声位置像素值设为最大值 1（白色）。将选中的椒噪声位置像素值设为最小值 0（黑色）。输出为与输入一致的类型：  加噪完成后将图像数据类型转换回 uint8，方便后续显示和存储。  **椒盐噪声的概率密度函数：**  **均值为**  **方差为**  **code：**   1. function noisy\_img = addSaltAndPepperNoise(img, density) 2. % 添加椒盐噪声 3. % 输入： 4. %   img - 原始图像（灰度或彩色图像） 5. %   density - 噪声密度（范围 [0, 1]，例如 0.02 表示 2% 的像素添加噪声） 6. % 输出： 7. %   noisy\_img - 加入椒盐噪声后的图像 9. % 将图像转换为 **double** 类型，便于计算 10. img = im2double(img); 12. % 初始化噪声图像为原图像 13. noisy\_img = img; 15. % 计算图像总像素数 16. num\_pixels = numel(img); 18. % 计算盐噪声和椒噪声的像素数量 19. num\_salt = round(density \* num\_pixels / 2); % 盐噪声数量 20. num\_pepper = round(density \* num\_pixels / 2); % 椒噪声数量 22. % 添加盐噪声（像素值设为1） 23. salt\_idx = randperm(num\_pixels, num\_salt); % 随机选择像素位置 24. noisy\_img(salt\_idx) = 1; 26. % 添加椒噪声（像素值设为0） 27. pepper\_idx = randperm(num\_pixels, num\_pepper); % 随机选择像素位置 28. noisy\_img(pepper\_idx) = 0; 30. % 转换回 uint8 类型输出 31. noisy\_img = im2uint8(noisy\_img); 32. end   **运行结果：**     1. **空域平滑滤波：**   **空域滤波主要指的是基于图像空间的邻域模板运算，也就是说滤波处理要考虑到图像中处理像素点与其周围像素点（邻点）之间的联系。**   * 1. **均值滤波**   **算法原理：**  又称邻域平均法。其基本思想是以某一像素为中心，在它的周围选择一邻域，将邻域内所有点的均值（灰度值相加求平均）来代替原来像素值，通过降低噪声点与周围像素点的差值以去除噪声点。  **函数：**  **S：点(x, y)为中心的邻域，包括点(x, y)**  **M：邻域 S 内总像素数目**  **code：**   1. function output\_img = meanFilter(input\_img, filter\_size) 2. % 创建均值滤波器 3. kernel = ones(filter\_size) / (filter\_size^2);  % 创建一个均值滤波核 5. % 将输入图像转换为 **double** 类型，防止溢出 6. input\_img = **double**(input\_img); 8. % 对图像进行卷积操作 9. **if** size(input\_img, 3) == 1 10. % 如果是灰度图像，直接使用conv2 11. output\_img = conv2(input\_img, kernel, 'same'); 12. **else** 13. % 如果是彩色图像，对每个通道进行卷积操作 14. output\_img = zeros(size(input\_img)); 15. **for** c = 1:size(input\_img, 3) 16. output\_img(:, :, c) = conv2(input\_img(:, :, c), kernel, 'same'); 17. end 18. end 20. % 将结果转换回 uint8 类型（如果输入是 uint8 类型） 21. output\_img = uint8(output\_img); 22. end   **运行结果：**     * 1. **中值滤波**   **算法原理：**  基本原理是，噪声点的出现，使被处理点比周围像素点暗（亮）许多。以被处理点为中心，周围选择一个窗口（邻域），把窗口内所有像素值排序，噪声点必位于序列前（后）端，中值一般为未受到噪声污染，取中值代替该点的值。  一组数：，把n个数按值的大小顺序排列，有：    **code：**   1. function output\_img = medianFilter(input\_img, filter\_size) 2. % input\_img: 输入图像，灰度图像或彩色图像 3. % filter\_size: 滤波器的大小，例如3表示3x3的滤波器 5. [rows, cols, channels] = size(input\_img); 7. % 转换为 **double** 类型，便于计算 8. input\_img = **double**(input\_img); 10. % 初始化输出图像 11. output\_img = zeros(size(input\_img)); 13. % 计算滤波器的半径 14. half\_size = floor(filter\_size / 2); 16. % 遍历每个通道（支持彩色图像） 17. **for** c = 1:channels 18. **for** i = 1 + half\_size : rows - half\_size 19. **for** j = 1 + half\_size : cols - half\_size 20. % 提取当前窗口 21. window = input\_img(i-half\_size:i+half\_size, j-half\_size:j+half\_size, c); 23. % 计算中值并赋值 24. output\_img(i, j, c) = median(window(:)); 25. end 26. end 27. end 29. % 转换回 uint8 类型 30. output\_img = uint8(output\_img); 31. end   **运行结果：**    **中值滤波不适于直接处理点线细节多的图像。**   * 1. **高斯滤波**   **算法原理：**  高斯滤波实质上是图像 f与高斯滤波器 H 的卷积运算。  **k , l值：根据所选邻域（窗口）大小确定；**  **H(r, s)：高斯平滑滤波器的响应函数；**  基本原理是，以被处理点为中心，选择一邻域；对邻域内的点，按照正态分布曲线统计，分配相应权值；将邻域内所有点的加权平均值代替被处理点的值。  高斯滤波器是对二维零均值高斯函数离散近似得到的。据分析，高斯函数的值约99.7%落在半径为的窗口内。因此，设计高斯滤波器的窗口大小：  **其中，6σ 取最近的奇数值。**    **其中，**  **H(i, j)的值要取整且归一化处理。**  **code：**   1. function output\_img = gaussianFilter(input\_img, filter\_size, sigma) 2. % input\_img: 输入图像，可以是灰度图像或彩色图像 3. % filter\_size: 高斯滤波器的大小（例如3表示3x3的滤波器） 4. % sigma: 高斯滤波器的标准差，决定了模糊程度 6. [rows, cols, channels] = size(input\_img); 8. % 生成高斯滤波器核 9. % 1. 创建一个 filter\_size x filter\_size 的高斯核 10. [X, Y] = meshgrid(-(filter\_size-1)/2:(filter\_size-1)/2, -(filter\_size-1)/2:(filter\_size-1)/2); 11. h = exp(-(X.^2 + Y.^2) / (2 \* sigma^2)); 12. h = h / sum(h(:));  % 归一化，确保滤波器的总和为1 14. % 如果是彩色图像，逐通道处理 15. output\_img = zeros(size(input\_img)); 17. **for** c = 1:channels 18. % 对每个通道应用卷积 19. output\_img(:,:,c) = conv2(input\_img(:,:,c), h, 'same'); 20. end 22. % 如果是彩色图像，结果需要转回 uint8 类型 23. output\_img = uint8(output\_img); 24. end   **运行结果：**    **对于高斯噪声，均值滤波、高斯滤波效果比较好。**  **高斯噪声图像中，每点都是污染点，中值滤波选不到合适的干净点。