به نام خدا

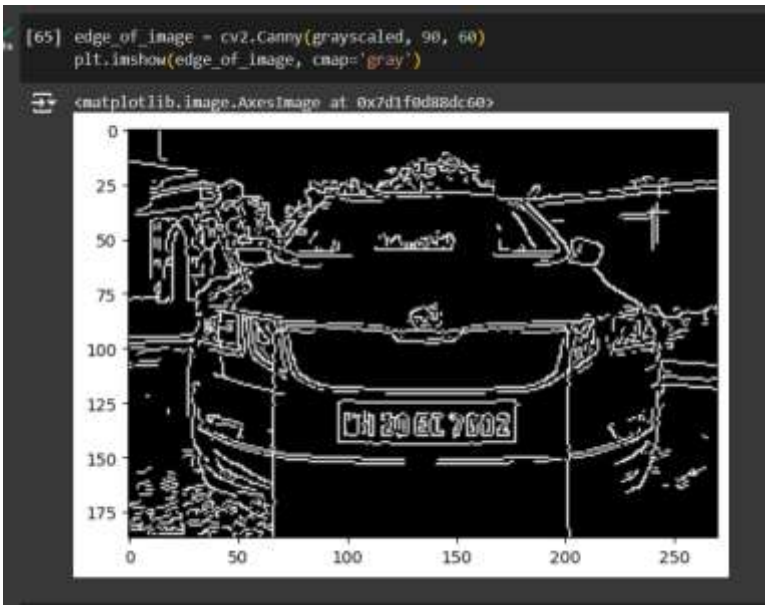
تمرین سری یکم مبانی بینایی کامپیوتر

معصومه پاسبانی 99243022

سوال اول)

لینک سوال اول

ابتدا تصویر را grayscale میکنیم تا لبه یاب اعمال شود. سپس با اعمال لب یاب کنی، مستطیل پلاک را شناسایی کرده.

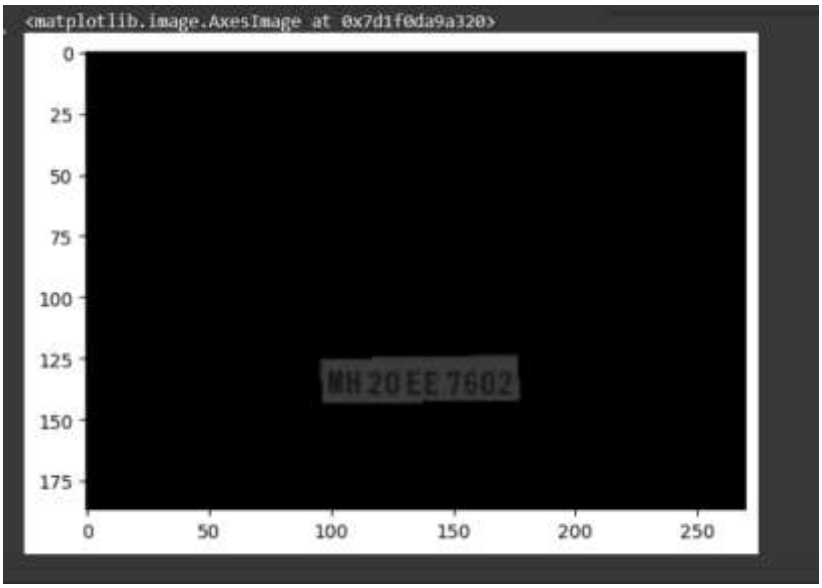


در اقدام بعدی، اشکال و خطوط تصویر را به کمک نقاط مهم آن از روی لبه ها پیدا کرده و اقدام به یافتن یک چهار ضلعی میکنیم.

