Computer Vision
Dr. Mohammadi
Fall 2022
Hoorieh Sabzevari – Elnaz Rezaee
Final Project

طرح مسئله:

از ما خواسته شده تا یک الگوریتم بینایی کامپیوتر برای تشخیص ترک‌های موجود در یک سطح کاشی ارائه دهیم.

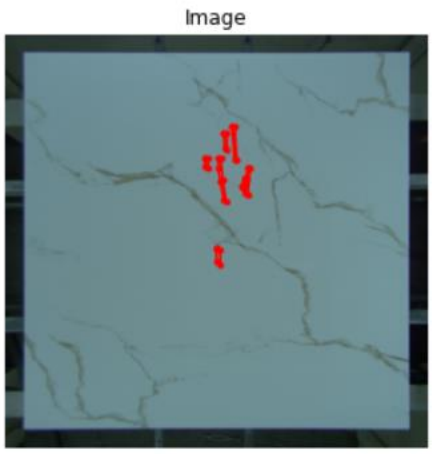
شرح کارها:

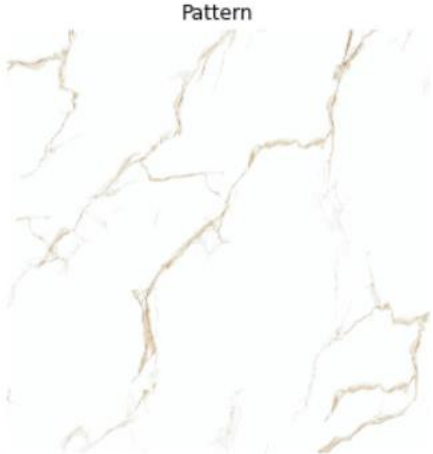
ابتدا تصاویر مربوط به کاشی‌ها و طرح‌ها را از لینک داده‌شده دانلود کرده و unzip می‌کنیم.

سپس با دستور زیر تمام تصاویر را در یک فایل جمع می‌کنیم تا از فایل‌های json جدا باشند.

```
filelist = [file for file in os.listdir('/content/Dataset') if file.endswith('.jpg') or file.endswith('.png')]
```

برای درک بهتر دیتاست، یک تصویر دلخواه را به همراه طرح آن و مختصات ترک‌ها نشان می‌دهیم.





همانطور که می‌بینیم علاوه بر تناژ رنگی متفاوت و بک‌گراند اضافی که عکس‌ها دارند، بلکه ممکن است دچار چرخش و تبدیل‌هایی از این قبیل نیز شده باشند. لذا ابتدا باید با انجام یک سری عملیات پیش‌پردازش، دو تصویر را به دو تصویر قابل‌مقایسه تبدیل کنیم.

۱) پیدا کردن نقاط کلیدی دو تصویر:

توابع کمکی زیر را تعریف می‌کنیم.

```
def detector(image1,image2):
    # creating ORB detector
    detect = cv2.ORB_create()
    # finding key points and descriptors of both images using detectAndCompute() function
    key_point1,descrip1 = detect.detectAndCompute(image1,None)
    key_point2,descrip2 = detect.detectAndCompute(image2,None)
    return (key_point1,descrip1,key_point2,descrip2)
```

این تابع برای پیدا کردن نقاط کلیدی دو تصویر تعریف شده است. بدین منظور از الگوریتم ORB استفاده کردیم. ORB تلفیقی از آشکارساز FAST و توصیف‌گر BRIEF با برخی ویژگی‌های اضافه شده برای بهبود عملکرد است. (لینک)

```
def BF_FeatureMatcher(des1,des2):
    brute_force = cv2.BFMatcher(cv2.NORM_HAMMING,crossCheck=True)
    no_of_matches = brute_force.match(des1,des2)

    # finding the humming distance of the matches and sorting them
    no_of_matches = sorted(no_of_matches,key=lambda x:x.distance)[:10]
    return no_of_matches
```