**  **因为高斯噪声是幅值近似正态分布，均值为0，所以根据统计数学，均值滤波、高斯滤波都可以消除高斯噪声。**  **实际上只能减弱，不能消除。**  **对于椒盐噪声，中值滤波去除效果较好，且模糊轻微，边缘保留较好。**  **椒盐噪声图像中除了污染点，有干净点。**  **中值滤波是选择适当的点来替代污染点的值。**  **椒盐噪声均值不为0，均值滤波、高斯滤波都不能很好地去除椒盐噪声点。**   * 1. **双边滤波**   **算法原理：**  双边滤波（Bilateral filter）是既保持边缘清晰又平滑去噪的滤波器。“双边”意味着平滑滤波同时考虑图像的空间位置邻近性和像素灰度的相似性。  双边滤波本质上是一种邻域像素值加权平均的滤波器，实质为图像与双边滤波器的卷积运算。  其中，  **空间邻近度高斯函数**  **表示点和的欧氏距离；为空间邻域标准差；**  **灰度邻近度高斯函数**  **表示点和的灰度值差；为像素灰度标准差；**  **为空间邻近度高斯函数与灰度邻近度高斯函数的乘积。**  通过这两个高斯函数滤波的同时作用，就是双边滤波。  **code：**   1. function output\_img = bilateralFilter(input\_img, filter\_size, sigma\_d, sigma\_r) 2. % input\_img: 输入图像，灰度图像或彩色图像 3. % filter\_size: 滤波器的大小，通常是一个奇数，例如 5 4. % sigma\_d: 空间域的标准差，控制邻域的影响范围 5. % sigma\_r: 灰度域的标准差，控制像素值的相似度 7. [rows, cols, channels] = size(input\_img); 8. % 将输入图像转换为 **double** 类型 9. input\_img = **double**(input\_img); 10. % 初始化输出图像 11. output\_img = zeros(size(input\_img)); 13. % 为每个通道处理 14. half\_size = floor(filter\_size / 2); 16. **for** c = 1:channels 17. **for** i = 1 + half\_size : rows - half\_size 18. **for** j = 1 + half\_size : cols - half\_size 19. % 初始化权重 20. weight\_sum = 0; 21. filtered\_pixel = 0; 23. % 遍历滤波器大小 24. **for** m = -half\_size:half\_size 25. **for** n = -half\_size:half\_size 26. % 获取邻域像素 27. ni = i + m; 28. nj = j + n; 30. % 计算空间权重（基于距离） 31. spatial\_weight = exp(-((m^2 + n^2) / (2 \* sigma\_d^2))); 33. % 计算灰度权重（基于像素值差异） 34. color\_weight = exp(-((input\_img(i,j,c) - input\_img(ni,nj,c))^2 / (2 \* sigma\_r^2))); 36. % 计算总的权重 37. weight = spatial\_weight \* color\_weight; 39. % 累加加权像素值 40. filtered\_pixel = filtered\_pixel + input\_img(ni, nj, c) \* weight; 41. weight\_sum = weight\_sum + weight; 42. end 43. end 45. % 计算输出像素值 46. output\_img(i, j, c) = filtered\_pixel / weight\_sum; 47. end 48. end 49. end 51. % 转换回 uint8 类型 52. output\_img = uint8(output\_img); 53. end   **运行结果：**    **双边滤波不适合处理椒盐噪声。**   1. **频域平滑滤波：**   频域滤波表达式：  噪声对应于高频部分，所以去噪可以采用低通滤波。  频域平滑滤波的核心关键为——设计合适的低通滤波器H(u,v)   * 1. **理想低通滤波**   **算法原理：**  当信号的频率低于某个阈值（称为截止频率）时，信号不会被衰减，而当信号的频率高于截止频率时，信号会被完全阻止。  截断频率为D0的理想低通滤波器转移函数：  **为频率平面原点到点(u,v)的距离；**  **code：**   1. function output\_img = idealLowPassFilter(input\_img, cutoff\_frequency) 2. % input\_img: 输入图像，灰度图像或彩色图像 3. % cutoff\_frequency: 截止频率（高于该频率的信号将被滤除） 5. % 如果是彩色图像，分离 RGB 通道 6. **if** size(input\_img, 3) == 3 7. R = input\_img(:, :, 1);  % 红色通道 8. G = input\_img(:, :, 2);  % 绿色通道 9. B = input\_img(:, :, 3);  % 蓝色通道 10. % 对每个通道分别进行处理 11. R\_filtered = idealLowPassFilterSingleChannel(R, cutoff\_frequency); 12. G\_filtered = idealLowPassFilterSingleChannel(G, cutoff\_frequency); 13. B\_filtered = idealLowPassFilterSingleChannel(B, cutoff\_frequency); 15. % 合并处理后的通道 16. output\_img = cat(3, R\_filtered, G\_filtered, B\_filtered); 17. **else** 18. % 如果是灰度图像，直接进行处理 19. output\_img = idealLowPassFilterSingleChannel(input\_img, cutoff\_frequency); 20. end 21. end 22. function output\_img = idealLowPassFilterSingleChannel(input\_img, cutoff\_frequency) 23. % 对单通道图像进行理想低通滤波 25. [rows, cols] = size(input\_img); 27. % 进行离散傅里叶变换 28. img\_dft = fft2(**double**(input\_img)); 30. % 将频谱移到中心 31. img\_dft\_shifted = fftshift(img\_dft); 33. % 创建理想低通滤波器 34. [X, Y] = meshgrid(1:cols, 1:rows); 35. centerX = floor(cols / 2) + 1; 36. centerY = floor(rows / 2) + 1; 38. % 计算每个频率点到中心的距离 39. distance = sqrt((X - centerX).^2 + (Y - centerY).^2); 41. % 创建滤波器，距离小于截止频率的部分为1，其它部分为0 42. H = **double**(distance <= cutoff\_frequency); 44. % 将滤波器应用于频谱 45. img\_dft\_filtered = img\_dft\_shifted .\* H; 47. % 进行逆傅里叶变换，得到滤波后的图像 48. img\_dft\_inv\_shifted = ifftshift(img\_dft\_filtered); 49. output\_img = real(ifft2(img\_dft\_inv\_shifted)); 51. % 将输出图像转换回uint8类型 52. output\_img = uint8(output\_img); 53. end   **运行结果：**     * 1. **指数低通滤波**   **算法原理：**  基于对输入信号应用一个指数衰减函数，使得低频成分得以保留，高频成分被衰减。与理想低通滤波器（其频率响应为理想的矩形函数，完全通过低频信号并完全衰减高频信号）不同，指数低通滤波器的衰减是连续且渐进的，并不会突然完全阻止高频成分。因此，它在实际应用中提供了一种较为平滑的频率响应，避免了理想低通滤波器在高频区域的过度衰减。  **截断频率为D0的指数低通滤波器转移函数：**  **code：**   1. function output\_img = exponentialLowPassFilter(input\_img, D0) 2. % input\_img: 输入的彩色图像 3. % D0: 截止频率，控制低通滤波器的频率范围 5. % 判断输入图像是否为灰度图或彩色图像 6. **if** size(input\_img, 3) == 1 7. % 如果是灰度图像，直接处理 8. output\_img = applyExponentialLowPassFilter(input\_img, D0); 9. **else** 11. % 分离RGB通道 12. R = input\_img(:,:,1);  % 红色通道 13. G = input\_img(:,:,2);  % 绿色通道 14. B = input\_img(:,:,3);  % 蓝色通道 16. % 对每个通道应用指数低通滤波 17. R\_filtered = applyExponentialLowPassFilter(R, D0); 18. G\_filtered = applyExponentialLowPassFilter(G, D0); 19. B\_filtered = applyExponentialLowPassFilter(B, D0); 20. % 合并处理后的通道 21. output\_img = cat(3, R\_filtered, G\_filtered, B\_filtered); 22. end 23. end 25. function output\_channel = applyExponentialLowPassFilter(channel, D0) 26. % 对单个通道应用指数低通滤波 28. % 获取图像尺寸 29. [rows, cols] = size(channel); 31. % 计算频域的中心点 32. u0 = floor(rows / 2); 33. v0 = floor(cols / 2); 35. % 进行傅里叶变换并将图像移到频域中心 36. F = fftshift(fft2(**double**(channel))); 38. % 构建指数低通滤波器 39. H = zeros(rows, cols); 40. **for** u = 1:rows 41. **for** v = 1:cols 42. D = sqrt((u - u0)^2 + (v - v0)^2);  % 距离频域中心的距离 43. H(u, v) = exp(-D / D0);  % 指数低通滤波器的频率响应 44. end 45. end 47. % 应用指数低通滤波器 48. F\_filtered = F .\* H; 50. % 进行逆傅里叶变换，返回处理后的图像 51. img\_filtered = ifft2(ifftshift(F\_filtered)); 53. % 取实部并转换为uint8类型 54. output\_channel = uint8(real(img\_filtered)); 55. end   **运行结果：**     1. **基于模糊技术的平滑滤波：**   **算法原理：**  在图像处理中，可以将一幅图像看成一个模糊集。  当图像被噪声高度污染时，其模糊不确定性增加，可应用模糊滤波来处理图像。  基于模糊数学思想，利用模糊隶属度函数的概念，通过对均值滤波器的权值加以优化，提高平滑高斯噪声的能力。  模糊加权均值滤波的具体算法步骤：  步骤1：计算以点(x, y) 为中心的邻域 S 内灰度变化  步骤2：计算每一邻域点对中心点的模糊隶属度  步骤3：计算当前窗口模糊加权均值滤波输出  步骤4：用 g(x, y) 来代替滤波窗口内中心点的灰度值。  **code：**   1. function result = fuzzy\_average\_filter(NoiseI, filterSize) 2. % 输入： 3. % NoiseI - 输入的噪声图像（可以是彩色图像） 4. % filterSize - 邻域模板大小，模板为 (filterSize x filterSize) 6. % 计算邻域模板的半径 7. n = (filterSize - 1) / 2; 9. % 将图像转换为 **double** 类型，方便计算 10. NoiseI = im2double(NoiseI); 12. % 获取图像的尺寸 13. [height, width, channels] = size(NoiseI); 15. % 扩展图像边界以便处理边缘像素 16. hh = height + 2 \* n; 17. ww = width + 2 \* n; 18. ff = zeros(hh, ww, channels); 20. % 将原图像填充到扩展后的图像中心 21. ff(n+1:hh-n, n+1:ww-n, :) = NoiseI; 23. % 初始化结果图像 24. result = zeros(height, width, channels); 26. % 对每个通道进行处理 27. **for** c = 1:channels 28. % 对每个像素进行模糊均值滤波 29. **for** i = n+1:hh-n 30. **for** j = n+1:ww-n 31. % 提取当前邻域 32. region = ff(i-n:i+n, j-n:j+n, c); 34. % 计算当前像素与邻域像素的灰度差平方 35. dd = (region - ff(i,j,c)).^2; 37. % 计算方差估计 38. bb = sum(dd(:)) / ((n\*2+1)^2 - 1); 40. % 计算权重矩阵 41. dd = exp(-dd / bb) / bb; 42. sumd = sum(dd(:)) - 1 / bb; 44. % 加权求和 45. weighted\_sum = sum(sum(dd .\* region)) - (1 / bb) \* ff(i,j,c); 47. % 存储结果 48. result(i-n, j-n, c) = weighted\_sum / sumd; 49. end 50. end 51. end 53. % 转换结果为 uint8，并返回 54. result = uint8(result \* 255); 55. end   **运行结果：**    **5.边缘提取**   1. **robert算子：**   **算法原理：**  检测图像中的边缘，特别是那些突变或不连续的区域。通过计算图像像素的局部梯度（水平和垂直方向），寻找强度变化较大的地方，从而识别边缘。  梯度： 图像中的梯度是描述图像强度变化的量。梯度的大小表示强度变化的程度，而梯度的方向则表示变化的方向。  Robert算子： 它使用两个简单的卷积核（模板）来计算图像的梯度：  水平梯度（Gx）：检测图像水平方向上的变化。  垂直梯度（Gy）：检测图像垂直方向上的变化。  **交叉求微分**  **用模板表示为**  对每个像素及其周围的2x2区域执行卷积，计算水平和垂直方向的梯度值。最终，通过计算水平和垂直梯度的平方和的平方根得到梯度幅值（即边缘强度）。  **code：**   1. function edgeImg = robertEdgeDetection(img) 2. % 检查输入图像是否为彩色图像，若是则转换为灰度图像 3. **if** size(img, 3) == 3 4. img = rgb\_to\_gray(img);  % 如果是彩色图像，则转换为灰度图 5. end 7. % 将图像转换为 **double** 类型，便于计算 8. img = **double**(img); 10. % 定义Robert算子的两个卷积核（模板） 11. Gx = [1 0; 0 -1];  % 水平梯度 12. Gy = [0 1; -1 0];  % 垂直梯度 14. % 获取图像的大小 15. [rows, cols] = size(img); 17. % 初始化梯度幅值图像 18. edgeImg = zeros(rows, cols); 20. % 对每个像素进行Robert算子卷积操作（排除边缘部分） 21. **for** i = 1:rows-1 22. **for** j = 1:cols-1 23. % 提取当前像素周围的2x2区域 24. region = img(i:i+1, j:j+1); 26. % 计算水平和垂直方向的梯度 27. Gx\_val = sum(sum(region .\* Gx));  % 水平梯度 28. Gy\_val = sum(sum(region .