روی کل تصویر یک تصویر سیاه انداخته و بخشس که سامل پلاک است را سفید کرده و با تصویر اصلی and میکنیم تا متن پلاک مشخص شود.

```
LH = cv2.findContours(edge_of_image.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
lines = imutils.grab_contours(LH)
lines2 = sorted(lines, key=cv2.contourArea, reverse=True)[:3]

for i in lines2:
    estimated = cv2.approxPolyDP(i, 5, True)
    if len(estimated) == 4:
        points = estimated
        break

black_image = np.zeros(grayscaled.shape, np.uint8)

pelak = cv2.drawContours(black_image, [points], 0, 255, -1)

pelak = cv2.bitwise_and(image_bgr, image_bgr, mask=black_image)

plt.imshow(cv2.cvtColor(pelak, cv2.COLOR_BGR2RGB))
```

```
<matplotlib.image.AxesImage at 0x7d1f0da9a320>
```



سپس دو نقطه اصلی مستطیل پلاک را پیدا کرده و آن مستطیل را trim میکنیم و برای وضوح بهتر پلاک، روشنایی تصویر را بالا میبریم.

```
(x, y) = np.where(black_image == 255)

x_top_left = np.min(x)
y_top_left = np.min(y)
x_down_right = np.max(x)
y_down_right = np.max(y)

trim = pelak[x_top_left:x_down_right, y_top_left:y_down_right]
```

```
[68] brightness_factor = 1.5
     brightened_image = np.clip(trim * brightness_factor, 0, 255).astype(np.uint8)
```

و در آخر با تابع reader متن پلاک را به انگلیسی میخوانیم.

```
[70] reader = easyocr.Reader(['en'])
     result = reader.readtext(brightened_image)

     print("OCR Results:", result)
     print('\n')

     if result:
         print("Detected Text:", result[0][-2])
         print('\n')

     plt.imshow(cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB))
```

و خروجی به صورت زیر خواهد بود:

```
WARNING:easyocr.easyocr:Neither CUDA nor MPS are available - defaulting to CPU. Note: This modu
OCR Results: [([[1, 5], [81, 5], [81, 19], [1, 19]], 'MH 20 EE 7602', 0.5323001630301985)]

Detected Text: MH 20 EE 7602

<matplotlib.image.AxesImage at 0x7d1f0ca95270>
```

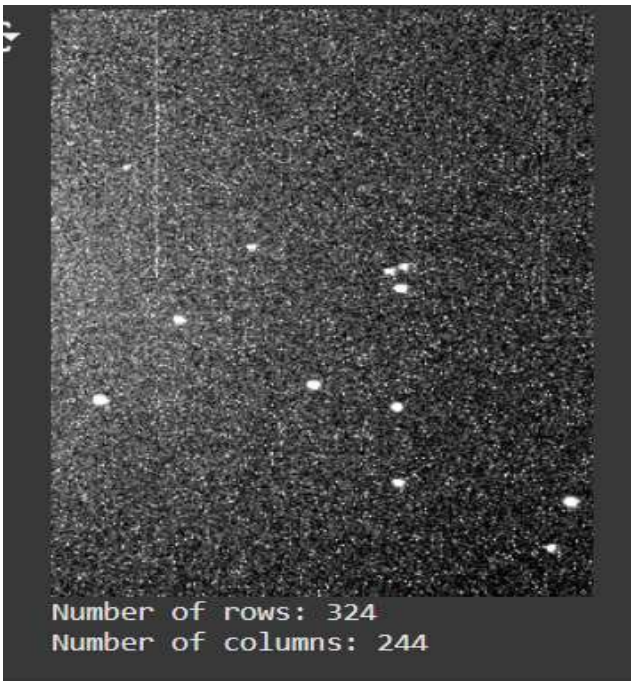
سوال دوم)

[لینک سوال دوم]

