به نام خدا

درس مبانی بینایی کامپیوتر

تمرین سری هفتم

مدرس درس:
جناب آقای دکتر محمدی

تهیه شده توسط:
الناز رضایی ۹۸۴۱۱۳۸۷

تاریخ ارسال: ۱۴۰۱/۰۸/۲۶

سوال ۱:

از دیوارهای یک ساختمان چند تصویر گرفته شده است. میخواهیم نمای کلی ساختمان را در یک تصویر مشاهده کنیم. با استفاده از توابع OpenCV این تصاویر را به یک دیگر متصل کنید. برای این منظور نوتبوک Q1 را تکمیل کنید و تصویر نمای کلی ساختمان را به دست آورید. (۲۵ نمره)

پاسخ ۱:

در ابتدا، ۷ تصویر داده شده را خوانده و سپس با استفاده از تابع subplot، آنها را در یک سطر نمایش می‌دهیم. دراینجا، چون سایز تصاویر متفاوت بود، وقتی آنها را نمایش می‌دادیم بزرگ و کوچک دیده می‌شدند. بنابراین با استفاده از تابع resize،سایز آنها را تغییر می‌دهیم تا اندازه‌اشان با هم برابر شود. کد نوشته شده برای این بخش در تصویر زیر آورده شده است.

```
# read input victoria images and show them in a row together
path = r'E:\University\Term7\FCV\Homeworks\HW7\Images'
imageName = ['victoria1.png', 'victoria2.png', 'victoria3.png', 'victoria4.png', 'victoria5.png', 'victoria6.png', 'victoria7.png']
images = [cv2.imread(os.path.join(path, imageName[i])) for i in range(7)]
images = [cv2.resize(images[i], (1000, 1000)) for i in range(7)]
fig, axs = plt.subplots(nrows=1, ncols=7, figsize=(20, 20), sharex=True, sharey=True)
for ax in range(7):
    axs[ax].set_axis_off()
    axs[ax].imshow(cv2.cvtColor(images[ax], cv2.COLOR_BGR2RGB))
    axs[ax].set_title('victoria' + str(ax + 1))
✓ 1.5s
```

نتیجه مربوط به این بخش، به شرح زیر می‌باشد.

در بخش دوم این سوال، از ما خواسته شده تا image stitcher را مقداردهی اولیه کنیم و تصاویر ورودی را اصطلاحا موزاییک کنیم. برای این کار، ابتدا از تابع Stitcher.create موجود در کتابخانه OpenCV استفاده می‌کنیم. این تابع، یک Stitcher پیکربندی شده در یکی از حالت‌های دوخت ایجاد می‌کند و یک نمونه کلاس Stitcher را برمی‌گرداند. سپس با استقاده از تابع stitch، سعی می کنیم تصاویر داده شده را بخیه بزنیم. ورودی این تابع، آرایه‌ای از تصاویر ورودی داده شده برای موزاییک کردن، و خروجی آن یک status code برای فهمیدن اینکه آیا موزاییک کردن تصویر انجام

شده است یا خیر یا با چه اروری برخورد کرده‌ایم، و تصویر موزاییک شده می‌باشد. کد این بخش نیز در تصویر زیر نمایان است.

```
# initialize OpenCV's image sticher object and then perform the image stitching on input images
sticher = cv2.Stitcher.create()
(ret, result) = sticher.stitch(images)
```
✓ 0.6s