این تابع برای تطبیق دو تصویر تعریف شده است. بدین صورت که از تطبیق‌گر brute force برای تطبیق دو عکس استفاده می‌کنیم تا نقاط تطبیق‌داده‌شده را روی هم بیندازیم. برای جلوگیری از مشکل رم و حافظه تنها ۱۰ نقطه‌ی ابتدایی را ذخیره کردیم.

```
def matching(img, img_pattern):
  key_pt1, descrip1, key_pt2, descrip2 = detector(img,img_pattern)

  matches = BF_FeatureMatcher(descrip1,descrip2)
  # tot_feature_matches = len(matches)
  # print(f'Total Number of Features matches found are {tot_feature_matches}')

  # display_output(gray_pic1, key_pt1, gray_pic2,key_pt2, matches)
  list_kp1 = [[int(key_pt1[mat.queryIdx].pt[0]), int(key_pt1[mat.queryIdx].pt[1])] for mat in matches]
  list_kp2 = [[int(key_pt2[mat.trainIdx].pt[0]), int(key_pt2[mat.trainIdx].pt[1])] for mat in matches]

  kp1 = np.array(list_kp1, np.float32)
  kp2 = np.array(list_kp2, np.float32)
  h, status = cv2.findHomography(kp1, kp2, cv.RANSAC)
  cropped = cv2.warpPerspective(img, h, (img.shape[1], img.shape[0]))

  matched = match_histograms(img_pattern, cropped, multichannel=True)
  return cropped, h
```

این تابع دو تصویر کاشی و طرح را با استفاده از توابع قبلی بر هم منطبق می‌کند و سپس با استفاده از تابع findHomograohy() و الگوریتم ransac تبدیل بین نقاط مچ‌شده را پیدا می‌کند. با تابع warpPerspective() هم عکس را با تبدیل یافت‌شده تغییر می‌دهیم. (لینک)

خروجی توابع بالا برای یک عکس نمونه:

سپس تابع زیر را برای تبدیل مختصات ترک‌های کاشی با استفاده از تبدیل یافت‌شده تعریف کردیم که ماتریس تبدیل را در تک تک نقاط ضرب می‌کند. (لینک)

```
def transform_point(point, t):
  x = t[0][0] * point[0] + t[0][1] * point[1] + t[0][2]
  y = t[1][0] * point[0] + t[1][1] * point[1] + t[1][2]
  z = t[2][0] * point[0] + t[2][1] * point[1] + t[2][2]
  x_transformed = np.int32(x / z)
  y_transformed = np.int32(y / z)
  return np.array([[x_transformed, y_transformed]])
```

تابع زیر را برای تولید mask برای هر کاشی تعریف کردیم. بدین صورت که یک عکس به ابعاد همان عکس اصلی تولید می‌کند که تمام مقادیر آن سیاه است و فقط داخل چندضلعی ترک‌ها سفید است.

```
def generate_mask(image, json_file, t):
  mask = np.zeros((image.shape[0], image.shape[1]))
  for shape in json_file['shapes']:
    points = np.array(shape['points'], dtype = "float32")
    points = np.array([points])
    for point in points:
      temp = transform_point(point, t)
      cv2.fillPoly(mask, pts=[temp.astype(np.int32)], color=(255, 255, 255))
  return mask
```

در انتهای این بخش نیز با استفاده از توابع قبلی، لیستی از تصاویر اصلی تطبیق‌داده‌شده، طرح کاشی مربوط به هر تصویر و ماسک مربوطه ایجاد می‌کنیم. هر تصویر را نیز با توجه به سایز ورودی مدل resize می‌کنیم.

```
images = []
patterns = []
masks = []

for img_name in filelist:
  json_name = img_name[:-3] + "json"
  f = open('/content/Dataset/' + json_name, encoding="utf8")
  json_file = json.load(f)
  f.close()
  img = cv.imread('/content/Dataset/' + img_name)
  img_pattern = cv.imread('/content/Patterns/' + json_file['pattern'])

  img_pattern = cv.resize(img_pattern, img.shape[:2][::-1])
  matched_img, t = matching(img, img_pattern)

  mask = generate_mask(matched_img, json_file, t)

  matched_img = cv.resize(matched_img, (256, 256))
  img_pattern = cv.resize(img_pattern, (256, 256))
  mask = cv.resize(mask, (256, 256))

  images.append(matched_img)
  patterns.append(img_pattern)
  masks.append(mask)
```

برای درست کردن دیتاست مناسب، عکس اصلی را با طرح آن concatenate کرده و به صورت ۶ کاناله در می‌آوریم. سپس دو لیست x_train و y_train را خروجی می‌دهیم.

```
def dataset_generator(images, patterns, masks):
  x_train = []
  y_train = []
  for i, img in enumerate(images):
    im = img
    p = patterns[i]
    y = masks[i]
    x = np.concatenate((im, p), axis=2)
    x_train.append(x)
    y_train.append(y)
  return x_train, y_train
```

