**贵州大学计算机科学与技术学院**

**计算机科学与技术及软件工程专业实验报告**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **《数字图像处理》实验报告** | | | | | | | |
| **题目** | 空间滤波器 | **姓名** | 李赛博 | **学号** | 2300770006 | **班级** | 计科2301 |
| **实验环境：**  Python 3.13 | | | | | | | |
| **实验内容与完成情况：**  **一、实验目的**  了解并掌握应用空域平滑和锐化滤波相关知识。   1. **实验内容**   **编程实践一：**  **基础实验内容：**  **编程实现一维高斯函数并设置不同的均值μ和标准差σ，在同一窗口中显示出相应的图形进行比较。**  **Code:**  import numpy as np  import matplotlib.pyplot as plt  plt.rcParams["font.family"] = ["SimHei", "WenQuanYi Micro Hei", "Heiti TC"]  plt.rcParams["axes.unicode\_minus"] = False  # 正确显示负号  # 定义一维高斯函数  def gaussian(x, mu, sigma):      """      计算一维高斯函数值      公式：f(x) = (1/(σ√(2π))) \* exp(-0.5\*((x-μ)/σ)²)      """      coefficient = 1 / (sigma \* np.sqrt(2 \* np.pi))  # 归一化系数      exponent = -0.5 \* ((x - mu) / sigma) \*\*2  # 指数部分      return coefficient \* np.exp(exponent)  x = np.linspace(-10, 10, 2000)  parameters = [      (0, 0.5, "σ=0.5（陡峭）"),      (0, 1, "σ=1（中等）"),      (0, 2, "σ=2（平缓）"),      (3, 1, "μ=3（右移）"),      (-2, 1, "μ=-2（左移）")  ]  # 创建画布  plt.figure(figsize=(12, 7))  plt.title("不同均值(μ)和标准差(σ)的一维高斯函数对比", fontsize=15)  plt.xlabel("x值", fontsize=12)  plt.ylabel("函数值 f(x)", fontsize=12)  plt.grid(linestyle="--", alpha=0.6)  # 网格线增强可读性  colors = ["#FF6B6B", "#4ECDC4", "#45B7D1", "#FFA07A", "#98D8C8"]  line\_styles = ["-", "--", "-.", ":", "-"]  for i, (mu, sigma, label) in enumerate(parameters):      y = gaussian(x, mu, sigma)      plt.plot(x, y,               color=colors[i],               linestyle=line\_styles[i],               linewidth=2.5,               label=f"μ={mu}, {label}")  plt.legend(fontsize=11, loc="upper right")  for mu, sigma, \_ in parameters:      plt.axvline(x=mu, color="gray", linestyle=":", alpha=0.3)  plt.tight\_layout()  plt.show()  结果：    **编程实践二：**  **基础实验内容：**  **2.1分析比较同一模板尺寸，不同σ的高斯滤波的结果。**  **2.2分析比较同一σ，不同模板尺寸的高斯滤波的结果。**  **Code:**  import cv2  import numpy as np  import matplotlib.pyplot as plt  # 设置中文显示  plt.rcParams["font.family"] = ["SimHei", "WenQuanYi Micro Hei", "Heiti TC"]  plt.rcParams["axes.unicode\_minus"] = False  # 生成带高斯噪声的测试图像  def add\_gaussian\_noise(image, mean=0, var=0.001):      """添加高斯噪声"""      image = np.array(image / 255, dtype=float)      noise = np.random.normal(mean, var\*\*0.5, image.shape)      noisy\_image = image + noise      noisy\_image = np.clip(noisy\_image, 0, 1)      return np.uint8(noisy\_image \* 255)  original = cv2.imread("fig.jpg", cv2.IMREAD\_GRAYSCALE)  if original is None:        original = np.zeros((256, 256), dtype=np.uint8)      cv2.rectangle(original, (50, 50), (200, 200), 128, -1)      cv2.circle(original, (128, 128), 50, 200, -1)  gaussian\_noisy = add\_gaussian\_noise(original, var=0.005)  # 高斯噪声（方差0.005）  sigma\_list = [0.5, 1.0, 2.0, 3.0]  gauss\_same\_size = [      cv2.GaussianBlur(gaussian\_noisy, (5, 5), sigmaX=sigma)      for sigma in sigma\_list  ]  size\_list = [(3, 3), (5, 5), (7, 7), (9, 9)]  gauss\_same\_sigma = [      cv2.GaussianBlur(gaussian\_noisy, size, sigmaX=1.0)      for size in size\_list  ]  # 可视化函数  def plot\_images(images, titles, rows, cols, figsize=(15, 10)):      """批量绘制图像"""      plt.figure(figsize=figsize)      for i in range(len(images)):          plt.subplot(rows, cols, i + 1)          plt.imshow(images[i], cmap="gray")          plt.title(titles[i])          plt.axis("off")      plt.tight\_layout()      plt.show()  plot\_images(      [gaussian\_noisy] + gauss\_same\_size,      ["原始高斯噪声图"] + [f"σ={s}" for s in sigma\_list],      1, 5, (16, 4)  )  plot\_images(      [gaussian\_noisy] + gauss\_same\_sigma,      ["原始高斯噪声图"] + [f"尺寸={s}" for s in size\_list],      1, 5, (16, 4)  )  **结果：**      **结果分析：**  在模板尺寸固定为 5×5 的情况下，σ（高斯核的标准差）对滤波效果的影响如下：  σ=0.5：噪声抑制效果较弱，图像中仍残留较多高斯噪声，边缘和细节保留相对清晰。因为小 σ 对应的高斯核权重集中在中心区域，对邻域像素的加权平均作用有限，仅能平滑极细微的噪声。  σ=1.0：噪声抑制效果有所提升，图像整体更干净，同时边缘和细节的模糊程度适中。此时 σ 与模板尺寸的匹配度较好，在噪声抑制和细节保留间取得了较好平衡。  σ=2.0：噪声抑制能力进一步增强，但图像边缘和细节的模糊程度明显增加。较大的 σ 使高斯核的权重分布更分散，对邻域像素的平滑作用更强，导致高频信息（边缘、细节）被过度抑制。  σ=3.0：噪声几乎被完全抑制，但图像出现严重的模糊，边缘轮廓变得十分柔和，细节丢失严重。超大的 σ 让高斯滤波的平滑作用占据主导，图像的纹理和细节几乎被抹平。  结论：在模板尺寸固定时，σ 越大，噪声抑制能力越强，但图像模糊程度越高、细节丢失越严重。需根据噪声强度和对细节保留的需求，选择合适的 σ 值。  