《计算机视觉》实验报告

# 实验08：图像修复

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 庞晓宇 | 学号 | 2020118100 |
| 实验地点 | 信息学院310 | 实验日期 | 2023年10月27日 |

**一、实验内容**

【1】任选图片，进行噪声模拟与统计排序滤波器的恢复实验。

【2】任选图片，用退化模型进行退化模拟，并通过维纳滤波进行复原，需测试噪声的影响。

【3】选合适的图片，进行破损图像修补的实验。

1. **实验过程以及结果分析**

计算机视觉的图像修复是指通过算法和技术手段对受损或受到噪声污染的图像进行恢复和改善的过程。图像修复在许多应用领域都有重要的作用，包括数字摄影、医学影像、监控系统等。 本次实验相关名词： - 噪声模拟（Noise Simulation）：噪声是图像中常见的干扰因素之一，例如高斯噪声、椒盐噪声等。噪声模拟是通过向原始图像添加已知类型和参数的噪声来模拟真实世界中的噪声情况。这样可以帮助研究人员评估和比较不同的图像修复算法的性能。 - 统计排序滤波（Statistical Order Filtering）：统计排序滤波是一种基于排序统计理论的图像去噪方法。它通过对局部区域内的像素值进行排序，然后根据排序结果进行像素值的更新。常见的统计排序滤波器包括中值滤波器、最大值滤波器和最小值滤波器等。这些滤波器能够有效地去除图像中的噪声，并保持图像的边缘信息。 - 退化模拟（Degradation Simulation）：退化模拟是指通过模拟图像在损坏、变形或降质过程中的特性，来研究和设计图像修复算法。例如，模拟图像在传感器噪声、运动模糊或光照变化等因素下的退化情况，可以帮助研究人员更好地理解和处理真实世界中的图像问题。 - 维纳滤波（Wiener Filtering）：维纳滤波是一种经典的图像恢复方法，它基于信号处理和统计学原理。维纳滤波器通过对图像进行频域分析，并根据图像的功率谱和噪声功率谱之间的关系，对图像进行滤波和恢复。维纳滤波器能够有效地减少噪声，并保持图像的细节信息。 - BSCB算法（Block-Based Sparse and Collaborative Representation）：BSCB算法是一种基于稀疏表示理论的图像修复方法。它将图像分成若干个重叠的小块，然后利用稀疏表示模型对每个小块进行重建。BSCB算法通过引入稀疏性和协作性的约束，能够有效地恢复受损图像的细节和结构。

本次实验选用下面的一些图片：  

解决了若干报错后的工具函数包：

import cv2  
import matplotlib.pyplot as plt  
import numpy as np  
import scipy  
  
  
# 高斯噪声  
def normalize(mask, cut=True):  
 if cut:  
 return np.clip(mask, 0, 255) / 255.0  
 return (mask - mask.min()) / (mask.max() - mask.min())  
  
  
def add\_gaussian\_noise(img, mu=0, sigma=25):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 image = np.array(img[:, :, i], dtype=float)  
 noise = np.random.normal(mu, sigma, image.shape)  
 # print(np.mean(noise), np.std(noise))  
 new\_img[:, :, i] = normalize(image + noise) \* 255  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
# 瑞利噪声  
def add\_rayleigh\_noise(img, mu=0, sigma=25):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 image = np.array(img[:, :, i], dtype=float)  
 noise = np.random.rayleigh(scale=sigma, size=image.shape)  
 noise = noise / np.std(noise) \* sigma  
 noise = noise - np.mean(noise) + mu  
 # print(np.mean(noise), np.std(noise))  
 new\_img[:, :, i] = normalize(image + noise) \* 255  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
# 伽马噪声  
def add\_gamma\_noise(img, mu=0, sigma=25):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 image = np.array(img[:, :, i], dtype=float)  
 a = 2 \* mu - np.sqrt(12 \* sigma)  
 b = 2 \* mu + np.sqrt(12 \* sigma)  
 noise = np.random.uniform(a, b, image.shape)  
 # showGrayHist(noise)  
 # print(np.mean(noise), np.std(noise))  
 # print(b\*scale, b\*\*0.5\*scale)  
 new\_img[:, :, i] = normalize(image + noise) \* 255  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
# 椒盐噪声  
def add\_salt\_pepper(img, ps=0.05, pp=0.05):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 h, w = img.shape[:2]  
 mask = np.random.choice((0, 0.5, 1), size=(h, w), p=[pp, (1 - pp - ps), ps])  
 img\_out = img  
 img\_out[mask == 1] = 255  
 img\_out[mask == 0] = 0  
 new\_img = img\_out  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
# 几何均值滤波  
def GeometricMeanOperator(roi):  
 roi = roi.astype(np.float64)  
 p = np.prod(roi)  
 return p \*\* (1 / (roi.shape[0] \* roi.shape[1]))  
  
  
def GeometricMeanAlogrithm(image):  
 # 几何均值滤波  
 new\_image = np.zeros(image.shape)  
 image = cv2.copyMakeBorder(image, 1, 1, 1, 1, cv2.BORDER\_DEFAULT)  
 for i in range(1, image.shape[0] - 1):  
 for j in range(1, image.shape[1] - 1):  
 new\_image[i - 1, j - 1] = GeometricMeanOperator(  
 image[i - 1 : i + 2, j - 1 : j + 2]  
 )  
 new\_image = (new\_image - np.min(image)) \* (255 / np.max(image))  
 return new\_image.astype(np.uint8)  
  
  
def rgbGemoetricMean(image):  
 r, g, b = cv2.split(image)  
 r = GeometricMeanAlogrithm(r)  
 g = GeometricMeanAlogrithm(g)  
 b = GeometricMeanAlogrithm(b)  
 return cv2.merge([r, g, b])  
  
