دانشکده مهندسی برق

بسمه تعالی

[طبقه بندی تصاویر 7MAPs Refractive]

مهدی وحیدمقدم

۴۰۲۱۳۰۷۴

پروژه پایانی درس یادگیری ماشین

فهرست مطالب

فهرست شکل‌ها

چکیده

در این گزارش،ابتدا توضیحاتی درباره مسئله بیان می‌شود و توضیحاتی درباره‌ی اهمیت تشخیص سریع این بیماری و تاثیر استفاده از هوش مصنوعی در آن داده خواهد شد.سپس درباره‌ی آماده‌سازی تصاویر گفته خواهد شد که دلیل انجام این کار چیست و چه امتیازاتی برای طبقه‌بندی تصاویر دارد و همچنین چه روندی برای حذف نویز و قسمت‌های اضافی از تصاویر انجام شده است.در قسمت بعد هم درباره‌ی مدل‌های مختلف تست شده بر روی این دیتاست اعم از مدل‌های ساخته شده توسط ما،مدل‌های پیش‌آموزش‌دیده [1]و یا ترکیبی از این مدل‌ها و نتایج حاصل شده از آن‌ها صحبت خواهد شد.همچنین درباره‌ی نحوه‌ی ذخیره‌کردن مدل و استفاده مجدد از وزن‌های آن نیز مطالبی بیان می‌شود.در آخر هم پیشنهاداتی درباره‌ی کارهایی که می‌توان انجام داد تا طبقه‌بندی بهتری انجام شود و نتایج بهتری داشته باشیم،داده خواهد شد.

کلمات کلیدی

قوز قرنیه،هوش مصنوعی،شبکه‌های CNN،پیش‌پردازش تصاویر،مدل‌های پیش‌آموزش‌دیده

[1] Pretrained models

مقدمه

در سال‌های اخیر استفاده از هوش مصنوعی در زمینه‌های علمی متعددی اهمیت پیدا کرده است.یکی از علومی که استفاده از هوش مصنوعی در آن در حال گسترش است،علم پزشکی است.تشخیص بیماری‌ها در مراحل اولیه‌ی آن که هنوز پیشرفت زیادی نکرده است،برای درمان بسیار حائز اهمیت است که تحقق آن با روش های هوش مصنوعی امکان‌پذیر است.یکی از این بیماری‌ها،بیماری قوز قرنیه [۲]است.قوز قرنیه، یکی از عوامل منع جراحی انکساری چشم مثل لازیک[۳] و فمتو[۴] می‌باشد.برای انجام این جراحی‌ها،جراح ابتدا باید مطمئن باشد که قرنیه چشم سالم و حتی مشکوک به قوز قرنیه نیست. پس در این کاربرد،استفاده از هوش مصنوعی و تشخیص زود و به موقع این بیماری بسیار حائز اهمیت می‌باشد.

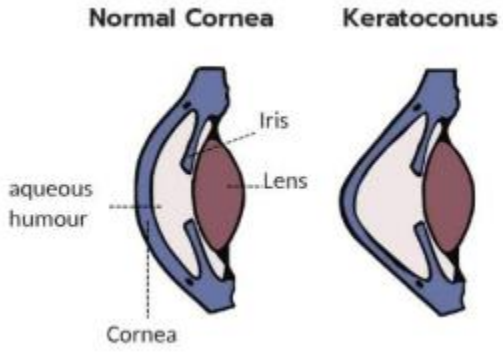
در این گزارش،از دیتای Kaggle که دارای حدود ۶۰۰ تصویر می باشد،برای طبقه بندی تصاویر استفاده می‌شود.هر case در این مجموعه داده حاوی ۷ تصویر مختلف است.در واقع در این حالت تصاویر به جای 4Map دارای 7Map هستند.در این گزارش،شبکه‌های مختلف از جمله شبکه‌ی دستی که توسط خود ما طراحی شده و تعدادی شبکه‌ی pretrained نیز برای طبقه‌بندی مورد بررسی قرار خواهند گرفت.در آخر ترکیبی از این شبکه‌ها نیز برای طبقه بندی مورد بررسی قرار خواهند گرفت.

[2] keratoconus
[3] Lasik
[4] Femto

بخش اول:طرح مسئله و توضیحات اولیه

تشریح بیماری

در بیماری قوز قرنیه،قرنیه‌ی فرد بیمار تغییر شکل می‌دهد و برآمدگی در قرنیه ایجاد می‌شود.این بیماری معمولا در دهه‌ی دوم یا سوم عمر فرد ایجاد می‌شود و تا نهایتا ۴۰سالگی پیشرفت می‌کند.علائم بالینی این بیماری،بسته به شدت بیماری متفاوت است.شکل۱ مقایسه‌ای از چشم فرد سالم و فرد دارای بیماری قوز قرنیه را نشان می‌دهد.



شکل۱. چشم فرد سالم در برابر چشم فرد بیمار

هر چقدر شکل قرنیه نامنظم‌تر می‌شود،نزدیک‌بینی و آستیگماتیسم نامنظم بیشتری ایجاد می‌شود و ممکن است پخش نور و حساسیت به نور نیز در بیمار بروز کند.

چالش‌های اصلی این بیماری

چالش اصلی بیماری قوز قرنیه برای پزشکان،تشخیص هر چه سریعتر بیماری در مراحل تحت بالینی می‌باشد.این بیماری به این دارای اهمیت بالایی است که یکی از عوامل منع جراحی انکساری چشم مثل لازیک[5] و

[5] LASEK or Laser-Assisted Sub-Epithelial Keratectomy

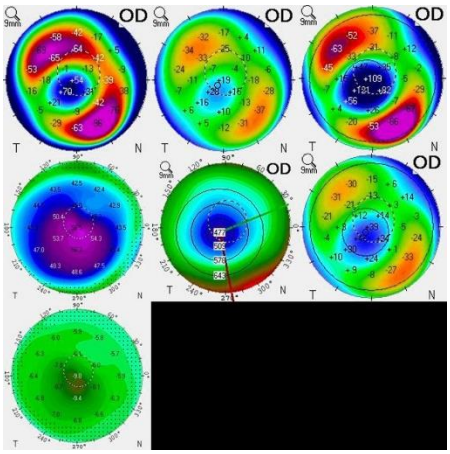
فمتو[6] است.به منظور انجام این جراحی،جراح باید از این موضوع مطمئن باشد که قرنیه چشم بیمار کاملا سالم است و حتی مشکوک به قوز قرنیه هم نیست.

چالش دیگر در این بیماری،این است که در سال‌های اولیه‌ی ایجاد این بیماری در فرد،معمولا به طور واضح علائم آن آشکار نمی‌شود و ممکن است در معاینات اولیه‌ی پزشک،به اشتباه بیماری‌هایی مانند ضعیفی چشم و یا آستیگماتیسم تشخیص داده شود.در صورتی که اگر این بیماری در مراحل اولیه تشخیص داده شود،با روش-هایی می‌توان پیشرفت این بیماری را کنترل کرد.

بخش دوم:تشریح دیتاست

تصاویر 7Maps Refractive

تصاویر مربوط به دیتاست موجود در Kaggle شامل نمونه‌هایی است که هر یک شامل ۷نقشه از تصاویر مربوط به چشم است.همه‌ی این ۷نقشه در یک تصویر برای نمونه در شکل۲ قابل مشاهده است:



شکل۲. تصویری از ۷نقشه‌ی نمونه‌ای از دیتا

[6] Femto (refers to “femtosecond” laser technology

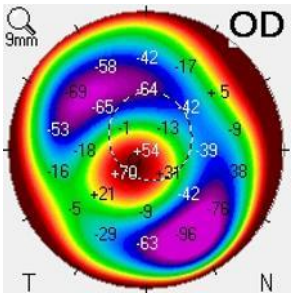
در واقع هر یک از این ۷نقشه تصاویر جداگانه‌ای هستند که می‌توان با هر یک از آن‌ها یک کار طبقه‌بندی جداگانه انجام داد.اما در این کار،هر ۷تصویر به شکل یک تصویر به صورت بالا هستند که کار طبقه‌بندی با این نوع دیتا انجام می‌شود.

در دیتاست kaggle،تصاویر دارای ۳کلاس Normal،Keratoconus و Suspect است.در کلاس‌های Keratoconus و Normal برای بخش آموزش ۱۵۰ نمونه و برای کلاس Suspect ۱۲۳ نمونه برای بخش آموزش وجود دارد.همچنین برای بخش ازریابی مدل،برای هر کلاس ۵۰ نمونه در دسترس است.

پیش‌پردازش دیتا و حذف قسمت‌های اضافی تصاویر

همان‌طور که مشاهده می‌شود،برخی اطلاعات عددی بر روی تصاویر وجود دارد که به پزشک کمک می‌کند تا تحلیل بهتری از تصاویر داشته باشد؛اما در کار طبقه‌بندی با شبکه‌های هوش مصنوعی،وجود این موارد اضافی نه تنها لزومی ندارد بلکه ممکن است کار طبقه‌بندی را با اختلال روبرو کند.در نتیجه سعی می‌شود تا این نویزها تا حدامکان از نمونه‌های دیتاست حذف شوند.در ادامه مراحلی که برای این کار طی شده را بررسی می‌کنیم.

یک نمونه از نقشه‌ها که برای حذف نویز انتخاب شده است در شکل۳ مشاهده می‌شود:



شکل۳. یک نمونه از نقشه‌ها برای انجام فرایند حذف نویز

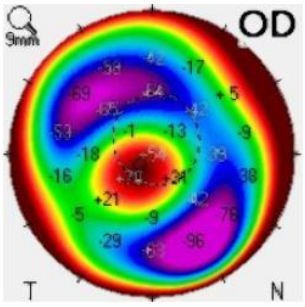
روش اول برای حذف نویز

اولین روشی که برای حذف نویز تست شد به صورت زیر است:

```
# Load the original image
original_image = cv2.imread('KCN_1_Elv_P.jpg')

# Convert the image to grayscale
```

```
gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)

# Threshold the grayscale image to create a binary mask
_, binary_mask = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)

# Apply median blur to the original image
blurred_image = cv2.medianBlur(original_image, 5)

# Replace white pixels in the original image with corresponding pixels
from the blurred image
result_image = original_image.copy()
result_image[binary_mask == 255] = blurred_image[binary_mask == 255]
```

در این روش ابتدا تصویر تبدیل به grayscale یا یک تصویر سیاه و سفید می‌شود.سپس یک binary mask از تصویر سیاه و سفید ساخته می‌شود.بعد از آن median blur بر روی تصویر اولیه اعمال شده و در نهایت پیکسل‌های سفید روی عکس اصلی جاگذاری می‌شود.با این تکنیک سعی شده قسمت‌های سیاه رنگ و سفیدرنگ که همان نوشته‌ها هستند از تصویر حذف شوند.خروجی به صورت شکل۴ است:



شکل۴. خروجی پیاده‌سازی روش اول روی تصویر

همان‌طور که مشاهده می‌شود،حذف نویز تا حدودی صورت گرفته است اما باز هم نمی‌توان گفت که این روش عملکرد خوبی داشته است.

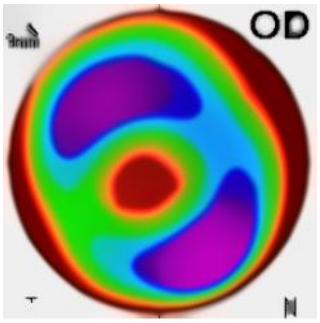
تکمیل روش اول برای حذف نویز

در این روش در ادامه همان روش اول،یک الگوریتم کاهش نویز دیگر روی تصویر اعمال می‌شود تا بتوان نویزهای باقی‌مانده روی تصویر را به طور کامل حذف کرد.الگوریتم مورداستفاده به صورت زیر است:

```
h = 22              # Strength of the noise reduction. Higher values mean
more aggressive denoising.
hForColor = 15      # Strength of the noise reduction for color components.
templateWindowSize = 7  # Size in pixels of the template patch.
searchWindowSize = 25   # Size in pixels of the window used to compute the
weighted average for a given pixel.

for i in range(5):
    templateWindowSize = templateWindowSize if templateWindowSize % 2 == 1
else templateWindowSize + 1
    searchWindowSize = searchWindowSize if searchWindowSize % 2 == 1 else
searchWindowSize + 1

    denoised_image = cv2.fastNlMeansDenoisingColored(result_image, None,
h, hForColor, templateWindowSize, searchWindowSize)

    result_image = denoised_image
```

این الگوریتم چهار پارامتر دارد که این پارامترها تعیین می‌کنند که چه میزان حذف نویز و با چه دقتی این کار انجام شود.بزرگ‌بودن بیش از حد پارامترها سبب از بین رفتن قسمت‌های اصلی تصویر و کوچک‌بودن بیش از حد آن‌ها موجب حذف نشدن نویز به طور کامل می‌شود.

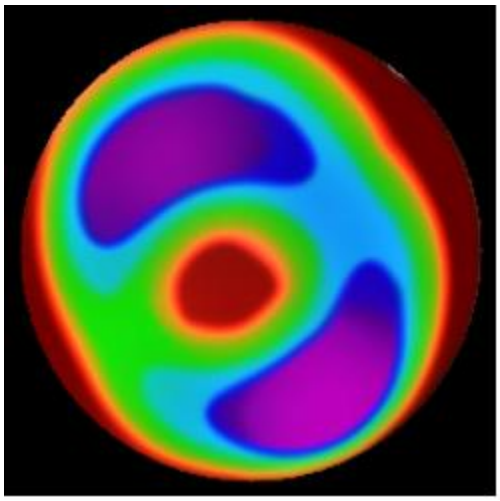
عملکرد این الگوریتم با مقادیر مشخص‌شده برای پارامترها در کد بالا به صورت شکل۵ است:



شکل۵. عملکرد روش تکمیلی اول روی تصویر

همان‌طور که مشاهده می‌شود،داده‌های عددی روی تصویر به کلی حذف شده‌اند و رنگ‌های موجود در تصویر نیز تا حد خوبی حفظ شده‌اند.