二、同一 σ（1.0），不同模板尺寸的高斯滤波结果分析  在 σ 固定为 1.0 的情况下，模板尺寸对滤波效果的影响如下：  尺寸 = 3×3：噪声抑制效果有限，图像中仍有明显噪声残留，不过边缘和细节的保留最为清晰。小尺寸模板的邻域范围小，加权平均的像素少，因此对噪声的平滑作用弱，对细节的破坏也小。  尺寸 = 5×5：噪声抑制效果提升，图像干净度增加，边缘和细节的模糊程度处于可接受范围。模板尺寸的增大让邻域像素参与加权平均的数量增加，噪声被更有效地平滑，同时对细节的影响仍在合理区间。  尺寸 = 7×7：噪声抑制能力进一步增强，图像整体更纯净，但边缘和细节的模糊程度明显加剧。更大的模板尺寸意味着更广泛的邻域像素参与平滑，高频的细节和边缘被持续抑制。  尺寸 = 9×9：噪声几乎被完全过滤，但图像模糊严重，边缘轮廓变得非常柔和，细节几乎不可见。超大的模板尺寸使邻域范围极大，加权平均的平滑作用被最大化，导致图像细节大量丢失。  结论：在 σ 固定时，模板尺寸越大，噪声抑制能力越强，但图像模糊程度越高、细节丢失越严重。需结合噪声水平和细节重要性，选择恰当的模板尺寸。  三、综合总结  高斯滤波的σ（标准差）和模板尺寸是两个核心参数，二者共同决定了噪声抑制能力和细节保留程度：  当需要强噪声抑制时，可选择较大的 σ 或较大的模板尺寸，但需承担图像模糊、细节丢失的代价；  当需要强细节保留时，应选择较小的 σ 或较小的模板尺寸，同时需接受噪声抑制不足的问题；  实际应用中，需根据图像的噪声强度、细节重要性等需求，对 σ 和模板尺寸进行权衡，以达到噪声抑制与细节保留的最佳平衡。  **2.3对同一幅（高斯和椒盐）噪声图像，分析比较均值滤波和高斯滤波的结果，均值滤波器模板大小（3\*3,5\*5,7\*7,9\*9）。**  **Code:**  import cv2  import numpy as np  import matplotlib.pyplot as plt  plt.rcParams["font.family"] = ["SimHei", "WenQuanYi Micro Hei", "Heiti TC"]  plt.rcParams["axes.unicode\_minus"] = False  plt.rcParams['axes.titlesize'] = 12  plt.rcParams['font.size'] = 12  def add\_gaussian\_noise(image, mean=0, var=0.001):      image = np.array(image / 255, dtype=float)      noise = np.random.normal(mean, var\*\*0.5, image.shape)      noisy\_image = image + noise      noisy\_image = np.clip(noisy\_image, 0, 1)      return np.uint8(noisy\_image \* 255)  def add\_salt\_pepper\_noise(image, prob=0.05):        output = np.zeros(image.shape, np.uint8)      thres = 1 - prob      for i in range(image.shape[0]):          for j in range(image.shape[1]):              rdn = np.random.random()              if rdn < prob:                  output[i][j] = 0              elif rdn > thres:                  output[i][j] = 255              else:                  output[i][j] = image[i][j]      return output  # 读取原始图像  original = cv2.imread("fig.jpg", cv2.IMREAD\_GRAYSCALE)  if original is None:      original = np.zeros((256, 256), dtype=np.uint8)      cv2.rectangle(original, (50, 50), (200, 200), 128, -1)      cv2.circle(original, (128, 128), 50, 200, -1)  # 生成带噪声的图像  gaussian\_noisy = add\_gaussian\_noise(original, var=0.005)  salt\_pepper\_noisy = add\_salt\_pepper\_noise(original, prob=0.08)  # 2. 滤波处理  mean\_kernels = [(3, 3), (5, 5), (7, 7), (9, 9)]  # 高斯噪声图像  mean\_gaussian = [cv2.blur(gaussian\_noisy, k) for k in mean\_kernels]  gauss\_gaussian = cv2.GaussianBlur(gaussian\_noisy, (5, 5), sigmaX=1.0)  # 椒盐噪声图像  mean\_saltpepper = [cv2.blur(salt\_pepper\_noisy, k) for k in mean\_kernels]  gauss\_saltpepper = cv2.GaussianBlur(salt\_pepper\_noisy, (5, 5), sigmaX=1.0)  def plot\_comparison(images, titles, figsize=(18, 12), nrows=2, ncols=3):      fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)         axes = axes.flatten()        for i in range(len(images)):          axes[i].imshow(images[i], cmap="gray")            axes[i].set\_title(titles[i], fontsize=16, fontweight='normal', pad=8)          axes[i].axis("off")        # 隐藏多余的子图      for i in range(len(images), len(axes)):          axes[i].axis("off")  plt.tight\_layout(pad=0.1, h\_pad=0.1, w\_pad=0.1)      plt.subplots\_adjust(top=0.95, bottom=0.05, left=0.05, right=0.95,                         hspace=0.15, wspace=0.001)      plt.show()  class PlotConfig:      def \_\_init\_\_(self):          self.figsize = (18, 12)          self.nrows = 2          self.ncols = 3          self.fontsize = 16          self.hspace = 0.15          self.wspace = 0.001  config = PlotConfig()  # 4.1 高斯噪声下：均值滤波 vs 高斯滤波  plot\_comparison(      images=[gaussian\_noisy, gauss\_gaussian] + mean\_gaussian,      titles=["原始高斯噪声图", "高斯滤波(5x5,σ=1)"] + [f"均值滤波{k}" for k in mean\_kernels],      figsize=config.figsize,      nrows=config.nrows,      ncols=config.ncols  )  # 4.2 椒盐噪声下：均值滤波 vs 高斯滤波  plot\_comparison(      images=[salt\_pepper\_noisy, gauss\_saltpepper] + mean\_saltpepper,      titles=["原始椒盐噪声图", "高斯滤波(5x5,σ=1)"] + [f"均值滤波{k}" for k in mean\_kernels],      figsize=config.figsize,      nrows=config.nrows,      ncols=config.ncols  )  **结果：**      **结果分析：**  一、高斯噪声图像的滤波结果分析  1. 高斯滤波（5×5，σ=1）的表现  高斯滤波对高斯噪声具有针对性的抑制效果，处理后的图像噪声明显减少，同时边缘和细节的模糊程度相对温和。它通过加权平均的方式平滑噪声，中心像素权重高、边缘像素权重低，因此在抑制噪声的同时，能一定程度上保留图像的细节层次，视觉上图像整体较为自然。  