\* Gy));  % 垂直梯度 30. % 计算梯度幅值（边缘强度） 31. edgeImg(i, j) = sqrt(Gx\_val^2 + Gy\_val^2); 32. end 33. end 35. % 将结果限制在 [0, 255] 范围内，并转换为 uint8 类型 36. edgeImg = uint8(mat2gray(edgeImg) \* 255); 37. end   **运行结果：**     1. **prewitt算子：**   **算法原理：**  Prewitt算子通过计算图像在水平和垂直方向的梯度来识别图像的边缘。它的基本思想是，通过对图像进行卷积操作，计算每个像素周围的局部梯度变化，从而找出图像中强度变化显著的地方。  水平梯度（Gx）：用于检测水平方向上的边缘。  垂直梯度（Gy）：用于检测垂直方向上的边缘。  **code：**   1. function edgeImg = prewittEdgeDetection(img) 2. % 检查输入图像是否为彩色图像，若是则转换为灰度图像 3. **if** size(img, 3) == 3 4. img = rgb\_to\_gray(img);  % 如果是彩色图像，则转换为灰度图 5. end 7. % 将图像转换为 **double** 类型，便于计算 8. img = **double**(img); 10. % 定义Prewitt算子的两个卷积核（模板） 11. Gx = [-1 0 1; -1 0 1; -1 0 1];  % 水平梯度 12. Gy = [-1 -1 -1; 0 0 0; 1 1 1];  % 垂直梯度 14. % 获取图像的大小 15. [rows, cols] = size(img); 17. % 初始化梯度幅值图像 18. edgeImg = zeros(rows, cols); 20. % 对每个像素进行Prewitt算子卷积操作（排除边缘部分） 21. **for** i = 2:rows-1 22. **for** j = 2:cols-1 23. % 提取当前像素周围的3x3区域 24. region = img(i-1:i+1, j-1:j+1); 25. % 计算水平和垂直方向的梯度 26. Gx\_val = sum(sum(region .\* Gx));  % 水平梯度 27. Gy\_val = sum(sum(region .\* Gy));  % 垂直梯度 29. % 计算梯度幅值（边缘强度） 30. edgeImg(i, j) = sqrt(Gx\_val^2 + Gy\_val^2); 31. end 32. end 34. % 将结果限制在 [0, 255] 范围内，并转换为 uint8 类型 35. edgeImg = uint8(mat2gray(edgeImg) \* 255); 36. end   **运行结果：**     1. **sobel算子：**   **算法原理：**  Sobel算子与其他边缘检测算法（如Prewitt算子）类似，基于梯度的变化来识别图像中强度变化大的区域，从而提取出边缘信息。  Sobel算子的核心是两个卷积核（模板），分别用于计算图像的水平和垂直梯度：  **水平梯度（Gx）：用于检测水平方向上的边缘。**  **垂直梯度（Gy）：用于检测垂直方向上的边缘。**  引入平均因素，对图像中随机噪声有一定的平滑作用  相隔两行或两列求差分，故边缘两侧的元素得到了增强，边缘显得粗而亮  **code：**   1. function edgeImg = sobelEdgeDetection(img) 2. % 检查输入图像是否为彩色图像，若是则转换为灰度图像 3. **if** size(img, 3) == 3 4. img = rgb\_to\_gray(img);  % 如果是彩色图像，则转换为灰度图 5. end 7. % 将图像转换为 **double** 类型，便于计算 8. img = **double**(img); 10. % 定义Sobel算子的两个卷积核（模板） 11. Gx = [-1 0 1; -2 0 2; -1 0 1];  % 水平梯度 12. Gy = [-1 -2 -1; 0 0 0; 1 2 1];  % 垂直梯度 14. % 获取图像的大小 15. [rows, cols] = size(img); 17. % 初始化梯度幅值图像 18. edgeImg = zeros(rows, cols); 20. % 对每个像素进行Sobel算子卷积操作（排除边缘部分） 21. **for** i = 2:rows-1 22. **for** j = 2:cols-1 23. % 提取当前像素周围的3x3区域 24. region = img(i-1:i+1, j-1:j+1); 26. % 计算水平和垂直方向的梯度 27. Gx\_val = sum(sum(region .\* Gx));  % 水平梯度 28. Gy\_val = sum(sum(region .\* Gy));  % 垂直梯度 30. % 计算梯度幅值（边缘强度） 31. edgeImg(i, j) = sqrt(Gx\_val^2 + Gy\_val^2); 32. end 33. end 35. % 将结果限制在 [0, 255] 范围内，并转换为 uint8 类型 36. edgeImg = uint8(mat2gray(edgeImg) \* 255); 37. end   **运行结果：**     1. **拉普拉斯算子：**   **算法原理：**  拉普拉斯算子是一种二阶导数算子，用于检测图像的边缘。它基于图像亮度变化的二阶导数进行边缘检测，能够高效地识别图像中的边缘。  拉普拉斯算子的核心思想是计算图像的二阶导数，也就是说，计算图像亮度变化的变化率。该算子通常应用于图像中高频部分（如边缘和细节），在这些区域，图像亮度发生较大的变化。  **拉普拉斯算子:**  **在二维空间中，拉普拉斯算子的一种常见形式是使用以下卷积核（模板）：**  **或**  **锐化模板：**  计算图像中某一像素周围的像素值与该像素值的差异，从而揭示图像的边缘。算子的响应在图像边缘处通常会有较大的变化，而在平坦区域变化较小。  **code：**   1. function edgeImg = laplaceEdgeDetection(img) 2. % 检查输入图像是否为彩色图像，若是则转换为灰度图像 3. **if** size(img, 3) == 3 4. img = rgb\_to\_gray(img);  % 如果是彩色图像，则转换为灰度图 5. end 7. % 将图像转换为 **double** 类型，便于计算 8. img = **double**(img); 10. % 定义拉普拉斯算子 11. laplaceKernel = [0 1 0; 1 -4 1; 0 1 0];  % 拉普拉斯核 13. % 获取图像的大小 14. [rows, cols] = size(img); 16. % 初始化梯度幅值图像 17. edgeImg = zeros(rows, cols); 19. % 对每个像素进行拉普拉斯算子卷积操作（排除边缘部分） 20. **for** i = 2:rows-1 21. **for** j = 2:cols-1 22. % 提取当前像素周围的3x3区域 23. region = img(i-1:i+1, j-1:j+1); 25. % 计算拉普拉斯算子的响应 26. edgeImg(i, j) = sum(sum(region .\* laplaceKernel)); 27. end 28. end 30. % 对拉普拉斯算子的输出应用绝对值 31. edgeImg = abs(edgeImg); 33. % 将结果限制在 [0, 255] 范围内 34. edgeImg = mat2gray(edgeImg) \* 255; 36. % 转换为 uint8 类型 37. edgeImg = uint8(edgeImg); 38. end   **运行结果：**    **6.目标提取**   1. **聚类分割：**   **算法原理：**  聚类是模式识别中对特征空间中数据进行分类的方法，取“物以类聚”的思想，把某些向量聚集为一组，每组具有相似的值。把图像分割看做对像素进行分类的问题，把像素表示成特征空间的数据，采用聚类算法划分数据为不同类别，对应原图则是实现对像素的分组，分组后利用“连通成分标记”找到连通区域。  K均值聚类：  通过迭代把特征空间分成K个聚集区域，首先确定K个初始聚类中心，然后根据各类样本到聚类中心的距离平方和最小的准则，不断调整聚类中心，直到聚类合理。  检查输入图像 I 的颜色空间。如果是彩色图像，将其转换为 Lab 颜色空间。Lab 颜色空间将颜色分为亮度信息（L通道）和色度信息（a和b通道）。如果是灰度图像，直接使用灰度信息进行处理。  使用K-means聚类算法将图像的每个像素分成2个类别（K=2），一个代表背景，另一个代表目标（鸟类）。K-means通过迭代优化簇的质心，使得同一簇中的像素尽可能相似，而不同簇中的像素差异尽可能大。  聚类完成后，生成一个 clusterImage，其中每个像素的值表示其所属的簇。然后将聚类结果以伪彩色图像的形式显示，方便选择鸟类的前景类别。通过点击图像中的一个点来选择代表鸟类的簇。  