سپس با استفاده از قطعه کد زیر، تصویر نهایی را نمایش می‌دهیم.

```
# show victoria panorama
plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```
✓ 0.1s

تصویر موزاییک شده حاصل از تصاویر ورودی، به شرح زیر است.

سوال ۲:

در این سوال می‌خواهیم فرمول مربوط به تخمین زاویه چرخش میان دو تصویر را محاسبه کنیم. قبلا در صفحه ۲۸ جلسه ۱۲ تخمین مختصات انتقالی بین دو تصویر با استفاده از رویکرد MSE محاسبه شده بود. در این سوال زاویه θ را با استفاده از این رویکرد به دست آورید. (۲۵ نمره)

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

پاسخ ۲:

برای پیدا کردن θ، نیاز داریم تا رابطه را به صورت خطی بنویسیم. بنابراین x_2 و y_2 به شکل زیر در می‌آیند.

$$x_2 = x_1 cos(\theta) - y_1 sin(\theta)$$
$$y_2 = x_1 sin(\theta) + y_1 cos(\theta)$$

حال تابع هزینه را برای این تبدیل محاسبه می‌کنیم.

$$cost(x) = \sum (x_2^n - x_1^n cos(\theta) + y_1^n sin(\theta))^2 + (y_2^n - x_1^n sin(\theta) - y_1^n cos(\theta))^2$$

سپس از cost نسبت به θ مشتق گرفته و آن را برابر با صفر قرار می‌دهیم تا مقدار بهینه آن را به دست بیاوریم.

$$\frac{d}{d\theta} cost = 2 \sum (x_1^n sin(\theta) + y_1^n cos(\theta))(x_2^n - x_1^n cos(\theta) + y_1^n sin(\theta)) +$$
$$(y_1^n sin(\theta) - x_1^n cos(\theta))(y_2^n - x_1^n sin(\theta) - y_1^n cos(\theta)) = 0$$

$$\frac{d}{d\theta} cost = \sum x_1^n x_2^n sin(\theta) - (x_1^n)^2 sin(\theta)cos(\theta) + x_1^n y_1^n sin^2(\theta) + y_1^n x_2^n cos(\theta)$$
$$- x_1^n y_1^n cos^2(\theta) + (y_1^n)^2 sin(\theta)cos(\theta) + y_1^n y_2^n sin(\theta) - x_1^n y_1^n sin^2(\theta)$$
$$- (y_1^n)^2 sin(\theta)cos(\theta) - y_2^n x_1^n cos(\theta) + (x_1^n)^2 sin(\theta)cos(\theta)$$
$$+ x_1^n y_1^n cos^2(\theta) = 0$$

همانطور که در عبارت بالا نیز مشخص است، عبارات رنگی دوبه‌دو با هم خط می‌خوردند. بنابراین داریم:

$$\sum x_1^n x_2^n sin(\theta) + y_1^n y_2^n sin(\theta) + y_1^n x_2^n cos(\theta) - y_2^n x_1^n cos(\theta) = 0$$

طبق خواص سیگما داریم:

$$sin(\theta) \sum x_1^n x_2^n + y_1^n y_2 = cos(\theta) \sum y_2^n x_1^n - y_1^n x_2^n$$

دو طرف را بر $cos(\theta)$ تقسیم می‌کنیم:

$$tan(\theta) \sum x_1^n x_2^n + y_1^n y_2^n = \sum y_2^n x_1^n - y_1^n x_2^n$$

حال برای به دست آوردن θ، کافیست تا سیگماها را به یک طرف برده و arctan بگیریم:

$$\theta = \arctan\left(\frac{\sum y_2^n x_1^n - y_1^n x_2^n}{\sum x_1^n x_2^n + y_1^n y_2^n}\right)$$

سوال ۳:

هدف از این تمرین طراحی الگوری تمی با کارکردی شبیه به برنامه CamScanner است. بدین منظور می‌بایست نوتبوک پیوست (Q3) را تکمیل کنید. برای این کار به بسیاری از نکات کلیدی که تاکنون در این درس آموخته‌اید نیاز خواهید داشت.
الگوریتم هدف ما از چند مرحله اساسی تشکیل میشود؛ مراحلی که با تبدیل‌هایی روی تصویر آغاز شده و با تغییر شکل آن خاتمه می‌یابند. این مراحل عبارتند از:

- الف) نگاشت سیاه‌ـسفید (Grayscale): به منظور شناسایی برگه درون کاغذ نیازی به دانستن مقدار دقیق رنگ‌های پیکسل‌ها نداریم و این کار تنها پیاده‌سازی ما را پیچیده‌تر می‌سازد. به همین منظور در ابتدای کار تصویر را نگاشت میکنیم. برای این کار از روش دلخواهتان استفاده کنید.

- ب) محوکردن (Blurring) تصویر: با این کار مولفه‌های فرکانس بالای تصویر حذف می‌شوند که مراحل آتی کار، به ویژه تشخیص لبه‌ها را ساده‌تر می‌سازد.

- پ) تشخیص لبه‌ها: این مرحله نخستین گام در شناسایی چارچوب هدف در تصویر است؛ چارچوبی که محدوده سند و کاغذ هدف را مشخص می‌کند.

- ت) تشخیص رئوس (Vertices) برگه: یک راه انجام این کار استفاده از رویکردهای تشخیص خط برای شناسایی اضلاع چهارضلعی حاضر در تصویر است. راهکار دیگری برای انجام این کار شناسایی Contourهای حاضر در تصویر است. با این مفهوم تاکنون تا حدودی در درس آشنا شده‌اید اما اسم آن را صریحاً نشنیده‌اید. شما می‌توانید برای شناسایی بزرگترین Contour حاضر در تصویر (که در شرایط تصویر برداری ایده‌آل می‌بایست همان برگه هدف باشد) مطابق راهنمایی نوتبوک از توابع مخصوص شناسایی Contourها استفاده کنید.

- ث) نگاشت دورنما (Perspective) و برش: بعد از تشخیص ناحیه محصور متناظر برگه حاضر در تصویر می‌بایست چارچوب متوازی‌الاضلاع آن را شناسایی کرده و با نگاشت دورنمای مناسب تصویر را به نحوی تغییر شکل دهیم که برگه در محدوده معین و مطلوب ما قرار گیرد تا بتوانیم آن را برش دهیم.

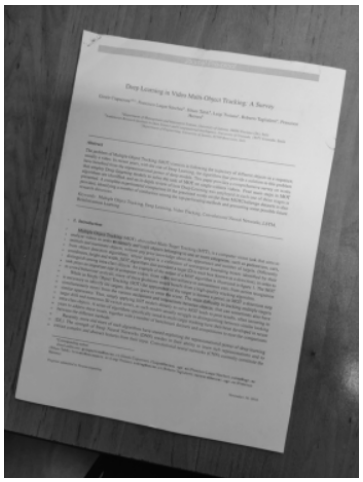
- ج) بهبود تصویر: پیش از برش تصویر اصلی می‌توانیم با کارهایی مانند افزایش غلظت (Saturation) رنگ‌ها آن را به یک اسکن حقیقی شبیه سازیم. در این‌باره می‌توانید درباره تبدیل Magic Color جست‌وجو کنید.

متناظر هر یک از این مراحل تابعی در نوتبوک پیوست تعبیه شده است. برای پیاده‌سازی هر یک از مراحل شما مجاز به استفاده از توابع کتابخانه‌های OpenCV و NumPy هستید. برای هر مرحله راهنمایی و توابعی پیشنهاد شده است اما شما می‌توانید به تشخیص خود توابع دیگری را جایگزین کنید که به نظر شما بهتر عمل می‌کنند. اگر بتوانید با تصاویر نمونه‌ای، ضعف توابع پیشنهادی و عملکرد بهتر رویکرد اتخاذ شده را اثبات کنید نمره تشویقی خواهید داشت. (۶۰ نمره)

پاسخ ۳:

- ۳ـالف) برای این بخش، از تابع cvtColor استفاده کردیم و چون در تابع imshow نوشته شده، تصویر را به حالت RGB تبدیل می‌کند، در داخل این تابع، از COLOR_RGB2GRAY استفاده می‌کنیم تا تصویر RGB را به GRAY تبدیل کنیم. کد نوشته شده این بخش به شرح زیر می‌باشد.