  
# 谐波均值滤波  
def HarmonicMeanOperator(roi):  
 roi = roi.astype(np.float64)  
 if 0 in roi:  
 roi = 0  
 else:  
 roi = scipy.stats.hmean(roi.reshape(-1))  
 return roi  
  
  
def HarmonicMeanAlogrithm(image):  
 # 谐波均值滤波  
 new\_image = np.zeros(image.shape)  
 image = cv2.copyMakeBorder(image, 1, 1, 1, 1, cv2.BORDER\_DEFAULT)  
 for i in range(1, image.shape[0] - 1):  
 for j in range(1, image.shape[1] - 1):  
 new\_image[i - 1, j - 1] = HarmonicMeanOperator(  
 image[i - 1 : i + 2, j - 1 : j + 2]  
 )  
 new\_image = (new\_image - np.min(image)) \* (255 / np.max(image))  
 return new\_image.astype(np.uint8)  
  
  
def rgbHarmonicMean(image):  
 r, g, b = cv2.split(image)  
 r = HarmonicMeanAlogrithm(r)  
 g = HarmonicMeanAlogrithm(g)  
 b = HarmonicMeanAlogrithm(b)  
 return cv2.merge([r, g, b])  
  
  
# 逆谐波均值滤波  
def Contra\_harmonicMeanOperator(roi, q):  
 roi = roi.astype(np.float64)  
 return np.mean(roi \*\* (q + 1)) / np.mean((roi) \*\* (q))  
  
  
def Contra\_harmonicMeanAlogrithm(image, q):  
 # 逆谐波均值滤波  
 new\_image = np.zeros(image.shape)  
 image = cv2.copyMakeBorder(image, 1, 1, 1, 1, cv2.BORDER\_DEFAULT)  
 for i in range(1, image.shape[0] - 1):  
 for j in range(1, image.shape[1] - 1):  
 new\_image[i - 1, j - 1] = Contra\_harmonicMeanOperator(  
 image[i - 1 : i + 2, j - 1 : j + 2], q  
 )  
 new\_image = (new\_image - np.min(image)) \* (255 / np.max(image))  
 return new\_image.astype(np.uint8)  
  
  
def rgbContra\_harmonicMean(image, q):  
 r, g, b = cv2.split(image)  
 r = Contra\_harmonicMeanAlogrithm(r, q)  
 g = Contra\_harmonicMeanAlogrithm(g, q)  
 b = Contra\_harmonicMeanAlogrithm(b, q)  
 return cv2.merge([r, g, b])  
  
  
# 图像退化恢复  
# 仿真运动模糊  
def get\_motion\_dsf(image\_sise, motion\_angle, motion\_dis):  
 PSF = np.zeros(image\_sise) # 点扩散函数  
 x\_center = (image\_sise[0] - 1) / 2  
 y\_center = (image\_sise[1] - 1) / 2  
 sin\_val = np.sin(motion\_angle \* np.pi / 180)  
 cos\_val = np.cos(motion\_angle \* np.pi / 180)  
 # 将对应角度上motion\_dis个点置成1  
 for i in range(motion\_dis):  
 x\_offset = round(sin\_val \* i)  
 y\_offset = round(cos\_val \* i)  
 PSF[int(x\_center - x\_offset), int(y\_center + y\_offset)] = 1  
 return PSF / PSF.sum() # 归一化  
  
  
# 仿真湍流模糊  
def cal\_distance(pa, pb): # 欧拉距离计算函数的定义  
 return np.sqrt((pa[0] - pb[0]) \*\* 2 + (pa[1] - pb[1]) \*\* 2)  
  
  
def get\_turbulence\_dsf(image\_sise, k=0.1):  
 center\_point = tuple(map(lambda x: (x - 1) / 2, image\_sise)) # 中心点  
 pos\_matrix = np.mgrid[0 : image\_sise[0], 0 : image\_sise[1]] # 生成位置矩阵  
 dis = cal\_distance(pos\_matrix, center\_point) # 计算距离矩阵  
 # PSF\_fft = dis  
 PSF\_fft = np.exp(-k \* (dis \*\* (5 / 6))) # 点扩散函数 fft  
 PSF = np.fft.ifft2(PSF\_fft) # image FFT multiplly PSF FFT  
 PSF = np.abs(np.fft.ifftshift(PSF)) # 中心化  
 return PSF / PSF.sum() # 归一化  
  
