تمرین سری دوم – بخش عملی

سوال اول – گزارش کار

مهرشاد فلاح اسطلخزیر – 401521462

منطق کد:

کرنل Identity:

```
image = cv2.imread("test.jpg")
"""
Apply identity kernel
"""
kernel1 = np.array([0.0, 0.0, 0.0,
                    0.0, 1.0, 0.0,
                    0.0, 0.0, 0.0]).reshape(3, 3)
# filter2D() function can be used to apply kernel to an image.
# Where ddepth is the desired depth of final image. ddepth is -1 if...
# ... depth is same as original or source image.
identity = cv2.filter2D(image, ddepth = -1 , kernel = kernel1)

# We should get the same image
cv2.imshow('Original', image)
cv2.imshow('Identity', identity)

cv2.waitKey(0)
print(1)
cv2.imwrite('identity.jpg', identity)
cv2.destroyAllWindows()
```

18.5s

ابتدا یک ndarray 3 در 3 با استفاده از کتابخانه numpy درست می‌کنم و بعد از تابع filter2D که در توضیحات آمده بود استفاده می‌کنیم و عکس را فیلتر می‌کنیم. توضیحات راجع به کرنل Identity از این سایت مشاهده شد.

کرنل Blur:

```
"""
Apply blurring kernel
"""
kernel2 = 1/25 * np.ones((5, 5) , dtype = np.float32)

img = cv2.filter2D(image, kernel = kernel2, ddepth = -1)

cv2.imshow('Original', image)
cv2.imshow('Kernel Blur', img)

cv2.waitKey()
cv2.imwrite('blur_kernel.jpg', img)
cv2.destroyAllWindows()
```

3.8s

برای این کرنل هم از توضیحات نحوه ساخت را فهمیدم به این صورت که با کمک np.ones که به ابعاد مورد نظر ما یک آرایه متشکل از یک می‌سازد یک آرایه با 1 ساختم و در مرحله بعد 1/25 را در این آرایه ضرب کردم که به دلیل پشتیبانی کتابخانه numpy از vectorization نیازی به ضرب این عبارت در تک تک عنصرها نبود و در نهایت از filter2D طبق خواسته سوال استفاده کردم.

کرنل GaussianBlur:

```
"""
Apply Gaussian blur
"""
# sigmaX is Gaussian Kernel standard deviation
# ksize is kernel size
gaussian_blur = cv2.GaussianBlur(image, ksize = (5, 5), sigmaX = 0, sigmaY = 0)


cv2.imshow('Original', image)
cv2.imshow('Gaussian Blurred', gaussian_blur)

cv2.waitKey()
cv2.imwrite('gaussian_blur.jpg', gaussian_blur)
cv2.destroyAllWindows()
```

54.8s

برای این کرنل صرفا از توضیحات آورده شده برای تابع cv2.GaussianBlur استفاده کردم و پارامترها را هم بر همان اساس مقداردهی کردم.

کرنل MedianBlur:

```
"""
Apply Median blur
"""
# medianBlur() is used to apply Median blur to image
# ksize is the kernel size
median = cv2.medianBlur(image, 5)

cv2.imshow('Original', image)
cv2.imshow('Median Blurred', median)

cv2.waitKey()
cv2.imwrite('median_blur.jpg', median)
cv2.destroyAllWindows()
```

✓ 7.1s

برای این کرنل هم صرفا از داک خود سوال استفاده کردم و مقداردهی را بر اساس تابع قبلی و کرنل سایز 5 انجام دادم.

کرنل Sharpening:

```
"""
Apply sharpening using kernel
"""
kernel3 = np.array([-1, -1, -1, -1, 9, -1, -1, -1, -1]).reshape(3, 3)
sharp_img = cv2.filter2D(image, kernel = kernel3, ddepth = -1)

cv2.imshow('Original', image)
cv2.imshow('Sharpened', sharp_img)

cv2.waitKey()
cv2.imwrite('sharp_image.jpg', sharp_img)
cv2.destroyAllWindows()
```

✓ 2.8s

این کرنل در واقع متفاوت از sharpen kernel است. (کرنل سمت راست sharpening و کرنل دوم از سمت راست sharpen است). برای این منظور ابتدا با استفاده از numpy یک ndarray 3 در 3 ساختم و با استفاده از filter2D آن را بر عکس اعمال کردم.

فیلتر bilateral:

```
"""
Apply Bilateral Filtering
"""
# Using the function bilateralFilter() where d is diameter of each...
# ...pixel neighborhood that is used during filtering.
# sigmaColor is used to filter sigma in the color space.
# sigmaSpace is used to filter sigma in the coordinate space.
bilateral_filter = cv2.bilateralFilter(image, 15, 75, 75)

cv2.imshow('Original', image)
cv2.imshow('Bilateral Filtering', bilateral_filter)

cv2.waitKey(0)
cv2.imwrite('bilateral_filtering.jpg', bilateral_filter)
cv2.destroyAllWindows()
✓ 6.1s
```

طبق توضیحات آورده شده در سوال و لینک این فیلتر وظیفه smooth کردن عکس و همچنین رفع نویز را دارد و برای مقدار دهی هم از لینک داده شده کمک گرفتم.

