

بنام خدا



دانشگاه تربیت مدرس

دانشکده مهندسی صنایع و سیستم‌ها

گروه فناوری اطلاعات

رشته سیستم‌های اطلاعاتی

گزارش پژوهه درس یادگیری عمیق

(3 CNN, 1 GAN, 1 AutoEncoder) وظیفه محول شده ۵

نگارنده

شايان رخوا و علی راشدی

استاد درس

خانم دکتر خطیبی

نیمسال اول سال تحصیلی ۱۴۰۳-۱۴۰۴

آبان و آذر ۱۴۰۳

۱ - توضیح وظایف محول شده

در این پروژه، ما باید پنج وظیفه را انجام دهیم. در وظیفه اول، سه مدل CNN پیاده‌سازی می‌کنیم که اولین مدل ترکیبی از لایه‌های Convolution و Pooling است. در وظیفه دوم، یک مدل پیش‌آماده وارد می‌شود که تمام پارامترهای آن فریز شده و تنها لایه‌ی کاملاً متصل باشد آموزش داده شود، این کار با توجه به یادگیری انتقالی موسوم به استخراج ویژگی انجام می‌شود. در وظیفه سوم، همان مدل وظیفه دوم را استفاده کرده و علاوه بر لایه کاملاً متصل، بخش انتهایی شبکه نیز باید آموزش ببیند. در وظیفه چهارم، باید یک شبکه عصبی GAN برای تولید داده‌های مصنوعی آموزش دهیم. در نهایت، در تسک پنجم نیز می‌باشد یک شبکه Auto Encoder نیز برای تولید داده‌های مصنوعی باکیفیت آموزش ببیند. ۳ وظیفه اول خیلی به هم مرتبط می‌باشند (برای مثال دارای پیش پردازش‌های یکسانی هستند و در بخش‌هایی همانند تعریف تابع زیان و بهینه ساز و ... نیز تقریباً یکسانند) بنابراین ۳ وظیفه اول با هم و سپس وظایف چهارم و پنجم نیز جداگانه شرح داده می‌شوند.

۲ - مجموعه داده

مجموعه داده‌ای که در این پروژه استفاده می‌شود، شامل تصاویر ریه بیماران است که هدف آن طبقه‌بندی دو کلاسه ریه نرمال و ذات‌الریه می‌باشد. این مجموعه داده شامل ۵۲۳۲ تصویر برای آموزش مدل و ۶۲۴ تصویر برای آزمون است. تصاویر مربوط به ریه بیماران به گونه‌ای انتخاب شده‌اند که بهطور دقیق ویژگی‌های مربوط به وضعیت‌های نرمال و ذات‌الریه را منعکس کنند و این امکان را فراهم می‌آورد که مدل یادگیری عمیق قادر به تمایز میان این دو کلاس باشد.

اهمیت این موضوع به دلیل نقش حیاتی طبقه‌بندی تصاویر پزشکی در تشخیص زودهنگام بیماری‌ها، بهویژه بیماری‌های ریوی مانند ذات‌الریه است. بیماری‌هایی نظیر ذات‌الریه می‌توانند به سرعت پیشرفت کنند و درمان بهموقع آن‌ها می‌تواند جان بیماران را نجات دهد. استفاده از مدل‌های یادگیری عمیق برای تحلیل تصاویر پزشکی این امکان را می‌دهد که فرآیند تشخیص خودکار، سریع‌تر و با دقت بالاتری انجام شود و این امر به پزشکان کمک می‌کند تا در زمان‌های بحرانی تصمیمات دقیقی اتخاذ کنند.

برای آموزش مدل، ۲۵ درصد از ۵۲۳۲ تصویر برای اعتبارسنجی (validation) انتخاب شده‌اند و ۷۵ درصد باقی مانده برای آموزش مدل استفاده می‌شود. این تقسیم‌بندی به این صورت است که ۲۹۲۴ تصویر برای آموزش و ۱۳۰۸ تصویر برای اعتبارسنجی در نظر گرفته شده‌اند. داده‌های آزمون نیز شامل ۶۲۴ تصویر هستند که برای ارزیابی نهایی عملکرد مدل پس از آموزش استفاده خواهند شد. این تقسیم‌بندی کمک می‌کند که مدل به‌طور بهینه آموزش ببیند و در عین حال قابلیت تعیین به داده‌های جدید را داشته باشد. شکل ۱ کد این بخش را نشان میدهد.

این مجموعه داده و فرآیند تقسیم آن به سه بخش اصلی آموزش، اعتبارسنجی و آزمون، به مدل اجازه می‌دهد که به طور دقیق آموزش ببیند، به خوبی بر روی داده‌های جدید ارزیابی شود و در نهایت به دقت بالاتری در تشخیص و طبقه‌بندی تصاویر ریه برسد.

```

▼ train & valid set

[ ] # Load datasets
train_dataset = datasets.ImageFolder(train_folder, transform=augment_transform)
valid_dataset = datasets.ImageFolder(train_folder, transform=main_transform)

# Split train_valid_set into train_set and valid_set
train_portion = 0.75
train_size = int(train_portion * len(train_dataset))
valid_size = len(train_dataset) - train_size

random_seed = 42
torch.manual_seed(random_seed)

train_indices, valid_indices = random_split(range(len(train_dataset)), [train_size, valid_size])

train_set = Subset(train_dataset, train_indices)
valid_set = Subset(valid_dataset, valid_indices)

[ ] len(train_set)
→ 3924

[ ] len(valid_set)
→ 1308

▼ A brief check, again!

[ ] print(f"train_set size: {len(train_set)}")
print(f"valid_set size: {len(valid_set)}")
print(f"test_set size: {len(test_set)}")

→ train_set size: 3924
valid_set size: 1308
test_set size: 624

```

شکل ۱ – تقسیم بندی داده های آموزشی و ارزیابی و آزمون

۳- پیش پردازش و داده سازی

در این بروژه، از فرآیند *Data Augmentation* برای بهبود عملکرد مدل در مرحله آموزش استفاده می شود. هدف اصلی این فرآیند افزایش تنوع داده های آموزشی با اعمال تغییرات تصادفی و کنترل شده بر روی تصاویر است. با این کار، مدل یادگیری عمیق قادر خواهد بود ویژگی های مهم تصاویر را به طور مؤثر تر استخراج کند و از وابستگی به ویژگی های خاص و غیرعمومی جلوگیری شود. این فرآیند کمک می کند که مدل نسبت به داده های جدید و چالش برانگیز تعمیم پذیری بهتری داشته باشد. در مقابل، داده های اعتبارسنجی و آزمون نباید تحت تأثیر تغییرات مصنوعی قرار گیرند؛ چراکه هدف این بخش ها ارزیابی عملکرد مدل بر روی داده های دنیای واقعی است.

شکل شماره ۲ کد مربوط به این بخش را نشان میدهد که در ادامه توضیحات آن را می آوریم: در *augment_transform* که برای داده های آموزشی به کار می رود، از چند تکنیک مهم استفاده شده است. ابتدا تصاویر با استفاده از *T.Resize* به ابعاد مشخص شده تغییر اندازه داده می شوند. سپس *T.RandomHorizontalFlip* با حداکثر چرخش ۳۰ درجه اعمال می شود تا مدل نسبت به تغییرات زاویه ای مقاوم شود. بعد از آن، *T.RandomRotation* با احتمال ۵۰ درصد تصاویر را به صورت افقی معکوس می کند تا انعطاف پذیری مدل نسبت به چیدمان تصاویر افزایش یابد. در ادامه، *T.ColorJitter* به کار

می‌رود که مقادیر روشنایی، کنتراست و رنگ تصاویر را به صورت جزئی تغییر می‌دهد تا مدل در برابر تغییرات جزئی رنگ مقاوم شود. سپس داده‌ها با استفاده از `T.Normalize` به فرمت تنسور تبدیل می‌شوند و در نهایت `T.ToTensor` برای نرمال‌سازی مقادیر پیکسل‌ها با استفاده از میانگین و انحراف معیار از پیش تعريف شده انجام می‌شود. از `Random Erasing` استفاده نشده است زیرا با پاک کردن بخش‌هایی از ریه، طبقه بندی بیش از حد نرمال دشوار می‌شود.

در `main_transform`، که برای داده‌های اعتبارسنجی و آزمون به کار می‌رود، تنها `FRAYINDEHAI` پایه مانند `T.Resize` برای تغییر اندازه، `T.ToTensor` برای تبدیل به فرمت تنسور و `T.Normalize` برای نرمال‌سازی استفاده می‌شوند. هدف این است که داده‌های اعتبارسنجی و آزمون همانند داده‌های دنیای واقعی باقی بمانند و تغییری در ساختار آن‌ها ایجاد نشود تا عملکرد واقعی مدل به طور دقیق ارزیابی شود.

لازم به ذکر است که چون در این پروژه از یادگیری انتقالی (Transfer Learning) استفاده می‌شود، میانگین (mean) و انحراف معیار (std) به کاررفته در فرآیند نرمال‌سازی از مقادیر مربوط به وزن‌های از پیش آموزش دیده مدل‌ها مانند `ImageNet` استفاده می‌شوند. این کار باعث می‌شود که داده‌های ورودی به ساختار مدل هماهنگ باشند و فرآیند یادگیری سریع‌تر و مؤثرتر صورت گیرد. به طور کلی، ترکیب این فرآیندهای `Data Augmentation` و پیش‌پردازش، به افزایش تعیین‌پذیری و عملکرد مدل در تشخیص دقیق داده‌های جدید کمک می‌کند.

شایان ذکر است که تمامی داده‌ها، چه داده‌های آموزشی چه ارزیابی و چه آزمون همگی به سایز ۲۵۶ تغییر داده شدند. تصاویر اولیه دارای سایزهای بزرگ (در حدود ۲۵۰۰ الی ۱۵۰۰) که برای کاربردهای پزشکی، منوط به داشتن سخت افزار و توان محاسباتی بالا مناسب است، لکن برای ما چنین چیزی مقدور نیست. بنابراین تمامی آنها به سایزی نرمال برای کارهای پردازش تصویر (همانند ۲۵۶ و ۱۲۸) تغییر سایز یافتند.

```

[ ] mean = [0.485, 0.456, 0.406]
      std = [0.229, 0.224, 0.225]      # for Normalization (Zero-Centering) - MEAN and STD are used from ImageNet

# This is the data augmentation part. One can modify to satisfy their specific needs and requirements.
# =====

# main_transform
main_transform = T.Compose([
    T.Resize((image_size, image_size)),   # Resizing
    T.ToTensor(),                      # PIL to Tensor
    T.Normalize(mean, std)             # Normalization / Zero-centering
])

# augment_transform
augment_transform = T.Compose([
    T.Resize((image_size, image_size)),   # Resizing
    T.RandomRotation(degrees = 30),       # Random Rotation with some degrees (up to 30 degree)
    T.RandomHorizontalFlip(p = 0.5),       # 50% of training sample will have Random Horizontal Flip
    T.ColorJitter(0.2, 0.2, 0.2),        # Transform image color with random changes (slightly)
    T.ToTensor(),                      # PIL to Tensor
    T.Normalize(mean, std),             # Normalization / Zero-centering
    # RandomErasing is NOT a good option in here
])

```

شکل ۲ – پیش‌پردازش داده و داده سازی

نکته شایان ذکر آنکه در این گزارش ممکن است تصاویر و اشکال بر حسب نیاز موضوع بیانند و در کد، ساختار کد را دنبال کنند. برای مثال شکل ۲ (پیش پردازش داده) قبل از `Creating Tensor Dataset` تعریف شده است زیرا باید مشخص شود که کدامیں داده‌ها `augment_transform` و کدامیں داده‌ها `main_transform` دارند.