در انتها نیز لیست‌های خروجی تابع قبلی را به np.array() تبدیل کرده و ابعاد را یکی می‌کنیم.

```
x_train, y_train = dataset_generator(images, patterns, masks)
```

```
x_train = np.array(x_train)
y_train = np.array(y_train)
y_train = np.expand_dims(y_train, axis=-1)
```

```
y_train.shape
```

```
(301, 256, 256, 1)
```

```
x_train.shape
```

```
(301, 256, 256, 6)
```

۲) بخش ساخت مدل:

برای این مسئله ما از شبکه‌ی U-net استفاده کردیم. ابتدا تابع double_conv_block() تعریف شده که از آن در توابع بعدی استفاده می‌شود. این تابع دو بلاک کانولوشنی با ابعاد فیلترهای ۳ در ۳، padding = same ، تابع فعالساز relu و توزیع "he_normal" برای مقداردهی اولیه‌ی کرنل‌ها استفاده شده است. تعداد فیلترها نیز به عنوان ورودی تابع دریافت می‌شود. یک تابع downsample_block() برای نمونه‌برداری پایین یا استخراج ویژگی تعریف شده تا در توابع بعدی استفاده شود. در این تابع با استفاده از تابع قبلی، دو بلاک کانولوشنی ساخته و تنها یک لایه max-pooling و یک لایه dropout با احتمال ۰.۳ به آن اضافه می‌شود. در نهایت، یک تابع upsample_block() برای مسیر گسترش U-Net تعریف می‌شود که ابتدا یک لایه conv2DTraspose با تعداد فیلترهایی که از ورودی تابع گرفته می‌شود، ابعاد فیلتر ۳ در ۳، padding = same و گام ۲ اضافه می‌شود. این کار یک تکنیک نمونه‌برداری است که اندازه‌ی تصویر را افزایش می‌دهد. سپس خروجی این بلاک با بلاک conv_features که پارامتر ورودی است، concatenate می‌شود. به دلیل این که اطلاعات لایه‌های قبلی را با هم ترکیب کنیم تا بتوانیم پیش‌بینی دقیق‌تری داشته باشیم. سپس یک لایه dropout با احتمال ۰.۳ و دو بلاک

کانولوشنی نیز اضافه می‌شود. تمام این لایه‌ها نیز به صورت sequential اضافه می‌شوند و توابع modular هستند.

```
def build_unet_model():
    inputs = layers.Input(shape=(256, 256, 6))
    f1, p1 = downsample_block(inputs, 64)
    f2, p2 = downsample_block(p1, 128)
    f3, p3 = downsample_block(p2, 256)
    f4, p4 = downsample_block(p3, 512)
    bottleneck = double_conv_block(p4, 1024)
    u6 = upsample_block(bottleneck, f4, 512)
    u7 = upsample_block(u6, f3, 256)
    u8 = upsample_block(u7, f2, 128)
    u9 = upsample_block(u8, f1, 64)
    outputs = layers.Conv2D(1, 1, padding="valid", activation = "sigmoid")(u9)
    unet_model = tf.keras.Model(inputs, outputs, name="U-Net")
    return unet_model
```

مدل خود را ساخته و با بهینه‌ساز adam، تابع ضرر binary_crossentropy و متریک accuracy کامپایل می‌کنیم و با تابع train_test_split از کتابخانه‌ی skleran، ۱۵ درصد از دیتا را برای داده‌های اعتبارسنجی کنار می‌گذاریم.

```
unet_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
x_train, x_test, y_train, y_test = train_test_split(x_train, y_train, test_size=0.15)
```

حال مدل خود را با طی ۵ ایپاک و سایز دسته ۳۲ آموزش می‌دهیم.

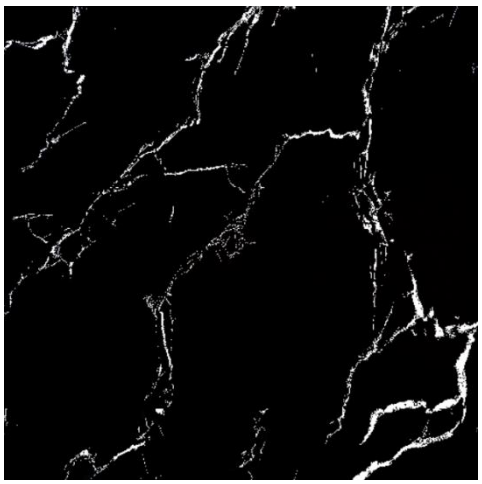
```
history = unet_model.fit(x_train, y_train,
                    batch_size=32,
                    validation_data=[x_test, y_test],
                    epochs=5)
```

```
Epoch 1/5
8/8 [==============================] - 77s 6s/step - loss: 37.9381 - accuracy: 0.8883 - val_loss: 5.9883 - val_accuracy: 0.9998
Epoch 2/5
8/8 [==============================] - 15s 2s/step - loss: 4.6211 - accuracy: 0.9741 - val_loss: 4.3019 - val_accuracy: 0.9998
Epoch 3/5
8/8 [==============================] - 15s 2s/step - loss: 3.4410 - accuracy: 0.9999 - val_loss: 2.6051 - val_accuracy: 0.9998
Epoch 4/5
8/8 [==============================] - 15s 2s/step - loss: 1.6622 - accuracy: 0.9999 - val_loss: 0.7945 - val_accuracy: 0.9998
Epoch 5/5
8/8 [==============================] - 15s 2s/step - loss: 0.6441 - accuracy: 0.9975 - val_loss: 0.2624 - val_accuracy: 0.9998
```