```
def to_grayscale(im):
    im = cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)
    return im
```

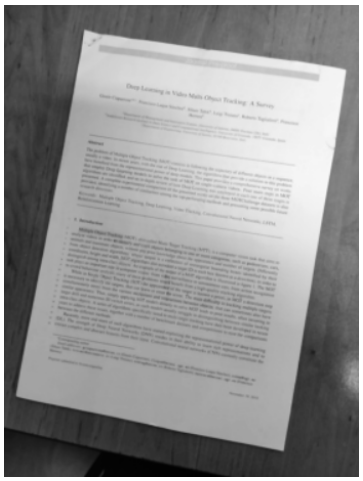


نتیجه این بخش، به شکل روبرو در می‌آید:

- ۳ـ ب) برای Blur کردن تصویر نیز از تابع bilateralFilter استفاده کردیم. ورودی‌های این تابع به ترتیب، تصویر ورودی، یک عدد از نوع int برای تعیین سایز kernel، عددی برای تعیین همسایگی برای فیلتر کردن در فضای رنگی و همچنین عددی برای تعیین همسایگی در فضای مختصات می‌باشد. معمولا سایز kernel ۵ پیشنهاد می‌شود؛ چرا که اگر این مقدار بیشتر از ۵ باشد، عملیات ما کند می‌شود. برای دو مقدار دیگر نیز، مقدار پیشنهادی بین ۱۰ تا ۱۵۰ می‌باشد؛ اگر کوچک‌تر از ۱۰ انتخاب کنیم، تاثیر زیادی ندارد و اگر بزرگ‌تر از ۱۵۰ در نظر بگیریم، تاثیر خیلی زیادی روی تصویر می‌گذارد و اصطلاحا تصویر را کارتونی می‌کند. کد مربوط به این بخش:

```
def blur(im):
    im = cv2.bilateralFilter(im, 5, 50, 50)
    return im
```

نتیجه این بخش، به شکل روبرو در می‌آید:



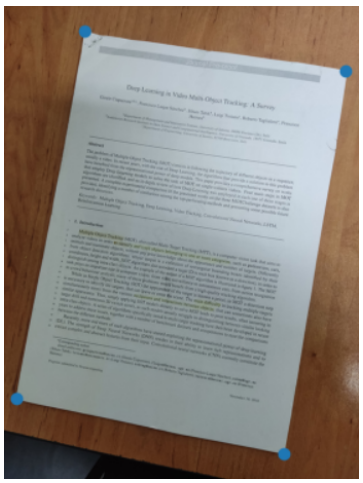
- ۳ـ پ) برای این بخش نیز از لبه‌یاب canny استفاده کردیم که ورودی‌هایش تصویر ورودی، حد آستانه اول و حد آستانه دوم می‌باشد. کد مربوط به این بخش:

```
def to_edges(im):
    im = cv2.Canny(im,20,50)
    return im
```

نتیجه این بخش، به شکل روبرو در می‌آید:

- ت) در ابتدا، با استفاده از دستور findContours، تمامی contourهای تصویر را پیدا می‌کنیم. ورودی‌های این تابع، تصویر ورودی، حالت بازیابی contour و روش تقریب contour می‌باشد. برای روش تقریب contour، دو حالت متداول داریم که یکی همه نقاط موجود را به ما می‌دهد؛ یعنی به عبارتی اضلاع را می‌دهد (CHAIN_APPROX_NONE) و دیگری فقط نقاط رئوس را می‌دهد (CHAIN_APPROX_SIMPLE). در اینجا چون ما مختصات رئوس را می‌خواهیم، از گزینه دوم استفاده می‌کنیم. در ادامه برای پیدا کردن بزرگترین contour که همان صفحه ما است، مقادیر area را با استفاده از تابع contourArea به ازای contourهای مختلف محاسبه می‌کنیم و بزرگ‌ترین آن را در نظر می‌گیریم. ورودی تابع contourArea نیز فقط همان contour است و خروجی‌اش، مساحت آن ناحیه می‌باشد. سپس با استفاده از تابع approxPolyDP() نقاط رئوس را با دقت مشخص تخمین می‌زنیم. ورودی‌های این تابع نیز شامل یک آرایه دو بعدی از نقاط، یک پارامتر برای مشخص کردن دقت و یک boolean که اگر true باشد، منحنی تقریبی بسته است (راس اول و آخر آن به هم متصل است). در غیر این صورت بسته نیست. در اینجا برای مشخص کردن دقت، از ضریبی از تابع arcLength استفاده کردیم. این تابع طول منحنی یا محیط کانتور بسته را محاسبه می کند. همچنین ورودی‌های آن نیز آرایه‌ای دو بعدی از نقاط و مشابه تابع قبل، یک boolean دارد که در صورت true بودن آن، یعنی منحنی بسته شده و اگر false باشد، بسته نشده است. با استفاده از خروجی این تابع، نقاط راس را پیدا کرده و در یک آرایه می‌ریزیم. در ادامه کد مربوط به این بخش آورده شده است.