2. 均值滤波的表现（不同模板尺寸）  3×3 模板：噪声抑制效果有限，图像中仍残留较多高斯噪声，但边缘和细节保留得最为清晰。因为小尺寸模板的邻域范围小，对像素的平均作用弱，所以细节破坏少，但噪声过滤不足。  5×5 模板：噪声抑制能力提升，图像干净度增加，同时边缘和细节的模糊程度处于可接受范围。模板尺寸的增大让更多邻域像素参与平均，噪声被更有效地平滑。  7×7 模板：噪声抑制能力进一步增强，图像整体更纯净，但边缘和细节的模糊程度明显加剧。更大的模板尺寸意味着更广泛的邻域像素参与平滑，高频的细节和边缘被持续抑制。  9×9 模板：噪声几乎被完全过滤，但图像模糊严重，边缘轮廓变得非常柔和，细节几乎不可见。超大的模板尺寸使邻域范围极大，平均作用被最大化，导致图像细节大量丢失。  二、椒盐噪声图像的滤波结果分析  1. 高斯滤波（5×5，σ=1）的表现  高斯滤波对椒盐噪声的抑制效果远不如对高斯噪声的效果。椒盐噪声是灰度的突变（纯黑或纯白像素），高斯滤波的加权平均难以有效消除这种突变，处理后的图像中仍能看到明显的椒盐噪声颗粒，同时图像边缘和细节也出现了不必要的模糊。  2. 均值滤波的表现（不同模板尺寸）  3×3 模板：能一定程度上抑制椒盐噪声，但噪声残留仍较明显，边缘和细节保留相对清晰。  5×5 模板：椒盐噪声抑制效果显著提升，图像中大部分椒盐噪声被过滤，不过边缘和细节的模糊程度也随之增加。  7×7 模板：椒盐噪声几乎被完全抑制，但图像模糊程度进一步加剧，边缘轮廓变得柔和，细节丢失较多。  9×9 模板：椒盐噪声被彻底过滤，但图像严重模糊，细节几乎不可辨识。  结论：均值滤波对椒盐噪声的抑制能力随模板尺寸增大而增强（因为大模板能更有效地 “覆盖” 椒盐噪声的突变像素），但代价是图像模糊和细节丢失也随之加剧。  三、均值滤波与高斯滤波的综合对比   | 滤波类型 | 高斯噪声抑制效果 | 椒盐噪声抑制效果 | 细节保留能力 | 模糊程度 | 适用场景 | | --- | --- | --- | --- | --- | --- | | 高斯滤波 | 优 | 差 | 较强 | 中（与参数有关） | 高斯噪声主导的图像去噪 | | 均值滤波（小核） | 一般 | 一般 | 强 | 低 | 对细节要求高、噪声较轻微的场景 | | 均值滤波（大核） | 较好 | 优 | 弱 | 高 | 椒盐噪声主导、对细节要求低的场景 |   四、总结  高斯滤波更适配高斯噪声，在抑制高斯噪声的同时能相对较好地保留细节；但对椒盐噪声效果不佳。  均值滤波对椒盐噪声的抑制能力随模板尺寸增大而增强，但对高斯噪声的抑制效果整体弱于同参数的高斯滤波（除非使用极大模板）；且均值滤波的细节保留能力随模板尺寸增大而急剧下降，模糊程度急剧上升。  实际应用中，需根据噪声类型（高斯或椒盐）、对细节保留的要求，选择合适的滤波方法及参数（如高斯滤波的 σ、均值滤波的模板尺寸），以实现噪声抑制与细节保留的最佳平衡。  **编程实践三**  **基础实验内容：**  **3.1利用双边滤波器对一幅图像实现填充后，并完成双边滤波，并对均值、高斯、双边滤波器的实验结果进行综合分析。**  **3.2自己设计一个均值滤波器，对一幅图像实现填充后，并完成均值平滑，并对实验结果进行分析。**  **3.3利用课件上关于锐化滤波器的代码，分别对一幅图像实现3\*3的sobel、prewitt滤波，Robert锐化滤波和拉普拉斯锐化滤波并对实验结果进行分析。**  **Code：**  import cv2  import numpy as np  import matplotlib.pyplot as plt  plt.rcParams['font.sans-serif'] = ['SimHei']  plt.rcParams['axes.unicode\_minus'] = False  # -------------------------- 实验一：双边滤波器及三种滤波器综合分析（参数优化） --------------------------  def experiment\_bilateral\_filter(img\_path):        img = cv2.imread(img\_path)      if img is None:          print(f"错误：无法读取图像 {img\_path}")          return      img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)      img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)      kernel\_size = 7      pad\_size = kernel\_size // 2      img\_padded = cv2.copyMakeBorder(          img\_gray,          top=pad\_size, bottom=pad\_size, left=pad\_size, right=pad\_size,          borderType=cv2.BORDER\_REPLICATE      )  # 3. 双边滤波  bilateral\_filtered = cv2.bilateralFilter(img\_padded, d=kernel\_size, sigmaColor=100, sigmaSpace=100)      bilateral\_filtered = bilateral\_filtered[pad\_size:-pad\_size, pad\_size:-pad\_size]      # 4. 均值滤波（    mean\_filtered = cv2.blur(img\_gray, ksize=(kernel\_size, kernel\_size))      # 5. 高斯滤波    gaussian\_filtered = cv2.GaussianBlur(img\_gray, ksize=(kernel\_size, kernel\_size), sigmaX=2.5)      plt.figure(figsize=(15, 10))      plt.subplot(2, 2, 1)      plt.imshow(img\_rgb)      plt.title("原图（RGB）")      plt.axis("off")      plt.subplot(2, 2, 2)      plt.imshow(mean\_filtered, cmap="gray")      plt.title(f"均值滤波（{kernel\_size}×{kernel\_size}）")      plt.axis("off")      plt.subplot(2, 2, 3)      plt.imshow(gaussian\_filtered, cmap="gray")      plt.title(f"高斯滤波（{kernel\_size}×{kernel\_size}，σ=2.5）")      plt.axis("off")      plt.subplot(2, 2, 4)      plt.imshow(bilateral\_filtered, cmap="gray")      plt.title(f"双边滤波（d={kernel\_size}, σ\_color=100, σ\_space=100）")      plt.axis("off")      plt.suptitle("实验一：均值、高斯、双边滤波器对比（参数优化）", fontsize=16, y=0.995)      plt.tight\_layout()      plt.show()  # -------------------------- 实验二：自定义均值滤波器（参数优化） --------------------------  def experiment\_custom\_mean\_filter(img\_path):      # 1. 读取图像      img = cv2.imread(img\_path)      if img is None:          print(f"错误：无法读取图像 {img\_path}")          return      img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)      img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)      # 2. 自定义均值滤波器函数      def custom\_mean\_filter(img, kernel\_size):          h, w = img.shape          pad\_size = kernel\_size // 2          img\_padded = cv2.copyMakeBorder(              img,              top=pad\_size, bottom=pad\_size, left=pad\_size, right=pad\_size,              borderType=cv2.BORDER\_REPLICATE          )          output = np.zeros\_like(img, dtype=np.uint8)          for i in range(h):              for j in range(w):                  window = img\_padded[i:i+kernel\_size, j:j+kernel\_size]                  output[i, j] = np.mean(window).astype(np.uint8)          return output  # 3. 