  
# 图像退化恢复  
# 对图片继续模糊  
def make\_blurred(img, PSF, eps):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 input\_fft = np.fft.fft2(img[:, :, i])  
 PSF\_fft = np.fft.fft2(PSF) + eps  
 blurred = np.fft.ifft2(input\_fft \* PSF\_fft)  
 blurred = np.abs(np.fft.ifftshift(blurred))  
 new\_img[:, :, i] = blurred  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
def inverse\_filter(img, PSF, eps):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 input\_fft = np.fft.fft2(img[:, :, i])  
 PSF\_fft = np.fft.fft2(PSF) + eps  
 result = np.fft.ifft2(input\_fft / PSF\_fft)  
 result = np.abs(np.fft.ifftshift(result))  
 new\_img[:, :, i] = result  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
def wiener\_filter(img, PSF, eps, K=0.01):  
 img = np.expand\_dims(img, axis=-1) if img.ndim == 2 else img  
 new\_img = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 input\_fft = np.fft.fft2(img[:, :, i])  
 PSF\_fft = np.fft.fft2(PSF) + eps  
 PSF\_fft\_1 = np.conj(PSF\_fft) / (np.abs(PSF\_fft) \*\* 2 + K)  
 result = np.fft.ifft2(input\_fft \* PSF\_fft\_1)  
 result = np.abs(np.fft.ifftshift(result))  
 new\_img[:, :, i] = result  
 if img.ndim == 2:  
 new\_img = np.squeeze(new\_img, -1)  
 return np.uint8(new\_img)  
  
  
def spectrum\_show(PSF, title):  
 PSF\_fft = np.fft.fft2(PSF)  
 PSF\_fft\_shift = np.fft.fftshift(PSF\_fft)  
 spectrum = 20 \* np.log(np.abs(PSF\_fft\_shift))  
 plt.imshow(spectrum, cmap="gray")  
 plt.title(title)  
 plt.axis("off")  
 plt.show()  
  
  
# 图像修补BSCB  
def normalize(mask, cut=False):  
 if cut:  
 return np.clip(mask, 0, 255) / 255.0  
 return (mask - mask.min()) / (mask.max() - mask.min())  
  
  
# BSCB https://www.cnblogs.com/jgg54335/p/14561720.html  
  
  
def BSCB\_inpaint(  
 pic\_array,  
 mask=None,  
 epsilon=0.1,  
 inpaint\_iters=6,  
 anidiffuse\_iters=6,  
 delta\_ts=0.02,  
 sensitivites=100,  
 diffuse\_coef=1,  
):  
 # BSCB算法  
 pic\_copy = pic\_array.copy()  
 # epsilon2 = epsilon \*\* 2  
 epsilon2 = epsilon \* epsilon  
 pic\_copy\_ = pic\_array.copy()  
 for i in range(anidiffuse\_iters): # 执行各向异性扩散  
 dx\_dy = np.gradient(pic\_copy) # 计算梯度  
 grad\_norm = (  
 dx\_dy[0] \*\* 2 + dx\_dy[1] \*\* 2 + epsilon2  
 ) \*\* 0.5 # epsilon2是为了防止分母为0  
 if diffuse\_coef == 0:  
 diffuse\_coefs = np.exp(-grad\_norm / sensitivites)  
 else:  
 diffuse\_coefs = 1 / (1 + grad\_norm / sensitivites)  
 dxx = np.gradient(dx\_dy[0], axis=0)  
 dyy = np.gradient(dx\_dy[1], axis=1)  
 laplacian = dxx + dyy  
 if not mask is None:  
 diffuse\_coefs = diffuse\_coefs \* mask  
 pic\_copy = pic\_copy + diffuse\_coefs \* laplacian  
 for i in range(inpaint\_iters): # 执行修补  
 dx\_dy = np.gradient(pic\_copy) # 计算梯度  
 grad\_norm = (dx\_dy[0] \*\* 2 + dx\_dy[1] \*\* 2 + epsilon2) \*\* 0.5  
 dxx = np.gradient(dx\_dy[0], axis=0)  
 dyy = np.gradient(dx\_dy[1], axis=1)  
 laplacian = dxx + dyy  
 dx\_dy\_ = np.gradient(laplacian)  
 if not mask is None:  
 delta\_ts = delta\_ts \* mask  
 delta\_ts = delta\_ts \* (grad\_norm > 0)  
 pic\_copy = (  
 pic\_copy  
 - delta\_ts \* (-dx\_dy[0] \* dx\_dy\_[0] + dx\_dy[1] \* dx\_dy\_[1]) / grad\_norm  
 )  
 # import time  
 # timestamp = time.time()  
 # print("time:", timestamp)  
 # cv2.imwrite(str(timestamp) + ".jpg", img=np.uint8(pic\_copy \* 255))  
 pic\_new = pic\_array.copy()  
 pic\_new[1:-1, 1:-1] = pic\_copy[1:-1, 1:-1] # 更新  
 return pic\_new  
  
  
def gen\_pic\_with\_mask(mask, origin\_pic):  
 origin\_pic[mask == 1.0] = 128  
 return origin\_pic  
  