تشخیص لبه با canny و sobel:

```python
import cv2

# Read the original image (test2.jpg) # This tiger image will be used for all the examples here.
img = cv2.imread("test2.jpg")
# Display original image
cv2.imshow('Original', img)
cv2.waitKey(0)

# Convert to graycsale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Blur the image for better edge detection
img_blur = cv2.GaussianBlur(img_gray, ksize = (3, 3), sigmaX = 0, sigmaY = 0)

# Sobel Edge Detection
sobelx = cv2.Sobel(img_blur, -1, 1, 0, ksize = 3)
sobely = cv2.Sobel(img_blur, -1, 0, 1, ksize = 3)
sobelxy = cv2.Sobel(img_blur, -1, 1, 1, ksize = 3)
# Display Sobel Edge Detection Images
cv2.imshow('Sobel X', sobelx)
cv2.waitKey(0)
cv2.imshow('Sobel Y', sobely)
cv2.waitKey(0)
cv2.imshow('Sobel X Y using Sobel() function', sobelxy)
cv2.waitKey(0)

# Canny Edge Detection
edges = cv2.Canny(img_blur, 100, 200)
# Display Canny Edge Detection Image
cv2.imshow('Canny Edge Detection', edges)
cv2.waitKey(0)

cv2.destroyAllWindows()
```

(function) sigmaX: Any

✓ 16.7s

برای اینکه برای عکس دوم تشخیص لبه را انجام دهیم. ابتدا عکس را می‌خوانیم سپس با استفاده از تابع cv2.cvtColor رنگ عکس را از BGR که دیفالت opencv هست تبدیل به GRAY می‌کنیم. در مرحله بعد همانطور که در بالا هم انجام شد از فیلتر GaussianBlur استفاده می‌کنیم و عکس را smooth می‌کنیم تا لبه‌ها بهتر پیدا شوند. در مرحله بعد از Sobel استفاده می‌کنیم. ابتدا در جهت x سپس در جهت y و در مرحله بعد در هر دو جهت x و y این کار را انجام می‌دهیم. نحوه مشخص کردن جهات هم با پارامتر 3 و 4 در تابع مشخص می‌شود (اگر یک باشد یعنی این جهت هم شامل شود). در آخرین مرحله هم از Canny با آستانه اندازه‌گیری پایین 100 و آستانه اندازه‌گیری بالای 200 استفاده می‌کنیم.

نتایج و تحلیل‌ها:

عکس اصلی

کرنل Identity:

همانطور که از شکل کرنل مشخص است عملا این کرنل کاری انجام نمی‌دهد و باید تصویر اولیه را برگرداند چرا که تمام سلول‌های همسایه سلول مرکزی را بی‌اثر می‌کند و مقدار همون سلول را به عنوان خروجی کانولوشن برمی‌گرداند.

فیلتر Identity

کرنل Blur:

این کرنل تصویر را smooth می‌کند اما اثر تمام خانه‌های همسایه سلول مرکزی را برابر قرار می‌دهد و همه به یک اندازه در پیکسل تاثیر دارند. به همین دلیل ممکن است در نقاطی که در همسایگی آن‌ها رنگ سیاه است تغییر رنگ شاهد باشیم. همچنین مقدار نویزها هم تقویت شده.

کرنل Blur

کرنل GaussianBlur:

این کرنل هم تصویر خروجی را smooth می‌کند اما هر چه فاصله همینگ یک سلول به سلول مرکزی کمتر باشد تاثیر آن سلول بیشتر می‌شود و بیشتر بر روی مقدار پیکسل خروجی تاثیر می‌گذارد. همانطور که در تصویر پایین هم مشخص است اثر نزدیکی بر هر پیکسل بیشتر مشخص است تا Blur عادی. البته که نویزها را هم تقویت کرده و میزان نویزها بیشتر شده.

کرنل GaussianBlur

کرنل medianBlur:

این کرنل مقدار میانه در همسایگی را به عنوان خروجی پیکسل مقداردهی می‌کند. و به همین دلیل در رفع یکسری نویز که در عکس وجود داشت با موفقیت عمل کرده البته که در برخی نقاط اطلاعات مهم تصویر را از بین برده و خطوط و لبه‌ها دیگر قابل مشاهده نیستند.