```
def find_vertices(im):
    cnts, h = cv2.findContours(im, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)
    max = -1
    for i in range(len(cnts)):
        area = cv2.contourArea(cnts[i])
        if area > max:
            cnt = cnts[i]
            max = area
    cnt = cv2.approxPolyDP(cnt,0.01*cv2.arcLength(cnt,True),True)
    coordinates = [[cnt[0, 0, 0], cnt[0, 0, 1]],
    [cnt[1, 0, 0], cnt[1, 0, 1]], [cnt[2, 0, 0], cnt[2, 0, 1]],
    [cnt[3, 0, 0], cnt[3, 0, 1]]]
    return coordinates
```

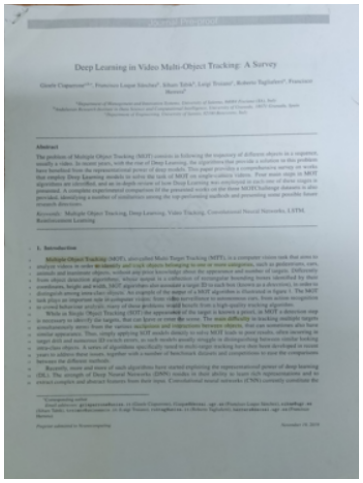


نتیجه این بخش، به شکل روبرو در می‌آید:

- ث) در ابتدا، ابعاد تصویر اصلی را با استفاده از دستور shape در می‌آوریم. سپس نقاط target که نقاط تعیین شده در بخش قبل باید بر این‌ها منطبق شوند را مشخص می‌کنیم و با استفاده از تابع getPerspectiveTransform، تابع تبدیل این نقاط را به دست می‌آوریم.

همچنین، نقاط مبدا و مقصد را به عنوان ورودی به این تابع می‌دهیم. سپس با استفاده از تابع warpPerspective، این نقاط را انتقال می‌دهیم. این تابع، تصویر اولیه، تابع تبدیل و طول و عرضی که تصویر انتقال داده شده را در آن می‌خواهیم، به عنوان ورودی دریافت می‌کند. مراحل توضیح داده شده در این بخش در قطعه کد زیر موجود است.

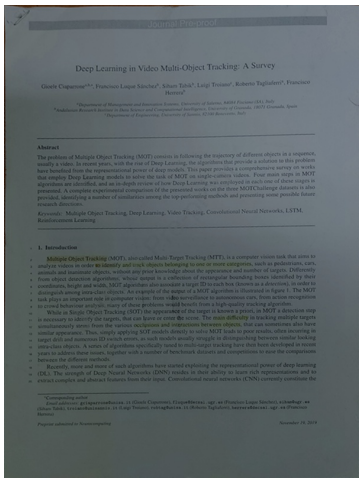
```
def crop_out(im, vertices):
    height, width = grayscale.shape
    target = [[0,0],[width,0],[0,height],[width,height]]
    vertices = reorder(vertices)
    target = reorder(target)
    transform = cv2.getPerspectiveTransform(vertices, target)
# get the top or bird eye view effect
    return cv2.warpPerspective(im, transform, (width, height))
```


Deep Learning in Video Multi-Object Tracking: A Survey

نتیجه این بخش، به شکل روبرو در می‌آید:

- ج) در این بخش، ابتدا تصویر خود را به HSV تبدیل می‌کنیم. برای افزایش saturation تصویر خود، یک ضریب که در اینجا ۱.۴ درنظر گرفتیم را در تصویر HSV خود ضرب می‌کنیم. همچنین برای کاهش روشنایی تصویر، یک ضریب کوچک‌تر از ۱ را در خروجی مرحله قبل ضرب می‌کنیم. در انتها تصویر خود را دوباره به حالت BGR برده و آن را به عنوان خروجی تابع return می‌کنیم. کد مربوط به این بخش در ادامه موجود است.

```
def enhance(im):
    hsv = cv2.cvtColor(im,cv2.COLOR_BGR2HSV)
    hsv[...,1] = hsv[...,1] * 1.4
    hsv[...,2] = hsv[...,2] * 0.6
    im = cv2.cvtColor(hsv,cv2.COLOR_HSV2BGR)
    return im
```



Deep Learning in Video Multi-Object Tracking: A Survey

نتیجه این بخش، به شکل روبرو در می‌آید:

نکته: برای حل این بخش از لینک زیر استفاده شده است.

https://answers.opencv.org/question/193336/how-to-make-an-image-more-vibrant-in-colour-using-opencv/