  
# 图像修补TV  
def normalize(mask, cut=False):  
 if cut:  
 return np.clip(mask, 0, 255) / 255.0  
 return (mask - mask.min()) / (mask.max() - mask.min())  
  
  
# TV https://www.cnblogs.com/hxjbc/p/6675901.html  
  
  
def tv\_inpaint(pic\_array, mask=None, epsilon=0.1, dt=0.1, lambda\_=0.1, withCCD=True):  
 # tv算法  
 pic\_copy = pic\_array.copy()  
 epsilon2 = epsilon \* epsilon  
 # 求梯度  
 dx\_dy = np.gradient(pic\_copy)  
 dx\_dy = dx\_dy / (dx\_dy[0] \*\* 2 + dx\_dy[1] \*\* 2 + epsilon2) \*\* 0.5  
 # 求散度 divergence(zx, zy) = zx\_x + zy\_y  
 dxx = np.gradient(dx\_dy[0], axis=0)  
 dyy = np.gradient(dx\_dy[1], axis=1)  
 div = dxx + dyy  
 if withCCD: # 是否曲率驱动  
 k = np.abs(div)  
 k = k / np.max(k) / dt  
 k = 0.3 + 0.7 \* k\*\*0.1  
 div = div \* k  
 if not mask is None:  
 div = div \* mask  
 # 迭代求解  
 pic\_copy = pic\_copy + dt \* div - lambda\_ \* (pic\_copy - pic\_array)  
 pic\_new = pic\_array.copy()  
 pic\_new[1:-1, 1:-1] = pic\_copy[1:-1, 1:-1] # 更新  
 return pic\_new  
  
  
def gen\_pic\_with\_mask(mask, origin\_pic):  
 origin\_pic[mask == 1.0] = 128  
 return origin\_pic

示例代码：

# 显示原图像  
plt.figure(1, figsize=(6, 6))  
plt.title('Original Image'), plt.imshow(image[..., [2, 1, 0]], 'gray')  
plt.xticks([]), plt.yticks([])  
  
  
# 进行模糊处理  
PSF = get\_motion\_dsf(image.shape[:2], -50, 100)  
PSF = get\_turbulence\_dsf(image.shape[:2])  
spectrum\_show(PSF, title='PSF Image') # 模糊PSF与谱  
blurred = make\_blurred(image, PSF, 1e-3)  
plt.figure(2, figsize=(8, 8))  
plt.subplot(231), plt.imshow(  
 blurred[..., [2, 1, 0]], 'gray'), plt.title('blurred')  
plt.xticks([]), plt.yticks([])  
  
# 逆滤波  
result = inverse\_filter(blurred, PSF, 1e-3)  
plt.subplot(232), plt.imshow(  
 result[..., [2, 1, 0]], 'gray'), plt.title('inverse deblurred')  
plt.xticks([]), plt.yticks([])  
  
# 维纳滤波  
result = wiener\_filter(blurred, PSF, 1e-3)  
plt.subplot(233), plt.imshow(result[..., [2, 1, 0]], 'gray'), plt.title(  
 'wiener deblurred(K=0.01)')  
plt.xticks([]), plt.yticks([])  
  
# 添加噪声,standard\_normal产生随机的函数  
blurred\_noisy = np.uint8(blurred + 0.1\*blurred.std()  
 \* np.random.standard\_normal(blurred.shape))  
  
# 显示添加噪声且模糊的图像  
plt.subplot(234), plt.imshow(  
 blurred\_noisy[..., [2, 1, 0]], 'gray'), plt.title('blurred and noisy')  
  
# 对添加噪声的图像进行逆滤波  
result = inverse\_filter(blurred\_noisy, PSF, 0.1+1e-3)  
plt.subplot(235), plt.imshow(  
 result[..., [2, 1, 0]], 'gray'), plt.title('inverse deblurred')  
plt.xticks([]), plt.yticks([])  
  
# 对添加噪声的图像进行维纳滤波  
result = wiener\_filter(blurred\_noisy, PSF, 0.1+1e-3)  
plt.subplot(236), plt.imshow(result[..., [2, 1, 0]], 'gray'), plt.title(  
 'wiener deblurred(k=0.01)')  
plt.xticks([]), plt.yticks([])  
  
plt.tight\_layout()  
plt.savefig('out.jpg', format='jpg', bbox\_inches='tight', dpi=96)  
plt.show()  
  
  
# ----------------------------------------------  
  
img = cv2.imread('dog\_defiled.jpg')  
# 图片二值化处理，把[0, 0, 200]~[70, 70, 255] 以外的颜色变成0  
thresh = cv2.inRange(img, np.array([0, 0, 200]), np.array([70, 70, 255]))  
# 创建形状和尺寸的结构元素  
kernel = np.ones((3, 3), np.uint8)  
# 扩张待修复区域  
mask = cv2.dilate(thresh, kernel, iterations=1)/255.0  
specturm\_show(mask)  
  
pic = gen\_pic\_with\_mask(mask, img)  
  
epsilon = 0.1  
inpaint\_iters = 6  
anidiffuse\_iters = 6  
delta\_ts = 0.2  
sensitivites = 100  
diffuse\_coef = 1  
  