۴- توزیع داده ها

توزیع داده ها در مجموعه های مختلف یکی از نکات حیاتی در آموزش مدل های یادگیری عمیق است. عدم یکنواختی و توزیع نامتوازن داده ها به خصوص در مجموعه آموزش می تواند فرآیند یادگیری مدل را به طور قابل توجهی تحت تأثیر قرار دهد. زمانی که یک کلاس به طور قابل توجهی بیشتر از کلاس دیگر باشد، مدل تمایل پیدا می کند که توجه بیشتری به آن کلاس غالب داشته باشد و در نتیجه عملکرد مدل در تشخیص کلاس های کم تعداد تر کاهش می یابد. همان طور که در شکل ۳ مشاهده می شود، در مجموعه آموزش، کلاس ذاتالریه (PNEUMONIA) با ۲۹۰۷ نمونه تقریباً ۷۴ درصد از داده ها را به خود اختصاص داده است در حالی که کلاس نرمال (NORMAL) تنها ۲۶ درصد از داده ها را شامل می شود. این عدم توازن می تواند منجر به سوگیری مدل و کاهش دقت آن در طبقه بندی کلاس نرمال شود.

با وجود این عدم توازن در داده های آموزشی، ما از روش های OverSampling یا UnderSampling برای یکسان سازی توزیع داده ها استفاده نکردیم، چرا که این روش ها در برخی موارد ممکن است اطلاعات موجود در داده ها را کاهش داده یا با ایجاد نمونه های تکراری و مصنوعی کیفیت آموزش را تحت تأثیر قرار دهند. به جای آن، برای بهبود تعادل یادگیری و جلوگیری از سوگیری مدل به کلاس غالب، تابع زیان (LOSS) (Function) را به گونه ای بهینه کردیم که با دادن وزن های معکوس به هر کلاس، توجه برابری به کلاس های مختلف داده شود. در این روش، کلاس های کم تعداد وزن بیشتری در محاسبه زیان دریافت می کنند و به این ترتیب مدل یاد می گیرد که اهمیت بیشتری به کلاس های کم نمونه تر داده و عملکرد کلی آن بهبود یابد.

Counts for Train Set:		
Train Set => Class: NORMAL	Index: 0	Count: 1017
Train Set => Class: PNEUMONIA	Index: 1	Count: 2907
NOTE: The total number of data in Train Set is: 3924		
Portion of class 1 to class 0: 2.86		
Counts for Validation Set:		
Validation Set => Class: NORMAL	Index: 0	Count: 332
Validation Set => Class: PNEUMONIA	Index: 1	Count: 976
NOTE: The total number of data in Validation Set is: 1308		
Portion of class 1 to class 0: 2.94		
Counts for Test Set:		
Test Set => Class: NORMAL	Index: 0	Count: 234
Test Set => Class: PNEUMONIA	Index: 1	Count: 390
NOTE: The total number of data in Test Set is: 624		
Portion of class 1 to class 0: 1.67		

شکل ۳ - توزیع داده ها

۵- دیتالودر

برای بارگذاری داده ها و آماده سازی آن ها برای آموزش مدل، از کلاس DataLoader استفاده کردیم که این امکان را فراهم می کند تا داده ها به صورت دسته ای (Batch) و تصادفی (Shuffle) بارگذاری شوند. در اینجا (شکل ۴)، مجموعه های آموزشی، اعتبار سنجی و آزمون با اندازه دسته ای برابر ۱۶ مقدار دهی شده اند تا پردازش داده ها به صورت بهینه و متعادل انجام شود. برای مثال، با استفاده از دستور next(iter(train_loader)) یک دسته از داده های آموزشی بارگذاری شده و مشاهده می شود که شکل ورودی X معادل [16, 3, 256, 256] است که نشان دهنده ۱۶ تصویر با ۳ کanal (RGB) و ابعاد ۲۵۶x۲۵۶ می باشد. خروجی Uniz که برچسب های مربوط به تصاویر را در بر می گیرد، دارای ابعاد [16, 3] است که نشان می دهد هر تصویر یک برچسب دارد. این ساختار سازمان یافته به مدل کمک می کند تا داده ها را به صورت کارآمد در طول فرآیند آموزش پردازش کند.

▼ Data Loader + Comprehension

```
[ ] train_loader = DataLoader(train_set , batch_size=16 , shuffle=True)
valid_loader = DataLoader(valid_set , batch_size=16 , shuffle=True)
eval_loader = DataLoader(test_set , batch_size=16 , shuffle=True)

# train_loader
# one batch / next batch
x , y = next(iter(train_loader))      # just one single batch from train_loader

# some info
print(f"Image size is {image_size}*{image_size}")
print("")

print("For Train_loader:")
print(f"x.shape is: {x.shape}")      # must be: [BS*3*image_size*image_size]
print(f"y.shape is: {y.shape}")      # must be: [BS]
print("")

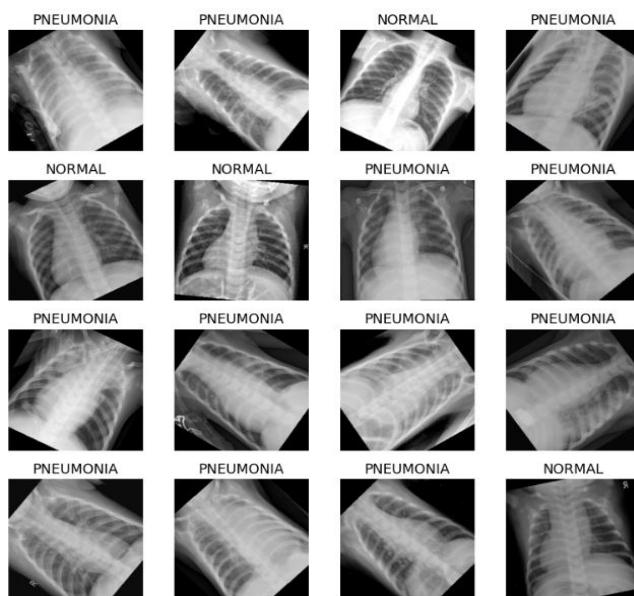


```

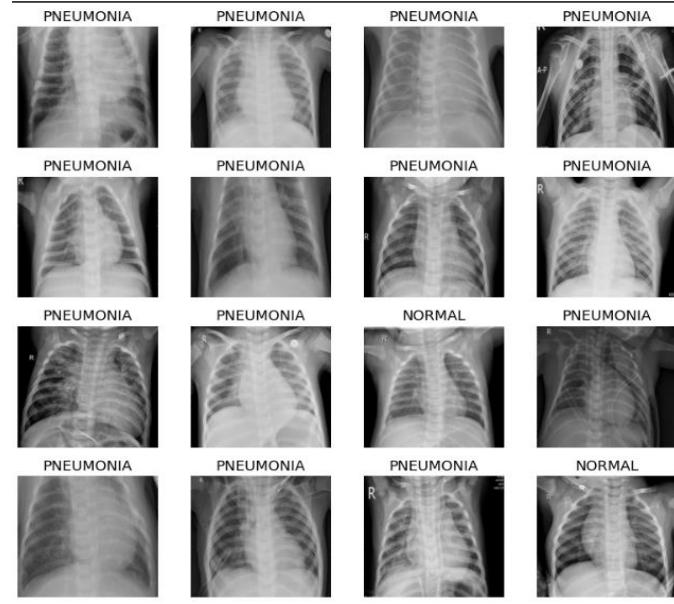
شکل ۴ - دیتالودر

۶- مصورسازی داده های هر سه مجموعه آموزش، ارزیابی و آزمون

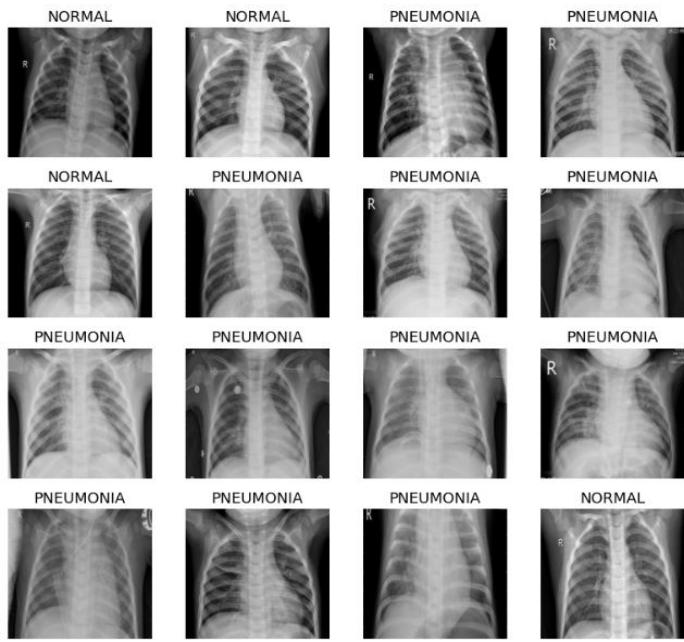
در اشکال ۵ الی ۷ داده های مجموعه آموزش، ارزیابی و آزمون مصور شده اند. این امر به دلیل اطلاع از درست انجام شدن داده سازی، چک کردن مجدد آنها، اطمینان از سایز تصویر و همچنین نبود بود ویژگی هایی خاص که بتواند ساختار تصویر را تحت تاثیر قرار دهد، حائز اهمیت است. همانطور که مشاهده میشود، وجود تصاویر چرخیده شده و فیلیپ شده نشان دهنده درست بودن اعمال augment_transform در بخش train هستند. ضمنا تصاویر کوچک نشان داده شده اند اما در اصل سایز بالای (۲۵۶) دارند.



شکل ۵ - ۱۰ بج از دیتالودر آموزشی، همراه با اعمال augment_transform



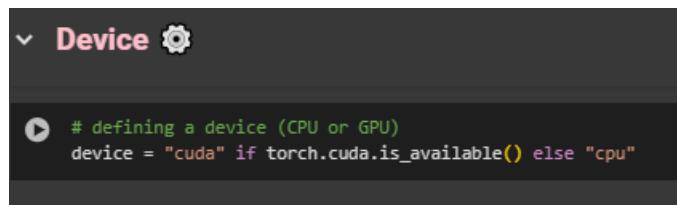
شکل ۱-۶ ۱ بج از دیتالودر ارزیابی، همراه با اعمال main_transform



شکل ۱-۷ ۱ بج از دیتالودر آزمون، همراه با اعمال main_transform

-۷ دستگاه (Device)

با توجه به استفاده از پایتورج، باید Device را مشخص کنیم. شکل ۸ کد نوشته شده را بگونه‌ای نشان میدهد که منعطف باشد و هم برای CPU و هم برای GPU مناسب باشد. البته برای این کارها GPU ضروری است.