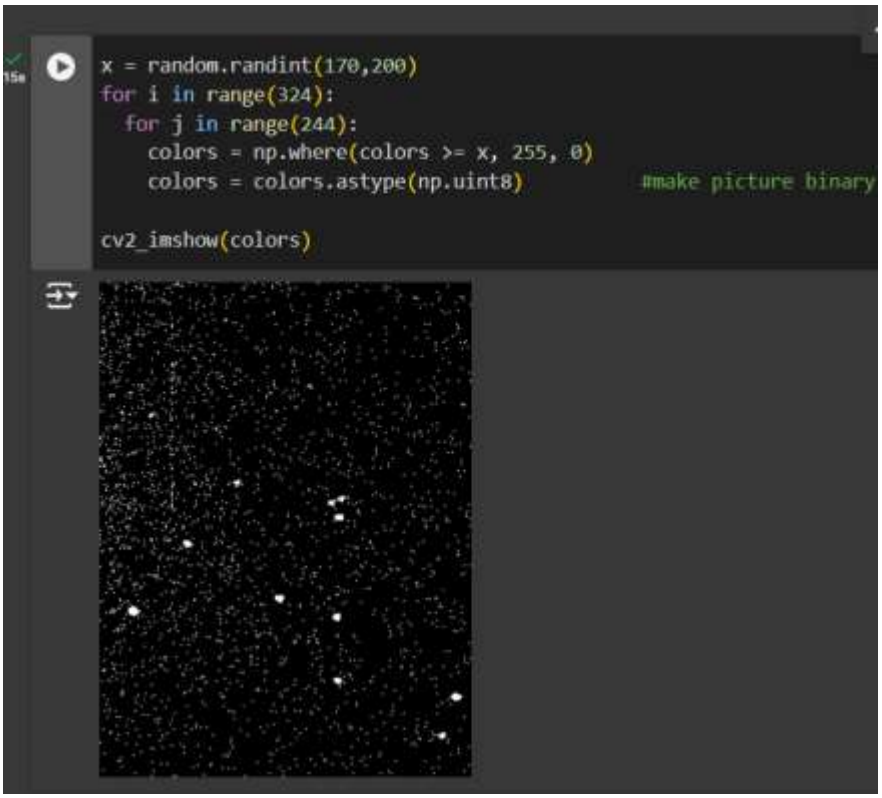
ابتدا باید از فایل ورودی تصویر ساخته شود.

با استفاده از اسپیس آرایه های 2بعدی در یک آرایه قرار میگیرند و تبدیل به یک آرایه 3بعدی میشود. یعنی هر سطر از تصویر ما یک عنصر آرایه است.

سپس این string ها را به کاراکتر و سپس به integer تبدیل میکنیم.

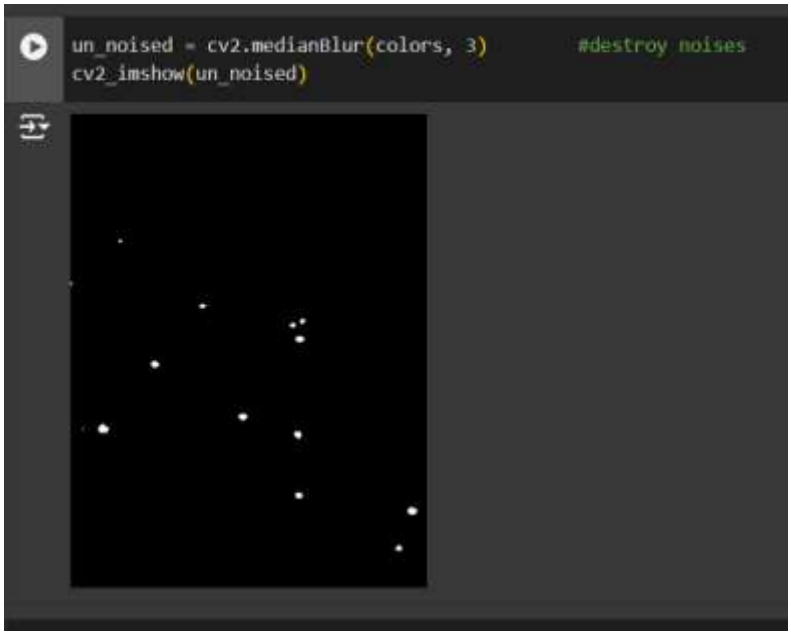


سپس در کل تصویر و تمام پیکسل ها پیمایش کرده و پیکسل ها را یا فید و یا سیاه کرده و تصویر را به حالت باینری می بریم. مرز رنگ هم عددی رندوم بین 170 تا 200 فرض میکنیم.