epochs = 201  
pic = (pic/255.0).astype(np.float)  
pic\_copy = np.zeros(pic.shape)  
for epoch in range(epochs):  
 # 每epochs次显示一次数据，保存一次数据  
 if epoch % 40 == 0:  
 print('epoch,当前的循环次数：', epoch, np.abs(pic-pic\_copy).max())  
 cv2.imwrite('dog\_filed'+str(epoch)+'.jpg', img=np.uint8(pic\*255))  
 pic\_copy = pic.copy()  
 if epoch < epoch-1:  
 for i in range(3):  
 pic[:, :, i] = BSCB\_inpaint(pic\_copy[:, :, i], mask=mask,  
 epsilon=epsilon, inpaint\_iters=inpaint\_iters, anidiffuse\_iters=anidiffuse\_iters, delta\_ts=delta\_ts,  
 sensitivites=sensitivites, diffuse\_coef=diffuse\_coef)  
pic = np.uint8(pic\*255)  
specturm\_show(pic)  
  
# ----------------------------------------------  
  
  
img = cv2.imread('dog\_defiled.jpg')  
# 图片二值化处理，把[240, 240, 240]~[255, 255, 255] 以外的颜色变成0  
thresh = cv2.inRange(img, np.array([0, 0, 200]), np.array([70, 70, 255]))  
# 创建形状和尺寸的结构元素  
kernel = np.ones((3, 3), np.uint8)  
# 扩张待修复区域  
mask = cv2.dilate(thresh, kernel, iterations=1)/255.0  
specturm\_show(mask)  
  
pic = gen\_pic\_with\_mask(mask, img)  
  
epsilon = 0.1  
dt = 0.1  
lambda\_ = 0.1  
epochs = 601  
pic = (pic/255.0).astype(np.float)  
pic\_copy = np.zeros(pic.shape)  
  
for epoch in range(epochs):  
 # 每epochs次显示一次数据，保存一次数据  
 if epoch % 100 == 0:  
 print('epoch,当前的循环次数：', epoch, np.abs(pic-pic\_copy).max())  
 cv2.imwrite('dog\_filed'+str(epoch)+'.jpg', img=np.uint8(pic\*255))  
 pic\_copy = pic.copy()  
 if epoch < epoch-1:  
 for i in range(3):  
 pic[:, :, i] = tv\_inpaint(pic\_copy[:, :, i], mask=mask,  
 epsilon=epsilon, dt=dt, lambda\_=lambda\_)  
pic = np.uint8(pic\*255)  
specturm\_show(pic)  
  
  
# 图像修补NS、FMM  
img = cv2.imread('dog\_defiled.jpg')  
# 图片二值化处理，把[240, 240, 240]~[255, 255, 255] 以外的颜色变成0  
thresh = cv2.inRange(img, np.array([0, 0, 200]), np.array([70, 70, 255]))  
# 创建形状和尺寸的结构元素  
kernel = np.ones((3, 3), np.uint8)  
# 扩张待修复区域  
mask = cv2.dilate(thresh, kernel, iterations=1)  
specturm\_show(mask)  
  
out = cv2.inpaint(img, mask, inpaintRadius=-1, flags=cv2.INPAINT\_TELEA)  
  
specturm\_show(out)  
cv2.imshow('dog\_mask.jpg', mask)  
cv2.imwrite('dog\_filed.jpg', out)  
  
# 图像修补PatchMatch

首先对图片进行噪声模拟，这里分别对两张图片进行四种不同的噪声模拟，分别是高斯噪声、瑞利噪声、伽马噪声和椒盐噪声。

任选图片，进行噪声模拟与统计排序滤波器的恢复实验。  
  
# 噪声模拟  
dogImg = cv2.imread('./imgs/dog.jpg')  
  
# 高斯狗  
gaussian\_noiseed = add\_gaussian\_noise(dogImg)  
cv2.imwrite('./noiseed/dog\_gaussian\_noiseed.jpg', gaussian\_noiseed)  
  
# 瑞利狗  
rayleigh\_noiseed = add\_rayleigh\_noise(dogImg)  
cv2.imwrite('./noiseed/dog\_rayleigh\_noiseed.jpg', rayleigh\_noiseed)  
  
# 伽马狗  
gamma\_noiseed = add\_gamma\_noise(dogImg)  
cv2.imwrite('./noiseed/dog\_gamma\_noiseed.jpg', gamma\_noiseed)  
  
# 椒盐狗  
salt\_peppered = add\_salt\_pepper(dogImg)  
cv2.imwrite('./noiseed/dog\_salt\_peppered.jpg', salt\_peppered)  
  
  
catImg = cv2.imread('./imgs/cat.jpg')  
  
# 高斯猫  
gaussian\_noiseed = add\_gaussian\_noise(catImg)  
cv2.imwrite('./noiseed/cat\_gaussian\_noiseed.jpg', gaussian\_noiseed)  
  
# 瑞利猫  
rayleigh\_noiseed = add\_rayleigh\_noise(catImg)  
cv2.imwrite('./noiseed/cat\_rayleigh\_noiseed.jpg', rayleigh\_noiseed)  
  