شکل - ۸

۸- معماری مدل ۱

مدل ۱ یا همان وظیفه محول شده اول پیاده سازی یک CNN از پایه است. این مدل شبکه عصبی کانولوشن (CNN) برای طبقه بندی تصاویر به صورت دودویی طراحی شده و از ترکیب لایه های Convolution، Batch Normalization، ReLU و MaxPooling استفاده می کند. ورودی شبکه تصاویر سه کanalه (RGB) با ابعاد ۲۵۶x۲۵۶ است. در اولین گام، از لایه Conv2d برای افزایش تعداد کanalها از ۳ به ۶۴ استفاده شده و با BatchNorm2d ترمال سازی انجام می شود تا سرعت همگرایی مدل بهبود یابد.تابع فعال سازی ReLU برای افزودن غیرخطی بودن شبکه به کار گرفته شده و سپس با استفاده از MaxPool2d ابعاد ویژگی ها به نصف کاهش می یابد. این روند با افزایش تدریجی تعداد کanalها از ۶۴ به ۱۲۸، سپس ۱۲۸ به ۲۵۶ و در نهایت ۲۵۶ به ۵۱۲ تکرار می شود. در هر مرحله، دو لایه کانولوشن برای استخراج ویژگی های پیچیده تر استفاده می شوند و پس از هر دو لایه، با MaxPooling ابعاد کاهش می یابد.

پس از عبور از لایه های کانولوشن، لایه AdaptiveAvgPool2d خروجی را به ابعاد ۱x۱ در هر کanal فشرده می کند. این لایه باعث می شود ویژگی های استخراج شده از هر کanal بدون توجه به اندازه تصویر به طور ثابت خلاصه شوند. در ادامه، خروجی ها توسط لایه Flatten به برداری یک بعدی با طول ۵۱۲ تبدیل می شوند تا برای لایه Fully Connected آماده شوند. این لایه در نهایت یک خروجی تولید می کند که برای طبقه بندی دودویی استفاده خواهد شد. لازم به ذکر است که از تابع فعال سازی Sigmoid استفاده نشده و به جای آن، در تابع زیان BCEWithLogitsLoss این عملیات مدیریت می شود که باعث افزایش پایداری محاسبات و بهبود کارایی مدل در پیش بینی خروجی می شود. ساختار مدل پیشنهادی در زیر آمده است که از CNN های اولیه همانند VGG الهام گرفته شده است.

```
def CNN():
    network = nn.Sequential(
        nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        # BS*3*256*256 => BS*64*256*256
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        # BS*64*256*256 => BS*64*256*256
        # MaxPOOL (size reduction)
        nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
        # BS*64*256*256 => BS*64*128*128
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
```

```
        nn.BatchNorm2d(128),  
        nn.ReLU(),  
        # BS*64*128*128 => BS*128*128*128  
        nn.Conv2d(in_channels=128 , out_channels=128 , kernel_size=(3,3) , stride=(1,1) , padding=(1,1)),  
        nn.BatchNorm2d(128),  
        nn.ReLU(),  
        # BS*128*128 => BS*128*128*128  
        # MaxPOOL (size reduction)  
        nn.MaxPool2d(kernel_size=(2,2) , stride=(2,2)),  
        # BS*128*128*128 => BS*128*64*64  
        nn.Conv2d(in_channels=128 , out_channels=256 , kernel_size=(3,3) , stride=(1,1) , padding=(1,1)),  
        nn.BatchNorm2d(256),  
        nn.ReLU(),  
        # BS*128*64*64 => BS*256*64*64  
        nn.Conv2d(in_channels=256 , out_channels=256 , kernel_size=(3,3) , stride=(1,1) , padding=(1,1)),  
        nn.BatchNorm2d(256),  
        nn.ReLU(),  
        # BS*256*64*64 => BS*256*64*64  
        # MaxPOOL (size reduction)  
        nn.MaxPool2d(kernel_size=(2,2) , stride=(2,2)),  
        # BS*256*64*64 => BS*256*32*32  
        nn.Conv2d(in_channels=256 , out_channels=512 , kernel_size=(3,3) , stride=(1,1) , padding=(1,1)),  
        nn.BatchNorm2d(512),  
        nn.ReLU(),  
        # BS*256*32*32 => BS*512*32*32  
        nn.Conv2d(in_channels=512 , out_channels=512 , kernel_size=(3,3) , stride=(1,1) , padding=(1,1)),  
        nn.BatchNorm2d(512),  
        nn.ReLU(),  
        # BS*512*32*32 => BS*512*32*32  
        nn.AdaptiveAvgPool2d(output_size=(1,1)) ,      # AdaptiveAvgPool (n*n) => (1*1)  
        # BS*512*32*32 => BS*512*1*1  
        # FLATTEN  
        nn.Flatten(),
```

```

# BS*512*1*1 => BS*512

# Fully Connected Layer

nn.Linear(in_features=512, out_features=1, bias=True)

# NOTE => I do NOT use Sigmoid in here. Instead, I will use "BCEWithLogitsLoss" for handing the Sigmoid.

()

return network

```

برای درک بهتر، ساختار مدل در شکل ۹ نیز آمده است.

```

def CNN():
    network = nn.Sequential(
        nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        # BS*3*256*256 => BS*64*256*256
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        # BS*64*256*256 => BS*64*256*256
        nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
        # BS*64*256*256 => BS*64*128*128
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        # BS*64*128*128 => BS*128*128*128
        nn.Conv2d(in_channels=128, out_channels=128, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        # BS*128*128*128 => BS*128*128*128
        nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
        # BS*128*128*128 => BS*128*64*64
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        # BS*128*64*64 => BS*256*64*64
        nn.Conv2d(in_channels=256, out_channels=256, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        # BS*256*64*64 => BS*256*64*64
        nn.MaxPool2d(kernel_size=(2,2), stride=(2,2)),
        # BS*256*64*64 => BS*256*32*32
        nn.Conv2d(in_channels=256, out_channels=512, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        # BS*256*32*32 => BS*512*32*32
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        # BS*512*32*32 => BS*512*32*32
        nn.AdaptiveAvgPool2d(output_size=(1,1)),                                # AdaptiveAvgPool (n*n) => (1*1)
        # BS*512*32*32 => BS*512*1*1
        # FLATTEN
        nn.Flatten(),
        # BS*512*1*1 => BS*512
        # Fully Connected Layer
        nn.Linear(in_features=512, out_features=1, bias=True)
        # NOTE => I do NOT use Sigmoid in here. Instead, I will use "BCEWithLogitsLoss" for handing the Sigmoid.
    )

```

شکل ۱۰ - ساختار و کدنویسی CNN مدل نوشته شده از پایه

۹- معماری و تعداد پارامترهای مدل ۱

با استفاده از دستور (`model.summary()`) معماری و تعداد پارامترهای مدل ۱ در شکل ۱۱ نشان داده شده اند. عدد منفی ۱ چیز خاصی را بیان نمی کندو بجای آن میتوان `Batch Size` را قرار داد.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 256, 256]	1,792
BatchNorm2d-2	[-1, 64, 256, 256]	128
ReLU-3	[-1, 64, 256, 256]	0
Conv2d-4	[-1, 64, 256, 256]	36,928
BatchNorm2d-5	[-1, 64, 256, 256]	128
ReLU-6	[-1, 64, 256, 256]	0
MaxPool2d-7	[-1, 64, 128, 128]	0
Conv2d-8	[-1, 128, 128, 128]	73,856
BatchNorm2d-9	[-1, 128, 128, 128]	256
ReLU-10	[-1, 128, 128, 128]	0
Conv2d-11	[-1, 128, 128, 128]	147,584
BatchNorm2d-12	[-1, 128, 128, 128]	256
ReLU-13	[-1, 128, 128, 128]	0
MaxPool2d-14	[-1, 128, 64, 64]	0
Conv2d-15	[-1, 256, 64, 64]	295,168
BatchNorm2d-16	[-1, 256, 64, 64]	512
ReLU-17	[-1, 256, 64, 64]	0
Conv2d-18	[-1, 256, 64, 64]	590,080
BatchNorm2d-19	[-1, 256, 64, 64]	512
ReLU-20	[-1, 256, 64, 64]	0
MaxPool2d-21	[-1, 256, 32, 32]	0
Conv2d-22	[-1, 512, 32, 32]	1,180,160
BatchNorm2d-23	[-1, 512, 32, 32]	1,024
ReLU-24	[-1, 512, 32, 32]	0
Conv2d-25	[-1, 512, 32, 32]	2,359,808
BatchNorm2d-26	[-1, 512, 32, 32]	1,024
ReLU-27	[-1, 512, 32, 32]	0
AdaptiveAvgPool2d-28	[-1, 512, 1, 1]	0
Flatten-29	[-1, 512]	0
Linear-30	[-1, 1]	513
<hr/>		
Total params: 4,689,729		
Trainable params: 4,689,729		
Non-trainable params: 0		
<hr/>		
Input size (MB): 0.75		
Forward/backward pass size (MB): 374.01		
Params size (MB): 17.89		
Estimated Total Size (MB): 392.65		
<hr/>		

شکل ۱۱- معماری و تعداد پارامترهای مدل شماره ۱

نهایتاً نیز با دستور (`model.to(device)`) مدل پیشنهادی به `Device` منتقل می شود.

۱۰-تابع زبان بهینه شده (Modified Loss Function)

در این پروژه به دلیل عدم توازن داده‌ها در مجموعه آموزشی، توجه ویژه‌ای به اصلاح تابع زیان (Loss Function) شده است. همان‌طور که مشخص است، کلاس نرمال (NORMAL) تنها ۲۶ درصد از داده‌های آموزشی را شامل می‌شود، در حالی که کلاس ذات‌الریه (PNEUMONIA) با ۷۴ درصد سهم بیشتری دارد. این عدم توازن می‌تواند باعث شود که مدل به کلاس غالب (PNEUMONIA) تمایل پیدا کند و عملکرد آن در تشخیص کلاس نرمال کاهش یابد. برای رفع این مشکل، وزن‌های معکوسی به هر کلاس اختصاص داده شد تا میزان توجه مدل به کلاس‌ها بر اساس توزیع داده‌ها متعادل شود. به این ترتیب، اهمیت بیشتری به کلاس نرمال داده شده و مدل از سوگیری نسبت به کلاس غالب جلوگیری می‌کند.

در پیاده‌سازی این اصلاح، تابع `BCEWithLogitsLoss` برای محاسبه زیان مورد استفاده قرار گرفت. وزن کلاس‌ها به‌طور دستی محاسبه شد و نسبت وزن کلاس نرمال به ذات‌الریه برابر $0.26 / 0.74 = 0.35$ است. با استفاده از پارامتر `pos_weight` در تابع `BCEWithLogitsLoss`، وزن بیشتری به کلاس مثبت (NORMAL) داده می‌شود تا توجه به کلاس‌های کمتر نمایش داده شده (PNEUMONIA) به‌طور متناسب افزایش یابد. این روش نه تنها نیاز به `OverSampling` یا `UnderSampling` را برطرف می‌کند بلکه از تکرار یا حذف داده‌ها جلوگیری کرده و به مدل اجازه می‌دهد تا تعادل لازم را در یادگیری هر دو کلاس بقرار کند.