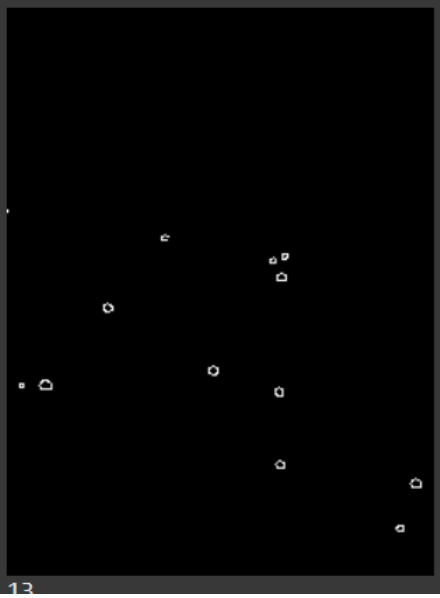
سپس با استفاده از فیلتر به کمک یک کرنل 3 در 3 اقدام به حذف نویزها میکنیم. اگر کرنل بزرگ شود تعداد ستاره های کمتری باقی می ماند و فقط نقاط بزرگ باقی می مانند.

```
un_noised = cv2.medianBlur(colors, 3)        #destroy noises
cv2_imshow(un_noised)
```



به کمک تابع canny و findcounters از طریق لبه های ستاره ها آن ها را شناسایی کرده و تعداد ستاره ها را به دست می آوریم.

```
detect = cv2.Canny(un_noised, 150, 200)
contours, hierarchy = cv2.findContours(detect,cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
num_stars = len(contours)
cv2_imshow(detect)
print(num_stars)
```



13

سپس از آنجا که آرایه ای از مختصات نقاط لبه های ستاره ها را داریم، اقدام به گرفتن میانگین از آن ها کرده که به ما مختصات مرکز هر ستاره را می دهد.

```
[ ]  mean = np.empty((num_stars, 2))

     for i in range(num_stars):
       mean[i] = np.array(np.average(contours[i], axis=0), int)

     print(mean)
```

```
[[224. 296.]
 [233. 271.]
 [155. 260.]
 [155. 218.]
 [  8. 215.]
 [ 21. 214.]
 [117. 206.]
 [ 57. 170.]
 [156. 153.]
 [151. 143.]
 [158. 141.]
 [ 89. 130.]
 [  0. 115.]]
```

سوال سوم)

الف) فیلتر *non-local means* (NLM) یک روش پیشرفته برای کاهش نویز در تصاویر است. برخلاف روش‌های محلی که تنها به پیکسل‌های مجاور توجه می‌کنند، این فیلتر به همه پیکسل‌های تصویر توجه می‌کند و وزن هر پیکسل را بر اساس شباهت آن با پیکسل هدف محاسبه می‌کند. این فیلتر به دلیل عملکردش در کاهش نویز و حفظ جزئیات تصویر، به‌ویژه در تصاویر با بافت‌های پیچیده محبوبیت یافته است.

برای محاسبه مقدار پیکسل فیلتر شده u(p) در نقطه p، از فرمول زیر استفاده می‌شود:

For an image, Ω, with discrete pixels, a discrete algorithm is required.

$$u(p) = \frac{1}{C(p)} \sum_{q \in \Omega} v(q) f(p,q)$$

where, once again, $v(q)$ is the unfiltered value of the image at point q. $C(p)$ is given by:

$$C(p) = \sum_{q \in \Omega} f(p,q)$$

Then, for a Gaussian weighting function,

$$f(p,q) = e^{-\frac{|B(q)^2 - B(p)^2|}{h^2}}$$

where $B(p)$ is given by:

$$B(p) = \frac{1}{|R(p)|} \sum_{i \in R(p)} v(i)$$

در اینجا:

v(q) مقدار پیکسل اصلی در نقطه q است.

f(p,q) تابع وزنی است که شباهت بین نقاط p و q را اندازه‌گیری می‌کند.

C(p) یک فاکتور نرمالیزه است که اطمینان حاصل می‌کند که وزن‌ها جمع‌شان برابر با ۱ می‌شود.

Ω مجموعه تمام نقاط در تصویر است.