应用自定义均值滤波器   kernel\_size = 5      custom\_mean\_filtered = custom\_mean\_filter(img\_gray, kernel\_size)      # 4. 与 OpenCV 内置均值滤波对比      opencv\_mean\_filtered = cv2.blur(img\_gray, ksize=(kernel\_size, kernel\_size))      # 5. 结果展示      plt.figure(figsize=(15, 8))      plt.subplot(1, 3, 1)      plt.imshow(img\_rgb)      plt.title("原图（RGB）")      plt.axis("off")      plt.subplot(1, 3, 2)      plt.imshow(custom\_mean\_filtered, cmap="gray")      plt.title(f"自定义均值滤波（{kernel\_size}×{kernel\_size}）")      plt.axis("off")      plt.subplot(1, 3, 3)      plt.imshow(opencv\_mean\_filtered, cmap="gray")      plt.title(f"OpenCV 均值滤波（{kernel\_size}×{kernel\_size}）")      plt.axis("off")      plt.suptitle("实验二：自定义均值滤波器与 OpenCV 均值滤波对比（参数优化）", fontsize=16, y=0.85)      plt.tight\_layout()      plt.show()  # -------------------------- 实验三：锐化滤波器（参数优化，增强效果差异） --------------------------  def experiment\_sharpening\_filters(img\_path):      # 1. 读取图像      img = cv2.imread(img\_path)      if img is None:          print(f"错误：无法读取图像 {img\_path}")          return      img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)      img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)      # 2. 定义锐化滤波器核      sobel\_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)      sobel\_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)      prewitt\_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype=np.float32)      prewitt\_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]], dtype=np.float32)      robert\_1 = np.array([[1, 0], [0, -1]], dtype=np.float32)      robert\_2 = np.array([[0, 1], [-1, 0]], dtype=np.float32)      laplacian = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]], dtype=np.float32)  # 3. 锐化滤波函数  def sharpen\_filter(img, kernel, weight=0.8):          edge = cv2.filter2D(img.astype(np.float32), -1, kernel)          edge = cv2.normalize(edge, None, 0, 255, cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)          sharpened = cv2.addWeighted(img, 1.0, edge, weight, 0)          return sharpened  # 4. 分别应用四种锐化滤波      sobel\_x\_sharp = sharpen\_filter(img\_gray, sobel\_x, 0.8)      sobel\_y\_sharp = sharpen\_filter(img\_gray, sobel\_y, 0.8)      sobel\_combined = cv2.addWeighted(sobel\_x\_sharp, 0.5, sobel\_y\_sharp, 0.5, 0)      prewitt\_x\_sharp = sharpen\_filter(img\_gray, prewitt\_x, 0.8)      prewitt\_y\_sharp = sharpen\_filter(img\_gray, prewitt\_y, 0.8)      prewitt\_combined = cv2.addWeighted(prewitt\_x\_sharp, 0.5, prewitt\_y\_sharp, 0.5, 0)      robert\_1\_sharp = sharpen\_filter(img\_gray, robert\_1, 0.8)      robert\_2\_sharp = sharpen\_filter(img\_gray, robert\_2, 0.8)      robert\_combined = cv2.addWeighted(robert\_1\_sharp, 0.5, robert\_2\_sharp, 0.5, 0)      laplacian\_sharp = sharpen\_filter(img\_gray, laplacian, 1.0)      # 5. 结果展示      plt.figure(figsize=(18, 12))      plt.subplot(3, 3, 1)      plt.imshow(img\_rgb)      plt.title("原图（RGB）")      plt.axis("off")      plt.subplot(3, 3, 2)      plt.imshow(sobel\_x\_sharp, cmap="gray")      plt.title("Sobel 水平锐化（强边缘）")      plt.axis("off")      plt.subplot(3, 3, 3)      plt.imshow(sobel\_y\_sharp, cmap="gray")      plt.title("Sobel 垂直锐化（强边缘）")      plt.axis("off")      plt.subplot(3, 3, 4)      plt.imshow(sobel\_combined, cmap="gray")      plt.title("Sobel 组合锐化（强边缘）")      plt.axis("off")      plt.subplot(3, 3, 5)      plt.imshow(prewitt\_x\_sharp, cmap="gray")      plt.title("Prewitt 水平锐化（宽边缘）")      plt.axis("off")      plt.subplot(3, 3, 6)      plt.imshow(prewitt\_y\_sharp, cmap="gray")      plt.title("Prewitt 垂直锐化（宽边缘）")      plt.axis("off")      plt.subplot(3, 3, 7)      plt.imshow(prewitt\_combined, cmap="gray")      plt.title("Prewitt 组合锐化（宽边缘）")      plt.axis("off")      plt.subplot(3, 3, 8)      plt.imshow(robert\_combined, cmap="gray")      plt.title("Robert 组合锐化（细边缘）")      plt.axis("off")      plt.subplot(3, 3, 9)      plt.imshow(laplacian\_sharp, cmap="gray")      plt.title("拉普拉斯锐化（细节增强）")      plt.axis("off")      plt.suptitle("实验三：四种锐化滤波器对比（参数优化）", fontsize=16, y=0.995)      plt.tight\_layout()      plt.show()  # -------------------------- 主函数：运行所有实验 --------------------------  if \_\_name\_\_ == "\_\_main\_\_":      image\_path = "fig.jpg"  # 替换为你的图像路径      print("正在运行实验一：双边滤波器及三种滤波器综合分析...")      