根据前景簇，创建一个二值掩码 birdMask，前景（鸟类）为1，背景为0。利用二值掩码提取出目标。  **code：**   1. function [birdMask, extracted\_img] = targetExtraction\_KMeans(I) 2. % targetExtraction 对鸟类图片进行目标提取 3. % 4. % 输入： 5. %   I - 输入图像，可以是彩色图像或灰度图像 6. % 7. % 输出： 8. %   birdMask - 二值掩码，前景（鸟类）为1，背景为0 9. %   extracted\_img - 应用掩码后的彩色图像，背景为黑色，前景为原色  12. % 检查输入图像是否为彩色图像，如果是，转换为 Lab 颜色空间 13. **if** size(I, 3) == 3 14. I\_lab = rgb2lab(I); 15. **else** 16. I\_lab = I; 17. end 19. % 将图像重塑为二维矩阵，每行是一个像素的特征 20. **if** size(I\_lab, 3) == 3 21. pixels = reshape(I\_lab, [], 3); 22. **else** 23. pixels = **double**(I\_lab(:)); 24. end 26. % 设置聚类数目为2（前景和背景） 27. K = 2; 29. % 应用 K-means 聚类 30. fprintf('正在进行 K-means 聚类，请稍候...\n'); 31. [cluster\_idx, ~] = kmeans(pixels, K, 'Replicates', 10, 'MaxIter', 400); 33. % 将聚类结果重新转换为图像格式 34. clusterImage = reshape(cluster\_idx, size(I,1), size(I,2)); 36. % 显示聚类结果供用户选择前景类别 37. figure; 38. imshow(label2rgb(clusterImage)); 39. title('K-means 聚类结果'); 41. % 让用户点击图像中的一个点以选择前景类别 42. disp('请点击鸟类所在的区域以选择前景类别...'); 43. [x, y] = ginput(1); 44. x = round(x); 45. y = round(y); 47. % 获取点击点的类别 48. selectedCluster = clusterImage(y, x); 50. % 创建二值掩码 51. birdMask = clusterImage == selectedCluster; 53. % 填充前景区域中的孔洞 54. birdMask = imfill(birdMask, 'holes'); 56. % 移除小的噪声区域 57. birdMask = bwareaopen(birdMask, 500); 59. % 关闭聚类结果图像 60. close(gcf); 62. % 应用掩膜到原始彩色图像 63. **if** size(I, 3) == 3 64. extracted\_img = I; 65. extracted\_img(repmat(~birdMask, [1, 1, 3])) = 0; 66. **else** 67. % 对于灰度图像，直接将非目标区域设置为0 68. extracted\_img = I; 69. extracted\_img(~birdMask) = 0; 70. end 72. end   **运行结果：**       1. **分水岭分割＋区域合并：**   **算法原理：**  分水岭分割：基于地形学概念的分割方法，实现可采用数学形态学的方法  梯度图像中各区域内部对应极小区域，边界对应高灰度，即分水岭。确定分水岭，即确定区域边界，实现分割。  首先使用Prewitt滤波器计算图像的梯度，通过计算水平方向和垂直方向的梯度，再合成总梯度 g，表示图像中的边缘强度。对梯度图像进行中值滤波，去除噪声并平滑梯度图。使用 watershed 函数对梯度图像 g 进行分水岭变换，得到标签矩阵 L，其中不同的标签表示不同的区域。  对每个区域，计算其平均灰度值，并将灰度值差异小于阈值（thresh）的区域合并。这有助于减少一些由于噪声或细节造成的过度分割。  提取目标区域。这里定义目标区域为标签大于1的区域，排除分水岭边界（标签为0）。  对于彩色图像，将目标区域的二值掩码扩展为三通道，并将背景区域设置为黑色，前景区域保持不变。  对于灰度图像，直接使用二值掩码进行点乘，背景区域设为黑色。  **code：**   1. function output\_image = targetExtract\_WatershedRegion(input\_image) 2. % 提取目标区域，输入是彩色图像，输出是目标区域 3. % input\_image: 输入的彩色图像 4. % output\_image: 输出的只包含目标区域，其他部分为黑色的图像 6. % 图像预处理 7. I = input\_image; 8. **if** size(input\_image, 3) == 3 9. input\_image = rgb\_to\_gray(input\_image); 10. end  13. image = im2double(input\_image); 14. hv = fspecial('prewitt'); 15. hh = hv.'; 17. % 计算梯度 18. gv = abs(imfilter(image, hv, 'replicate')); 19. gh = abs(imfilter(image, hh, 'replicate')); 20. g = sqrt(gv.^2 + gh.^2);  % 总梯度 22. g = medfilt2(g, [5, 5]); 24. % 分水岭变换 25. L = watershed(g);  % 进行分水岭变换 27. % 提取目标区域 28. num = max(L(:));  % 获取最大标签数，即区域数量 30. % 计算每个区域的平均灰度值 31. avegray = zeros(num, 1); 32. **for** i = 1:num 33. avegray(i) = mean(image(L == i));  % 计算每个区域的平均灰度值 34. end 36. % 合并相似灰度值的区域 37. thresh = 0.3; 38. [N, M] = size(L); 39. **for** i = 2:M-1 40. **for** j = 2:N-1 41. **if** L(j, i) == 0  % 如果当前像素是分水岭边界 42. neighbor = [L(j-1, i+1), L(j, i+1), L(j+1, i+1), L(j-1, i), L(j+1, i), ... 43. L(j-1, i-1), L(j, i-1), L(j+1, i-1)]; 44. neicode = unique(neighbor);  % 获取相邻区域的标签 45. neicode = neicode(neicode ~= 0);  % 去除分水岭边界（标签为0的部分） 46. neinum = length(neicode);  % 获取相邻区域的数量 47. **for** n = 1:neinum - 1 48. **for** m = n + 1:neinum 49. **if** abs(avegray(neicode(m)) - avegray(neicode(n))) < thresh  % 如果灰度值差异小于阈值 50. L(L == neicode(m)) = neicode(n);  % 合并区域 51. end 52. end 53. end 54. end 55. end 56. end 58. % 再次处理分水岭边界 59. **for** i = 2:M-1 60. **for** j = 2:N-1 61. **if** L(j, i) == 0  % 如果当前像素是分水岭边界 62. neighbor = [L(j-1, i+1), L(j, i+1), L(j+1, i+1), L(j-1, i), L(j+1, i), ... 63. L(j-1, i-1), L(j, i-1), L(j+1, i-1)]; 64. neicode = unique(neighbor);  % 获取相邻区域的标签 65. neicode = neicode(neicode ~= 0);  % 去除分水岭边界（标签为0的部分） 66. neinum = length(neicode);  % 获取相邻区域的数量 67. **if** neinum == 1  % 如果只有一个相邻区域 68. L(j, i) = neicode(neinum);  % 将当前像素标记为唯一的相邻区域标签 69. end 70. end 71. end 72. end 74. % 提取目标区域 75. target\_area = (L > 1);  % 目标区域的标签大于1，排除分水岭边界  78. **if** size(I, 3) == 3 79. % 使用点乘操作，将目标区域提取出来，背景部分设为黑色 80. output\_image = **double**(I) .