کرنل medianBlur

کرنل Sharpening:

این کرنل لبه‌ها را بولدتر می‌کند و مقادیر آنها را افزایش می‌دهد و برای لبه‌یابی مناسب‌تر است و برعکس کرنل‌های بالا که وظیفه smooth کردن تصویر را داشتند همچین وظیفه‌ای ندارد اما به دلیل وجود نویز بالا در عکس جزئیات زیادی در تصویر قرار گرفته. اگر در تصویر اولیه دقت شود ممکن است میزان نویز نمک فلفل موجود در تصویر به درستی مشاهده شود و از آنجا که با medianBlur هم جزئیات تصویر از بین می‌رود باید به دنبال یک راه جایگزین برای این موضوع بود.

کرنل *Sharpening*

کرنل Bilaterial:

این کرنل که توضیحات آن را در بالاتر فرستادم وظیفه بلور کردن و از بین بردن نویز را همزمان انجام می‌دهد و همانطو که در پایین مشاهده می‌شود تصویر خروجی آن در همچین موردی بهتر خواهد بود. همانطور که در عکس پایین مشاهده می‌شود علاوه بر اینکه جزئیات تصویر از بین نرفته از میزان نویز عکس هم کاشته شده و این تصویر در نهایت برای لبه‌یابی و تشخیص اشکال و تحلیل کردن به طور کلی مناسب‌تر است تا نسبت به باقی عکس‌هایی که در بالای صفحه به آن‌ها اشاره شد.

کرنل *Bilaterial*

لبه‌یابی:

در عکس پایین تصویر اصلی قابل مشاهده است.

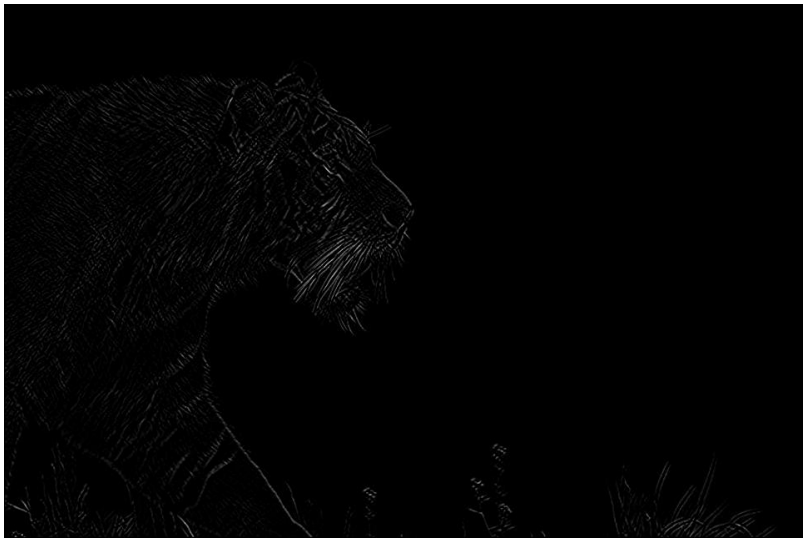
خروجی Sobel_x:

همانطور که در عکس بالا مشخص است لبه‌یابی عمودی بیشتر در این عکس مشاهده شدند و خروجی داده شده‌اند. چرا که گرادیان در جهت افقی بر تصویر اعمال می‌شود و تفاوت رنگ عمودی را به عنوان لبه در نظر می‌گیرد.

خروجی Sobel_y:

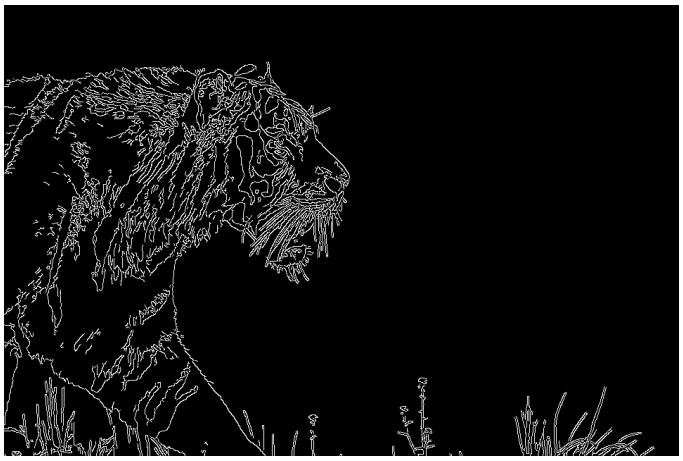
خروجی این تصویر بیشتر لبه‌های افقی را پیدا کرده و دلیلش هم جهت گرادیان عمودی بوده که تغییرات را در جهت افقی مشاهده می‌کند و به عنوان لبه در نظر می‌گیرد.

خروجی Sobel_xy:



این خروجی Sobel در هر دو جهت x و y است و مشاهده می‌شود که بعضی خطوط به خوبی مشاهده نشده و لبه‌یابی نشده‌اند.

خروجی Canny:



همانطور که برای Canny مشاهده می‌شود تقریبا تمام لبه‌های تصویر مشاهده شده و می‌توان مشاهده کرد که آستانه‌گذاری به خوبی رعایت شده و تصویر خروجی مناسبی دارد.