در آخر هم گوشه‌های تصویر هم باید سیاه‌رنگ شوند تا قسمت‌های اضافی گوشه هم از بین بروند.خروجی نهایی به صورت شکل۶ است:



شکل۶. تصویر نهایی نویزگیری شده

از آن‌جا که ممکن است برای نقشه‌های مختلف نیاز به نویزگیری با پارامترهای متفاوت باشد،تابعی درنظر گرفته شده است تا میزان شباهت تصویر نویزگیری شده و تصویر اصلی را بدهد.این تابع به صورت زیر است:

```
def similarity_2picture(image1 , image2):
```

```
  from skimage.metrics import structural_similarity as ssim
  gray_pic1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
  gray_pic2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
  ssim_index = ssim(gray_pic1, gray_pic2)
  return ssim_index
```

همچنین تابع دیگری درنظر گرفته شده تا برای یک تصویر میزان مشابهت تصویر اصلی و تصویر نویزگیری شده را به ازای بازه‌ای از چهار پارامتری که گفته شد حساب کرده و در نهایت با توجه به این عدد پارامترهایی که بهترین درصد مشابهت را می‌دهند به عنوان خروجی به ما داده شوند.این تابع به صورت زیر است:

```
def parameter_search(image_name):
  import numpy as np
  image_org = cv2.imread(image_name)
  height, width = image_org.shape[:2]

  center_x = width // 2
  center_y = height // 2

  radius = 105
  mask = np.zeros((height, width), dtype=np.uint8)
  cv2.circle(mask, (center_x, center_y), radius, 255, -1)

  mask_inv = cv2.bitwise_not(mask)

  image_org[mask_inv == 255] = [0, 0, 0]
  for i in range(7):
    for j in range(7):
        image_den = denoise_image(image_path = image_name , h = 15+i ,
hForColor = 8+j , templateWindowSize = 7 , searchWindowSize = 25 , epochs
= 5)
        A = similarity_2picture(image1 = image_org , image2 = image_den)
        if A > 0.6 and A < 0.7:
          print(similarity_2picture(image1 = image_org , image2 =
image_den))
          print(f" h = {15+i} | hForColor = {7+j}")
```

بخش سوم:مدل‌های مختلف استفاده شده برای طبقه‌بندی تصاویر

بهترین مدل‌هایی که برای طبقه‌بندی تصاویر استفاده می‌شوند،شبکه‌های CNN[7] هستند.این شبکه‌ها بهترین شبکه‌ها برای استخراج ویژگی از داده‌های تصویری و یا داده‌های ویدیویی هستند.یک شبکه‌ی CNN می‌تواند توسط خود ما طراحی شود و برای طبقه‌بندی استفاده شود.همچنین شبکه‌های از پیش آموزش دیده نیز وجود دارند که شبکه‌هایی بزرگ با لایه‌های زیاد هستند که روی داده‌های تصویری خیلی بزرگ آموزش دیده‌اند و قادر به طبقه‌بندی با دقت بالای انواع داده‌های تصویری هستند.البته با توجه به مسئله‌ای که با آن مواجه هستیم ممکن است استفاده از یکی از این شبکه‌ها به تنهایی کارساز نباشد و نیاز باشد که از ترکیب این شبکه‌ها با یکدیگر و یا ترکیب با روش‌های دیگر استفاده کرد.

شبکه‌ی CNN عادی

در قسمت اول نتایج حاصل از شبکه‌ای که خودمان ساختیم را بررسی می‌کنیم.

این شبکه به صورت زیر است:

```
class CNN_Model(nn.Module):
  def __init__(self, input_shape: int, hidden_units: int, output_shape:
int):
        super().__init__()
        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=hidden_units,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
```

[7] Convolutional neural network

```
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=hidden_units*64*64,
                      out_features=output_shape)
        )

  def forward(self, x: torch.Tensor):
      x = self.block_1(x)
      x = self.block_2(x)
      x = self.classifier(x)
      return x
```

این شبکه از دو لایه‌ی کلی کانولوشنال تشکیل شده است.هر یک از این دو لایه به ترتیب شامل یک لایه‌ی Conv2d،یک لایه‌ی تابع ReLU،مجدد یک لایه‌ی Conv2d،مجدد یک لایه تابع ReLU و در نهایت یک MaxPool2d می‌باشد که برای استخراج ویژگی از تصاویر است.همچنین در لایه‌ی این شبکه،Classifier قرار دارد تا طبقه‌بندی را برای ما انجام دهد.

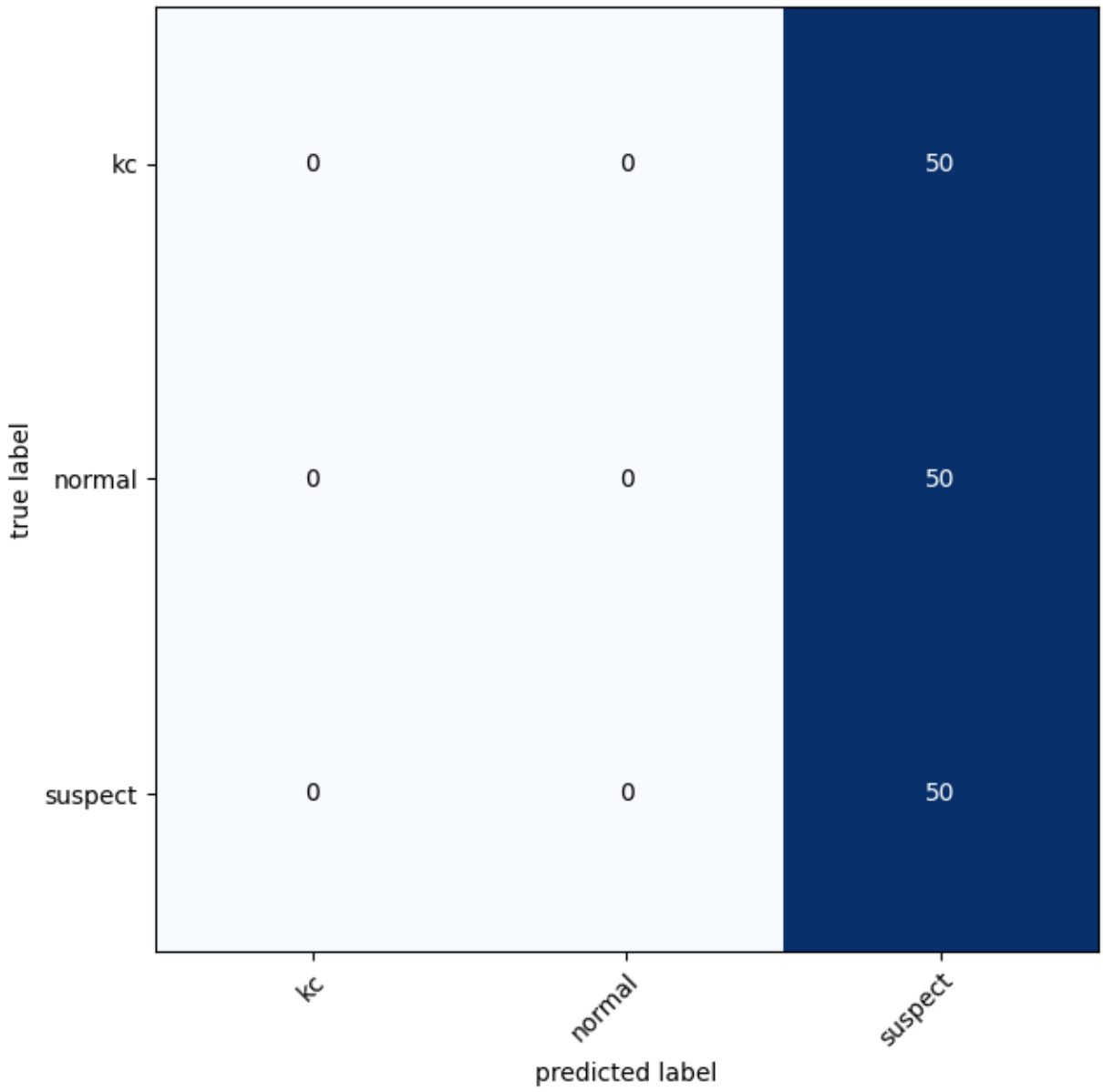
همچنین مدل ما به صورت زیر تعریف می‌شود:

```
model4 = CNN_Model(input_shape=3,
    hidden_units=40,
    output_shape=3).to(device)
optimizer_model4 = torch.optim.Adam(model4.parameters(), lr=0.01)
```

نتیجه آموزش و ارزیابی این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 1.0994 | train_acc: 0.2913 | test_loss: 1.0982 |
test_acc: 0.3400
Epoch: 2 | train_loss: 1.0995 | train_acc: 0.2913 | test_loss: 1.0982 |
test_acc: 0.3400
Epoch: 3 | train_loss: 1.0995 | train_acc: 0.2904 | test_loss: 1.0983 |
test_acc: 0.3333
Epoch: 4 | train_loss: 1.0995 | train_acc: 0.2910 | test_loss: 1.0983 |
test_acc: 0.3400
Epoch: 5 | train_loss: 1.0994 | train_acc: 0.2908 | test_loss: 1.0985 |
test_acc: 0.3333
Total training time: 908.855 seconds
```

همان‌طور که مشاهده می‌شود این مدل اصلا عملکرد خوبی نداشته است و مقدار اتلاف اصلا تغییری نکرده است.همچنین دقت تقریبا تغییری نداشته و ۳۳درصد است.حال ماتریس درهم‌ریختگی مربوط به این مدل در شکل۷ مشاهده می‌شود:



شکل۷. ماتریس درهم‌ریختگی مدل عادی

در این ماتریس هم می‌بینیم که مدل عملا پیش‌بینی را نتوانسته انجام دهد و همه‌ی داده‌ها را در کلاس suspect تشخیص داده است.

استفاده از شبکه‌ی پیش‌آموزش دیده

شبکه‌ای که در این طبقه‌بندی از آن استفاده می‌شود،شبکه‌ی efficientnet_b2 است که از شبکه‌های پیش‌آموزش دیده می‌باشد.همچنین وزن‌های مدل نیز همان وزن‌های پیش‌فرض این مدل است.مراحل تعریف مدل به صورت زیر است:

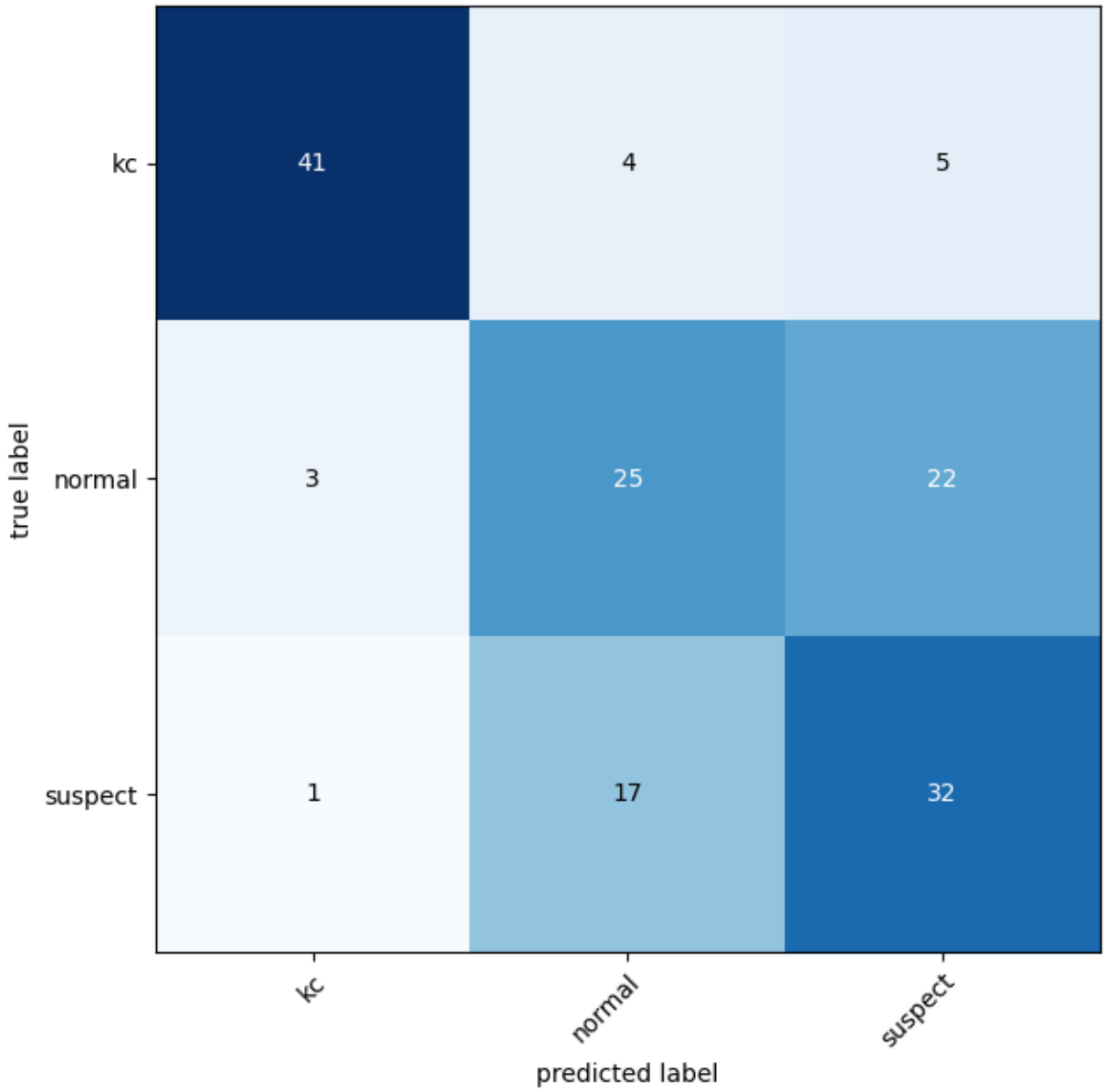
```
weights1 = torchvision.models.EfficientNet_B2_Weights.DEFAULT
model1 = torchvision.models.efficientnet_b2(weights=weights1).to(device)
model1.classifier = torch.nn.Sequential(
    torch.nn.Dropout(p=0.2, inplace=True),
    torch.nn.Linear(in_features=1408,
                    out_features=output_shape,
                    bias=True)).to(device)
optimizer_model1 = torch.optim.Adam(model1.parameters(), lr=0.01)
```

نتایج برای این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 0.5176 | train_acc: 0.7731 | test_loss: 1.4243 |
test_acc: 0.3867
Epoch: 2 | train_loss: 0.5245 | train_acc: 0.7331 | test_loss: 0.9047 |
test_acc: 0.5133
Epoch: 3 | train_loss: 0.4991 | train_acc: 0.7733 | test_loss: 0.8020 |
test_acc: 0.6733
Epoch: 4 | train_loss: 0.4979 | train_acc: 0.7532 | test_loss: 0.7799 |
test_acc: 0.6267
Epoch: 5 | train_loss: 0.5466 | train_acc: 0.7584 | test_loss: 1.4405 |
test_acc: 0.5133
Epoch: 6 | train_loss: 0.4886 | train_acc: 0.7709 | test_loss: 1.1171 |
test_acc: 0.5600
Epoch: 7 | train_loss: 0.4081 | train_acc: 0.8245 | test_loss: 1.3825 |
test_acc: 0.6000
Epoch: 8 | train_loss: 0.3132 | train_acc: 0.8649 | test_loss: 0.8542 |
test_acc: 0.6467
Epoch: 9 | train_loss: 0.4017 | train_acc: 0.8157 | test_loss: 1.5632 |
test_acc: 0.5267
Epoch: 10 | train_loss: 0.2991 | train_acc: 0.8886 | test_loss: 1.0253 |
test_acc: 0.6867
```