experiment\_bilateral\_filter(image\_path)      print("正在运行实验二：自定义均值滤波器...")      experiment\_custom\_mean\_filter(image\_path)      print("正在运行实验三：锐化滤波器...")      experiment\_sharpening\_filters(image\_path)      print("所有实验运行完毕！")  **结果：**        **结果分析：**  **实验3.1：均值、高斯、双边滤波器对比分析**  1. 均值滤波（7×7）  均值滤波对图像的平滑效果极强，图像整体呈现明显的模糊感，边缘和细节（如头发纹理、猫咪的轮廓）几乎被抹平。这是因为均值滤波对邻域内所有像素 “无差别平均”，彻底抑制了高频信息（边缘、细节），虽然能有效消除噪声，但代价是图像细节的大量丢失。  2. 高斯滤波（7×7，σ=2.5）  高斯滤波的模糊程度较均值滤波有所减轻，边缘和细节的保留更优（如头发的层次感、猫咪的面部细节比均值滤波更清晰）。它通过 “加权平均”（中心像素权重高、边缘像素权重低）在噪声抑制和细节保留间取得了平衡，对高斯噪声的抑制效果尤其突出，但仍存在一定程度的边缘模糊。  3. 双边滤波（d=7，σ\_color=100，σ\_space=100）  双边滤波在三者中细节保留能力最强，边缘（如人物的发丝边缘、猫咪的轮廓）几乎未出现明显模糊，同时噪声也得到了有效抑制。它通过同时考虑空间距离和灰度差异，对边缘区域（灰度突变处）降低滤波权重，从而实现了 “保边平滑” 的效果，是三者中综合性能最优的滤波器。  综合结论   | 滤波器类型 | 噪声抑制能力 | 细节保留能力 | 模糊程度 | 适用场景 | | --- | --- | --- | --- | --- | | 均值滤波 | 强 | 弱 | 高 | 对细节要求低、需强噪声抑制的场景 | | 高斯滤波 | 较强 | 较强 | 中 | 高斯噪声主导、需平衡噪声与细节的场景 | | 双边滤波 | 较强 | 强 | 低 | 需保留边缘细节、同时抑制噪声的场景 |   **实验3.2：自定义均值滤波器与 OpenCV 均值滤波对比分析**  自定义均值滤波器（5×5）与 OpenCV 内置均值滤波（5×5）的结果几乎完全一致，图像的模糊程度、细节丢失情况无明显差异。这验证了自定义均值滤波器的正确性—— 其核心逻辑（邻域像素平均）与 OpenCV 均值滤波的原理完全吻合，且在代码实现中（图像填充、滑动窗口遍历、均值计算）未出现偏差。  从效果上看，5×5 均值滤波使图像出现了中度模糊，边缘和细节（如头发纹理、猫咪的面部）有一定程度的丢失，符合均值滤波 “以细节换噪声抑制” 的特性。  **实验3.3：四种锐化滤波器（Sobel、Prewitt、Robert、拉普拉斯）对比分析**  1. Sobel 锐化（3×3）  水平锐化：对垂直边缘（如人物的发丝竖线、猫咪的垂直轮廓）增强效果显著，边缘清晰且对比度高；  垂直锐化：对水平边缘（如人物的发际线、猫咪的水平轮廓）增强效果突出；  组合锐化：综合了水平和垂直方向的边缘增强，整体细节（如头发的层次感、猫咪的面部纹理）最为丰富，边缘清晰度在四种锐化方法中居首。  2. Prewitt 锐化（3×3）  水平 / 垂直锐化的边缘较 Sobel 更 “宽”，视觉上边缘的过渡更柔和，细节增强的同时噪声也会被一定程度放大；  组合锐化的整体效果偏向 “柔和的细节增强”，适合需要保留边缘层次但又不希望边缘过于锐利的场景。  3. Robert 锐化（2×2）  Robert 锐化的边缘较细且对比度高，对对角线方向的边缘（如头发的斜向纹理）增强效果明显，但边缘的连续性不如 Sobel 和 Prewitt，且对噪声的敏感度更高（易放大噪声）。  4. 拉普拉斯锐化（3×3）  拉普拉斯锐化对细节的增强效果最突出（如发丝的细微纹理、猫咪的毛发细节），但同时也会显著放大噪声。它通过 “突出灰度突变” 的方式增强细节，视觉上图像的纹理层次感极强，但对噪声的鲁棒性较差。  综合结论   | 锐化滤波器 | 边缘清晰度 | 细节增强程度 | 噪声放大程度 | 适用场景 | | --- | --- | --- | --- | --- | | Sobel | 高 | 较强 | 中 | 需清晰边缘、综合细节增强的场景 | | Prewitt | 中 | 中 | 中 | 需柔和边缘增强的场景 | | Robert | 中（细边缘） | 中 | 高 | 对角线边缘增强、对噪声不敏感的场景 | | 拉普拉斯 | 中 | 强 | 高 | 需极致细节增强、可接受噪声放大的场景 |   全实验总结  平滑类滤波器（均值、高斯、双边）的核心是抑制高频信息（边缘、细节）以实现噪声消除，其中双边滤波的 “保边” 特性使其在细节保留上优势显著；  锐化类滤波器（Sobel、Prewitt、Robert、拉普拉斯）的核心是增强高频信息以突出边缘和细节，不同滤波器在边缘风格（清晰 / 柔和 / 纤细）、细节增强程度、噪声敏感性上各有侧重；  实际应用中需根据需求（噪声类型、细节重要性、边缘风格）选择合适的滤波器及参数，以达到 “噪声抑制” 与 “细节保留”、“边缘增强” 与 “噪声鲁棒性” 的最佳平衡。  **提高实验内容：**  **3.4试着设计一个均值滤波改进算法，减弱边界模糊现象，并编写程序验证。**  **Code:**  import cv2  import numpy as np  import matplotlib.pyplot as plt  plt.rcParams['font.sans-serif'] = ['SimHei']  plt.rcParams['axes.unicode\_minus'] = False  def improved\_mean\_filter\_strong(img\_gray, kernel\_size=7, edge\_weight=0.2, edge\_threshold=30):      h, w = img\_gray.shape      pad\_size = kernel\_size // 2      img\_padded = cv2.copyMakeBorder(          img\_gray,          pad\_size, pad\_size, pad\_size, pad\_size,          cv2.BORDER\_REPLICATE      )      output = np.zeros\_like(img\_gray, dtype=np.uint8)        sobel\_x = cv2.Sobel(img\_gray, cv2.CV\_64F, 1, 0, ksize=5)      sobel\_y = cv2.Sobel(img\_gray, cv2.CV\_64F, 0, 1, ksize=5)      edge = np.sqrt(sobel\_x\*\*2 + sobel\_y\*\*2)      edge\_mask = edge > edge\_threshold  # 严格判定边缘        for i in range(h):          for j in range(w):              window = img\_padded[i:i+kernel\_size, j:j+kernel\_size]              if edge\_mask[i, j]:                    output[i, j] = np.uint8(edge\_weight \* np.mean(window) + (1 - edge\_weight) \* img\_gray[i, j])              else:                    output[i, j] = np.uint8(np.mean(window))      return output  def verify\_improved\_mean\_filter\_strong(img\_path):      img = cv2.imread(img\_path)      if img is None:          print("图像读取失败！")          