▼ MODIFIED loss function:

```
[ ] # Explanation using print statements
print("ATTENTION!")
print("In our training set, 26% of the data belongs to the NORMAL class (index 0),")
print("while the remaining 74% belongs to the PNEUMONIA class (index 1).")
print("Since the dataset is imbalanced, with NORMAL being underrepresented, the loss function")
print("needs to be modified to assign more importance to the NORMAL class so that")
print("the model's performance is not biased toward the majority class (PNEUMONIA).")
```

☞ ATTENTION!

```
In our training set, 26% of the data belongs to the NORMAL class (index 0),
while the remaining 74% belongs to the PNEUMONIA class (index 1).
Since the dataset is imbalanced, with NORMAL being underrepresented, the loss function
needs to be modified to assign more importance to the NORMAL class so that
the model's performance is not biased toward the majority class (PNEUMONIA).
```

```
[ ] # Calculate pos_weight for PNEUMONIA (positive class) based on class imbalance
pos_weight = torch.tensor([0.26 / 0.74], dtype=torch.float32).to(device) # NORMAL : PNEUMONIA ratio

# Modify the BCEWithLogitsLoss to include pos_weight
loss_function = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
```

شکل ۱۲ - تابع زیان بهینه شده

نکته مهم: نکته قابل توجه در این پروژه این است که به دلیل ماهیت آموزشی و تمرکزی که بر تعادل یادگیری داریم، وزن‌های کلاس‌ها به‌طور برابر در نظر گرفته شده‌اند تا مدل بتواند به‌طور یکسان به هر دو کلاس توجه کند. اما در کاربردهای عملی، اهمیت کلاس‌ها باید با توجه به زمینه و هزینه خطای پیش‌بینی تعیین شود. برای مثال، در تشخیص بیماری‌های حساس مانند ذات‌الریه، پیش‌بینی اشتباه یک نمونه نرمال به‌عنوان ذات‌الریه ممکن است منجر به انجام آزمایش‌ها و درمان‌های غیرضروری شود که هزینه‌بر است، اما این خطأ در مقایسه با پیش‌بینی اشتباه یک بیمار ذات‌الریه به‌عنوان نرمال که ممکن است منجر به مرگ بیمار شود، بسیار ناچیز‌تر است. در چنین مواردی، وزن بیشتری به کلاس ذات‌الریه داده می‌شود تا مدل حساسیت بیشتری نسبت به این کلاس داشته باشد و از وقوع این نوع خطای پرهزینه جلوگیری کند.

مثال‌های دیگری نیز در حوزه‌های مختلف می‌توان یافت. برای مثال، در تشخیص تقلب در تراکنش‌های مالی، بهتر است مدل در پیش‌بینی اشتباه یک تراکنش معتبر به عنوان تقلب (خطای مثبت کاذب) حساس‌تر باشد، زیرا در صورت نادیده گرفتن یک تراکنش تقلبی، ممکن است خسارات مالی قابل توجهی به وجود آید. در حوزه امنیت سایبری نیز، پیش‌بینی اشتباه یک فعالیت مخرب به عنوان فعالیت عادی بسیار خطرناک است، چراکه می‌تواند منجر به نفوذ به سیستم‌ها و از دست رفتن داده‌های حساس شود. بنابراین، هزینه خطأ و کاربرد مدل نقش تعیین‌کننده‌ای در تنظیم وزن کلاس‌ها دارد و باید در هر پروژه متناسب با شرایط و نیازها تعیین شود.

۱۱- بهینه ساز و نرخ یادگیری کاهنده با ایپاک

برای بهینه‌سازی مدل از الگوریتم SGD (Stochastic Gradient Descent) استفاده شده است که یکی از پرکاربردترین بهینه‌سازها در یادگیری عمیق به شمار می‌رود. نرخ یادگیری اولیه (initial_lr) برابر ۰.۰۱ در نظر گرفته شده که به عنوان مهم‌ترین هایپرپارامتر برای کنترل سرعت بهروزرسانی وزن‌ها استفاده می‌شود. البته این نرخ یادگیری ۰.۰۱ بهترین نرخ یادگیری برای این شبکه با این معماری یافته شد. در اصل، باید بگوییم که ۰.۰۳ نرخ یادگیری ۰.۰۰۱ و ۰.۰۰۰۱ آزمایش شدند و ۰.۰۰۱ از همگی بهتر یافته شد. نرخ یادگیری ۰.۰۰۰۱ بزرگ بود و سریع به نوسان می‌انجامید و نرخ یادگیری ۰.۰۰۱ نیز کند یافت شد که اصلاً مناسب نیست زیرا با LR_Schduler که مدام نیز کمتر می‌شود، سیار کند می‌شود. علاوه بر این، برای جلوگیری از نوسانات و تسريع همگرایی، از Momentum برابر ۰.۹ استفاده شده که وزن بهروزرسانی‌های قبلی را در بهروزرسانی‌های فعلی تأثیر می‌دهد و در نتیجه مسیر حرکت به سمت بهینه محلی هموارتر و سریع‌تر می‌شود. همچنین برای کنترل پیچیدگی Nesterov مدل و جلوگیری از بیش‌برازش (Overfitting)، از Weight Decay به مقدار ۱e-۵ استفاده شده است. در نهایت، Momentum فعال شده که نسخه‌ای بهبود یافته از Momentum محسوب می‌شود و به مدل کمک می‌کند تا با پیش‌بینی وزن‌های بعدی، به سمت نقطه بهینه دقیق‌تری حرکت کند.

علاوه بر بهینه‌ساز، از StepLR به عنوان LR Scheduler برای کاهش نرخ یادگیری در طول ایپاک‌ها استفاده شده است. در این تنظیمات، مقدار step_size برابر ۵ تعیین شده که به این معناست پس از هر ۵ ایپاک، نرخ یادگیری کاهش می‌یابد. پارامتر gamma برابر ۰.۵ است که نرخ یادگیری را در هر مرحله به نصف مقدار فعلی کاهش می‌دهد. استفاده از StepLR باعث می‌شود که مدل در ابتدای آموزش با نرخ یادگیری بالا سریع‌تر به روزرسانی شود و در ادامه، با کاهش تدریجی نرخ یادگیری، فرآیند همگرایی به سمت بهینه سراسری دقیق‌تر انجام گیرد. این ترکیب بهینه‌ساز SGD و تنظیم کننده نرخ یادگیری StepLR باعث می‌شود که مدل بتواند بهطور مؤثر وزن‌ها را یاد گرفته و بهینه‌سازی پایدارتر و کارآمدتری داشته باشد. شکل ۱۳ کدنویسی همه این بخش‌ها را نشان می‌دهد.

```

▼ Optimizer 🔗
▶ # Optimizer => LR is the most influential parameter
initial_lr = 0.01
momentum    = 0.9
WD          = 1e-5
optimizer = optim.SGD(model_1.parameters(), lr=initial_lr, momentum=momentum, nesterov=True, weight_decay=WD)

▼ LR Scheduler 🚧
[ ] # LR Scheduler
step_size = 5
gamma     = 0.5
lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

```

شکل ۱۳ - بهینه ساز و نرخ یادگیری کاهش یافته با زمان (همراه با ابرمولفه ها)

۱۲-توابع Evaluation (eval_one_epoch) و Train_one_epoch

در اینجا تابع evaluation (معادل با همان test_one_epoch) آورده شده اند تا در طول فرآیند آموزش و ارزیابی استفاده شوند.

```
13-     # Import metrics
14-     # torchmetrics
15-     from torchmetrics import Accuracy    # Accuracy
16-     from torchmetrics import Precision, Recall, F1Score    #
17-         Precision, Recall, F1Score
18-     def train_one_epoch(model, loader, loss_function, optimizer,
19-         epoch=None):
20-             # Train phase, a flag (This is essential)
21-             model.train()
22-             # AverageMeter for LOSS train
23-             loss_train = AverageMeter()
24-
25-             # Metrics: Accuracy, Precision, Recall, F1Score
26-             accuracy_train = Accuracy(task="binary").to(device)
27-             precision_train = Precision(task="binary").to(device)
28-             recall_train = Recall(task="binary").to(device)
29-             f1score_train = F1Score(task="binary").to(device)
30-
31-             # Reset metrics at the start of each epoch (essential)
32-             loss_train.reset()
33-             accuracy_train.reset()
34-             precision_train.reset()
35-             recall_train.reset()
36-             f1score_train.reset()
37-
38-             # FOR loop => across different batches
39-             for inputs, targets in tqdm(loader, unit="batch"):    # tqdm
40-                 with batches
41-                     # both inputs and targets => to device
42-                     inputs, targets = inputs.to(device),
43-                         targets.to(device).float()    # Convert targets to float
44-                         targets = targets.unsqueeze(1)    # Ensure targets have
45-                             shape [batch_size, 1]
46-
47-                         # inputs => model => outputs
48-                         outputs = model(inputs)
```

```

48-         loss = loss_function(outputs, targets)
49-
50-         # back propagation, updating weights
51-         optimizer.zero_grad()
52-         loss.backward()
53-         optimizer.step()
54-
55-         # For accuracy and metrics, apply sigmoid activation
56-         # and threshold
56-         predictions = (outputs.sigmoid() > 0.5).int()
57-
58-         # updating and saving metrics for train
59-         loss_train.update(loss.item(), inputs.size(0))
60-         accuracy_train.update(predictions, targets.int()) #
61-             Convert targets back to int for metrics
61-         precision_train.update(predictions, targets.int())
62-         recall_train.update(predictions, targets.int())
63-         f1score_train.update(predictions, targets.int())
64-
65-         # Compute average metrics over all batches in this 1 epoch
66-         avg_loss_train = loss_train.avg
67-         avg_accuracy_train = accuracy_train.compute().item()
68-         avg_precision_train = precision_train.compute().item()
69-         avg_recall_train = recall_train.compute().item()
70-         avg_f1score_train = f1score_train.compute().item()
71-
72-         # Print metrics (optional for debugging purposes)
73-         print(f"Epoch {epoch}: Loss: {avg_loss_train:.4f},"
74-             Accuracy: {avg_accuracy_train:.4f}, "
74-                 f"Precision: {avg_precision_train:.4f}, Recall:"
74-                 {avg_recall_train:.4f}, F1Score: {avg_f1score_train:.4f})")
75-
76-         # Return metrics: LOSS, ACC, PRECISION, RECALL, F1
77-         return avg_loss_train, avg_accuracy_train,
78-             avg_precision_train, avg_recall_train, avg_f1score_train

```

```

78-     # Import metrics
79-     # torchmetrics
80-     from torchmetrics import Accuracy # Accuracy
81-     from torchmetrics import Precision, Recall, F1Score # #
81-         Precision, Recall, F1Score
82-
83-     def evaluation(model, loader, loss_function):
84-         """Evaluation phase with loss, accuracy, precision,
84-         recall, and F1Score."""