همان‌طور که می‌بینیم،در این مدل مقدار اتلاف کاهش یافته است و در نهایت در قسمت آموزش به دقت ۰.۸۹ و در قسمت ارزیابی به دقت حدود ۰.۶۹ رسیده ایم که نتیجه‌ی نسبتا خوبی است.

حال ماتریس درهم‌ریختگی مربوط به این مدل را در شکل۸ می‌بینیم:



شکل۸. ماتریس درهم‌ریختگی مدل پیش‌آموزش‌دیده‌ی efficientnetb2

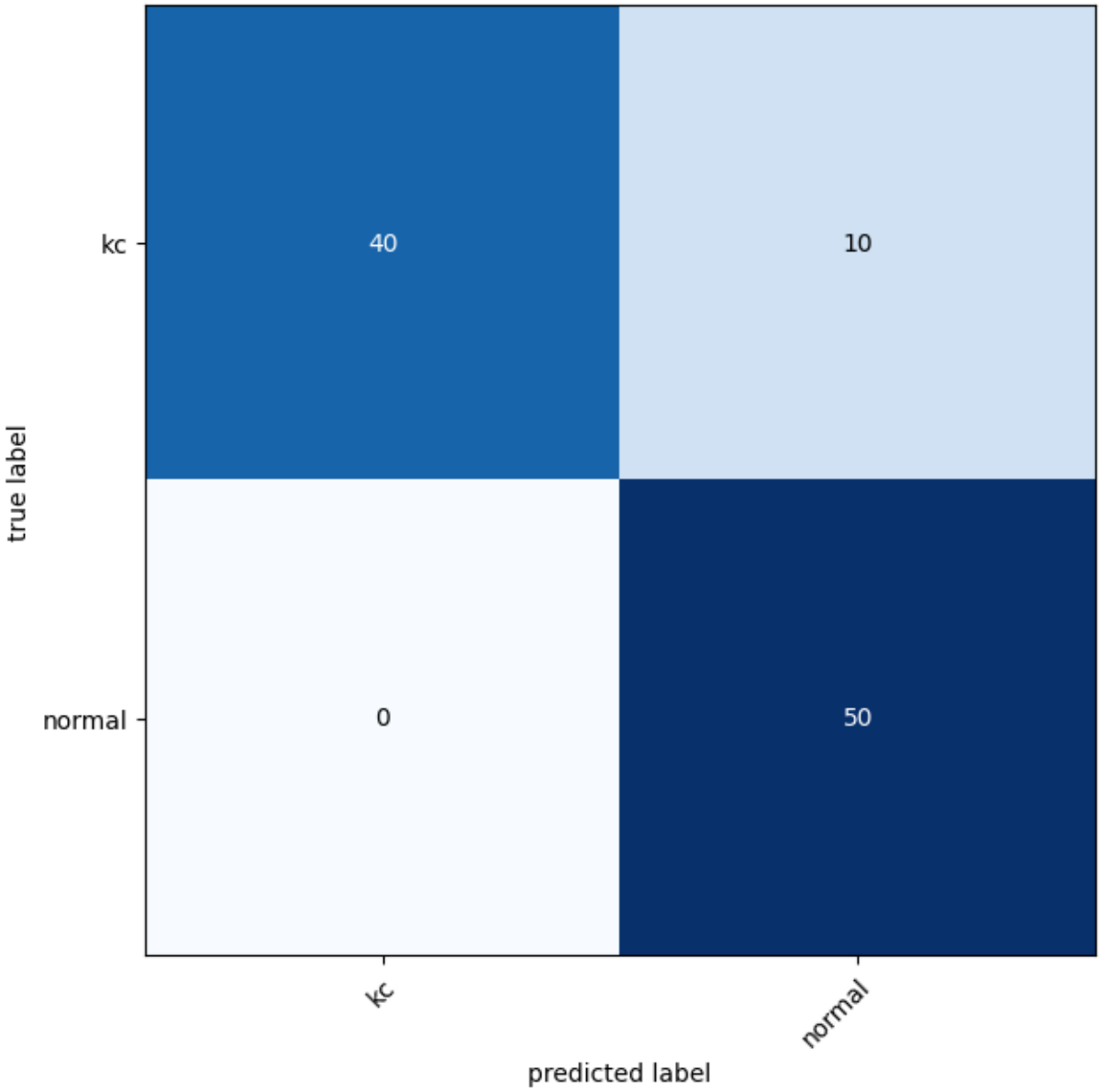
همان‌طور که می‌بینیم،این مدل تشخیص خیلی خوبی برای داده‌های کلاس kc داشته است.اما در کلاس-های suspect و normal به دلیل شباهت تصاویر آن‌ها،عملکرد خیلی خوبی نداشته است.

حال همین مدل را روی داده‌ای که دارای دو کلاس kc و normal است،تست می‌کنیم.نتیجه به صورت زیر است:

```
Epoch: 1 | train_loss: 0.4960 | train_acc: 0.8033 | test_loss: 15880.6392
| test_acc: 0.5000
Epoch: 2 | train_loss: 0.2783 | train_acc: 0.9200 | test_loss: 4405.8959 |
test_acc: 0.4900
Epoch: 3 | train_loss: 0.2843 | train_acc: 0.9167 | test_loss: 70.9551 |
test_acc: 0.5000
Epoch: 4 | train_loss: 0.2535 | train_acc: 0.9267 | test_loss: 1.4179 |
test_acc: 0.8500
Epoch: 5 | train_loss: 0.2108 | train_acc: 0.9267 | test_loss: 1.8811 |
test_acc: 0.6600
Epoch: 6 | train_loss: 0.1512 | train_acc: 0.9433 | test_loss: 0.6057 |
test_acc: 0.6600
Epoch: 7 | train_loss: 0.1388 | train_acc: 0.9733 | test_loss: 0.6903 |
test_acc: 0.6800
Epoch: 8 | train_loss: 0.0652 | train_acc: 0.9800 | test_loss: 0.1969 |
test_acc: 0.8800
Epoch: 9 | train_loss: 0.0337 | train_acc: 0.9867 | test_loss: 1.7162 |
test_acc: 0.7000
Epoch: 10 | train_loss: 0.0386 | train_acc: 0.9900 | test_loss: 0.3636 |
test_acc: 0.9000
Total training time: 1291.264 seconds
```

همان‌طور که مشاهده می‌شود،در این حالت دقت به ۹۰ درصد رسیده است و عملکرد مدل خیلی خوب بوده است.

در شکل۹ ماتریس درهم‌ریختگی را در این حالت می‌بینیم:



شکل۹. ماتریس درهم‌ریختگی مدل پیش‌آموزش‌دیده با داده‌ی دارای دو کلاس

استفاده از ترکیبی از شبکه‌های پیش‌آموزش‌دیده

در این قسمت،تنها از یک مدل پیش‌آموزش‌دیده استفاده نمی‌شود و از ترکیبی از ۳مدل پیش‌آموزش‌دیده استفاده می‌شود.این مدل به صورت زیر است:

```
model11 = models.resnet18(pretrained=True)
model22 = models.vgg16(pretrained=True)
model33 = models.efficientnet_b0(pretrained=True)
```

```
for param in model11.parameters():
    param.requires_grad = False
for param in model33.parameters():
    param.requires_grad = False

num_classes=len(class_names)

class FusionModel(nn.Module):
    def __init__(self, model1, model2, model3):
        super(FusionModel, self).__init__()

        self.fc1 = nn.Linear(3 * 1000, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.model1 = model11
        self.model2 = model22
        self.model3 = model33

    def forward(self, x):

        out1 = self.model1(x)
        out2 = self.model2(x)
        out3 = self.model3(x)

        out = torch.cat((out1, out2, out3), dim=1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out


device = "cuda" if torch.cuda.is_available() else "cpu"
fusion_model = FusionModel(model11 , model22 , model33).to(device)
```

در این‌جا ما سه مدل resnet18 و vgg16 و efficientnetb0 به صورت ترکیبی کار طبقه‌بندی را برای ما انجام خواهند داد.یعنی ورودی که تصاویر ما هستند،به هر یک از این شبکه‌ها داده می‌شود و خروجی این شبکه‌ها ترکیب شده و تصمیم‌گیری با آن انجام می‌شود.در این کار،پارامترهای مدل‌های resnet18 و efficientnetb0 به اصطلاح منجمد شده‌اند.[8]با این کار پارامترها و وزن‌های مدل در فرایند آموزش update نمی‌شوند و این باعث می‌شود که سرعت مدل بیشتر شود و میزان حافظه‌ی استفاده شده نیز کاهش یابد.