return      img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)      img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)        # 原始均值滤波      mean\_filtered = cv2.blur(img\_gray, (7, 7))  # 改进均值滤波  improved\_filtered = improved\_mean\_filter\_strong(          img\_gray,          kernel\_size=7,          edge\_weight=0.2,          edge\_threshold=30      )      plt.figure(figsize=(15, 6))      plt.subplot(1, 3, 1)      plt.imshow(img\_rgb)      plt.title("原图（RGB）")      plt.axis("off")        plt.subplot(1, 3, 2)      plt.imshow(mean\_filtered, cmap="gray")      plt.title("原始均值滤波（7×7）")      plt.axis("off")        plt.subplot(1, 3, 3)      plt.imshow(improved\_filtered, cmap="gray")      plt.title("改进均值滤波（7×7）")      plt.axis("off")        plt.suptitle("改进均值滤波（增强版）vs 原始均值滤波（效果对比强化）", fontsize=16, y=1.05)      plt.tight\_layout()      plt.show()      # 局部放大对比（聚焦发丝和猫咪边缘）      plt.figure(figsize=(12, 4))      # 原始均值滤波局部      plt.subplot(1, 2, 1)      plt.imshow(mean\_filtered[50:200, 50:200], cmap="gray")      plt.title("原始均值滤波（局部：发丝+猫咪边缘）")      plt.axis("off")      # 改进均值滤波局部      plt.subplot(1, 2, 2)      plt.imshow(improved\_filtered[50:200, 50:200], cmap="gray")      plt.title("改进均值滤波（局部：发丝+猫咪边缘）")      plt.axis("off")      plt.tight\_layout()      plt.show()  # 运行验证  if \_\_name\_\_ == "\_\_main\_\_":      img\_path = "fig.jpg"  # 替换为你的图像路径      verify\_improved\_mean\_filter\_strong(img\_path)  **结果：**      **结果分析：**  整体效果对比  从实验结果可以直观看到，原始均值滤波（7×7） 使图像出现了严重的整体模糊，人物发丝的纹理、猫咪的面部细节几乎完全丢失，边缘轮廓变得极其柔和；而改进均值滤波（7×7） 在噪声抑制的同时，成功保留了大量边缘细节，人物发丝的纹理、猫咪的轮廓和面部特征清晰可辨，边界模糊现象被显著削弱。  局部细节对比（发丝 + 猫咪边缘）  在局部放大的对比图中，差异更为明显：  原始均值滤波：发丝边缘完全融合在背景中，猫咪的轮廓过渡极其模糊，细节层次丢失严重；  改进均值滤波：发丝的纹理清晰可区分，猫咪的边缘轮廓锐利且层次分明，几乎接近原图的细节表现。  算法改进原理与优势  该改进算法通过边缘感知 + 权重调整实现了边界模糊的减弱：  边缘检测：利用 Sobel 算子（ksize=5）增强梯度敏感性，通过edge\_threshold=30严格判定边缘区域，确保仅对真实边缘进行特殊处理；  权重调整：在边缘区域仅用edge\_weight=0.2的邻域均值参与计算，保留 80% 的原始像素值，极大减少了边缘的模糊程度；  非边缘区域：仍采用普通均值滤波，保证了噪声抑制效果。  结论  改进后的均值滤波算法在不牺牲噪声抑制能力的前提下，成功解决了传统均值滤波 “边界模糊” 的核心缺陷，实现了 “噪声抑制” 与 “边缘保留” 的平衡，对比效果非常显著，验证了算法设计的有效性。  **3.5编写程序实现对一幅真彩色图像的双边滤波。**  **Code:** import cv2  import numpy as np  import matplotlib.pyplot as plt  # 设置中文显示  plt.rcParams['font.sans-serif'] = ['SimHei']  plt.rcParams['axes.unicode\_minus'] = False  def bilateral\_filter\_color\_image(img\_path, d=9, sigmaColor=75, sigmaSpace=75):      # 读取真彩色图像      img = cv2.imread(img\_path)      if img is None:          print(f"错误：无法读取图像 {img\_path}")          return      # 转换为RGB格式      img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)        # 分离BGR通道      b, g, r = cv2.split(img)      b\_filtered = cv2.bilateralFilter(b, d, sigmaColor, sigmaSpace)      g\_filtered = cv2.bilateralFilter(g, d, sigmaColor, sigmaSpace)      r\_filtered = cv2.bilateralFilter(r, d, sigmaColor, sigmaSpace)        # 合并滤波后的通道      img\_filtered = cv2.merge([b\_filtered, g\_filtered, r\_filtered])      img\_filtered\_rgb = cv2.cvtColor(img\_filtered, cv2.COLOR\_BGR2RGB)        # 展示结果      plt.figure(figsize=(12, 6))      plt.subplot(1, 2, 1)      plt.imshow(img\_rgb)      plt.title("原图（真彩色）")      plt.axis("off")        plt.subplot(1, 2, 2)      plt.imshow(img\_filtered\_rgb)      plt.title(f"双边滤波后")      plt.axis("off")        plt.suptitle("真彩色图像双边滤波效果对比", fontsize=16, y=1.05)      plt.tight\_layout()      plt.show()  if \_\_name\_\_ == "\_\_main\_\_":        image\_path = "fig.jpg"      bilateral\_filter\_color\_image(image\_path, d=9, sigmaColor=75, sigmaSpace=75)  **结果：**    结果分析：  整体效果与模糊性分析  从实验结果可见，双边滤波后图像确实存在一定程度的模糊，这与双边滤波的参数设置密切相关。本次实验中 d=9、sigmaColor=75、sigmaSpace=75 的参数组合，对空间域的平滑作用较强，因此在实现 “保边” 的同时，仍会对非边缘区域的细节产生一定模糊影响。  细节层面的保边与模糊差异  边缘区域：人物发丝的轮廓、猫咪的毛发边缘依然保持锐利，未出现明显的边缘模糊，这体现了双边滤波 “保边” 的核心特性 —— 对灰度（色彩）突变的边缘区域，算法会降低滤波权重，从而保留边缘细节。  非边缘区域：如人物头发的内部纹理、猫咪毛发的细腻层次，在滤波后出现了一定程度的模糊，这是因为非边缘区域的像素灰度（色彩）差异小，算法对其应用了较强的空间域平滑，导致细节丢失。  参数对模糊程度的影响  d（滤波核直径）：d=9 属于较大的核尺寸，邻域范围广，平滑作用强，是导致模糊的主要因素之一；  sigmaSpace（空间域标准差）：sigmaSpace=75 较大，进一步强化了空间域的平滑权重，使得非边缘区域的像素被更广泛地 “平均化”；  sigmaColor（色彩域标准差）：sigmaColor=75 较大，对色彩差异的容忍度高，虽能保证边缘不被过度模糊，但也让算法对非边缘区域的色彩过渡平滑程度要求更高，间接加剧了细节模糊。  结论与优化建议  双边滤波的 “模糊程度” 可通过调整参数进行控制：  若希望减少模糊、保留更多细节，可减小 d、sigmaSpace 的值（如将 d 设为 5，sigmaSpace 设为 30），同时适当降低 sigmaColor（如设为 50）；  本次实验的参数更偏向 “强平滑” 效果，若需平衡 “保边” 与 “细节保留”，需对参数进行精细化调整。  