\* **double**(repmat(target\_area, [1, 1, 3])); 81. **else** 82. % 对于灰度图像，直接使用二值掩码进行点乘 83. output\_image = **double**(I) .\* **double**(target\_area); 84. end 86. % 将输出图像转换为 uint8 类型 87. output\_image = uint8(output\_image);  % 转换回 uint8 类型  90. end   **运行结果**     1. **基于双峰直方图的阈值化方法：**   **算法原理：**  若图像的灰度直方图为双峰分布，表明图像的内容大致为两部分，分别为灰度分布的两个山峰的附近。选择阈值为两峰间的谷底对应灰度值。  通过检测图像直方图中的两个明显的峰值（即前景和背景的灰度值分布），并基于这两个峰值之间的低谷来确定合适的阈值。双峰直方图的关键是，前景和背景的灰度值通常会在直方图上形成两个清晰的峰，而这两个峰之间的低谷即为最佳的阈值。  检查每个灰度值邻域的值，判断是否存在局部最大值。若直方图上有两个明显的峰值，则认为检测到双峰，并返回两个峰值的索引。一旦双峰被检测到，算法通过找到这两个峰之间的低谷来计算阈值。低谷位置即为最适合分割前景和背景的灰度值。最终的阈值是该低谷的灰度值，它用于将图像二值化。  **code：**   1. function [result] =targetExtract\_BimodalThresholding(I) 2. % BimodalThresholding 使用基于双峰直方图的阈值化方法 3. % 输入： 4. %   I - 输入图像，可以是彩色图像或灰度图像 5. % 6. % 输出： 7. %   result - 处理后的二值化图像（前景为1，背景为0） 9. % 检查输入图像是否为彩色图像，如果是，转换为灰度图像 10. img = I; 11. **if** size(I, 3) == 3 12. I = rgb\_to\_gray(I); 13. end 15. % 基于双峰分布的直方图选择阈值 16. [thresh, hist1] = BimodalThreshold(I); 18. % 执行阈值化操作 19. binary\_mask = I > thresh; 21. % 如果输入图像是彩色图像，则扩展二值掩码到三通道 22. **if** size(img, 3) == 3 23. % 确保图像为 **double** 类型进行计算，然后点乘 24. extracted\_img = **double**(img) .\* **double**(repmat(binary\_mask, [1, 1, 3])); 25. **else** 26. % 对于灰度图像，直接使用二值掩码进行处理 27. extracted\_img = **double**(img) .\* **double**(binary\_mask); 28. end 30. % 将处理结果转换回 uint8 类型 31. extracted\_img = uint8(extracted\_img); 33. % 输出提取后的图像 34. result = extracted\_img;  37. end 39. function [thresh, hist1] = BimodalThreshold(I) 40. % BimodalThreshold 使用基于双峰直方图的阈值化方法 41. % 输入： 42. %   I - 输入灰度图像 43. % 44. % 输出： 45. %   thresh - 阈值 46. %   hist1  - 归一化的直方图 48. % 计算图像的直方图 49. hist1 = computeHistogram(I); 50. hist2 = hist1; 51. iter = 0; 53. % 迭代平滑过程，直到满足条件 54. **while** **true** 55. [is, peak] = Bimodal(hist1); 56. **if** is == 0 57. hist2(1) = (hist1(1)\*2 + hist1(2)) / 3; 58. **for** j = 2:255 59. hist2(j) = (hist1(j-1) + hist1(j) + hist1(j+1)) / 3; 60. end 61. hist2(256) = (hist1(255) + hist1(256)\*2) / 3; 62. hist1 = hist2; 63. iter = iter + 1; 64. **if** iter > 1000 65. **break**; 66. end 67. **else** 68. **break**; 69. end 70. end 72. % 找到双峰的低谷并计算阈值 73. [trough, pos] = min(hist1(peak(1):peak(2))); 74. thresh = pos + peak(1); 75. end  78. function [is, peak] = Bimodal(histgram) 79. % Bimodal 检测直方图中的双峰 80. % 81. % 输入： 82. %   histgram - 输入的直方图 83. % 84. % 输出： 85. %   is - 返回1表示检测到双峰，0表示未检测到 86. %   peak - 双峰的峰值索引 88. count = 0; 89. **for** j = 2:255 90. **if** histgram(j-1) < histgram(j) && histgram(j+1) < histgram(j) 91. count = count + 1; 92. peak(count) = j; 93. **if** count > 2 94. is = 0; 95. **return**; 96. end 97. end 98. end 99. **if** count == 2 100. is = 1; 101. **else** 102. is = 0; 103. end 104. end   **运行结果**    **7.特征提取**   1. **LBP特征：**   **算法原理：**  3×3的窗口内，灰度值大于中心像素的位置记为1，否则为0，产生8位无符号二进制数，转换为十进制数，即为该窗口中心像素点的LBP值。  通常将图像分为n×n的子区域，对子区域内的像素点计算LBP值，并统计其直方图，以直方图作为其判别特征。  **code：**   1. function lbp = computeLBP(image) 2. % computeLBP 计算图像的LBP特征图 3. % 输入： 4. %   image - 输入的图像，可以是彩色图像或灰度图像 5. % 输出： 6. %   lbp - 计算得到的LBP特征图 8. % 如果是彩色图像，先转换为灰度图像 9. **if** size(image, 3) == 3 10. image = rgb\_to\_gray(image); 11. end 13. % 获取图像的尺寸 14. [N, M] = size(image); 15. lbp = zeros(N, M);  % 初始化LBP特征图 17. % 对每个像素计算LBP值（不处理边缘） 18. **for** j = 2:N-1 19. **for** i = 2:M-1 20. % 获取当前像素周围8个邻域的坐标 21. neighbor = [j-1 i-1; j-1 i; j-1 i+1; j i+1; j+1 i+1; j+1 i; j+1 i-1; j i-1]; 22. count = 0; 23. % 遍历邻域像素并计算LBP值 24. **for** k = 1:8 25. **if** image(neighbor(k,1), neighbor(k,2)) > image(j, i) 26. count = count + 2^(8 - k); 27. end 28. end 29. lbp(j, i) = count;  % 将计算得到的LBP值赋给特征图 30. end 31. end 33. lbp = uint8(lbp);  % 将结果转换为无符号8位整数 35. end   **运行结果**    **左图：**  **描述了整幅图像的局部纹理特征，包括前景和背景。前景区域的纹理信息更清晰，背景区域的特征更随机。**  **右图：**  **聚焦于目标区域，展示了鸟的纹理特征和局部结构，去除了背景干扰，更好地突出目标形状特征。