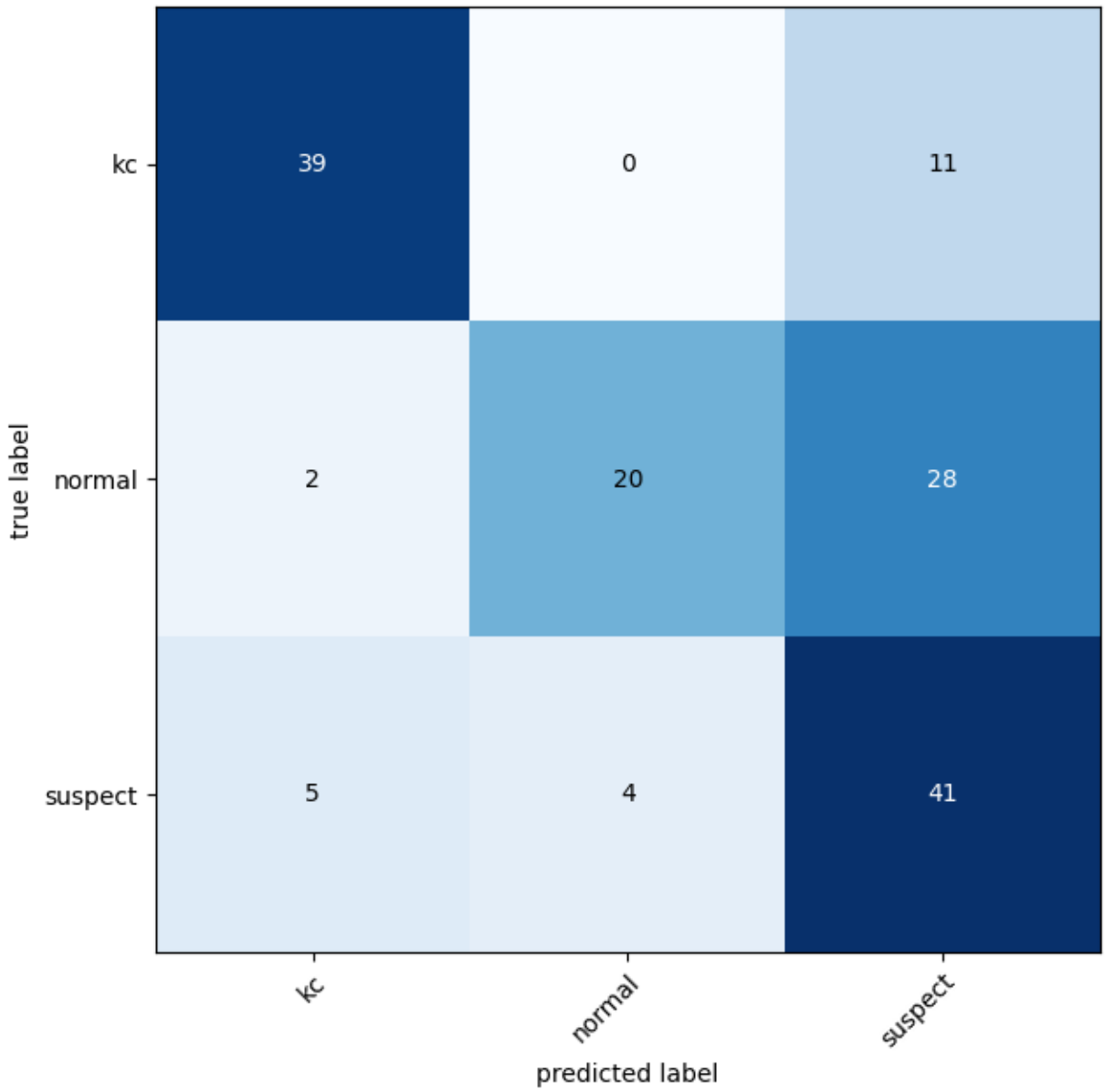
[8] Freezing parameters

نتیجه‌ی نهایی به صورت زیر است:

```
Epoch: 1 | train_loss: 0.3180 | train_acc: 0.8792 | test_loss: 1.0122 | test_
acc: 0.6533
Epoch: 2 | train_loss: 0.2518 | train_acc: 0.9025 | test_loss: 0.9597 | test_
acc: 0.6533
Epoch: 3 | train_loss: 0.2331 | train_acc: 0.9127 | test_loss: 1.0741 | test_
acc: 0.6133
Epoch: 4 | train_loss: 0.2309 | train_acc: 0.9153 | test_loss: 1.0367 | test_
acc: 0.6600
Epoch: 5 | train_loss: 0.2196 | train_acc: 0.9012 | test_loss: 1.0828 | test_
acc: 0.6467
Epoch: 6 | train_loss: 0.2221 | train_acc: 0.9008 | test_loss: 1.0919 | test_
acc: 0.6667
Epoch: 7 | train_loss: 0.1844 | train_acc: 0.9247 | test_loss: 1.0633 | test_
acc: 0.6733
Epoch: 8 | train_loss: 0.2120 | train_acc: 0.9027 | test_loss: 1.1995 | test_
acc: 0.6133
Epoch: 9 | train_loss: 0.1905 | train_acc: 0.9174 | test_loss: 1.1049 | test_
acc: 0.6467
Epoch: 10 | train_loss: 0.1876 | train_acc: 0.9233 | test_loss: 0.9496 | test
_acc: 0.6767
Total training time: 107.197 seconds
```

همان‌طور که می‌بینیم،در این حالت تقریبا نتیجه مشابه مدل قبلی است اما کمی بهتر است.همچنین در قسمت آموزش،دقت مدل بهتر شده است.

ماتریس درهم‌ریختگی این مدل به صورت شکل ۱۰ مشاهده می‌شود:



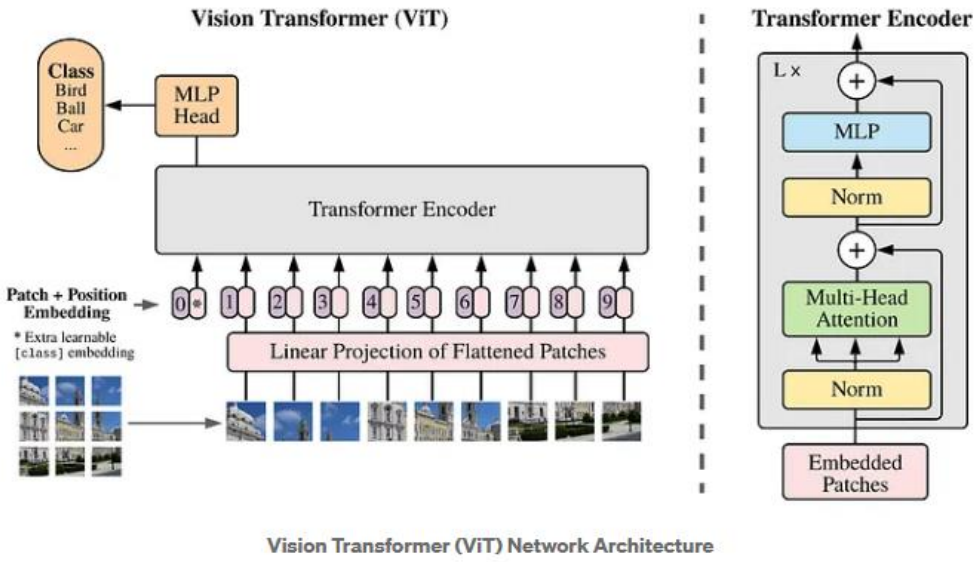
شکل۱۰. ماتریس درهم‌ریختگی مدل ترکیبی از سه مدل پیش‌آموزش‌دیده

در این حالت هم مشکل طبقه‌بندی،در تشخیص بین کلاس Normal و Suspect است.در واقع،مدل داده-هایی که کلاس kc یا suspect بوده است را تقریبا خوب تشخیص داده است.اما در داده‌هایی که در کلاس Normal بوده‌اند،باز هم تشخیص خوب نبوده است.این عدم تشخیص می‌تواند به علت شباهتی که بین این دو کلاس وجود دارد باشد.

استفاده از شبکه‌ی Vision Transformer(ViT)

در این بخش مدل‌های ViT را بررسی می‌کنیم.این مدل‌ها بر پایه‌ی Patch کردن تصاویر ورودی و استخراج ویژگی از این Patch ها با استفاده از Encoder عمل می‌کنند.

یک نمونه از ساختار این مدل به شکل زیر است:



شکل ۱۱. ساختار کلی مدل Vision Transformer(ViT)

مدل‌هایی که در این بخش بررسی می‌شوند،یک مدل عادی ViT،مدل vit_b_32 و ترکیبی از مدل vit_b_16 با مدل‌های دیگر می‌باشد.

مدل ViT عادی

مدل اولی که بررسی می‌شود،یک مدل ViT عادی است که خود ما آن را طراحی کرده‌ایم.شکل کلی ساختار این مدل در پیوست آمده است.

کدی که برای ساخت این مدل در نظر گرفته شده به صورت زیر است:

```
class PatchEmbedding(nn.Module):
```

```
    def __init__(self,
                 in_channels:int=3,
                 patch_size:int=32,
                 embedding_dim:int=768):
        super().__init__()

        self.patcher = nn.Conv2d(in_channels=in_channels,
                                 out_channels=embedding_dim,
                                 kernel_size=patch_size,
                                 stride=patch_size,
                                 padding=0)
        self.patch_size = patch_size
        self.flatten = nn.Flatten(start_dim=2,
                                  end_dim=3)

    def forward(self, x):
        image_resolution = x.shape[-1]
        assert image_resolution % self.patch_size == 0, f"Input image size
must be divisble by patch size, image shape: {image_resolution}, patch
size: {patch_size}"

        x_patched = self.patcher(x)
        x_flattened = self.flatten(x_patched)
        return x_flattened.permute(0, 2, 1)
```

این کد برای استخراج Patch هایی با اندازه‌ی 32*32 از تصویر ورودی است.در واقع برای استخراج Patch،از یک لایه‌ی Conv2D که فراپارامترهای kernel_size و stride آن برابر Patch_size و padding آن برابر صفر است برای استخراج این Patch ها استفاده می‌شود.

```
class TransformerEncoderBlock(nn.Module):
    """Creates a Transformer Encoder block."""
    def __init__(self,
                 embedding_dim:int=768,
                 num_heads:int=12,
                 mlp_size:int=3072,
                 mlp_dropout:float=0.1,
                 attn_dropout:float=0):
        super().__init__()
```

```
        self.msa_block =
MultiheadSelfAttentionBlock(embedding_dim=embedding_dim,
                                                        num_heads=num_heads,
                                                        attn_dropout=attn_dro
pout)

        self.mlp_block =  MLPBlock(embedding_dim=embedding_dim,
                                   mlp_size=mlp_size,
                                   dropout=mlp_dropout)

    def forward(self, x):

        x =  self.msa_block(x) + x

        x = self.mlp_block(x) + x

        return x
```

این بخش از کد هم برای ساختن همان Encoder که در ساختار کلی ViT دیدیم،استفاده می‌شود.

```
class ViT(nn.Module):
    """Creates a Vision Transformer architecture with ViT-Base
hyperparameters by default."""
    def __init__(self,
                 img_size:int=224,
                 in_channels:int=3,
                 patch_size:int=32,
                 num_transformer_layers:int=12,
                 embedding_dim:int=768,
                 mlp_size:int=3072,
                 num_heads:int=12,
                 attn_dropout:float=0,
                 mlp_dropout:float=0.1,
                 embedding_dropout:float=0.1,
                 num_classes:int=1000):
        super().__init__()

        assert img_size % patch_size == 0, f"Image size must be divisible
by patch size, image size: {img_size}, patch size: {patch_size}."

        self.num_patches = (img_size * img_size) // patch_size**2
```

```
        self.class_embedding = nn.Parameter(data=torch.randn(1, 1,
embedding_dim),
                                            requires_grad=True)

        self.position_embedding = nn.Parameter(data=torch.randn(1,
self.num_patches+1, embedding_dim),
                                               requires_grad=True)

        self.embedding_dropout = nn.Dropout(p=embedding_dropout)

        self.patch_embedding = PatchEmbedding(in_channels=in_channels,
                                              patch_size=patch_size,
                                              embedding_dim=embedding_dim)

        self.transformer_encoder =
nn.Sequential(*[TransformerEncoderBlock(embedding_dim=embedding_dim,

  num_heads=num_heads,

  mlp_size=mlp_size,

  mlp_dropout=mlp_dropout) for _ in range(num_transformer_layers)])

        self.classifier = nn.Sequential(
            nn.LayerNorm(normalized_shape=embedding_dim),
            nn.Linear(in_features=embedding_dim,
                      out_features=num_classes)
        )

    def forward(self, x):

        batch_size = x.shape[0]

        class_token = self.class_embedding.expand(batch_size, -1, -1)
        x = self.patch_embedding(x)

        x = torch.cat((class_token, x), dim=1)

        x = self.position_embedding + x

        x = self.embedding_dropout(x)

        x = self.transformer_encoder(x)

        x = self.classifier(x[:, 0])
```

```
        return x
```

در این کلاس تعریف‌شده،توابعی که در بالا تعریف شدند،با هم ترکیب شده و در نهایت مدل ViT را برای ما می‌سازند.

```
vit_manual_model = ViT(num_classes=len(class_names)).to(device)

from module_funcs import engine

optimizer_manual_vit =
torch.optim.Adam(params=vit_manual_model.parameters(),
                                lr=3e-3,
                                betas=(0.9, 0.999),
                                weight_decay=0.3)

loss_fn = torch.nn.CrossEntropyLoss()

torch.manual_seed(42)
torch.cuda.manual_seed(42)

results = engine.train(model=vit_manual_model,
                       train_dataloader=train_dataloader,
                       test_dataloader=test_dataloader,
                       optimizer=optimizer_manual_vit,
                       loss_fn=loss_fn,
                       epochs=10,
                       device=device)
```

در این بخش هم optimizer و loss function برای مدل تعریف شده و مدل آموزش خواهد دید.نتایج به صورت زیر است:

```
Epoch: 1 | train_loss: 2.8933 | train_acc: 0.3618 | test_loss: 1.3469 |
test_acc: 0.3333
Epoch: 2 | train_loss: 1.1510 | train_acc: 0.3640 | test_loss: 1.1093 |
test_acc: 0.3333
Epoch: 3 | train_loss: 1.0997 | train_acc: 0.3648 | test_loss: 1.1789 |
test_acc: 0.3333
Epoch: 4 | train_loss: 1.1313 | train_acc: 0.3446 | test_loss: 1.0993 |
test_acc: 0.3333
Epoch: 5 | train_loss: 1.1753 | train_acc: 0.2981 | test_loss: 1.1278 |
test_acc: 0.3333
Epoch: 6 | train_loss: 1.1854 | train_acc: 0.3589 | test_loss: 1.1071 |
test_acc: 0.3333
```

```
Epoch: 7 | train_loss: 1.1400 | train_acc: 0.3687 | test_loss: 1.1525 |
test_acc: 0.3333
Epoch: 8 | train_loss: 1.1127 | train_acc: 0.3351 | test_loss: 1.1189 |
test_acc: 0.3333
Epoch: 9 | train_loss: 1.1192 | train_acc: 0.3238 | test_loss: 1.1142 |
test_acc: 0.3333
Epoch: 10 | train_loss: 1.1231 | train_acc: 0.3263 | test_loss: 1.1137 |
test_acc: 0.3333
```

همان‌طور که مشاهده می‌شود،این مدل همانند مدل عادی CNN عملکرد خوبی نداشته است و در نهایت به دقت ۳۳درصد رسیده‌ایم.

استفاده از vit_b_16

مدل vit_b_16 در واقع شبیه به همان مدل ViT عادی است؛با این تفاوت که در این مدل،از لایه‌های بیشتری استفاده شده است و این مدل روی دیتاست تصویری بزرگ ImageNet آموزش دیده است.در واقع نوعی از مدل ViT پیش‌آموزش‌دیده است.

مراحل استفاده از این مدل به صورت زیر است:

```
pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT

pretrained_vit =
torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)

for parameter in pretrained_vit.parameters():
    parameter.requires_grad = False

torch.manual_seed(42)
torch.cuda.manual_seed(42)

pretrained_vit.heads = nn.Linear(in_features=768,
out_features=len(class_names)).to(device)
pretrained_vit_transforms = pretrained_vit_weights.transforms()
optimizer_vit = torch.optim.Adam(params=pretrained_vit.parameters(),
                                 lr=0.0001)
loss_fn = torch.nn.CrossEntropyLoss()
train_dataloader_pretrained, test_dataloader_pretrained, class_names =
data_setup.create_dataloaders(train_dir="/content/train_data_kc/train_data
_kc",
```

```
                                test_dir="/content/test_data_kc/test_data_kc",
                                transform=pretrained_vit_transforms,
                                batch_size=32)
```

تفاوت این مدل با سایر مدل‌هایی که تاکنون داشتیم این است که از transform خود این مدل به جای transform ای که خودمان در نظر گرفته بودیم،استفاده شده است.همچنین قابل به ذکر است که اندازه Patch ها در این مدل 16*16 است.