```

```
85-      # Evaluation phase, a flag (essential)
86-      model.eval()
87-
88-      # AverageMeter for LOSS eval
89-      loss_eval = AverageMeter()
90-
91-      # Metrics: Accuracy, Precision, Recall, F1Score
92-      accuracy_eval = Accuracy(task="binary").to(device)
93-      precision_eval = Precision(task="binary").to(device)
94-      recall_eval = Recall(task="binary").to(device)
95-      f1score_eval = F1Score(task="binary").to(device)
96-
97-      # Reset metrics at the start of each epoch (essential)
98-      loss_eval.reset()
99-      accuracy_eval.reset()
100-     precision_eval.reset()
101-     recall_eval.reset()
102-     f1score_eval.reset()
103-
104-     # NO gradient is required for evaluation (test)
105-     with torch.no_grad(): # essential
106-         # FOR loop => across different batches
107-         for inputs, targets in tqdm(loader, unit="batch"):
108-             # both inputs and targets => to device
109-             inputs, targets = inputs.to(device),
110-             targets.to(device).float() # Convert targets to float
111-             targets = targets.unsqueeze(1) # Ensure targets
112-             have shape [batch_size, 1]
113-
114-             # inputs => model => outputs
115-             outputs = model(inputs)
116-
117-             # loss eval => this is for each batch
118-             loss = loss_function(outputs, targets)
119-             predictions = (outputs.sigmoid() > 0.5).int()
120-
121-             # Updating and saving metrics for
122-             # validation/evaluation (no train process)
123-             loss_eval.update(loss.item(), inputs.size(0))
124-             accuracy_eval.update(predictions,
125-             targets.int()) # Convert targets to int for metrics
126-             precision_eval.update(predictions, targets.int())
127-
```

```

125-             recall_eval.update(predictions, targets.int())
126-             f1score_eval.update(predictions, targets.int())
127-
128-         # Compute average metrics over all batches for this 1
129-         epoch
130-         avg_loss_eval = loss_eval.avg
131-         avg_accuracy_eval = accuracy_eval.compute().item()
132-         avg_precision_eval = precision_eval.compute().item()
133-         avg_recall_eval = recall_eval.compute().item()
134-         avg_f1score_eval = f1score_eval.compute().item()
135-
136-         # Print metrics (optional for debugging purposes)
137-         print(f"Evaluation Results: Loss: {avg_loss_eval:.4f},"
138-               f"Accuracy: {avg_accuracy_eval:.4f}, "
139-               f"Precision: {avg_precision_eval:.4f}, Recall:"
140-               f"{avg_recall_eval:.4f}, F1Score: {avg_f1score_eval:.4f}")
141-

```

در تابع آموزش مدل، ابتدا شبکه در حالت `train` قرار می‌گیرد تا پارامترهای آن به روزرسانی شوند. برای شروع هر دوره (Epoch)، مقادیر زیان و معیارهای ارزیابی مانند دقت، دقت مثبت (Precision)، بازخوانی (Recall) و F1Score بازنگاری می‌شوند تا مقادیر هر ایپاک به طور مجزا محاسبه شوند. در حلقه آموزش، داده‌ها به صورت دسته‌ای (Batch) از مجموعه داده‌ها خوانده و به دستگاه محاسباتی منتقل می‌شوند. پس از عبور داده‌ها از مدل و محاسبه خروجی‌ها، تابع زیان محاسبه می‌شود و الگوریتم پس انتشار برای به روزرسانی وزن‌ها اجرا می‌شود. برای محاسبه معیارهای ارزیابی، خروجی‌های مدل از طریق تابع سیگموید به احتمال بین صفر و یک تبدیل شده و با استفاده از یک آستانه (Threshold) مقدار نهایی پیش‌بینی شده مشخص می‌شود. این مقادیر برای هر دسته محاسبه و ذخیره می‌شوند تا در پایان هر دوره، میانگین زیان و معیارها بر اساس تمام داده‌های آموزشی محاسبه و نمایش داده شوند. هدف از این فرآیند به روزرسانی مدام وزن‌های مدل برای کاهش زیان و بهبود عملکرد مدل در طول دوره‌های آموزشی است.

در تابع ارزیابی مدل، ابتدا شبکه در حالت `eval` قرار می‌گیرد و فرآیند محاسبات بدون نیاز به به روزرسانی وزن‌ها انجام می‌شود. برای جلوگیری از تغییرات ناخواسته در وزن‌ها، گرادیان‌ها غیرفعال می‌شوند تا محاسبات تنها برای ارزیابی انجام شوند. در این حالت نیز داده‌ها به صورت دسته‌ای از مجموعه اعتبارسنجی یا آزمون خوانده و به دستگاه منتقل می‌شوند. خروجی مدل پس از عبور داده‌ها محاسبه و برای محاسبه زیان به تابع زیان داده می‌شود. سپس برای ارزیابی عملکرد، خروجی‌ها به احتمال تبدیل و با استفاده از آستانه مشخص به مقادیر صفر و یک نکاشت می‌شوند تا با برجسب‌های واقعی مقایسه شوند. معیارهای ارزیابی شامل دقت، دقت مثبت، بازخوانی و F1Score برای هر دسته به روزرسانی و ذخیره می‌شوند. در پایان، میانگین زیان و معیارها بر اساس تمام داده‌های ارزیابی محاسبه و نمایش داده می‌شود. این فرآیند به طور دقیق عملکرد مدل را بر روی داده‌های جدید و خارج از مجموعه آموزش ارزیابی می‌کند و نشان می‌دهد که مدل تا چه حد به طور مؤثر تعیین یافته است.

در این پژوهش، از سه مجموعه داده مجزا شامل مجموعه آموزش (Train)، اعتبارسنجی (Validation) و آزمون (Test) استفاده می‌شود. مدل ابتدا با استفاده از مجموعه داده‌های آموزشی، فرآیند یادگیری و به روزرسانی وزن‌ها را انجام می‌دهد. سپس، عملکرد مدل در هر دوره با استفاده

از مجموعه اعتبارسنجی ارزیابی می‌شود تا معیارهای عملکرد مانند دقت و زیان بررسی شده و هایپرپارامترها بهینه‌سازی شوند. لازم به ذکر است که داده‌های اعتبارسنجی به مدل در فرآیند آموزش کمک می‌کنند تا از بیش‌برازش (Overfitting) جلوگیری شود. در نهایت، مدل تنها یک بار و فقط یک بار روی مجموعه داده‌های آزمون اجرا می‌شود تا عملکرد نهایی مدل بر روی داده‌های کاملاً جدید و دیده‌نشده سنجیده شود. این رویکرد تضمین می‌کند که ارزیابی مدل بر اساس داده‌های آزمون، دقیق و بدون تأثیر از فرآیند یادگیری یا تنظیمات هایپرپارامترها صورت گیرد.

۱۳- چرخه آموزش

در این بخش از فرآیند، مجموعه‌ای از اقدامات برای ارزیابی اولیه، آموزش، و اعتبارسنجی مدل تعریف شده‌اند که شامل بررسی معیارهای اولیه، ثبت تاریخچه یادگیری، و اجرای حلقه اصلی آموزش می‌شود. هر یک از این مراحل دقیق و با جزئیات برنامه‌ریزی شده‌اند تا عملکرد مدل در طول آموزش به‌طور مستمر مورد ارزیابی قرار گیرد و بهترین ایپاک (Epoch) ذخیره شود.

ابتدا با توجه به تصاویر اول، در مرحله ارزیابی اولیه (Epoch0) مدل بدون هیچ آموزشی بر روی مجموعه‌های آموزش و اعتبارسنجی اجرا می‌شود تا عملکرد اولیه مدل محاسبه و ثبت گردد. در این مرحله، معیارهایی مانند زیان (Loss)، دقت (Accuracy)، دقت مثبت (Precision)، بازخوانی (Recall) و F1Score محاسبه می‌شوند. این معیارها به ما کمک می‌کنند تا متوجه شویم مدل در ابتدای کار (بدون یادگیری) چگونه عمل می‌کند و عملکرد آن در حالت خام چقدر از تعادل برخوردار است.

در ادامه، یک تاریخچه برای ذخیره مقادیر معیارهای آموزش و اعتبارسنجی تعریف می‌شود. این تاریخچه شامل مقادیر زیان، دقت، Precision و Recall در هر ایپاک است. با ثبت این مقادیر در طول آموزش، می‌توان روند بهبود عملکرد مدل را مشاهده و تحلیل کرد. این روند به ما کمک می‌کند تا بینیم مدل چگونه با هر ایپاک یادگیری خود را بهبود می‌بخشد.

حلقه اصلی آموزش مدل به‌طور دقیق طراحی شده است. در این حلقه، برای هر ایپاک فرآیند آموزش و ارزیابی اعتبارسنجی انجام می‌شود. ابتدا مدل برای هر دسته داده در مجموعه آموزش اجرا شده و وزن‌های آن با استفاده از الگوریتم بهینه‌ساز SGD به‌روز می‌شوند. در پایان هر ایپاک، مدل بر روی مجموعه اعتبارسنجی اجرا می‌شود تا عملکرد آن روی داده‌های دیده‌نشده محاسبه شود. این ارزیابی به ما کمک می‌کند تا از یادگیری بیش‌برازش (Overfitting) جلوگیری کنیم. برای بهینه‌سازی فرآیند آموزش، از کاهش نرخ یادگیری (LR Scheduler) استفاده می‌شود که هر چند ایپاک نرخ یادگیری را کاهش می‌دهد تا مدل به تدریج به بهترین نقطه همگرایی برسد.

در پایان هر ایپاک، نتایج شامل زیان، دقت، Precision و F1Score برای داده‌های آموزش و اعتبارسنجی نمایش داده می‌شوند. همچنین، نرخ یادگیری فعلی ثبت می‌شود تا بتوان کاهش آن را در طول ایپاک‌ها مشاهده کرد. بهترین عملکرد مدل، که بر اساس بیشترین دقت روی مجموعه اعتبارسنجی سنجیده می‌شود، به‌طور خودکار ذخیره می‌گردد. این کار تضمین می‌کند که بهترین مدل ممکن برای ارزیابی نهایی انتخاب شده باشد.

خروجی‌های نهایی حلقه آموزش نشان می‌دهند که مدل به تدریج در طول ایپاک‌ها بهبود می‌یابد. زیان کاهش می‌یابد و دقت، Precision و F1Score افزایش می‌یابند. این روند نشان‌دهنده این است که مدل با مشاهده بیشتر داده‌های آموزشی و تنظیم مناسب وزن‌ها توانسته است به تعمیم‌پذیری و عملکرد مطلوبی دست یابد.

```

# Evaluate model performance without any training (Epoch 0)
print("Initial Metrics (Before Training, Epoch = 0)\n")

# Evaluate on Train Loader
print("Train Loader:")
initial_train_loss, initial_train_accuracy, initial_train_precision, initial_train_recall, initial_train_f1score = evaluation(
    model_1, train_loader, loss_function
)
print(f"Initial Train Loss      (no train): {initial_train_loss:.4f}")
print(f"Initial Train Accuracy  (no train): {initial_train_accuracy:.4f}")
print(f"Initial Train Precision (no train): {initial_train_precision:.4f}")
print(f"Initial Train Recall    (no train): {initial_train_recall:.4f}")
print(f"Initial Train F1 Score (no train): {initial_train_f1score:.4f}")
print("")

# Evaluate on Validation Loader
print("Validation Loader:")
initial_valid_loss, initial_valid_accuracy, initial_valid_precision, initial_valid_recall, initial_valid_f1score = evaluation(
    model_1, valid_loader, loss_function
)
print(f"Initial Validation Loss   (no train): {initial_valid_loss:.4f}")
print(f"Initial Validation Accuracy (no train): {initial_valid_accuracy:.4f}")
print(f"Initial Validation Precision (no train): {initial_valid_precision:.4f}")
print(f"Initial Validation Recall  (no train): {initial_valid_recall:.4f}")
print(f"Initial Validation F1 Score (no train): {initial_valid_f1score:.4f}")
print("")
```

Initial Metrics (Before Training, Epoch = 0)

Train Loader:
100%|██████████| 246/246 [01:06<00:00, 3.68batch/s]
Evaluation Results: Loss: 0.3601, Accuracy: 0.4294, Precision: 0.8638, Recall: 0.2728, F1Score: 0.4146
Initial Train Loss (no train): 0.3601
Initial Train Accuracy (no train): 0.4294
Initial Train Precision (no train): 0.8638
Initial Train Recall (no train): 0.2728
Initial Train F1 Score (no train): 0.4146

Validation Loader:
100%|██████████| 82/82 [00:17<00:00, 4.74batch/s]Evaluation Results: Loss: 0.3577, Accuracy: 0.4694, Precision: 0.9796, Recall: 0.2951, F1Score: 0.4535
Initial Validation Loss (no train): 0.3577
Initial Validation Accuracy (no train): 0.4694
Initial Validation Precision (no train): 0.9796
Initial Validation Recall (no train): 0.2951
Initial Validation F1 Score (no train): 0.4535

شکل ۱۴ - عملکرد اولیه مدل (بدون آموزش دیدن) مستقیم در فاز ارزیابی

▼ Training loop (train and validation)