تلاش‌های ناموفق انجام شده:

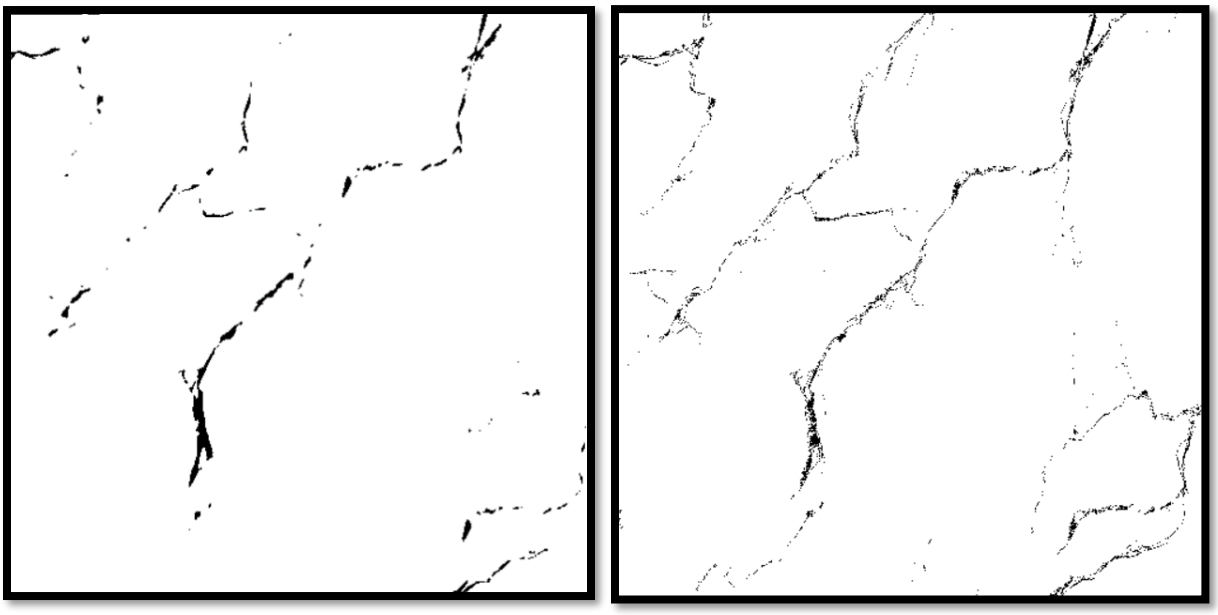
۱. **حذف طرح کاشی:** در ابتدای امر، ما تلاش کردیم تا با استفاده از histogram matching، رنگ کاشی و pattern را match کنیم و سپس با استفاده از subtract، پیکسل به پیکسل کاشی را از طرحش کم کنیم تا طرح حذف شده و تنها ترک‌های کاشی باقی بماند. اما چالشی که در اینجا با آن روبرو شدیم، وجود مقداری از تصویر background پس از استفاده از findhomography بود که آن را با crop از بین بردیم؛ اما در حین این کار، بخشی از اطلاعات تصویر نیز از بین رفت. در نتیجه تصویر pattern و کاشی پیکسل به پیکسل تطابق نداشت و عملیات تفریق ناموفقیت آمیز بود.