**   1. **HOG特征：**   **算法原理：**  HOG的基本思想是通过图像的局部梯度信息来描述图像的局部形状和边缘结构。是特征直方图的一种，通过统计梯度方向直方图，用于表征图像局部梯度方向和梯度强度分布特性。在边缘具体位置未知的情况下，边缘方向的分布也可以很好的表示目标的外形轮廓。  **特征提取步骤：**  **图像灰度化**  **图像归一化：**  **计算图像每个像素的梯度大小和方向**  **为沿x、y方向的梯度**  **划分图像为若干方格单元，计算每一个方格单元的梯度方向直方图**  **将相邻单元组成块，计算块中的HOG特征向量**  **块HOG特征向量归一化**  **生成图像的HOG特征向量**  **code：**   1. function [feature, image\_hog\_with\_arrows] = computeHOG(Image, step, K) 2. % computeHOG - 计算HOG特征并返回带有箭头的图像 3. % 4. % 语法: [feature, image\_hog\_with\_arrows] = computeHOG(Image, step, K) 5. % 6. % 输入参数: 7. %    Image - 输入的彩色或灰度图像 8. %    step - HOG计算时的cell大小（步长） 9. %    K - 梯度方向的数量（即直方图的分桶数） 10. % 11. % 输出参数: 12. %    feature - HOG特征向量的单元数组 13. %    image\_hog\_with\_arrows - 带有HOG箭头的图像 15. % 如果是彩色图像，则转换为灰度图 16. **if** size(Image, 3) == 3 17. Image\_gray = rgb\_to\_gray(Image); 18. **else** 19. Image\_gray = Image; 20. end 22. % 将图像转换为**double**类型 23. Image\_gray = **double**(Image\_gray); 25. [N, M] = size(Image\_gray); 26. Image\_processed = sqrt(Image\_gray);  % 对图像进行开根号操作 28. % 计算梯度 29. Hy = [-1 0 1]; 30. Hx = Hy'; 31. Gy = imfilter(Image\_processed, Hy, 'replicate'); 32. Gx = imfilter(Image\_processed, Hx, 'replicate'); 33. Grad = sqrt(Gx.^2 + Gy.^2);  % 计算梯度幅值 34. Phase = atan2d(Gy, Gx);  % 计算梯度方向（度） 35. Phase(Phase < 0) = Phase(Phase < 0) + 180;  % 将角度转换到[0, 180]范围内 37. % HOG计算参数 38. angle = 180 / K;  % 每个方向的角度大小 39. numCellsX = floor(M / step); 40. numCellsY = floor(N / step); 41. Cell = cell(numCellsY, numCellsX);  % 预分配单元格数组 43. % 计算HOG特征 44. **for** i = 1:numCellsX 45. **for** j = 1:numCellsY 46. col\_start = (i-1)\*step + 1; 47. col\_end = i\*step; 48. row\_start = (j-1)\*step + 1; 49. row\_end = j\*step; 50. Gtmp = Grad(row\_start:row\_end, col\_start:col\_end); 51. sumGrad = sum(Gtmp(:)); 52. **if** sumGrad == 0 53. Gtmp\_norm = Gtmp; 54. **else** 55. Gtmp\_norm = Gtmp / sumGrad;  % 对每个cell的梯度进行归一化 56. end 57. Hist = zeros(1, K);  % 初始化梯度直方图 58. **for** x = 1:step 59. **for** y = 1:step 60. ang = Phase(row\_start + y -1, col\_start + x -1); 61. **if** ang <= 180 62. bin = floor(ang / angle) + 1; 63. **if** bin > K  % 处理角度为180度的情况 64. bin = K; 65. end 66. Hist(bin) = Hist(bin) + Gtmp\_norm(y, x); 67. end 68. end 69. end 70. Cell{j, i} = Hist;  % 存储每个cell的HOG特征 71. end 72. end 74. % 特征提取 75. feature = cell(1, (numCellsX-1)\*(numCellsY-1));  % 初始化HOG特征向量的单元数组 76. idx = 1; 77. **for** i = 1:numCellsX-1 78. **for** j = 1:numCellsY-1 79. f = [Cell{j, i} Cell{j, i+1} Cell{j+1, i} Cell{j+1, i+1}]; 80. f = f / sum(f);  % 归一化特征向量 81. feature{idx} = f;  % 存储特征向量 82. idx = idx + 1; 83. end 84. end 86. % 初始化箭头线段列表 87. lines = [];  % 每行是 [x1 y1 x2 y2] 89. % 计算箭头的起点和方向 90. **for** i = 1:numCellsX 91. **for** j = 1:numCellsY 92. Hist = Cell{j, i}; 94. % 将梯度直方图转换为HOG方向图 95. **for** k = 1:K 96. angle\_start = (k - 1) \* angle;  % 每个方向的起始角度 97. angle\_end = k \* angle;          % 每个方向的结束角度 98. intensity\_k = Hist(k);         % 每个方向的强度 99. **if** intensity\_k == 0 100. **continue**;  % 跳过强度为0的方向 101. end 102. length = intensity\_k \* 25;      % 可以调节条的长度 104. % 计算方向条的极坐标 105. angle\_center = (angle\_start + angle\_end) / 2; 106. [dx, dy] = pol2cart(deg2rad(angle\_center), length);  % 转换为笛卡尔坐标 108. % 计算箭头的起点 109. startX = (i - 0.5) \* step;  % cell中心的x坐标 110. startY = (j - 0.5) \* step;  % cell中心的y坐标 112. % 计算箭头的终点 113. endX = startX + dx; 114. endY = startY + dy; 116. % 将箭头的起点和终点加入lines列表 117. lines = [lines; startX, startY, endX, endY]; 118. end 119. end 120. end 122. % 检查是否有箭头需要绘制 123. **if** isempty(lines) 124. disp('No arrows generated. Returning original image.'); 125. **return**; 126. end 128. % 使用 insertShape 批量绘制所有箭头 129. image\_hog\_with\_arrows = insertShape(image\_hog\_with\_arrows, 'Line', lines, 'Color', 'blue', 'LineWidth', 1); 131. end   **运行结果：**    **源图像HOG特征提取：**  **图中覆盖的蓝色箭头表示每个Cell中的主梯度方向和强度，箭头的密度由Cell的大小（step 参数）决定。**  **背景特征：**  **图像背景中的箭头较为稀疏且随机，背景区域梯度较弱，方向无明显规律。**  **前景特征：**  **在鸟的轮廓、头部、喙部等区域，箭头方向密集且呈现较为一致的方向模式。