مراحل کار:

1. **محاسبه شباهت** :برای هر پیکسل، شباهت آن با سایر پیکسل‌ها محاسبه می‌شود. این شباهت معمولاً با استفاده از فاصله‌های بین پیکسل‌ها در یک ناحیه خاص محاسبه می‌شود.
2. **محاسبه وزن‌ها** :بر اساس شباهت‌ها، وزن هر پیکسل برای تأثیرگذاری بر پیکسل هدف تعیین می‌شود. پیکسل‌های مشابه وزن بیشتری خواهند داشت.
3. **تجمیع نهایی** :با استفاده از وزن‌ها، مقدار نهایی پیکسل فیلتر شده محاسبه می‌شود که شامل مجموع مقادیر پیکسل‌های اصلی است که با وزن‌هایشان ضرب شده‌اند.

این فرآیند باعث می‌شود که فیلتر توانایی بهبود کیفیت تصویر را داشته باشد و جزئیات و بافت‌ها را حفظ کند.

ب)

در averaging تنها به پیکسل‌های مجاور (معمولاً در یک ناحیه محلی) توجه می‌کند و میانگین آن‌ها را محاسبه می‌کند. این کار می‌تواند باعث محو شدن جزئیات در تصویر شود، زیرا همه پیکسل‌ها به یک اندازه در محاسبه تأثیر دارند. این فیلتر ممکن است جزئیات تصویر و بافت‌ها را محو کند، زیرا نویزهای محلی با اطلاعات واقعی مخلوط می‌شوند.Averaging سریع و ساده است، اما ممکن است در تصاویر با نویز بالا و بافت‌های پیچیده کارایی کمتری داشته باشد.

Non-local Meansبه همه پیکسل‌ها در تصویر توجه دارد و تأثیر هر پیکسل بر پیکسل هدف را بر اساس شباهت محاسبه می‌کند. این روش اطلاعات بیشتری از تصویر را برای تصمیم‌گیری درباره پیکسل‌ها استفاده می‌کند. این فیلتر جزئیات و بافت‌ها را بهتر حفظ می‌کند، زیرا تنها پیکسل‌های مشابه (با توجه به ویژگی‌هایشان) وزن بیشتری دارند و در محاسبه تأثیر می‌گذارند. Non-local Meansبه دلیل استفاده از اطلاعات سراسری تصویر، می‌تواند زمان‌برتر باشد، اما نتایج بهتری در کاهش نویز و حفظ کیفیت تصویر ارائه می‌دهد.

فیلتر non-local meansبه عنوان یک روش پیشرفته‌تر نسبت به averagingشناخته می‌شود که توانایی بهبود کیفیت تصویر را بدون از دست دادن جزئیات دارد.

ج)

پیاده‌سازی فیلتر Non-Local Means (NLM) بر روی GPU می‌تواند به سرعت و کارایی در پردازش تصویر کمک کند. استفاده از GPU به دلیل توانایی‌های موازی آن، می‌تواند زمان پردازش را به شدت کاهش دهد. GPU می‌تواند چندین محاسبه را به طور همزمان انجام دهد، که برای الگوریتم‌های مانند NLM که به بررسی و مقایسه‌ی پیکسل‌های متعدد نیاز دارد، بسیار مناسب است.

قبل از اجرای فیلترNLM ، تصویر ورودی باید به شکل مناسب برای پردازش روی GPU آماده شود. این شامل تبدیل تصویر به یک ماتریس داده‌ای مناسب و انتقال آن به حافظه GPU است. سپس برای هر پیکسل، فاصله‌های بین پیکسل‌های مجاور و غیرمجاور باید محاسبه شود. این محاسبات به صورت موازی بر روی GPU انجام می‌شود و وزن هر پیکسل بر اساس

فاصله آن از پیکسل هدف و تشابه با ناحیه‌ای که پیکسل هدف در آن قرار دارد، محاسبه می‌شود. این وزن‌ها به صورت یک ماتریس ذخیره می‌شوند. وزن‌ها و پیکسل‌ها باید به طور مناسب ترکیب شوند تا مقدار جدید هر پیکسل محاسبه شود. این مرحله شامل انجام محاسبات جمع و ضرب برای تعیین مقدار جدید هر پیکسل است.