نتایج حاصل از آموزش و ارزیابی این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 0.4680 | train_acc: 0.8109 | test_loss: 1.0314 |
test_acc: 0.5250
Epoch: 2 | train_loss: 0.4403 | train_acc: 0.8304 | test_loss: 1.0499 |
test_acc: 0.5062
Epoch: 3 | train_loss: 0.4339 | train_acc: 0.8259 | test_loss: 1.0343 |
test_acc: 0.5000
Epoch: 4 | train_loss: 0.4426 | train_acc: 0.8326 | test_loss: 1.0142 |
test_acc: 0.5125
Epoch: 5 | train_loss: 0.4362 | train_acc: 0.8326 | test_loss: 1.0346 |
test_acc: 0.5125
Epoch: 6 | train_loss: 0.4375 | train_acc: 0.8246 | test_loss: 1.0279 |
test_acc: 0.5125
Epoch: 7 | train_loss: 0.4445 | train_acc: 0.8189 | test_loss: 1.0457 |
test_acc: 0.5125
Epoch: 8 | train_loss: 0.4264 | train_acc: 0.8326 | test_loss: 1.0521 |
test_acc: 0.5125
Epoch: 9 | train_loss: 0.4315 | train_acc: 0.8268 | test_loss: 1.0271 |
test_acc: 0.5125
Epoch: 10 | train_loss: 0.4432 | train_acc: 0.8202 | test_loss: 1.0451 |
test_acc: 0.5125
Total training time: 99.118 seconds
```

این مدل،در مرحله‌ی آموزش به دقت ۸۲ درصد و در مرحله‌ی ارزیابی،به دقت ۵۱ درصد رسیده است.

مدل vit_b_32

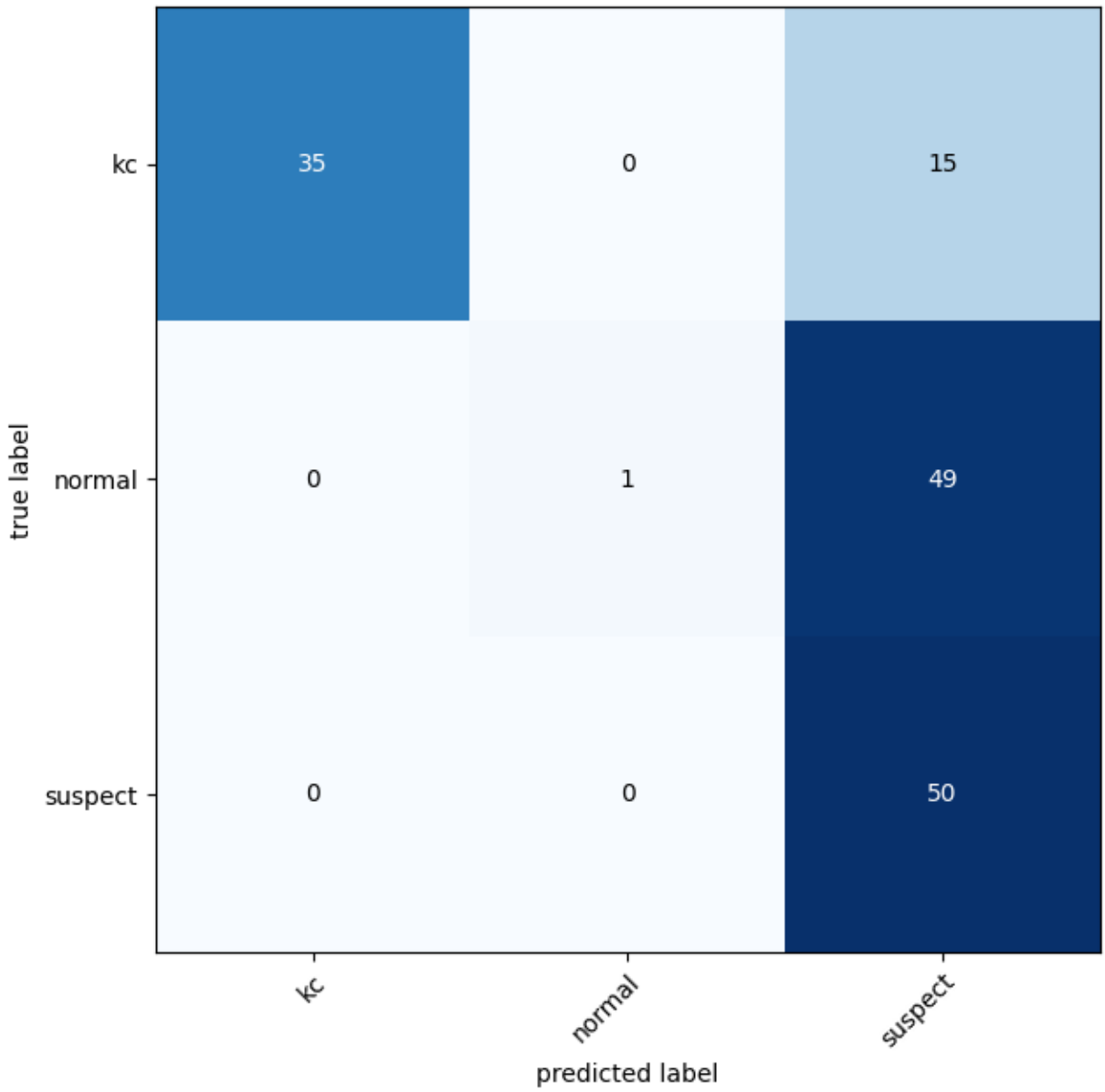
تفاوت این مدل با مدل قبل در این است که اندازه Patch های تولیدشده،32*32 است و به طور کلی از نظر تعداد لایه‌ها مدل بزرگتری است.

از نظر کد،این مدل با مدل قبلی تفاوت زیادی ندارد پس فقط در این‌جا نتایج را بررسی می‌کنیم.

نتایج برای این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 0.8517 | train_acc: 0.6113 | test_loss: 0.8075 |
test_acc: 0.6125
Epoch: 2 | train_loss: 0.6249 | train_acc: 0.7117 | test_loss: 0.7470 |
test_acc: 0.6062
Epoch: 3 | train_loss: 0.5459 | train_acc: 0.7567 | test_loss: 0.8283 |
test_acc: 0.6062
Epoch: 4 | train_loss: 0.5494 | train_acc: 0.7564 | test_loss: 0.8744 |
test_acc: 0.6062
Epoch: 5 | train_loss: 0.5251 | train_acc: 0.7701 | test_loss: 0.8367 |
test_acc: 0.6062
Epoch: 6 | train_loss: 0.4994 | train_acc: 0.7911 | test_loss: 0.8206 |
test_acc: 0.6000
Epoch: 7 | train_loss: 0.4869 | train_acc: 0.7832 | test_loss: 0.8266 |
test_acc: 0.6000
Epoch: 8 | train_loss: 0.4672 | train_acc: 0.8112 | test_loss: 0.7874 |
test_acc: 0.6062
Epoch: 9 | train_loss: 0.4540 | train_acc: 0.8214 | test_loss: 0.8421 |
test_acc: 0.6000
Epoch: 10 | train_loss: 0.4551 | train_acc: 0.8033 | test_loss: 0.8109 |
test_acc: 0.6000
Total training time: 79.225 seconds
```

دقت داده‌های ارزیابی این مدل،۶۰ درصد است.این دقت نسبت به حالت قبل،بهتر شده است.شکل ماتریس درهم‌ریختگی برای این مدل به صورت زیر است:



شکل ۱۲. ماتریس درهم‌ریختگی مدل vit_b_32

همان‌طور که می‌بینیم،این مدل در تشخیص داده‌های کلاس kc تقریبا خوب و در تشخیص داده‌های کلاس suspect خیلی خوب عمل کرده است و ضعف این مدل،در تشخیص داده‌های کلاس normal است.

یک طبقه‌بندی با مدل vit_h_14 هم انجام شده که این مدل با اندازه Patch های 14*14 کار می‌کند.دقت این مدل هم در نهایت ۶۰ درصد بود که به دلیل سنگین بودن زیاد این مدل،فقط به همین شبیه‌سازی اکتفا شده است.

در ادامه سعی می‌شود با ترکیب مدل ViTبا سایر مدل‌ها،عملکرد مدل را بهتر کرد.

ترکیب مدل‌های vit_b_16 و efficientnet_b0

در این قسمت،برای بهتر شدن مدل و حل مشکل عدم توانایی تشخیص داده‌های کلاس normal، مدل vit_b_16 با efficientnet_b0 ترکیب می‌شوند.دلیل استفاده از vit_b_16 به جای vit_b_32 سبک‌تر بودن مدل و استفاده کمتر از memory و RAM می‌باشد.

ترکیب این دو مدل به این صورت است که خروجی‌های دو مدل با هم concatenate می‌شوند و سپس با استفاده از خروجی نهایی،کار طبقه‌بندی انجام می‌شود.

کدی که برای ترکیب این دو مدل استفاده شده است،به صورت زیر است:

```
efficientnet = efficientnet_b0(pretrained=True)
efficientnet.classifier = nn.Identity()

for param in efficientnet.parameters():
    param.requires_grad = False

vit = vit_b_16(pretrained=True)
vit.heads = nn.Identity()

for param in vit.parameters():
    param.requires_grad = False
```

در این قسمت مدل‌ها تعریف شده‌اند.در هر دو مدل،پارامترها منجمد شده‌اند تا حافظه‌ی کمتری مورداستفاده قرار گیرد.

ترکیب این دو مدل به صورت زیر انجام می‌شود:

```
class CombinedModel(nn.Module):
    def __init__(self, efficientnet, vit, feature_dim, vit_dim,
num_classes):
        super(CombinedModel, self).__init__()
        self.efficientnet = efficientnet
        self.vit = vit
        self.combined_dim = feature_dim + vit_dim
        self.classifier = nn.Linear(self.combined_dim, num_classes)
```

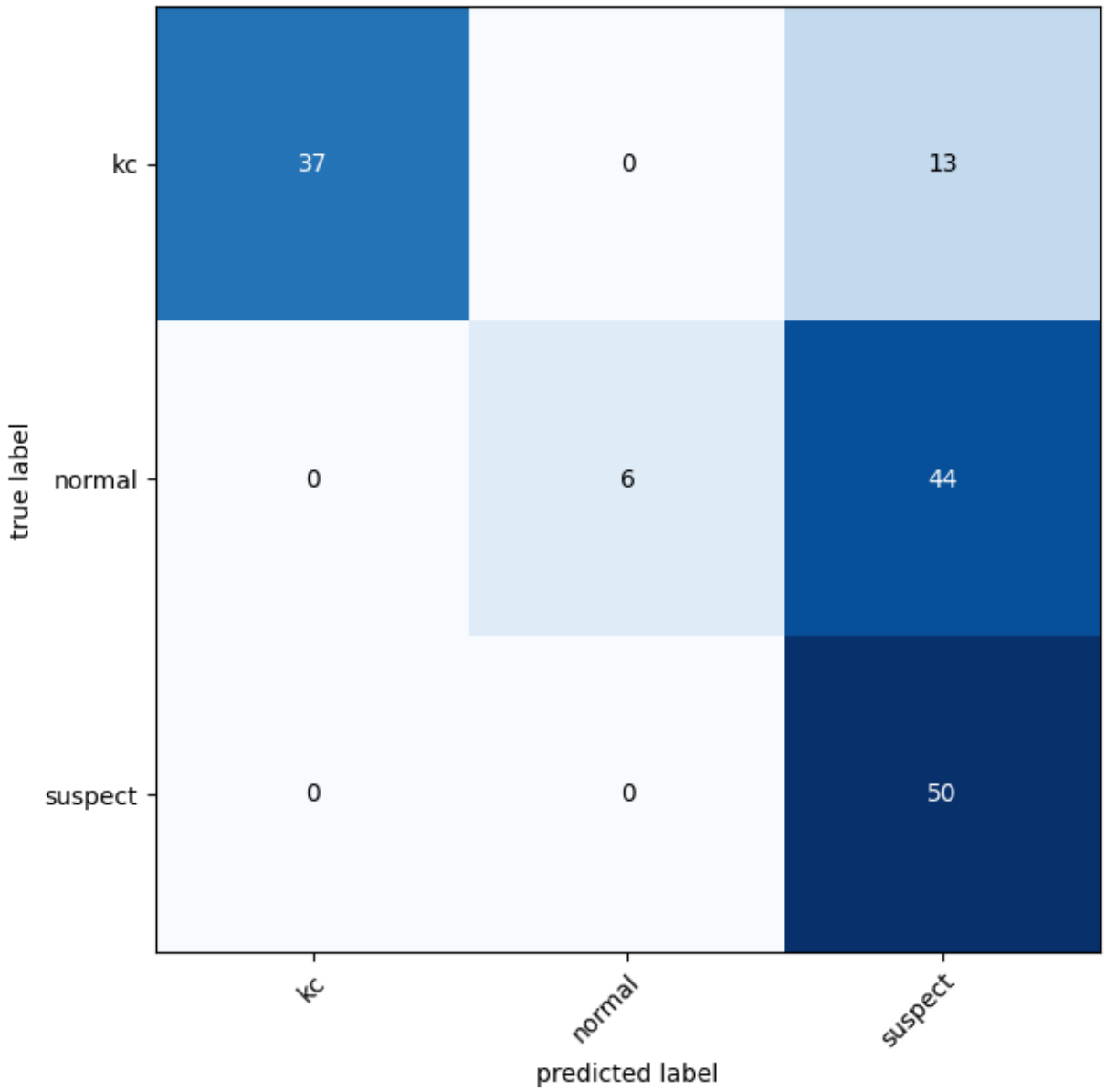
```
    def forward(self, x):
        with torch.no_grad():
            efficientnet_features = self.efficientnet(x)

        with torch.no_grad():
            vit_embeddings = self.vit(x)

        combined_features = torch.cat((efficientnet_features,
vit_embeddings), dim=1)

        output = self.classifier(combined_features)
        return output

combined_model_vit_eff = CombinedModel(efficientnet, vit,
feature_dim=1280, vit_dim=768, num_classes=3).to(device)
```