▼ Initial check

```

model = model_1.to(device)

[ ] # Initial Check
print()
print(f"To know: within optimizer: initial learning rate is {initial_lr} , L2 regularization is {WD} , momentum is 0.9 , and nesterov is True")
print(f"To know: during learning rate scheduler : gamma is {gamma} , step size is {step_size}")
print()
```

To know: within optimizer: initial learning rate is 0.01 , L2 regularization is 1e-05 , momentum is 0.9 , and nesterov is True
To know: during learning rate scheduler : gamma is 0.5 , step size is 5

شکل ۱۵ - مقدار ابرمولفه های مدل پیش از یادگیری

کدنویسی فرآیند آموش

```
import time

# Saving the best epoch
# The best epoch => is the highest validation accuracy
#
=====
best_acc_valid = 0
best_epoch = 0
#
=====

# The process => Training the model, extracting metrics for each epoch, finding the best
validation accuracy, and printing the results
#
=====
for epoch in range(num_epochs):
    epoch += 1 # Counting epochs for the best one

    # Separating the info of each epoch
    print("=====")
    =====

    # Start timing (tic)
    epoch_start_time = time.time()

    # Training and extracting the results
    #
    =====
    # Train
    loss_train, accuracy_train, precision_train, recall_train, f1score_train = train_one_epoch(
```

```
    model, train_loader, loss_function, optimizer, epoch)

    # Validation

    loss_valid, accuracy_valid, precision_valid, recall_valid, f1score_valid = evaluation(
        model, valid_loader, loss_function)

    # lr_scheduler

    lr_scheduler.step() # Must NOT be forgotten

    # =====

# Filling the history with metrics

# =====

# LOSS

loss_train_history.append(loss_train)

loss_valid_history.append(loss_valid)

# ACCURACY

accuracy_train_history.append(accuracy_train)

accuracy_valid_history.append(accuracy_valid)

# PRECISION

precision_train_history.append(precision_train)

precision_valid_history.append(precision_valid)

# RECALL

recall_train_history.append(recall_train)

recall_valid_history.append(recall_valid)

# F1 SCORE

f1score_train_history.append(f1score_train)

f1score_valid_history.append(f1score_valid)

# =====

# Printing the results

# =====
```

```

# Epoch
print("")

print(f"Epoch: {epoch} / {num_epochs}")

# LOSS - ACC - Precision - Recall - F1 Score

print(f"Train:    LOSS = {loss_train:.4}, Accuracy = {accuracy_train:.4}, Precision = {precision_train:.4}, Recall = {recall_train:.4}, F1 Score = {f1score_train:.4}")

print(f"Validation:  LOSS = {loss_valid:.4}, Accuracy = {accuracy_valid:.4}, Precision = {precision_valid:.4}, Recall = {recall_valid:.4}, F1 Score = {f1score_valid:.4}")

# Printing the LR

print(f"Learning Rate: {optimizer.param_groups[0]['lr']:.6f}")

# End timing (toc)

epoch_end_time = time.time()

elapsed_time_minutes = (epoch_end_time - epoch_start_time) / 60

print(f"Time taken for this epoch: {elapsed_time_minutes:.2f} minutes")

# =====

# Finding the best epoch = the highest validation accuracy

# =====

if accuracy_valid > best_acc_valid:

    torch.save(model.state_dict(), f'Optimal_Model_from_the_ground_up_saved.pt') # Save
only state dict in Colab VM

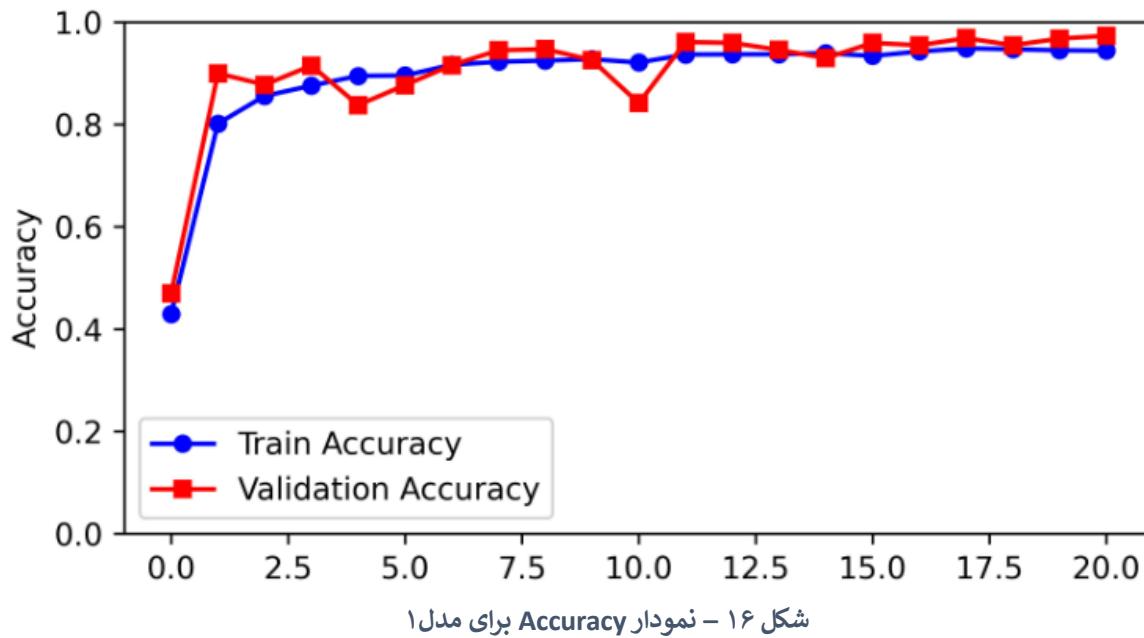
    best_acc_valid = accuracy_valid

    best_epoch = epoch

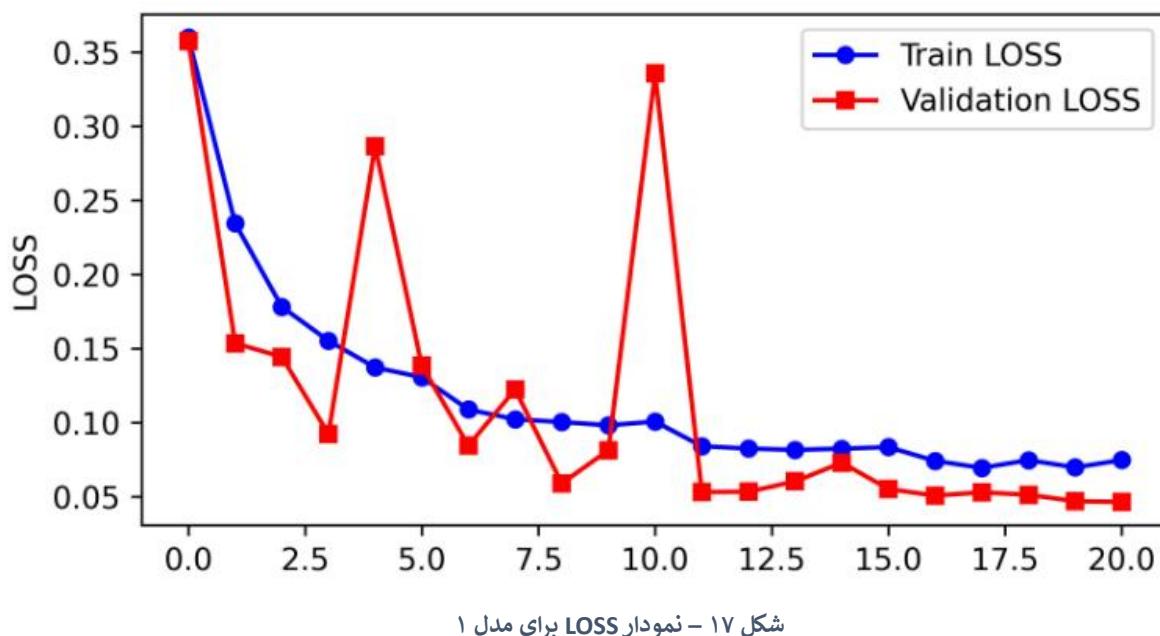
# =====

#
=====
```

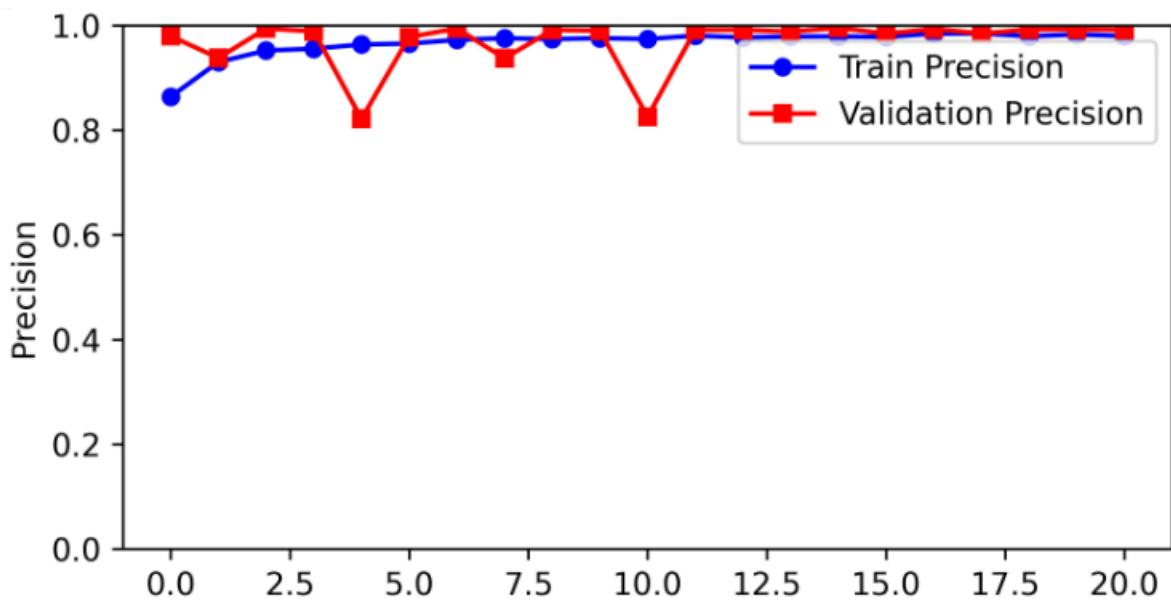
۱۴- نتایج نمودارها (مدل شماره ۱) (مجموعه های آموزش و ارزیابی)