简言之，双边滤波的模糊性是 “平滑强度” 与 “细节保留” 的权衡结果，可通过参数优化实现不同需求下的效果平衡。  **3.6获取DnCNN代码，对代码进行训练和测试，然后对核心代码进行逐行注释。**  **Code:**  import torch  import torch.nn as nn  import torch.optim as optimfrom torch.utils.data  import DataLoader, Datasetfrom torchvision  import datasets, transformsimport numpy as np  import matplotlib.pyplot as pltfrom skimage.metrics  import peak\_signal\_noise\_ratio as psnr  # -------------------------- 1. 定义 DnCNN 模型 --------------------------class DnCNN(nn.Module):  def \_\_init\_\_(self, in\_channels=1, out\_channels=1, num\_layers=17, num\_features=64):  """  Args:  in\_channels: 输入图像通道数（灰度图为1，RGB为3）  out\_channels: 输出图像通道数（与输入一致）  num\_layers: 网络总层数（建议17层，原始论文设计）  num\_features: 中间层卷积核数量（64为原始论文推荐值）  """  # 继承PyTorch的nn.Module基类（所有模型必须继承）  super(DnCNN, self).\_\_init\_\_()    # 第一个卷积层：将输入通道数转为64通道（提取初步噪声特征）  # kernel\_size=3：3x3卷积核（平衡感受野和计算量）  # stride=1：步长1（不改变图像尺寸）  # padding=1：填充1像素（保证输入输出尺寸一致）  # bias=False：后续有BN层，BN会处理均值偏移，故禁用偏置（减少参数）  # 原始论文：第一层不使用BN，避免破坏输入图像的原始分布  layers = [nn.Conv2d(in\_channels, num\_features, kernel\_size=3, stride=1, padding=1, bias=False)]    # 中间层：共15层（总层数17 - 输入层1 - 输出层1），每一层是「Conv + BN + ReLU」组合  for \_ in range(num\_layers - 2):  # 64通道→64通道卷积（深化特征提取，保持通道数稳定）  layers.append(nn.Conv2d(num\_features, num\_features, kernel\_size=3, stride=1, padding=1, bias=False))  # 批量归一化（BN）：标准化特征图（均值0，方差1），加速训练收敛，防止过拟合  layers.append(nn.BatchNorm2d(num\_features))  # ReLU激活函数：引入非线性，让模型能学习复杂的噪声模式（丢弃负特征，保留正特征）  # inplace=True：直接在原张量上修改，节省内存（不创建新张量）  layers.append(nn.ReLU(inplace=True))    # 最后一个卷积层：64通道→输出通道（回归噪声，无激活函数）  # 无激活函数原因：噪声值可正可负（干净图=带噪图-噪声），ReLU会截断负值，导致噪声预测不准  layers.append(nn.Conv2d(num\_features, out\_channels, kernel\_size=3, stride=1, padding=1, bias=False))    # 用nn.Sequential封装所有层，简化前向传播（按顺序执行层计算）  self.model = nn.Sequential(\*layers)    # 初始化网络权重（保证训练稳定性，避免梯度消失/爆炸）  self.\_initialize\_weights()    def \_initialize\_weights(self):  """初始化网络权重（针对Conv2d和BatchNorm2d层）"""  # 遍历网络中所有模块（层）  for m in self.modules():  # 如果是卷积层  if isinstance(m, nn.Conv2d):  # Kaiming正态初始化：适合ReLU激活函数的权重初始化方法  # mode='fan\_out'：让输出特征图的方差一致，避免梯度消失  # nonlinearity='relu'：针对ReLU的特性调整初始化方差  nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')  # 如果卷积层有偏置（本模型中禁用，此处仅为通用写法）  if m.bias is not None:  nn.init.constant\_(m.bias, 0) # 偏置初始化为0  # 如果是BN层  elif isinstance(m, nn.BatchNorm2d):  nn.init.constant\_(m.weight, 1) # BN权重初始化为1（保持方差）  nn.init.constant\_(m.bias, 0) # BN偏置初始化为0（保持均值）    def forward(self, x):  """前向传播：输入带噪图 → 输出预测噪声（核心逻辑）"""  # 输入x：带噪图像（形状：[batch\_size, in\_channels, height, width]）  # 模型计算：通过17层网络提取特征，最终输出预测的噪声  noise = self.model(x)  # 返回预测噪声（去噪图像 = 带噪图像 - 预测噪声，这是残差学习的核心思想）  return noise  # -------------------------- 2. 数据准备（MNIST数据集+添加高斯噪声） --------------------------class NoisyMNIST(Dataset):  def \_\_init\_\_(self, root, train=True, transform=None, sigma=25):  """  自定义数据集：为MNIST图像添加高斯噪声，生成「带噪图-真实噪声」对  Args:  root: ./MNIST  train: True=训练集，False=测试集  transform: 图像预处理变换（如转为Tensor）  sigma: 噪声强度（0-255，对应原始像素值范围）  """  # 加载原始MNIST数据集（干净图像）  self.mnist = datasets.MNIST(root=root, train=train, download=True, transform=transform)  # 噪声强度归一化：MNIST图像转为Tensor后值域为[0,1]，故sigma归一化为sigma/255  self.sigma = sigma / 255.0    def \_\_len\_\_(self):  """返回数据集总样本数（继承Dataset必须实现）"""  return len(self.mnist)    def \_\_getitem\_\_(self, idx):  """获取单个样本（继承Dataset必须实现）：返回（带噪图，真实噪声）"""  # 获取干净图像和标签（标签无用，仅用图像）  clean\_img, \_ = self.mnist[idx] # clean\_img形状：[1, 28, 28]（灰度图）  # 生成高斯噪声：均值=0，标准差=self.sigma，形状与干净图像一致  # torch.randn\_like(clean\_img)：生成与clean\_img同形状的标准正态分布噪声（均值0，方差1）  # 乘以self.sigma：调整噪声强度到目标标准差  noise = torch.randn\_like(clean\_img) \* self.sigma  # 生成带噪图像：干净图像 + 噪声  # torch.clamp：将结果限制在[0.0, 1.0]（避免像素值超出Tensor的有效值域）  noisy\_img = torch.clamp(clean\_img + noise, 0.0, 1.0)  # 返回（带噪图，真实噪声）：模型输入是带噪图，标签是真实噪声（用于计算损失）  return noisy\_img, noise  # 图像预处理变换：仅转为Tensor（MNIST图像本身是灰度图，无需额外处理）# transforms.ToTensor()：将PIL图像转为Tensor，同时自动归一化到[0,1]（除以255）  transform = transforms.Compose([  transforms.ToTensor(),])  # 构建训练集和测试集  train\_dataset = NoisyMNIST(root='./data', train=True, transform=transform, sigma=25)  test\_dataset = NoisyMNIST(root='./data', train=False, transform=transform, sigma=25)  # 构建数据加载器（批量读取数据，支持多线程）# batch\_size=64：每次训练/测试读取64个样本（平衡内存和训练效率）# shuffle=True：训练集打乱样本（避免模型学习样本顺序，提升泛化能力）# shuffle=False：测试集不打乱（便于结果复现）# num\_workers=2：用2个线程加载数据（加速数据读取，避免CPU瓶颈）  train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True, num\_workers=2)  test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False, num\_workers=2)  # -------------------------- 3. 