نتایج حاصل از این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 0.8009 | train_acc: 0.6805 | test_loss: 0.8853 |
test_acc: 0.5250
Epoch: 2 | train_loss: 0.5473 | train_acc: 0.7742 | test_loss: 0.8699 |
test_acc: 0.5813
Epoch: 3 | train_loss: 0.4709 | train_acc: 0.8291 | test_loss: 0.8416 |
test_acc: 0.6125
Epoch: 4 | train_loss: 0.4721 | train_acc: 0.8099 | test_loss: 0.9804 |
test_acc: 0.5750
Epoch: 5 | train_loss: 0.4233 | train_acc: 0.8214 | test_loss: 0.9961 |
test_acc: 0.5938
Epoch: 6 | train_loss: 0.3937 | train_acc: 0.8594 | test_loss: 1.0081 |
test_acc: 0.5813
Epoch: 7 | train_loss: 0.3841 | train_acc: 0.8434 | test_loss: 0.9109 |
test_acc: 0.6062
Epoch: 8 | train_loss: 0.3397 | train_acc: 0.8871 | test_loss: 0.8478 |
test_acc: 0.6375
Epoch: 9 | train_loss: 0.3316 | train_acc: 0.8705 | test_loss: 0.9303 |
test_acc: 0.6125
Epoch: 10 | train_loss: 0.3107 | train_acc: 0.9139 | test_loss: 0.8146 |
test_acc: 0.6438
Total training time: 123.437 seconds
```

همان‌طور که می‌بینیم،این مدل در نهایت در قسمت آموزش به دقت حدود ۹۱درصد و در بخش ارزیابی،به دقت حدود ۶۵ درصد رسیده است.

در شکل زیر ماتریس درهم‌ریختگی را برای این مدل می‌بینیم:



شکل ۱۳. ماتریس درهم‌ریختگی مدل ترکیبی مدل‌های efficientnet_b0 و vit_b_16

همان‌طور که مشاهده می‌شود،این مدل هم اندکی بهبود در تشخیص داده‌های کلاس kc داشته و هم در تشخیص داده‌های کلاس normal بهبود داشته است.

ترکیب مدل‌های efficientnet_b0 و resnet50 و vgg16 و vit_b_16

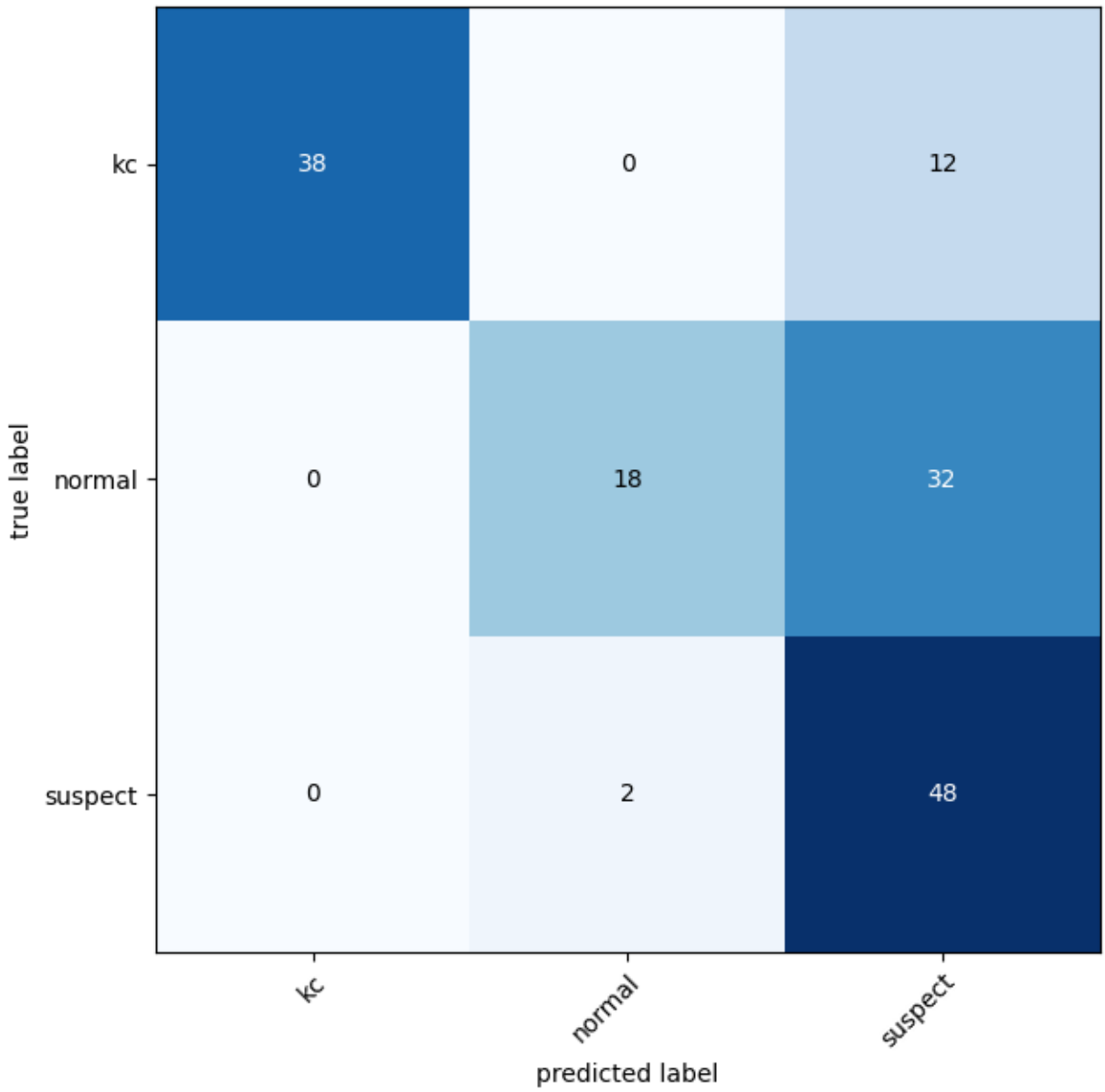
این مدل شبیه به مدل قبل ساخته می‌شود.فقط با این تفاوت که مدل‌های resnet50 و vgg16 اضافه شده‌اند و در طبقه‌بندی به کار گرفته خواهند شد.

نتایج حاصل از این مدل به صورت زیر است:

```
Epoch: 1 | train_loss: 0.7724 | train_acc: 0.6301 | test_loss: 1.9255 |
test_acc: 0.4625
Epoch: 2 | train_loss: 0.7172 | train_acc: 0.6996 | test_loss: 1.2793 |
test_acc: 0.5312
Epoch: 3 | train_loss: 0.5117 | train_acc: 0.8013 | test_loss: 0.6202 |
test_acc: 0.7318
Epoch: 4 | train_loss: 0.5167 | train_acc: 0.7675 | test_loss: 0.8552 |
test_acc: 0.6188
Epoch: 5 | train_loss: 0.4277 | train_acc: 0.7985 | test_loss: 0.8714 |
test_acc: 0.6312
Epoch: 6 | train_loss: 0.4824 | train_acc: 0.7809 | test_loss: 0.6952 |
test_acc: 0.6750
Epoch: 7 | train_loss: 0.3708 | train_acc: 0.8256 | test_loss: 1.1528 |
test_acc: 0.5750
Epoch: 8 | train_loss: 0.3410 | train_acc: 0.8469 | test_loss: 0.6675 |
test_acc: 0.6909
Epoch: 9 | train_loss: 0.3432 | train_acc: 0.8527 | test_loss: 0.9084 |
test_acc: 0.6034
Epoch: 10 | train_loss: 0.3313 | train_acc: 0.8390 | test_loss: 0.6871 |
test_acc: 0.7068
Total training time: 163.677 seconds
```

این مدل در نهایت در قسمت آموزش به دقت ۸۴ درصد و در قسمت ارزیابی به دقت حدود ۷۱ درصد رسیده است.این دقت بیشترین دقتی است که تاکنون داشته ایم.

حال در شکل زیر ماتریس درهم‌ریختگی این مدل را می‌بینیم:



شکل ۱۴. ماتریس درهم‌ریختگی ترکیب ۴مدل

با توجه به این ماتریس می‌بینیم که مدل جدید در تشخیص داده‌های کلاس normal خیلی بهتر شده است.

یک مدل دیگر در نظر گرفته شده است که با استفاده از سه مدل efficientnet_b0 و resnet50 و vgg16 ویژگی از داده‌ها استخراج شود و با استفاده از مدل ViT طبقه‌بندی انجام شود که به دلیل محدودیت RAM نتوانستیم از آن خروجی بگیریم.

ذخیره‌کردن و load کردن مدل ذخیره‌شده

دلیل اهمیت ذخیره کردن مدل

مدل‌هایی که ما پیاده‌سازی می‌کنیم،باید بتوان در یک دستگاه یا نرم‌افزار یا ... دیگر مورداستفاده قرار داد.مثلا فرض کنید مدلی که پیاده‌سازی شده است،قرار است در یک اپلیکیشن موبایل مورداستفاده قرار گیرد.در این شرایط وزن‌های مدل ذخیره می‌شوند و در ساختار برنامه‌نویسی آن اپلیکیشن مورداستفاده قرار خواهد گرفت.پس نتیجه نهایی در پیاده‌سازی یک مدل،زمانی است که ذخیره می‌شود و در یک کار مورد استفاده قرار می‌گیرد.

نحوه‌ی ذخیره‌کردن مدل

کدی که برای ذخیره‌کردن مدل استفاده شده است،به صورت زیر است:

```
train_dataloaders = train_dataloader
num_epochs = [5 , 10]
models = ["effnetb0", "effnetb2"]
experiment_number = 0
dataloader_name = 'org_dataloader'

for epochs in num_epochs:

    for model_name in models:

        experiment_number += 1
        print(f"[INFO] Experiment number: {experiment_number}")
        print(f"[INFO] Model: {model_name}")
        print(f"[INFO] DataLoader: {dataloader_name}")
        print(f"[INFO] Number of epochs: {epochs}")

        if model_name == "effnetb0":
            model = create_effnetb0()
        else:
            model = create_effnetb2()

        loss_fn = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)

        train_write.train_writer(model=model,
              train_dataloader=train_dataloader,
```

```
                test_dataloader=test_dataloader,
                optimizer=optimizer,
                loss_fn=loss_fn,
                epochs=epochs,
                device=device,
                writer=create_writer(experiment_name=dataloader_name,
                                     model_name=model_name,
                                     extra=f"{epochs}_epochs"))

        save_filepath =
f"model_{model_name}_{dataloader_name}_{epochs}_epochs.pth"
        save_model(model=model,
                   target_dir="models",
                   model_name=save_filepath)
        print("-"*50 + "\n")
```

در این کد برای تعداد دوره آموزش،مدل مورداستفاده در آموزش و دیتای استفاده شده در آموزش لیست-هایی در نظر گرفته می‌شود تا به ازای حالت‌های مختلفی از آن‌ها آموزش مدل انجام شود و ذخیره‌سازی انجام شود تا بتوان بهرین مدل را از میان این حالت‌ها انتخاب کرد.

در مدل ما،تنها یک دیتاست وجود دارد.پس آموزش فقط با یک دیتاست انجام می‌شود.اما برای تعداد دوره آموزش دو عدد ۵ و ۱۰ و برای مدل‌ها دو مدل efficeintnetb0 و efficientnetb2 در نظر گرفته شده-اند.البته محدودیتی برای تعداد این حالت‌ها وجود ندارد و می‌توان حالت‌های بیشتری را در نظر گرفت.

در قسمت ذخیره‌ی مدل،فایل‌هایی با نام مرتبط با مدل استفاده شده،دیتاست مورداستفاده و تعداد دوره‌ی آموزش موردنظر ایجاد و هر مدل در فایل مربوط به ویژگی‌های خود ذخیره می‌شود تا برای انتخاب مدل،بدانیم بهترین مدل کدام بوده است.

نتیجه به صورت زیر است:

```
[INFO] Experiment number: 1
[INFO] Model: effnetb0
[INFO] DataLoader: org_dataloader
[INFO] Number of epochs: 5
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2024-05-
03/org_dataloader/effnetb0/5_epochs...
Epoch: 1 | train_loss: 0.9373 | train_acc: 0.5842 | test_loss: 0.8592 |
test_acc: 0.6000
Epoch: 2 | train_loss: 0.7083 | train_acc: 0.6925 | test_loss: 0.7752 |
test_acc: 0.6067
Epoch: 3 | train_loss: 0.6088 | train_acc: 0.7543 | test_loss: 0.7910 |
test_acc: 0.6200
```

```
Epoch: 4 | train_loss: 0.5352 | train_acc: 0.7949 | test_loss: 0.7684 |
test_acc: 0.6467
Epoch: 5 | train_loss: 0.5372 | train_acc: 0.7757 | test_loss: 0.7605 |
test_acc: 0.6467
[INFO] Saving model to: models/model_effnetb0_org_dataloader_5_epochs.pth
--------------------------------------------------

[INFO] Experiment number: 2
[INFO] Model: effnetb2
[INFO] DataLoader: org_dataloader
[INFO] Number of epochs: 5
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2024-05-
03/org_dataloader/effnetb2/5_epochs...
Epoch: 1 | train_loss: 0.9306 | train_acc: 0.5772 | test_loss: 0.8966 |
test_acc: 0.5667
Epoch: 2 | train_loss: 0.7172 | train_acc: 0.7111 | test_loss: 0.7907 |
test_acc: 0.5867
Epoch: 3 | train_loss: 0.6324 | train_acc: 0.7476 | test_loss: 0.7634 |
test_acc: 0.5933
Epoch: 4 | train_loss: 0.5968 | train_acc: 0.7535 | test_loss: 0.7361 |
test_acc: 0.6000
Epoch: 5 | train_loss: 0.5748 | train_acc: 0.7709 | test_loss: 0.7509 |
test_acc: 0.5800
[INFO] Saving model to: models/model_effnetb2_org_dataloader_5_epochs.pth
--------------------------------------------------