برای بهینه‌سازی عملکرد، می‌توان از تکنیک‌های مختلفی مانند shared memory، multiscale techniques و ... استفاده کرد. پیاده‌سازی NLM بر روی CPU معمولاً کندتر است زیرا به دلیل عدم وجود پردازش موازی، زمان محاسبات افزایش می‌یابد. بررسی عملکرد بر روی GPU نسبت به CPU می‌تواند به صورت تجربی انجام شود تا کارایی را نشان دهد.

د) لینک سوال سوم

```
print("CUDA Available:", tf.config.list_physical_devices('GPU'))
```

ابتدا بررسی می کنیم که آیا GPU دارای قابلیت CUDA در محیط را دارد یا خیر. این موضوع برای استفاده از تسریع محاسبات با استفاده از GPU در TensorFlow مهم است

```
@tf.function(jit_compile=True)
def add_gaussian_noise(image, mean=0, sigma=25):
    noise = tf.random.normal(shape=tf.shape(image), mean=mean, stddev=sigma, dtype=tf.float32)
    noisy_image = tf.clip_by_value(image + noise, 0, 255)
    return tf.cast(noisy_image, tf.uint8)
```

.

سپس تابع افزودن نویز گوسی که نویز تصادفی را با میانگین و انحراف معیار مشخص تولید می‌کند، را پیاده میکینم. ین نویز به تصویر اضافه می‌شود و سپس مقادیر آن به گونه‌ای برش داده می‌شوند که در محدوده شدت پیکسل‌های معتبر (۰ تا ۲۵۵) باقی بمانند و در نهایت، تصویر نویزی به فرمت عدد صحیح ۸ بیتی بدون علامت تبدیل می‌شود.

```
def get_window(img, x, y, size):
    half_size = size // 2
    return img[y - half_size:y + half_size + 1, x - half_size:x + half_size + 1]
```

تابع get_window ، تابع کمکی است که یک پنجره مربعی از پیکسل‌ها را در اطراف یک مختصات مشخص (x, y) از تصویر استخراج می‌کند. اندازه پنجره با پارامتر sizeتعیین می‌شود که به الگوریتم اجازه می‌دهد همسایگی پیکسل‌ها را در طول پردازش ارزیابی کند.

```
def NL_means(img, h=8.5, f=4, t=11):
    N = 2 * f + 1
    S = 2 * t + 1
    sigma_h = h
    pad_img = np.pad(img, t + f)
    h, w = img.shape

    neigh_mat = np.zeros((h + S - 1, w + S - 1, N, N))

    for y in range(h + S - 1):
        for x in range(w + S - 1):
            neigh_mat[y, x] = np.squeeze(get_window(pad_img[:, :, np.newaxis], x + f, y + f, 2 * f + 1))

    output = np.zeros(img.shape)
    prog = tqdm(total=(h - 1) * (w - 1), position=0, leave=True)

    for Y in range(h):
        for X in range(w):
            x = X + t
            y = Y + t
            a = get_window(neigh_mat.reshape((h + S - 1, w + S - 1, N * N)), x, y, S)
            b = neigh_mat[y, x].flatten()
            c = a - b
            d = c * c
            e = np.sqrt(np.sum(d, axis=2))
            F = np.exp(-e / (sigma_h * sigma_h))
            Z = np.sum(F)

            im_part = np.squeeze(get_window(pad_img[:, :, None], x + f, y + f, S))
            NL = np.sum(F * im_part)
```

```
            im_part = np.squeeze(get_window(pad_img[:, :, None], x + f, y + f, S))
            NL = np.sum(F * im_part)
            output[Y, X] = NL / Z
            prog.update(1)

    return output
```

در nlm ابتدا یک آرایه ۴ بعدی برای ذخیره همسایگی پیکسل‌ها برای هر پیکسل در تصویر پد شده مقداردهی می‌شود. حلقه‌های تو در تو این ماتریس را با استخراج پنجره‌های پیکسل برای تمام موقعیت‌های ممکن در تصویر پد شده پر می‌کنند. سپس یک آرایه خروجی برای ذخیره مقادیر پیکسل‌های کاهش‌یافته نویز ایجاد می‌شود.