نتیجه اختلاف تصویر pattern و کاشی پس از هیستوگرام مچینگ:



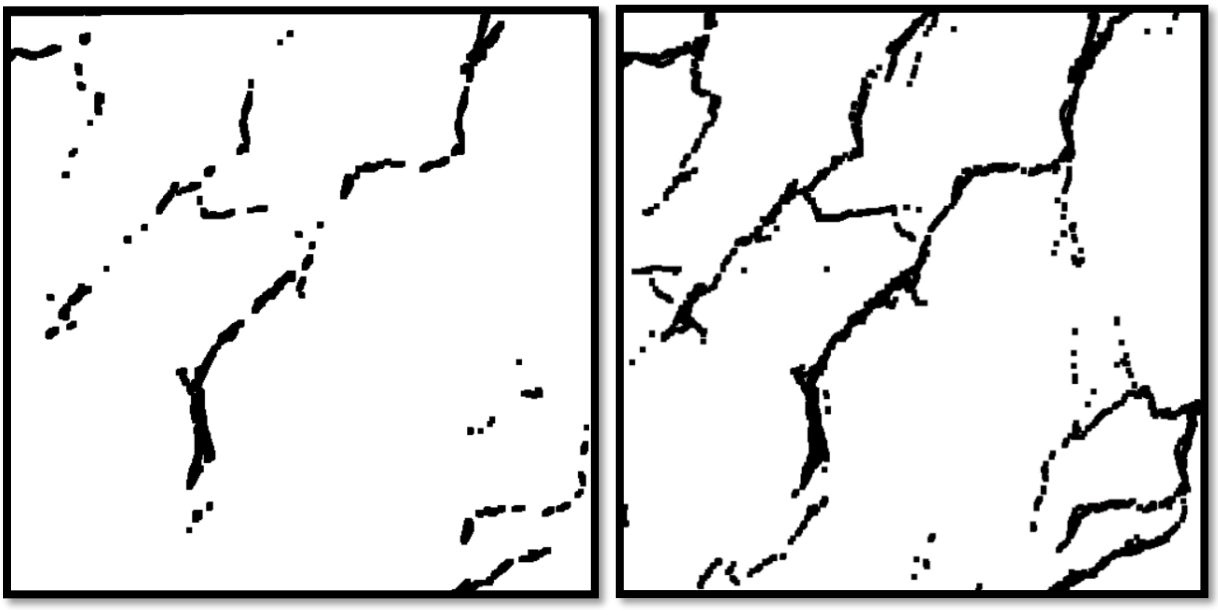
۲. **حذف طرح کاشی با استفاده از روش باینری کردن:** پس از انجام عملیات ناموفق مرحله قبل، تصمیم به باینری کردن تصویر و سپس به محاسبه اختلاف بین تصویر کاشی و pattern پرداختیم. در این مرحله نیز تصویر کاشی و طرح باینری شده گسستگی‌های ناهماهنگ داشتند.

نتیجه تصاویر باینری شده:



۳. **حذف طرح کاشی با استفاده از روش باینری کردن و سایش:** بنابراین ابتدا یک بار هر دو تصویر را erode کردیم تا بعد از آن اختلاف را محاسبه کنیم. اما این روش نیز جواب نداد و دو تصویر نتایج غیرهمسان داشتند.

نتیجه تصاویر erode شده pattern و کاشی:



۴. **Compile کردن مدل با استفاده از تابع ضرر focal loss**: با استفاده از این روش، به loss برابر با not a number رسیدیم و دقت هم به مقدار زیادی کاهش پیدا کرد.

۵. **استفاده از معیار f1-score برای دقت**: این معیار نیز دقت نزدیک ۰.۰۰۵ می‌داد و loss هم نزدیک به ۰.۹ بود. لذا برای دقت نیز از همان متریک accuracy استفاده کردیم.

نتیجه‌گیری:

با استفاده از شبکه‌ی U-net نتیجه گرفتیم که این شبکه برای این نوع مسئله کار نمی‌کند. زیرا خروجی شبکه باید پیش‌بینی کند که مقدار هر پیکسل سیاه(بک‌گراند) است یا سفید(ترک)؛ ولی در اینجا تمامی پیسکل‌ها به عنوان بک‌گراند تشخیص داده می‌شوند و چون صرفا دو رنگ داریم و یک نوع مسئله‌ی binary classification محسوب می‌شود، لذا بهتر بود از شبکه‌های دیگر استفاده می‌کردیم.

یک روش دیگر برای حل این مسئله، می‌توانست استفاده از شبکه‌ی Siamese باشد. بدین صورت که با روش پنجره‌ی لغزان، هر ناحیه از دو تصویر باهم مقایسه شده و مقدار شباهت سنجیده شود. سپس با استفاده از آستانه‌گذاری روی میزان شباهت هر دو ناحیه، نواحی‌ای که شباهت کمتری دارند را شناسایی کرده و برای پیدا کردن محل دقیق ترک‌ها، سایز پنجره را کاهش می‌دهیم تا به دقت خوبی برسیم.