[INFO] Experiment number: 3
[INFO] Model: effnetb0
[INFO] DataLoader: org_dataloader
[INFO] Number of epochs: 10
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2024-05-
03/org_dataloader/effnetb0/10_epochs...
Epoch: 1 | train_loss: 0.9373 | train_acc: 0.5842 | test_loss: 0.8592 |
test_acc: 0.6000
Epoch: 2 | train_loss: 0.7083 | train_acc: 0.6925 | test_loss: 0.7752 |
test_acc: 0.6067
Epoch: 3 | train_loss: 0.6088 | train_acc: 0.7543 | test_loss: 0.7910 |
test_acc: 0.6200
Epoch: 4 | train_loss: 0.5352 | train_acc: 0.7949 | test_loss: 0.7684 |
test_acc: 0.6467
Epoch: 5 | train_loss: 0.5372 | train_acc: 0.7757 | test_loss: 0.7605 |
test_acc: 0.6467
Epoch: 6 | train_loss: 0.5036 | train_acc: 0.8039 | test_loss: 0.7773 |
test_acc: 0.6600
Epoch: 7 | train_loss: 0.5160 | train_acc: 0.7919 | test_loss: 0.7661 |
test_acc: 0.6667
Epoch: 8 | train_loss: 0.4656 | train_acc: 0.8041 | test_loss: 0.8018 |
test_acc: 0.6267
Epoch: 9 | train_loss: 0.4821 | train_acc: 0.7968 | test_loss: 0.7572 |
test_acc: 0.6867
Epoch: 10 | train_loss: 0.4556 | train_acc: 0.8129 | test_loss: 0.7901 |
test_acc: 0.6267
[INFO] Saving model to: models/model_effnetb0_org_dataloader_10_epochs.pth
--------------------------------------------------
```

```
[INFO] Experiment number: 4
[INFO] Model: effnetb2
[INFO] DataLoader: org_dataloader
[INFO] Number of epochs: 10
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2024-05-
03/org_dataloader/effnetb2/10_epochs...
Epoch: 1 | train_loss: 0.9306 | train_acc: 0.5772 | test_loss: 0.8966 |
test_acc: 0.5667
Epoch: 2 | train_loss: 0.7172 | train_acc: 0.7111 | test_loss: 0.7907 |
test_acc: 0.5867
Epoch: 3 | train_loss: 0.6324 | train_acc: 0.7476 | test_loss: 0.7634 |
test_acc: 0.5933
Epoch: 4 | train_loss: 0.5968 | train_acc: 0.7535 | test_loss: 0.7361 |
test_acc: 0.6000
Epoch: 5 | train_loss: 0.5748 | train_acc: 0.7709 | test_loss: 0.7509 |
test_acc: 0.5800
Epoch: 6 | train_loss: 0.5165 | train_acc: 0.8105 | test_loss: 0.7481 |
test_acc: 0.5933
Epoch: 7 | train_loss: 0.4986 | train_acc: 0.7784 | test_loss: 0.7354 |
test_acc: 0.6067
Epoch: 8 | train_loss: 0.5097 | train_acc: 0.8180 | test_loss: 0.7474 |
test_acc: 0.5867
Epoch: 9 | train_loss: 0.4574 | train_acc: 0.8370 | test_loss: 0.7508 |
test_acc: 0.6067
Epoch: 10 | train_loss: 0.4806 | train_acc: 0.7943 | test_loss: 0.7525 |
test_acc: 0.6200
[INFO] Saving model to: models/model_effnetb2_org_dataloader_10_epochs.pth
--------------------------------------------------
```

در این خروجی ابتدا درباره‌ی این که چه مدلی استفاده شده است و تعداد دوره‌های آموزش چندتاست اطلاعاتی داده می‌شود.بعد از آن روند آموزش مثل قبل آمده است و اطلاعات درباره کاهش یا افزایش اتلاف و افزایش یا کاهش دقت در طول روند آموزش داده می‌شود و در نهایت با مقدار دقت نهایی می‌توان بهترین مدل را انتخاب کرد.

Load کردن مدل ذخیره‌شده

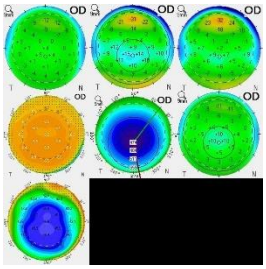
برای load کردن مدل ابتدا همان مدل استفاده شده را تعریف می‌کنیم و سپس مدل ذخیره‌شده را در آن مدل load می‌کنیم.این کار به صورت زیر است:

```
best_model_path =
"/content/models/model_effnetb0_org_dataloader_10_epochs.pth"
best_model = create_effnetb0()
best_model.load_state_dict(torch.load(best_model_path))
```

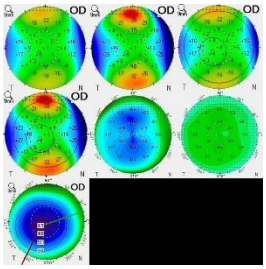
حال تابعی را در نظر می‌گیریم تا تصاویری را گرفته و با استفاده از مدلی که load کردیم،پیش‌بینی روی آن‌ها انجام دهیم و ببینیم که مدل ذخیره‌شده به خوبی کار می‌کند یا خیر.تابع به صورت زیر است:

```
def pred_and_plot_image(model: torch.nn.Module,
                        image_path: str,
                        class_names: List[str],
                        image_size: Tuple[int, int] = (224, 224),
                        transform: torchvision.transforms = None,
                        device: torch.device=device):

    img = Image.open(image_path)
    if transform is not None:
        image_transform = transform
    else:
        image_transform = transforms.Compose([
            transforms.Resize(image_size),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225]),
        ])

    model.to(device)
    model.eval()
    with torch.inference_mode():
      transformed_image = image_transform(img).unsqueeze(dim=0)

      target_image_pred = model(transformed_image.to(device))
    target_image_pred_probs = torch.softmax(target_image_pred, dim=1)
    target_image_pred_label = torch.argmax(target_image_pred_probs, dim=1)