هر پیکسل پردازش می‌شود و همسایگی آن استخراج و وزن‌ها بر اساس شباهت محاسبه می‌شود. هسته گوسی برای محاسبه وزن‌ها برای لکه‌های مشابه بر اساس فاصله اقلیدسی اعمال می‌شود ، مجموع وزنی پیکسل‌ها محاسبه شده و نرمالیزه می‌شود تا مقدار نهایی کاهش‌یافته نویز برای هر پیکسل به دست آید. و در نهایت تابع تصویر کاهش‌یافته نویز را به عنوان خروجی برمی‌گرداند.

```
2]  def process_image_cpu(image):
        noisy_img = add_gaussian_noise(image)

        with tf.device('/CPU:0'):
            start_time_cpu = time.time()
            denoised_cpu = NL_means(noisy_img, h=9, f=4, t=12)
            cpu_time = time.time() - start_time_cpu

        return noisy_img, denoised_cpu, cpu_time
```

در تابع process_image_cpu لبتدا، به تصویر ورودی نویز گوسی با استفاده از تابع add_gaussian_noise اضافه می‌شود و تصویر نویزی در متغیر noisy_img ذخیره می‌شود. با استفاده از with tf.device('/CPU:0'):، تضمین

می‌شود که عملیات زیر بر روی CPU انجام شود. زمان شروع پردازش ذخیره می‌شود و پس از اجرای تابع NL_means که تصویر نویزی را کاهش می‌دهد، زمان سپری‌شده محاسبه می‌شود. در انتها، تصویر نویزی، تصویر denoise شده ، و زمان پردازش CPU به عنوان خروجی بازگشت داده می‌شود.

```
def display_results(original, noisy, denoised, title="Results"):
    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(original, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 3, 2)
    plt.title("Noisy Image")
    plt.imshow(noisy, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 3, 3)
    plt.title("Denoised Image")
    plt.imshow(denoised, cmap='gray')
    plt.axis('off')

    plt.suptitle(title)
    plt.show()
```

تابع نمایش نتایج که برای نمایش تصاویر اصلی، نویزی و denoise شده طراحی شده است.

```
def print_timings(cpu_time=None, gpu_time=None):
    if cpu_time is not None:
        print(f'CPU Time: {cpu_time:.4f} seconds')
    if gpu_time is not None:
        print(f'GPU Time: {gpu_time:.4f} seconds')
```

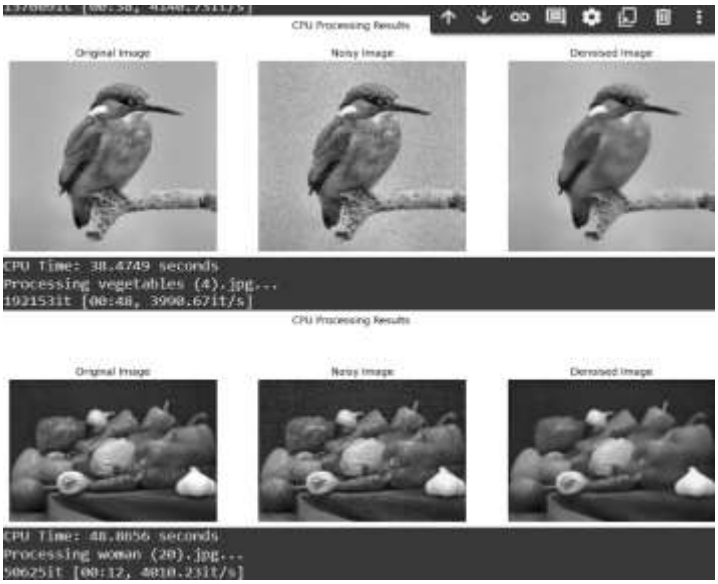
تابع print_timings برای چاپ زمان‌های پردازش CPU و GPU طراحی شده است. اگر زمان CPU ارائه شده باشد (غیر ازNone)، زمان پردازش CPU به فرمت مناسب چاپ می‌شود. به همین ترتیب، اگر زمان GPU ارائه شده باشد، زمان پردازش GPU نیز چاپ می‌شود.

```
uploaded_files = files.upload()
```

```
Choose Files  3 files
 • bird.jpg(image/jpeg) - 24533 bytes, last modified: 10/22/2024 - 100% done
 • vegetables.jpg(image/jpeg) - 46697 bytes, last modified: 10/22/2024 - 100% done
 • woman.jpg(image/jpeg) - 8006 bytes, last modified: 12/10/2023 - 100% done
Saving bird.jpg to bird (4).jpg
Saving vegetables.jpg to vegetables (4).jpg
Saving woman.jpg to woman (20).jpg
```

```
[ ]
for filename, file_data in uploaded_files.items():
    print(f'Processing {filename}...')

    img = Image.open(BytesIO(file_data)).convert('L')
    img = np.array(img, dtype=np.float32)

    # For CPU Processing
    noisy_cpu, denoised_cpu, cpu_time = process_image_cpu(img)
    display_results(img, noisy_cpu, denoised_cpu, title="CPU Processing Results")
    print_timings(cpu_time=cpu_time)
```