训练模型 --------------------------def train(model, train\_loader, optimizer, criterion, device, epochs=5):  """  训练DnCNN模型  Args:  model: DnCNN模型实例  train\_loader: 训练数据加载器  optimizer: 优化器（如Adam）  criterion: 损失函数（如MSE）  device: 训练设备（cuda/GPU或cpu）  epochs: 训练轮数  """  # 切换模型到训练模式：启用BN层的批量统计更新、启用Dropout（本模型无Dropout，仅为规范写法）  model.train()  # 遍历每个训练轮次  for epoch in range(epochs):  running\_loss = 0.0 # 累计当前轮次的总损失  # 遍历训练集中的每个批次  for i, (noisy\_imgs, true\_noises) in enumerate(train\_loader):  # 将带噪图和真实噪声移到指定设备（GPU/CPU）  noisy\_imgs = noisy\_imgs.to(device)  true\_noises = true\_noises.to(device)    # 清零优化器梯度（避免上一轮梯度累积影响当前轮）  optimizer.zero\_grad()    # 前向传播：输入带噪图，得到模型预测的噪声  pred\_noises = model(noisy\_imgs)    # 计算损失：预测噪声与真实噪声的均方误差（MSE）  # 原因：噪声回归是连续值预测任务，MSE是最常用的损失函数（惩罚大误差，梯度平滑）  loss = criterion(pred\_noises, true\_noises)    # 反向传播：计算损失对模型参数的梯度  loss.backward()    # 优化器更新参数：根据梯度调整权重（最小化损失）  optimizer.step()    # 累计损失：loss.item()是当前批次的平均损失，乘以批次大小得到总损失  running\_loss += loss.item() \* noisy\_imgs.size(0)    # 计算当前轮次的平均损失（总损失 / 训练集总样本数）  epoch\_loss = running\_loss / len(train\_loader.dataset)  # 打印轮次和损失（便于观察训练进度）  print(f'Epoch [{epoch+1}/{epochs}], Loss: {epoch\_loss:.6f}')  # -------------------------- 4. 测试模型 --------------------------def test(model, test\_loader, device, sigma=25):  """  测试DnCNN模型：计算去噪图像的平均PSNR（峰值信噪比，评估图像质量）  Args:  model: 训练好的DnCNN模型  test\_loader: 测试数据加载器  device: 测试设备  sigma: 噪声强度（与训练一致）  """  # 切换模型到评估模式：禁用BN层更新、禁用Dropout（保证测试结果稳定）  model.eval()  total\_psnr = 0.0 # 累计所有测试样本的PSNR  sigma\_norm = sigma / 255.0 # 噪声强度归一化（与训练一致）    # 禁用梯度计算（测试时无需反向传播，节省内存和计算时间）  with torch.no\_grad():  # 遍历测试集中的每个批次  for noisy\_imgs, true\_noises in test\_loader:  # 数据移到指定设备  noisy\_imgs = noisy\_imgs.to(device)  true\_noises = true\_noises.to(device)    # 前向传播：预测噪声  pred\_noises = model(noisy\_imgs)    # 计算去噪图像：带噪图 - 预测噪声（残差学习核心）  # clamp到[0.0,1.0]：避免像素值超出有效范围，保证图像可视化正常  denoised\_imgs = torch.clamp(noisy\_imgs - pred\_noises, 0.0, 1.0)    # 计算真实干净图像：带噪图 - 真实噪声（用于与去噪图对比）  clean\_imgs = torch.clamp(noisy\_imgs - true\_noises, 0.0, 1.0)    # 逐张计算PSNR（需将Tensor转为numpy数组）  # zip(clean\_imgs, denoised\_imgs)：遍历每张干净图和对应去噪图  # cpu().numpy()：将Tensor从GPU移到CPU，并转为numpy数组  for c, d in zip(clean\_imgs.cpu().numpy(), denoised\_imgs.cpu().numpy()):  # 计算MSE（均方误差）：干净图与去噪图的像素差值平方的均值  mse = np.mean((c - d) \*\* 2)  # 计算PSNR：10 \* log10( (max\_pixel^2) / MSE )  # max\_pixel=1：图像归一化后最大像素值为1  # +1e-10：避免MSE=0时除以零报错  psnr\_val = 10 \* np.log10(1.0 / (mse + 1e-10))  # 累计PSNR  total\_psnr += psnr\_val    # 计算测试集平均PSNR（总PSNR / 测试集总样本数）  avg\_psnr = total\_psnr / len(test\_loader.dataset)  print(f'Test Average PSNR: {avg\_psnr:.2f} dB')  return avg\_psnr  # -------------------------- 5. 主函数（执行训练和测试） --------------------------if \_\_name\_\_ == "\_\_main\_\_":  # 配置训练参数  # 自动选择设备：有GPU用GPU（cuda），无GPU用CPU  device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  sigma = 25 # 噪声强度（可调整为15、25、50等，越大噪声越强）  epochs = 5 # 训练轮数（建议至少5轮，轮数越多效果越好，10轮后PSNR基本稳定）  lr = 1e-3 # 学习率（Adam优化器常用值，太大易震荡，太小收敛慢）    # 实例化DnCNN模型（输入输出通道数=1，对应灰度图），并移到指定设备  model = DnCNN(in\_channels=1, out\_channels=1).to(device)    # 定义损失函数和优化器  criterion = nn.MSELoss() # 均方误差损失（适合噪声回归任务）  # Adam优化器：结合Momentum和RMSprop的优点，收敛快且稳定  # model.parameters()：优化模型所有可训练参数  # lr=lr：学习率  optimizer = optim.Adam(model.parameters(), lr=lr)    # 开始训练模型  print("Start training...")  train(model, train\_loader, optimizer, criterion, device, epochs=epochs)    # 开始测试模型  print("Start testing...")  test(model, test\_loader, device, sigma=sigma)    # 保存模型权重（仅保存参数，不保存模型结构，节省空间）  # 文件名格式：dncnn\_噪声强度\_训练轮数.pth（便于后续调用和对比）  torch.save(model.state\_dict(), f'dncnn\_sigma{sigma}\_epochs{epochs}.pth')  print(f"Model saved as 'dncnn\_sigma{sigma}\_epochs{epochs}.pth'") | | | | | | | |