# 伽马猫  
gamma\_noiseed = add\_gamma\_noise(catImg)  
cv2.imwrite('./noiseed/cat\_gamma\_noiseed.jpg', gamma\_noiseed)  
  
# 椒盐猫  
salt\_peppered = add\_salt\_pepper(catImg)  
cv2.imwrite('./noiseed/cat\_salt\_peppered.jpg', salt\_peppered)

下面是四种方式处理的狗的图片：    

对噪声图像进行统计排序滤波器的恢复实验。这里使用了几何均值滤波、谐波均值滤波和逆谐波均值滤波多种方法进行恢复。

# 统计排序滤波器的恢复  
dog\_gaussian\_noiseed = cv2.imread("./noiseed/dog\_gaussian\_noiseed.jpg")  
dog\_rayleigh\_noiseed = cv2.imread("./noiseed/dog\_rayleigh\_noiseed.jpg")  
dog\_gamma\_noiseed = cv2.imread("./noiseed/dog\_gamma\_noiseed.jpg")  
dog\_salt\_peppered = cv2.imread("./noiseed/dog\_salt\_peppered.jpg")  
  
# 几何均值滤波  
fixed\_dog\_gaussian\_noiseed = rgbGemoetricMean(dog\_gaussian\_noiseed)  
cv2.imwrite("./fixed/GemoetricMean-dog\_gaussian\_noiseed.jpg",  
 fixed\_dog\_gaussian\_noiseed)  
  
fixed\_dog\_rayleigh\_noiseed = rgbGemoetricMean(dog\_rayleigh\_noiseed)  
cv2.imwrite("./fixed/GemoetricMean-dog\_rayleigh\_noiseed.jpg",  
 fixed\_dog\_rayleigh\_noiseed)  
  
fixed\_dog\_gamma\_noiseed = rgbGemoetricMean(dog\_gamma\_noiseed)  
cv2.imwrite("./fixed/GemoetricMean-dog\_gamma\_noiseed.jpg",  
 fixed\_dog\_gamma\_noiseed)  
  
fixed\_dog\_salt\_peppered = rgbGemoetricMean(dog\_salt\_peppered)  
cv2.imwrite("./fixed/GemoetricMean-dog\_salt\_peppered.jpg",  
 fixed\_dog\_salt\_peppered)  
  
  
# 谐波均值滤波  
fixed\_dog\_gaussian\_noiseed = rgbHarmonicMean(dog\_gaussian\_noiseed)  
cv2.imwrite("./fixed/HarmonicMean-dog\_gaussian\_noiseed.jpg",  
 fixed\_dog\_gaussian\_noiseed)  
  
  
fixed\_dog\_rayleigh\_noiseed = rgbHarmonicMean(dog\_rayleigh\_noiseed)  
cv2.imwrite("./fixed/HarmonicMean-dog\_rayleigh\_noiseed.jpg",  
 fixed\_dog\_rayleigh\_noiseed)  
  
fixed\_dog\_gamma\_noiseed = rgbHarmonicMean(dog\_gamma\_noiseed)  
cv2.imwrite("./fixed/HarmonicMean-dog\_gamma\_noiseed.jpg",  
 fixed\_dog\_gamma\_noiseed)  
  
fixed\_dog\_salt\_peppered = rgbHarmonicMean(dog\_salt\_peppered)  
cv2.imwrite("./fixed/HarmonicMean-dog\_salt\_peppered.jpg",  
 fixed\_dog\_salt\_peppered)  
  
  
# 逆谐波均值滤波  
fixed\_dog\_gaussian\_noiseed = rgbContra\_harmonicMean(dog\_gaussian\_noiseed, 1.5)  
cv2.imwrite("./fixed/InverseHarmonicMean-dog\_gaussian\_noiseed.jpg",  
 fixed\_dog\_gaussian\_noiseed)  
  
fixed\_dog\_rayleigh\_noiseed = rgbContra\_harmonicMean(dog\_rayleigh\_noiseed, 1.5)  
cv2.imwrite("./fixed/InverseHarmonicMean-dog\_rayleigh\_noiseed.jpg",  
 fixed\_dog\_rayleigh\_noiseed)  
  
fixed\_dog\_gamma\_noiseed = rgbContra\_harmonicMean(dog\_gamma\_noiseed, 1.5)  
cv2.imwrite("./fixed/InverseHarmonicMean-dog\_gamma\_noiseed.jpg",  
 fixed\_dog\_gamma\_noiseed)  
  
fixed\_dog\_salt\_peppered = rgbContra\_harmonicMean(dog\_salt\_peppered, 1.5)  
cv2.imwrite("./fixed/InverseHarmonicMean-dog\_salt\_peppered.jpg",  
 fixed\_dog\_salt\_peppered)

可以看到对于椒盐噪声，几种方法都不能特别理想的恢复图片，而对于高斯噪声，几何均值滤波和谐波均值滤波效果较好，逆谐波均值滤波效果较差，而对于瑞利噪声，几何均值滤波和谐波均值滤波效果较差，逆谐波均值滤波效果较好。