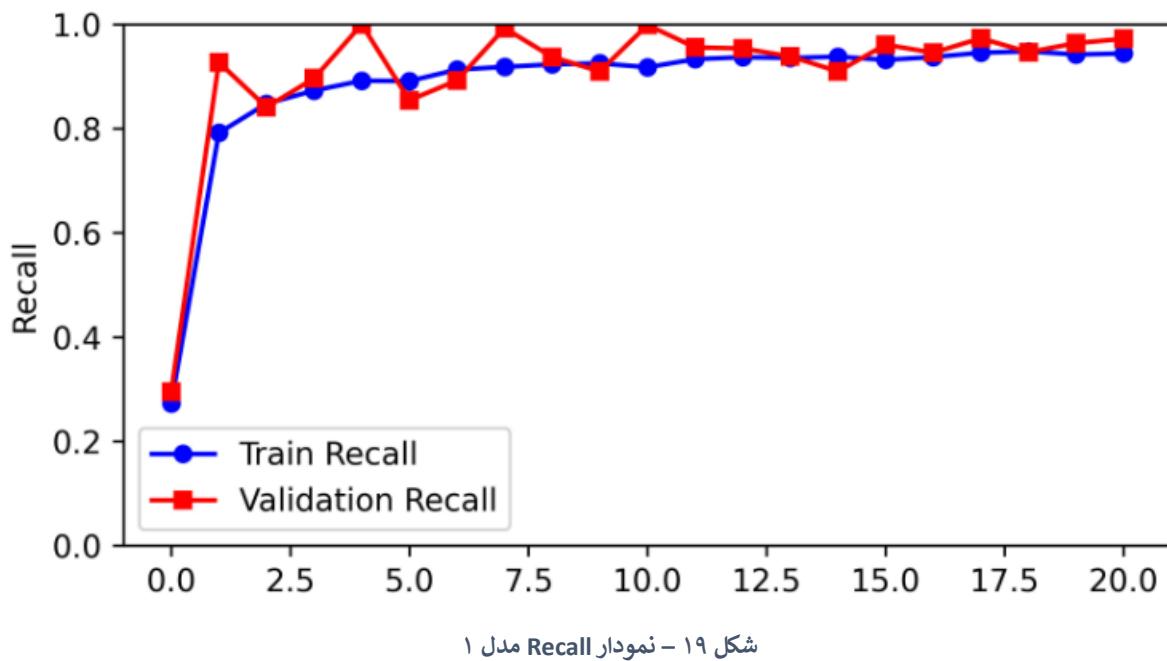
شکل ۱۶ - نمودار Accuracy برای مدل ۱



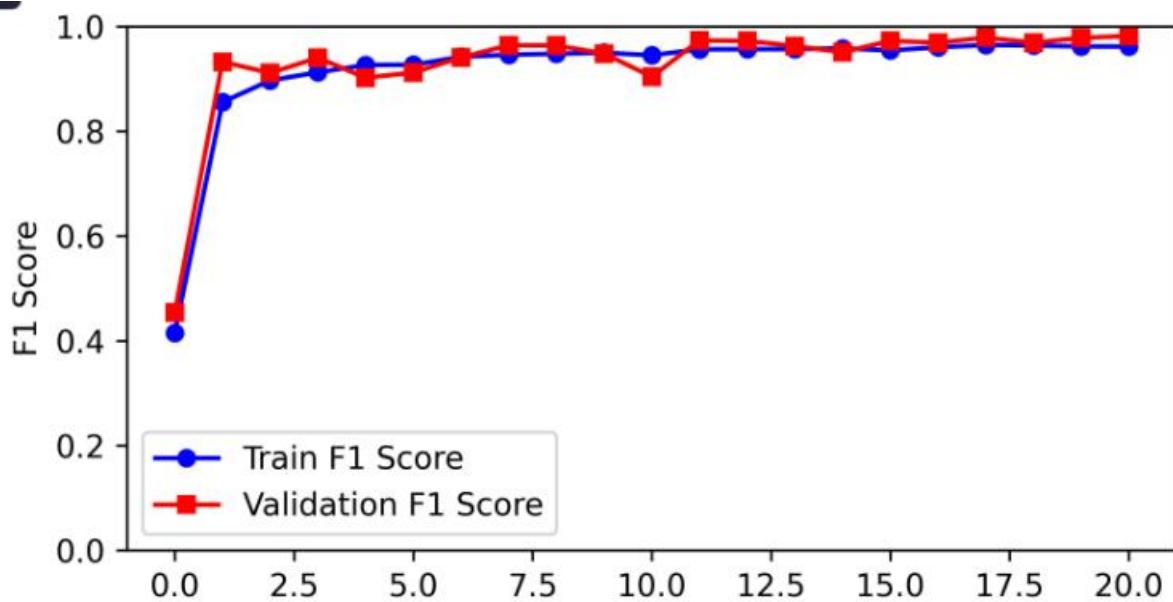
شکل ۱۷ - نمودار LOSS برای مدل ۱



شکل ۱۸ - نمودار Precision مدل ۱



شکل ۱۹ - نمودار Recall مدل ۱



شکل ۲۰ – نمودار F1Score مدل ۱

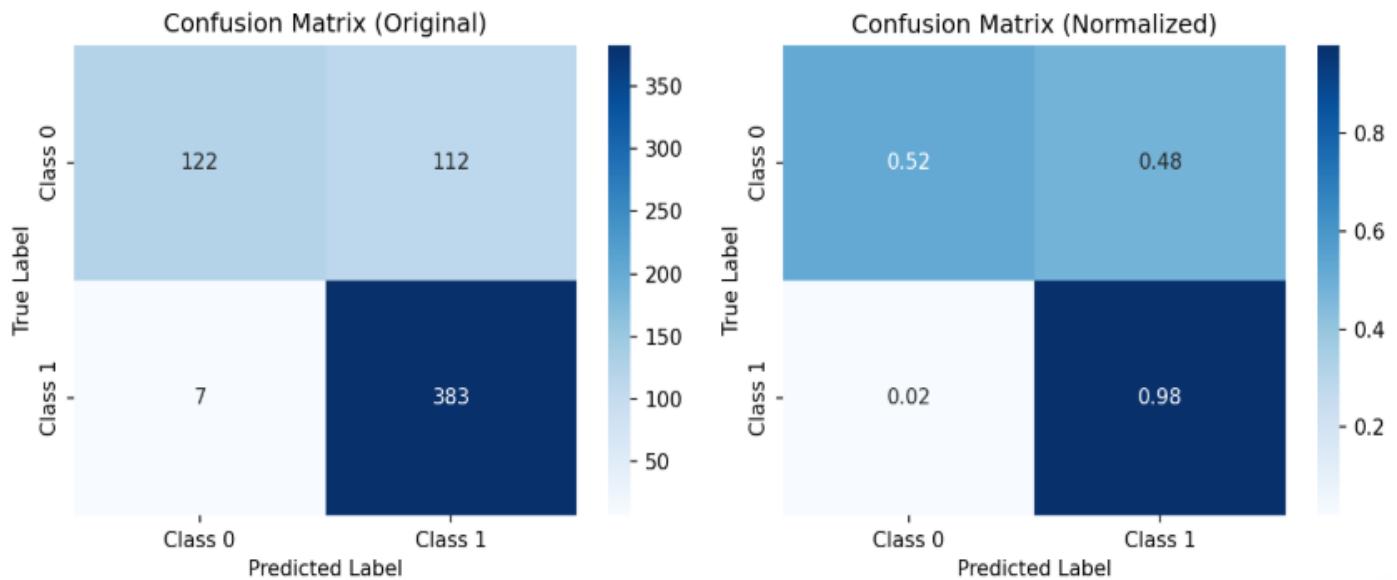
لازم به ذکر است که اینها برای داده های آموزشی (train) و ارزیابی (validation) است. مدل با داده های آموزشی آموزش دیده است، هایپرپارامترهای آن بهینه نمودارهای میشود و نمودارهای فوق بر اساس عملکرد داده های آموزشی و ارزیابی است. سپس مدل بهینه ذخیره شده و فقط یک مرتبه برروی داده های آزمون، که دیده نشده است، ارزیابی میشود.

۱۵- ارزیابی برروی داده های آزمون (فقط یک مرتبه)

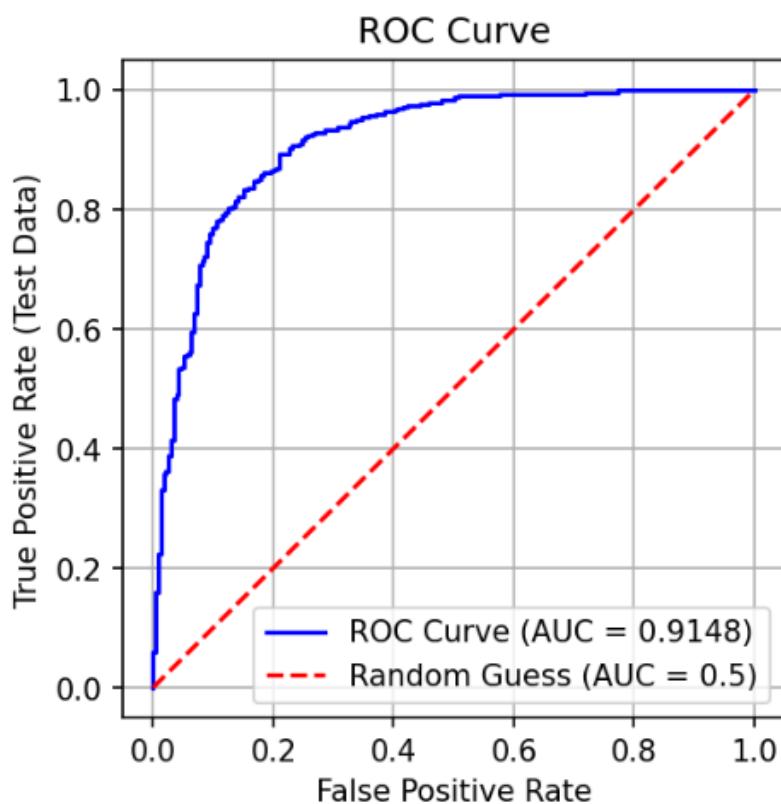
مدل بهینه ذخیره شده فقط یک مرتبه برروی داده های آزمون آزمایش میشود که نتایج آن در اشکال ۲۱، ۲۲ و ۲۳ آمده است.

```
--- Final Evaluation Metrics ---
Evaluation Accuracy: 0.8093
Evaluation Precision: 0.7737
Evaluation Recall: 0.9821
Evaluation F1 Score: 0.8655
```

شکل ۲۰ – ارزیابی فقط یک بار روی داده های آزمون (دیده نشده) – مدل ۱



شکل ۲۱ – ماتریس درهم ریختگی فقط بر روی داده های آزمون مدل ۱



شکل ۲۲ – نمودار ROC-AUC برای داده های آزمون مدل ۱

مدل ۱ به پایان رسید. به سراغ دیگر مدل ها (۲ و ۳) میرویم. شایان ذکر است که فقط بخش هایی که در بین مدل ها تغییر میکنند دوباره گفته میشوند.

۱۶ - مدل ۲ (افیشننت نت ب ۷، استخراج ویژگی)

کد مربوط به این بخش در شکل ۲۳ آمده است. در این کد از مدل EfficientNet-B7 به عنوان مدل از پیش آموزش دیده استفاده شده که وزن های آن بر اساس مجموعه داده ImageNet مقداردهی اولیه شده اند. ابتدا تمام لایه های مدل منجمد (Freeze) می شوند تا وزن های آن ها در فرآیند آموزش ثابت بمانند و تنها بخش های جدید یادگیری را انجام دهد. هدف از این کار استفاده از ویژگی های از پیش استخراج شده مدل در مراحل اولیه است تا نیاز به آموزش کامل مدل کاهش یابد و سرعت همگرایی افزایش یابد. در ادامه، لایه طبقه بند (Classifier) موجود در مدل حذف و با یک طبقه بند جدید مناسب برای طبقه بندی دودویی جایگزین می شود. طبقه بند جدید شامل یک لایه Dropout برای جلوگیری از بیش برآذش و یک لایه خطی (Linear) با یک نرون خروجی است که برای تولید مقدار احتمال (Logit) جهت طبقه بندی دودویی استفاده می شود.