پس از upload تصاویر، تصویر باز شده و پس از gray شدن به یک آرایه NumPy با نوع داده float32 تبدیل می‌شود که برای پردازش‌های بعدی مناسب است. سپس تابع process_image_cpu برای پردازش تصویر ورودی و کاهش نویز آن با استفاده از CPU فراخوانی می‌شود و با استفاده از تابع displsay_results نتیجه شامل تصویر اصلی، تصویر نویزی و تصویر denoise شده نمایش داده میشوند و زمان پردازش cpu هم با فراخوانی تابع print_timings چاپ میشود.



```python
def process_image_gpu(image):
    image_float32 = tf.convert_to_tensor(image, dtype=tf.float32)

    noisy_img = add_gaussian_noise(image_float32)

    def wrapped_NL_means(noisy_img_np):
        return NL_means(noisy_img_np).astype(np.float32)

    with tf.device('/GPU:0'):
        start_time_gpu = time.time()
        denoised_gpu = tf.numpy_function(
            wrapped_NL_means, [noisy_img.numpy()], tf.float32
        )
        gpu_time = time.time() - start_time_gpu

    return noisy_img, denoised_gpu, gpu_time
```

تابع process_image_gpu برای پردازش تصویر ورودی با استفاده از GPU طراحی شده است. تصویر ورودی به یک تنسور TensorFlow با نوع داده float32 تبدیل می‌شود. سپس به تصویر تنسوری نویز گوسی اضافه می‌شود. تابع wrapped_NL_means به عنوان یک تابع کمکی تعریف می‌شود که NL_means را فراخوانی می‌کند و خروجی آن را به نوع float32 تبدیل می‌کند. این تابع به عنوان ورودی به tf.numpy_function داده می‌شود.

با استفاده از with tf.device('/GPU:0'), تضمین می‌شود که عملیات بعدی بر روی GPU انجام شود. زمان شروع پردازش ذخیره می‌شود و پس از فراخوانی تابع کاهش نویز، زمان سپری‌شده محاسبه می‌شود. در انتها، تصویر نویزی، تصویر کاهش‌یافته نویز و زمان پردازش GPU به عنوان خروجی بازگشت داده می‌شود.

```python
for filename, file_data in uploaded_files.items():
    print(f'Processing {filename}...')

    img = Image.open(BytesIO(file_data)).convert('L')
    img = np.array(img)
    noisy_gpu, denoised_gpu, gpu_time = process_image_gpu(img)

    display_results(img, noisy_gpu, denoised_gpu, title="GPU Processing Results")
    print_timings(gpu_time=gpu_time)
```

تصویر به آرایه numpy تبدیل شده و تابع process_image_gpu برای پردازش تصویر ورودی و کاهش نویز آن با استفاده از GPU فراخوانی می‌شود، مانند مورد قبل با تابع display_results تصاویر خروجی نمایش داده میشوند و با تابع print_timings زمان پردازش هر کدام از تصاویر نمایش داده میشود.

که مشاهده میکنیم زمان های پردازش در GPU کمتر از زمان های پردازش در CPU هستند.