在实验的第二部分，我选择了猫的图片进行退化模拟：

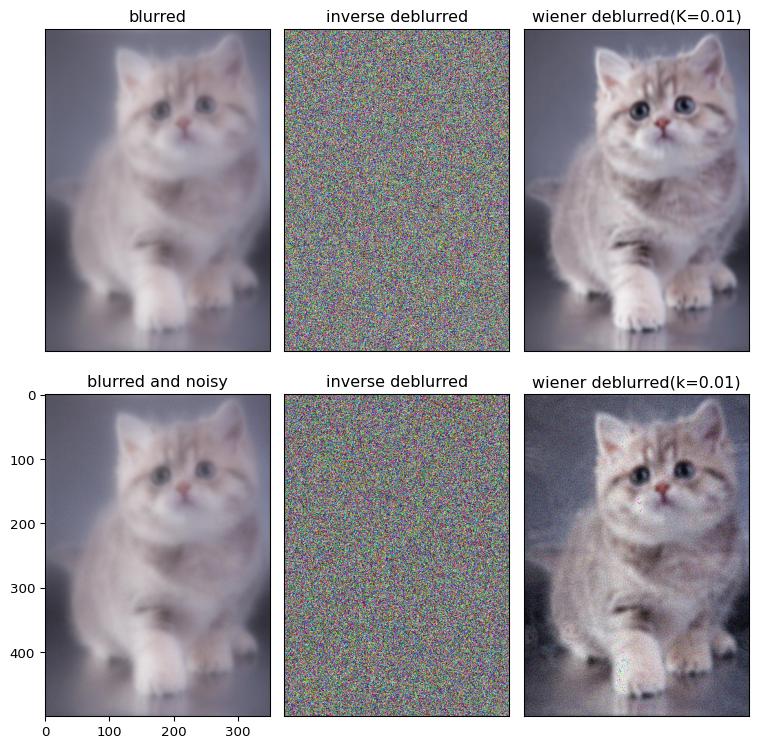
# 退化模拟  
image = cv2.imread('./imgs/cat.jpg')  
  
# 显示原图像  
plt.figure(1, figsize=(6, 6))  
plt.title('Original Image'), plt.imshow(image[..., [2, 1, 0]], 'gray')  
plt.xticks([]), plt.yticks([])  
  
  
# 进行模糊处理  
PSF = get\_motion\_dsf(image.shape[:2], -50, 100)  
PSF = get\_turbulence\_dsf(image.shape[:2])  
spectrum\_show(PSF, title='PSF Image') # 模糊PSF与谱  
blurred = make\_blurred(image, PSF, 1e-3)  
plt.figure(2, figsize=(8, 8))  
plt.subplot(231), plt.imshow(  
 blurred[..., [2, 1, 0]], 'gray'), plt.title('blurred')  
plt.xticks([]), plt.yticks([])

之后使用逆滤波和维纳滤波尝试对图像进行了修复

# 逆滤波  
result = inverse\_filter(blurred, PSF, 1e-3)  
plt.subplot(232), plt.imshow(  
 result[..., [2, 1, 0]], 'gray'), plt.title('inverse deblurred')  
plt.xticks([]), plt.yticks([])  
  
# 维纳滤波  
result = wiener\_filter(blurred, PSF, 1e-3)  
plt.subplot(233), plt.imshow(result[..., [2, 1, 0]], 'gray'), plt.title(  
 'wiener deblurred(K=0.01)')  
plt.xticks([]), plt.yticks([])

下面是对于噪声影响的测试：

# 添加噪声,standard\_normal产生随机的函数  
blurred\_noisy = np.uint8(blurred + 0.1\*blurred.std()  
 \* np.random.standard\_normal(blurred.shape))  
  
# 显示添加噪声且模糊的图像  
plt.subplot(234), plt.imshow(  
 blurred\_noisy[..., [2, 1, 0]], 'gray'), plt.title('blurred and noisy')  
  
# 对添加噪声的图像进行逆滤波  
result = inverse\_filter(blurred\_noisy, PSF, 0.1+1e-3)  
plt.subplot(235), plt.imshow(  
 result[..., [2, 1, 0]], 'gray'), plt.title('inverse deblurred')  
plt.xticks([]), plt.yticks([])  
  
# 对添加噪声的图像进行维纳滤波  
result = wiener\_filter(blurred\_noisy, PSF, 0.1+1e-3)  
plt.subplot(236), plt.imshow(result[..., [2, 1, 0]], 'gray'), plt.title(  
 'wiener deblurred(k=0.01)')  
plt.xticks([]), plt.yticks([])  
  
plt.tight\_layout()  
plt.savefig('out.jpg', format='jpg', bbox\_inches='tight', dpi=96)  
plt.show()

测试结果如下： 

在实验的第三部分，我们进行破损图像修补的实验，这部分我们继续选用狗的图片进行实验。

# def specturm\_show(img: cv2.UMat):  
# # 转换为灰度图像  
# gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
  
# # 计算傅里叶变换  
# f = np.fft.fft2(gray)  
# fshift = np.fft.fftshift(f)  
  
# # 计算频谱  
# magnitude\_spectrum = 20\*np.log(np.abs(fshift))  
  
# # 显示频谱  
# plt.imshow(magnitude\_spectrum, cmap='gray')  
# plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])  
# plt.show()  
  