这是因为这些区域包含了明显的边缘信息，HOG能够捕获这些梯度变化。**  **目标图像HOG特征提取：**  **前景区域的特征提取：**  **特征更加集中于目标区域，蓝色箭头主要分布在鸟的轮廓边缘、头部、喙部等区域，且梯度方向与鸟的形状紧密相关。**  **细节分析：**  **鸟的喙部分：箭头集中分布且方向一致，反映了梯度变化的强烈方向性。**  **鸟的身体边缘：箭头沿着鸟的轮廓方向排列，表示梯度的主方向与物体的边界线一致。**  **背景移除的效果：背景部分不再显示特征，说明特征提取算法能够有效地聚焦于目标区域，而不受无关信息的干扰。**  **8.图像分类（鸟类识别）**  **使用基于 PyTorch 和 torchvision 的深度学习方法，处理 CUB\_200\_2011 数据集（细粒度鸟类图像分类数据集），完成从数据加载到模型训练、验证和测试的任务。**  **EfficientNet-B0：**  使用预训练模型的特征提取部分，将其冻结（requires\_grad=False），只训练分类器部分。  新的分类器使用 全连接层（Linear）和激活函数（SiLU）构建。  输出层的神经元数量与类别数相同。   1. model.classifier = nn.Sequential( 2. nn.Linear(n\_inputs, 2048), 3. nn.SiLU(), 4. nn.Dropout(0.3), 5. nn.Linear(2048, 2048), 6. nn.SiLU(), 7. nn.Dropout(0.3), 8. nn.Linear(2048, len(classes)) 9. )   **数据增强：**  随机水平翻转：RandomHorizontalFlip，增强模型对左右翻转的不变性。  随机垂直翻转：RandomVerticalFlip，增强模型对上下翻转的不变性。  颜色抖动：ColorJitter，模拟颜色变化，增加数据多样性。  中心裁剪：CenterCrop 和 Resize，标准化输入图像大小为224×224。  归一化：使用 ImageNet 数据集的均值和标准差归一化，确保特征分布一致。  **训练过程：**  模型训练与评估模式切换（train 和 eval）。  梯度清零、前向传播、反向传播、参数更新。  动态学习率调整（StepLR）。  保存每个 epoch 的检查点，包括模型权重、优化器状态等。  **模型评估与测试：**  对测试集进行预测，计算总体的准确率以及每个类别的分类准确率。  使用混淆矩阵的形式输出每个类别的分类性能。  **预测接口：**  定义了 predict 函数，可以对图像进行推理，返回预测类别。  **模型训练：**    **部分测试结果：**    **分类预测**   1. import os 2. os.environ['KMP\_DUPLICATE\_LIB\_OK'] = 'True' 3. import torch 4. import torchvision 5. from torchvision import datasets, transforms 6. from torch import nn 7. from PIL import Image 9. # 获取类别名称 10. def get\_classes(data\_dir): 11. all\_data = datasets.ImageFolder(data\_dir) 12. **return** all\_data.classes  15. def MyModel(classes): 16. model = torchvision.models.efficientnet\_b0(pretrained=False) 17. n\_inputs = model.classifier[1].in\_features 18. model.classifier = nn.Sequential( 19. nn.Linear(n\_inputs, 2048),  # 增加第一个全连接层的大小 20. nn.SiLU(), 21. nn.Dropout(0.3), 22. nn.Linear(2048, 2048),  # 增加另一个全连接层 23. nn.SiLU(), 24. nn.Dropout(0.3), 25. nn.Linear(2048, len(classes))  # 调整输出大小以匹配类的数量 26. ) 27. **return** model  30. def apply\_test\_transforms(): 31. # 使用 Compose 统一处理 32. **return** transforms.Compose([ 33. transforms.Resize((224, 224)), 34. transforms.ToTensor(), 35. transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) 36. ])  39. def predict\_with\_checkpoint(checkpoint\_path, image\_path, device='cuda'): 40. # 获取类 41. dataset\_path = 'D:/\_laboratory/pythonProject/DIP/CUB\_200\_2011/images' 42. classes = get\_classes(dataset\_path) 44. # 加载模型和检查点 45. checkpoint = torch.load(checkpoint\_path, map\_location=device) 46. model = MyModel(classes) 48. model.load\_state\_dict(checkpoint['model\_state\_dict'])  # 加载模型权重 49. model.to(device) 50. model.eval()  # 设置为评估模式 52. # 图像预处理 53. transform = apply\_test\_transforms() 54. im = Image.open(image\_path) 55. image\_tensor = transform(im).to(device) 57. # 推理 58. with torch.no\_grad(): 59. minibatch = torch.stack([image\_tensor]) 60. outputs = model(minibatch) 61. \_, predicted\_class = torch.max(outputs, 1)  # 获取预测类别索引 63. # 返回类别名称 64. **return** classes[predicted\_class.item()]   **运行结果：**     1. **实验结论及收获建议**   通过本次大作业，我系统性地复习并实践了数字图像处理的基础知识：  掌握了灰度图直方图的生成与分析，并熟练实现了直方图均衡化和直方图匹配，从视觉效果上验证了这些算法对图像亮度分布的优化。  理解了图像灰度变换的作用，并完成了线性变换和非线性变换（对数变换和指数变换）的对比度增强操作，能够直观感受到这些算法在增强图像细节上的不同表现。  实现了图像的缩放、旋转等几何变换，巩固了对图像坐标系和插值算法的理解。  在 MATLAB的图形化界面中动态实现这些变换，帮助我深刻理解了几何变换的实际应用场景。  通过实现图像加噪（如高斯噪声、椒盐噪声等）以及空域和频域滤波去噪处理，我对图像噪声的特性有了更加深入的认识。  尤其是在空域滤波中，理解了均值滤波、中值滤波对不同噪声类型的适用性，以及频域滤波如何通过频谱信息有效消除噪声。  这部分实践让我更好地掌握了空域滤波和频域滤波的优劣，以及选择滤波器时的注意事项。  在作业中，我实现了Robert算子、Prewitt算子、Sobel算子和拉普拉斯算子进行图像边缘检测。  通过对比不同算子提取的边缘结果，我对这些算子的计算原理、方向敏感性及其适用场景有了更清晰的理解。  此外，实际编写这些算子的卷积运算代码，帮助我更加熟练地掌握了 MATLAB 的矩阵操作。  我学习并应用了多种方法进行目标提取，例如阈值分割、分水岭算法、K-means 聚类等，完成了从原始图像中提取目标的任务。  特别是在实践中，我发现不同方法的目标提取效果依赖于图像特性，这使我更加注重算法的选择与参数调整的重要性。  实现了基于LBP和HOG的图像特征提取，并用直观的方式展示了特征提取后的结果（如特征图和方向梯度）。我学习到：LBP 适合捕捉图像的纹理信息。HOG 更注重目标的形状与轮廓信息。通过对特征图的观察，理解了这些方法如何将图像转化为数值特征，方便后续分类任务使用。  在加分项中，我尝试了基于深度学习的图像分类任务：使用预训练模型（如EfficientNet）实现了智能模型的训练、测试。  本次作业涉及多个知识点的整合，帮助我在数字图像处理方面得到了显著提升。 |