    plt.figure()
    plt.imshow(img)
    plt.title(f"Pred: {class_names[target_image_pred_label]} | Prob:
{target_image_pred_probs.max():.3f}")
    plt.axis(False);
```

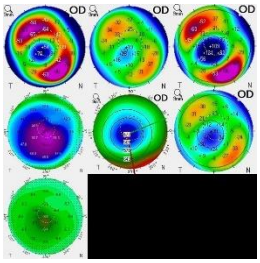
تصاویری که برای تست مدل استفاده شده‌اند به صورت زیر هستند:



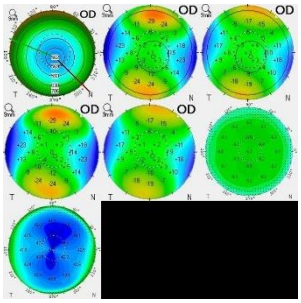
suspect



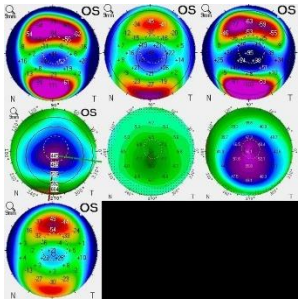
suspect



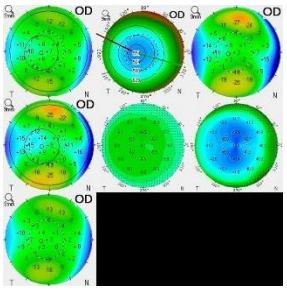
kc



Normal



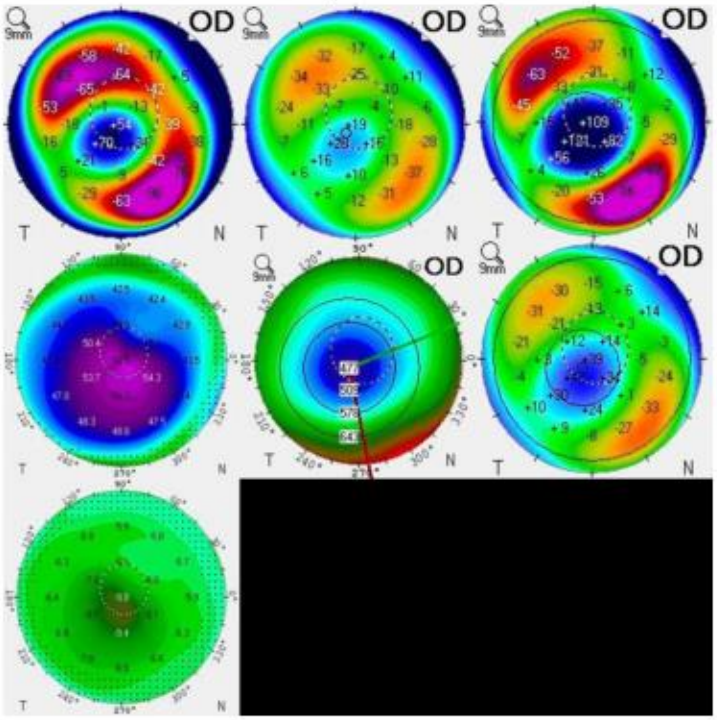
kc



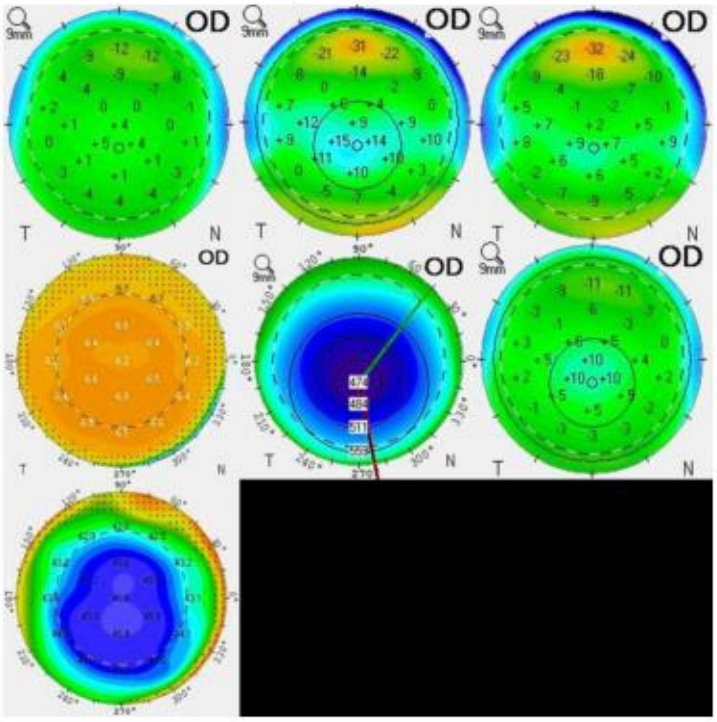
normal

پیش‌بینی‌های مدل به صورت زیر است:

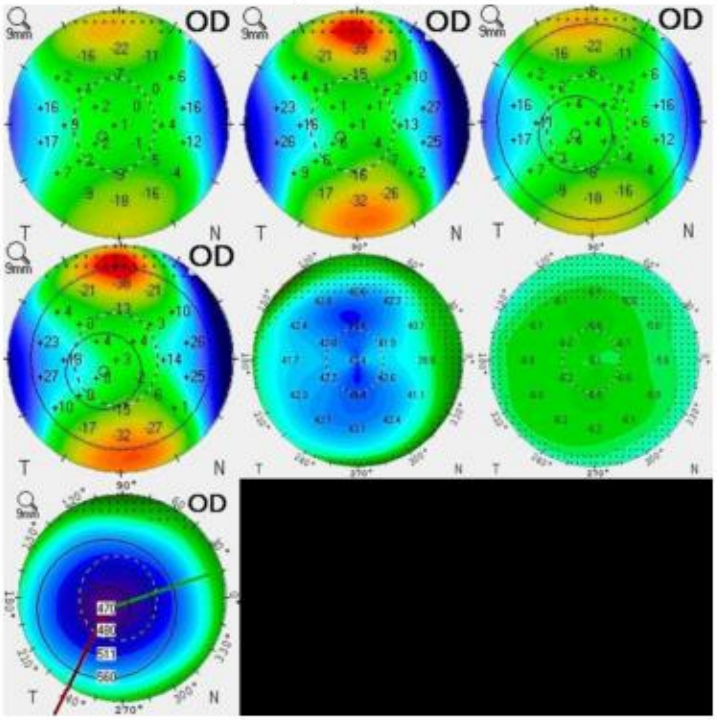
Pred: kc | Prob: 0.935



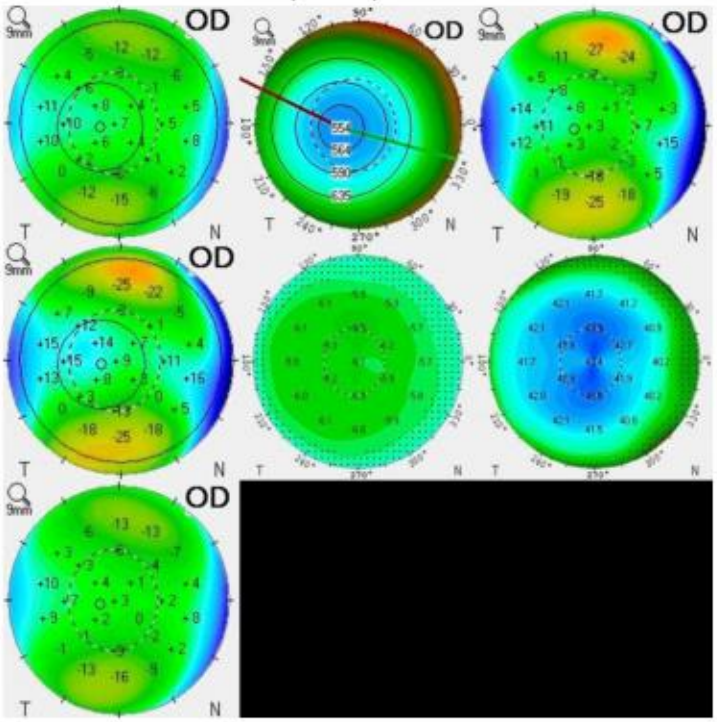
Pred: suspect | Prob: 0.809



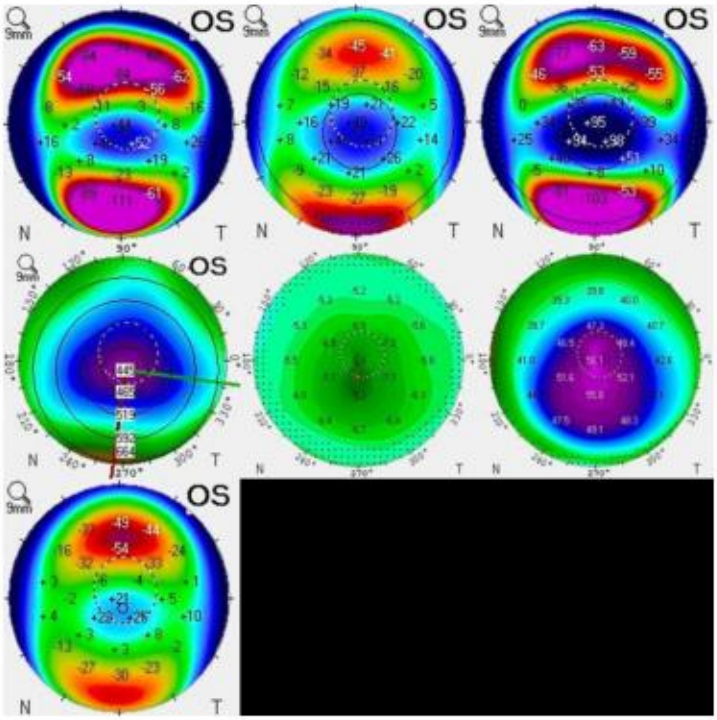
Pred: suspect | Prob: 0.696



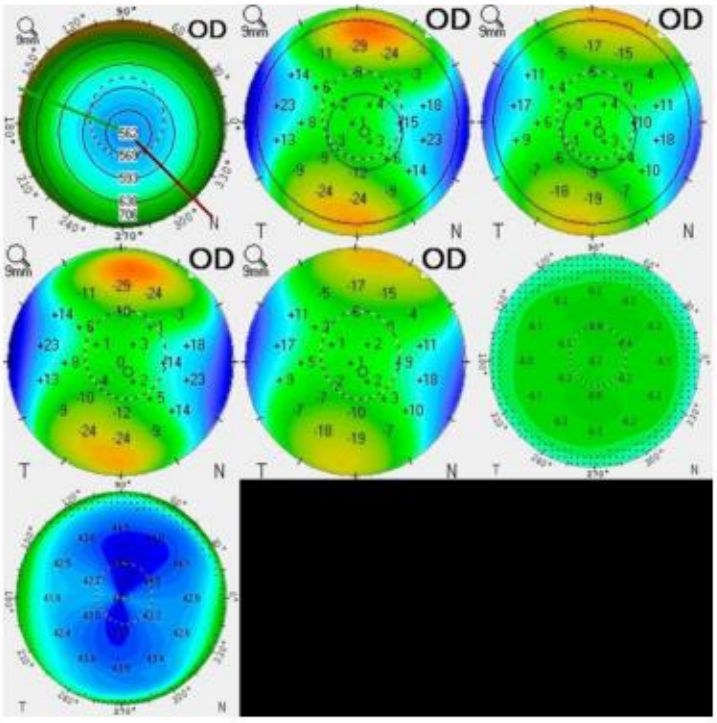
Pred: suspect | Prob: 0.697



Pred: kc | Prob: 0.971



Pred: normal | Prob: 0.492



همان‌طور که می‌بینیم،از ۶ تصویری که برای تست انتخاب شده است،۵ تصویر درست تشخیص داده شده‌اند.

در این پیش‌بینی،علاوه بر پیش‌بینی کلاس احتمالی که با آن این تصویر پیش‌بینی شده است نیز آورده شده است(در واقع این احتمال همان آرگومان ماکزیمم است که از خروجی مدل برای هر داده داریم).می‌بینیم که در احتمالات آورده شده،مقادیر برای کلاس kc، بزرگ است که نشان می‌دهد تصاویر این کلاس با اطمینان بالا تشخیص داده شده‌اند اما برای دو کلاس دیگر،این عدد کمتر است که نشان می‌دهد تشخیص بین دو کلاس suspect و normal دشوارتر بوده است.

پیشنهادات

در این بخش مشکلاتی که می‌تواند در این کار وجود داشته باشد،بررسی می‌شوند و راه‌حل‌هایی برای آن‌ها مطرح خواهند شد.

اولین مسئله کم بودن تعداد داده‌ها است.کم بودن تعداد داده‌ها می‌تواند باعث شود که آموزش به خوبی انجام نشود.اگر تعداد داده‌ها بیشتر شود،احتمالا نتایج بهتری را خواهیم داشت.

نکته‌ی دومی که می‌تواند نتیجه را ارتقا دهد،از بین بردن نویز و قسمت‌های اضافی همه تصاویر و تشکیل دیتاست بدون نویز است.

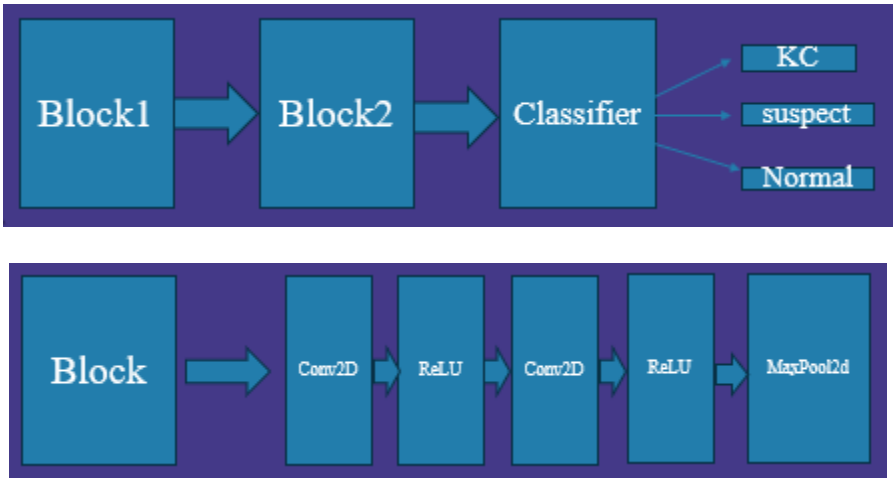
در بخش مدل،اگر این مدل با برخی فیلترها(مثل لحظه‌ی زرنیکه[9]) و برخی دیگر روش‌ها مانند استفاده از encoder و decoder استفاده شود،نتایج بهتری حاصل خواهد شد.

می‌توان ترکیب مدل‌های دیگر حالت‌های مختلف را آزمایش کرد تا مشکل طبقه‌بندی در برخی کلاس‌ها حل شود و به دقت بیشتری دست یافت.

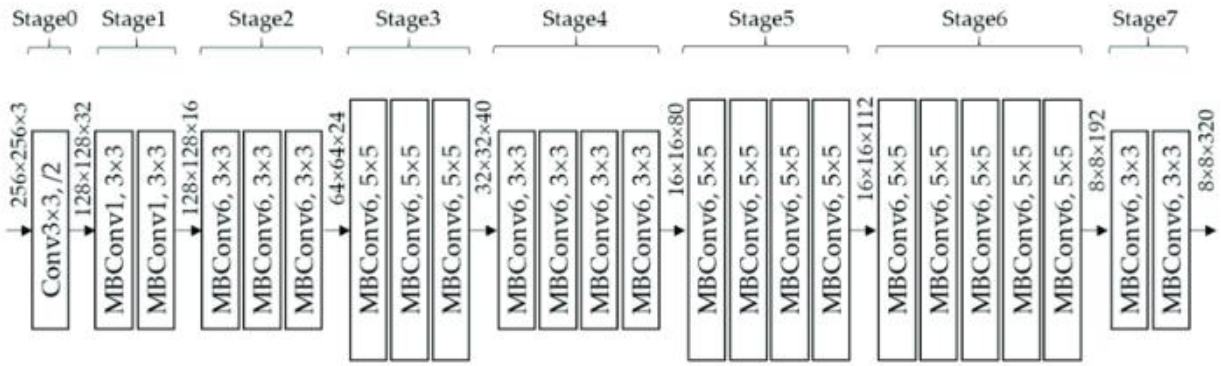
[9] Zernike moment

پیوست‌ها

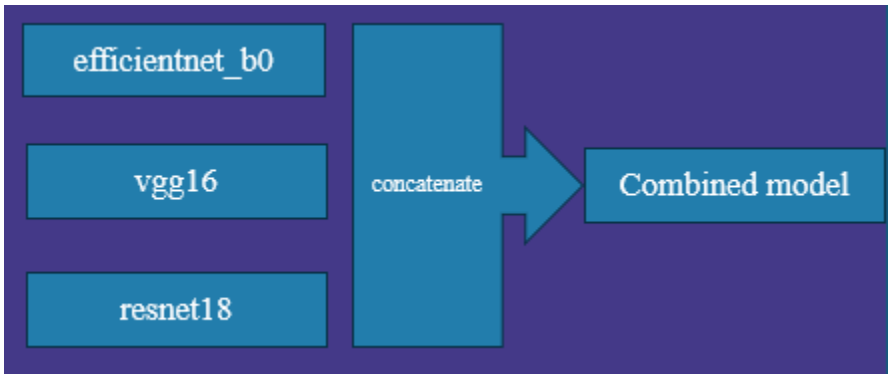
ساختار کلی شبکه‌ی CNN عادی



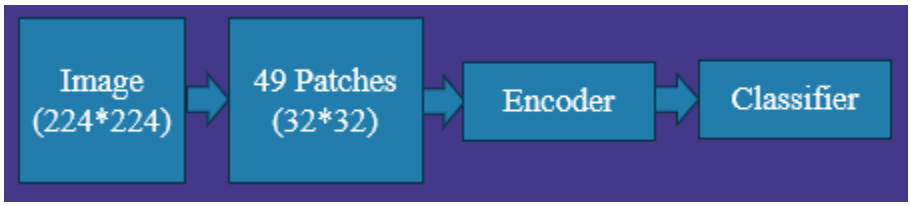
ساختار کلی شبکه‌ی efficientnet_b2



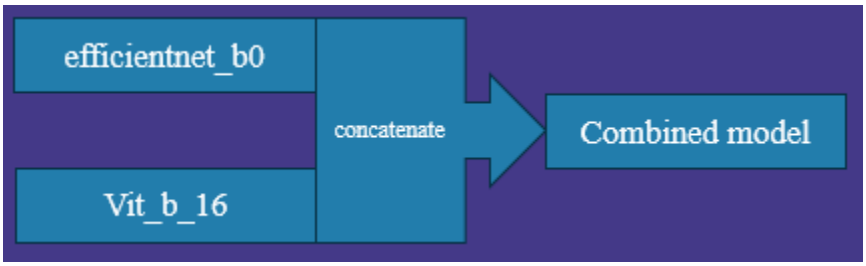
ساختار کلی شبکه‌ی ترکیبی سه مدل efficientnet_b0 و resnet18 و vgg16



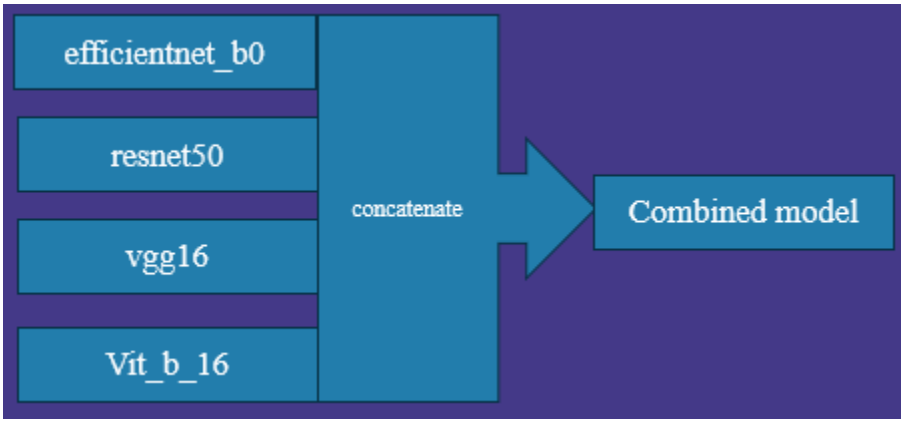
ساختار کلی شبکه‌ی ViT عادی



ساختار کلی شبکه‌ی ترکیبی vit_b_16 و efficientnet_b0



ساختار کلی شبکه‌ی ترکیبی vit_b_16 و efficientnet_b0 و resnet50 و vgg16



لینک کدها

https://colab.research.google.com/drive/1WPK-m1PNlBRek4ffkIjw5DNjEeGMjw7S?usp=sharing

https://colab.research.google.com/drive/1_qLpvbLB5Ati1VP9AjbjtUifamh0pCQN?usp=sharing

https://colab.research.google.com/drive/1VJ9yfFgEyuDorI3PRVSr2hUyvCwJUF1b?usp=sharing