  
# 【3】选合适的图片，进行破损图像修补的实验。  
img = cv2.imread("./imgs/dog.jpg")  
# 图片二值化处理，把[0, 0, 200]~[70, 70, 255] 以外的颜色变成0  
# thresh = cv2.inRange(img, np.array([0, 0, 200]), np.array([70, 70, 255]))  
thresh = cv2.inRange(img, np.array([110, 50, 50]), np.array([130, 255, 255]))  
# 创建形状和尺寸的结构元素  
kernel = np.ones((3, 3), np.uint8)  
# 扩张待修复区域  
mask = cv2.dilate(thresh, kernel, iterations=1) / 255.0  
  
# specturm\_show(mask)  
# spectrum\_show(mask, "title")  
  
pic = gen\_pic\_with\_mask(mask, img)  
cv2.imwrite("destroyedDog.jpg", img=pic)  
  
  
# cv2.imshow("original", img)  
# cv2.imshow("mask", mask)  
# cv2.imshow("pic", pic)

经过处理后的图片如下： 

epsilon = 0.1  
inpaint\_iters = 6  
anidiffuse\_iters = 6  
delta\_ts = 0.2  
sensitivites = 100  
diffuse\_coef = 1  
  
epochs = 201  
# pic = (pic/255.0).astype(np.float)  
pic = (pic / 255.0).astype(float)  
pic\_copy = np.zeros(pic.shape)  
for epoch in range(epochs):  
 # 每epochs次显示一次数据，保存一次数据  
 if epoch % 40 == 0:  
 print("epoch,当前的循环次数：", epoch, np.abs(pic - pic\_copy).max())  
 cv2.imwrite("dog\_filed" + str(epoch) + ".jpg", img=np.uint8(pic \* 255))  
 pic\_copy = pic.copy()  
 if epoch < epochs - 1:  
 for i in range(3):  
 pic[:, :, i] = BSCB\_inpaint(  
 pic\_copy[:, :, i],  
 mask,  
 epsilon,  
 inpaint\_iters,  
 anidiffuse\_iters,  
 delta\_ts,  
 sensitivites,  
 diffuse\_coef,  
 )  
pic = np.uint8(pic \* 255)  
  
# specturm\_show(pic)  
# spectrum\_show(pic, "title")

修复后的图片如下（从上到下每40次迭代输出一张图片）     

可以看到算法较好的修复了图片的破损部分。

1. **实验总结**

本次实验主要涉及数字图像处理中的三个重要问题：噪声去除、图像复原和图像修补。通过对不同类型的图像进行模拟和处理，我们深入理解了这些问题的本质，掌握了相应的算法和技术，并为实际应用提供了参考。

在噪声去除方面，我们使用了几何均值滤波、谐波均值滤波和逆谐波均值滤波等方法来处理高斯、瑞利、伽马和椒盐四种噪声。实验结果显示，不同噪声和不同滤波方式对图像的影响差异明显，特别是在滤除椒盐噪声方面，逆谐波滤波效果更好。

在图像复原方面，我们模拟了图像的退化过程，如模糊和加噪声，并使用维纳滤波器进行图像复原。实验结果表明，维纳滤波器可以在一定程度上恢复退化图像，但受到噪声的干扰。

在图像修补方面，我们使用了基于纹理合成和插值的方法来修补破损图像，并测试了不同参数对修补效果的影响。实验结果显示，这些方法可以在一定程度上恢复破损图像，但也存在一些缺陷，如边缘过渡不自然、纹理不连续等。

通过本次实验，我们深入了解了图像处理中的噪声去除、图像复原和图像修补问题，并掌握了相应的算法和技术。我们发现图像处理的效果受到多种因素的影响，如阈值选择、结构元素的形状和算法的参数设置等。因此，在实际应用中，需要根据具体情况选择合适的方法和参数来实现更好的图像处理效果。

在实验过程中，我们也遇到了一些编译问题和bug，这提醒我们在编程过程中要保持仔细谨慎，注意细节。同时，我们也意识到编程是数字图像处理中的关键环节，需要不断学习和提升编程技术，以应对实际应用中的挑战。

1. **材料提交**

1.实验要求内容完备(实验代码、实验结果及分析)、格式规范、排版美观。

2.实验过程中遇到问题需记录具体问题和解决方法；

3.把相关材料(包括实验报告、实验代码、实验使用到的图片等数据)压缩打包为“计算机视觉实验08\_学号\_姓名.zip”，提交到邮箱pengshenglin@nwu.edu.cn；

4.截止时间为实验课当周周日24点前(如实验课在周六周日，截止时间为下周周二24点前)。

5.不要迟交，不要抄袭(迟交当次作业最多70分，抄袭整个课程记0分！)。实验报告整体雷同且存在以下情况判为抄袭：程序仅有极少字符与变量的不同且；程序仅有空格和分行的不同；存在从网页复制导致的乱码，全角符号，非ASCII符号，&nbsp;等；代码高度相似并且程序存在完全相同的错误。(重复教材上的代码不计入抄袭)