سپس، اطمینان حاصل می شود که لایه های جدید (FC Layer) برای آموزش آماده هستند و گرادیان آن ها محاسبه خواهد شد. مدل به دستگاه محاسباتی مناسب GPU (CPU) یا (GPU منتقل می شود و خلاصه ای از ساختار جدید مدل چاپ می شود تا تغییرات اعمال شده تأیید شوند. در ادامه، با استفاده از تابع count_parameters، تعداد کل پارامترهای مدل و تعداد پارامترهای قابل آموزش محاسبه می شوند. خروجی نشان می دهد که در کل مدل 63,789,521 پارامتر دارد، در حالی که تنها 2,561 پارامتر برای یادگیری در دسترس هستند. این کاهش چشمگیر در تعداد پارامترهای قابل آموزش، ناشی از منجمد کردن لایه های اولیه مدل است که باعث می شود مدل بتواند به طور مؤثر و سریع تر با حجم داده کمتر آموزش ببیند و در عین حال از قدرت ویژگی های از پیش یادگیری شده بهره مند شود.

۱۷ - سایر بخش ها برای مدل ۲

سایر بخش های مدل ۲ همانند تابع خطا بهینه شده، توابع evaluation (test_one_epoch) و train_one_epoch عین هم هستند. تنها فرق در اینجا این است که در این مدل تمامی پارامترهای خود شبکه (به جز تک نuron مربوط به FC که وظیفه طبقه بندی دو کلاسه را برعهده دارد) همگی بر روی اصل مدل که خود بر روی ImageNet آموزش دیده است کپی شده و همگی فریز (منجمد) شده است و فقط و فقط تک نuron طبقه بند دسته بندی دو کلاسه فعال است و آموزش می بیند. یک نکته کوچک دیگر آنکه به سبب استفاده از یادگیری انتقالی علاوه بر نرخ یادگیری های ۰.۰۰۰۱ و ۰.۰۰۵ نرخ یادگیری ۰.۰۰۰۵ نیز بر روی داده های ارزیابی آزمون شد و مشاهده شد که ۰.۰۰۵ از هر دو بهتر است، البته همچنان همانند قبل نرخ یادگیری پویا است و در هر ۵ ایپاک نصف می شود. این کاهش نرخ یادگیری برای این حالت را میتوان نتیجه استفاده از یادگیری انتقالی و شبکه ای که نسبتاً چیزی یاد گرفته است دانست. ضمناً بدیهی است که تعداد پارامترهای آموزش پذیر این مدل در این حالت بسیار کم است. به طور دقیق تر میتوان گفت EfficientNetB7 backbone که مسئول استخراج ویژگی هاست و دارای تعداد زیادی پارامتر است، مسئول حدود ۶۳.۷ میلیون پارامتر غیر آموزش پذیر است. تعداد ۲۵۶۱ پارامتر آموزش پذیر نیز مربوط به لایه FC است. بدیهی است که عملکرد مدل چه در حالت initial_metrics و چه در حالات دیگر متفاوت با بخش ۱ است.

۱۸ - عملکرد مدل ۲ بر روی داده های آموزشی (train) و ارزیابی (validation)

اشکال ۲۴، ۲۵، ۲۶ و ۲۷ آنها را نشان می دهند. این اشکال روند آموزش مدل بر روی داده های آموزشی و ارزیابی را نشان می دهند.

```

import torch
import torch.nn as nn
from torchvision.models import efficientnet_b7

# Load the pretrained EfficientNet-B7 model
model_2 = efficientnet_b7(weights='IMAGENET1K_V1') # Use the pretrained weights for ImageNet

# Freeze all layers to make them non-trainable
for param in model_2.parameters():
    param.requires_grad = False

# Modify the classifier for binary classification
# Replace the existing classifier with a new one
model_2.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(model_2.classifier[1].in_features, 1) # Keep the dropout layer as in the original model
) # Binary classification: 1 output neuron for logit

# Ensure the new FC layer is trainable
for param in model_2.classifier.parameters():
    param.requires_grad = True

# Move the model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_2 = model_2.to(device)

# Print model summary to verify changes
print(model_2)

```

Show hidden output

Parameter counts

```

[] def count_parameters(model):
    # Total parameters
    total_params = sum(p.numel() for p in model.parameters())

    # Trainable parameters
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

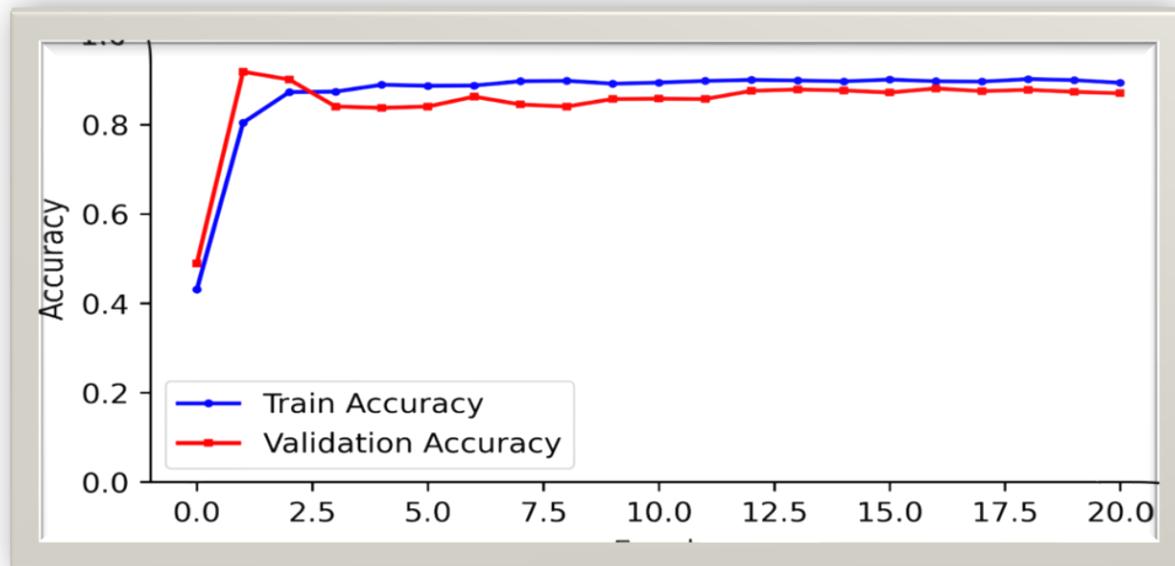
    print(f"Total Parameters: {total_params}")
    print(f"Trainable Parameters: {trainable_params}")

# Call the function for model_2
count_parameters(model_2)

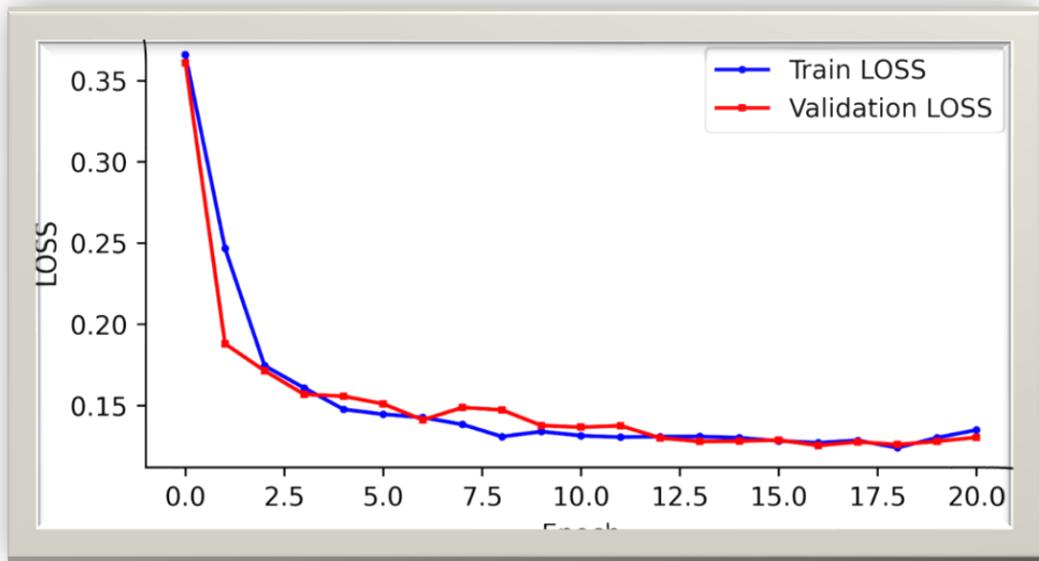
```

Total Parameters: 63789521
Trainable Parameters: 2561

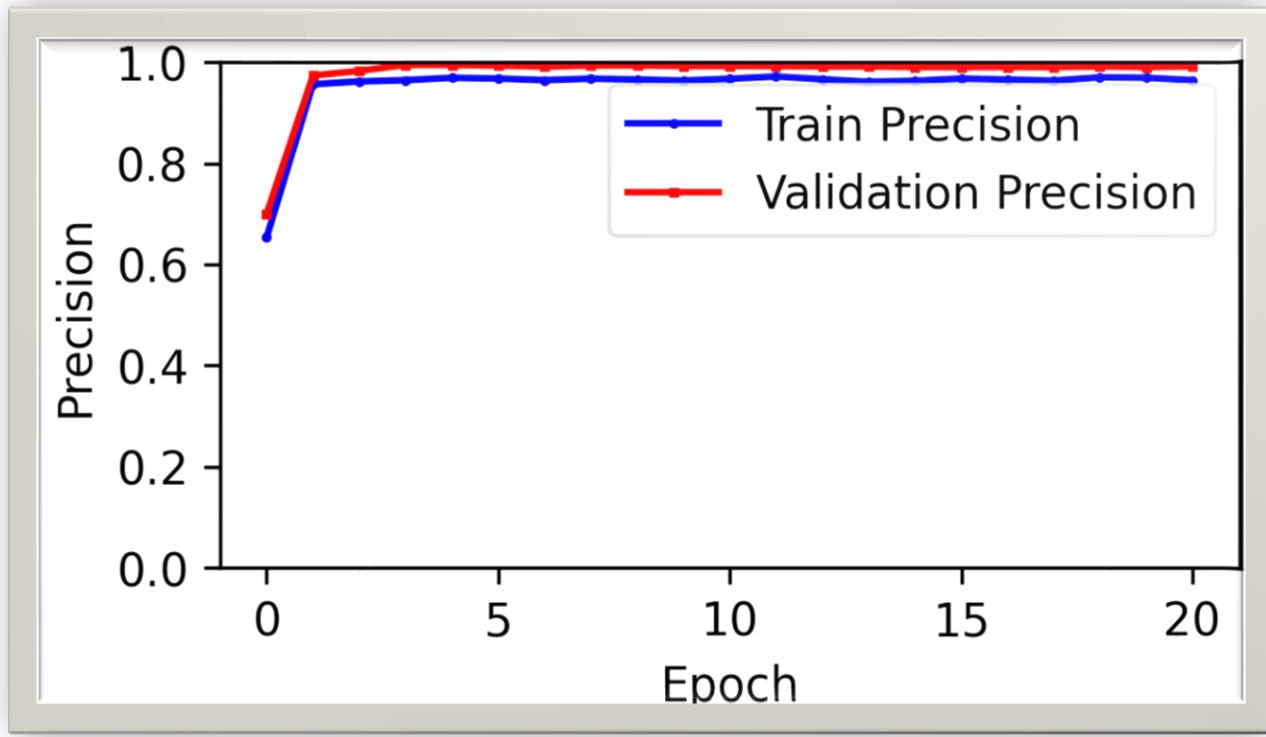
شکل ۲۳ – مدل فاز Feature Extraction در EfficientNetB7



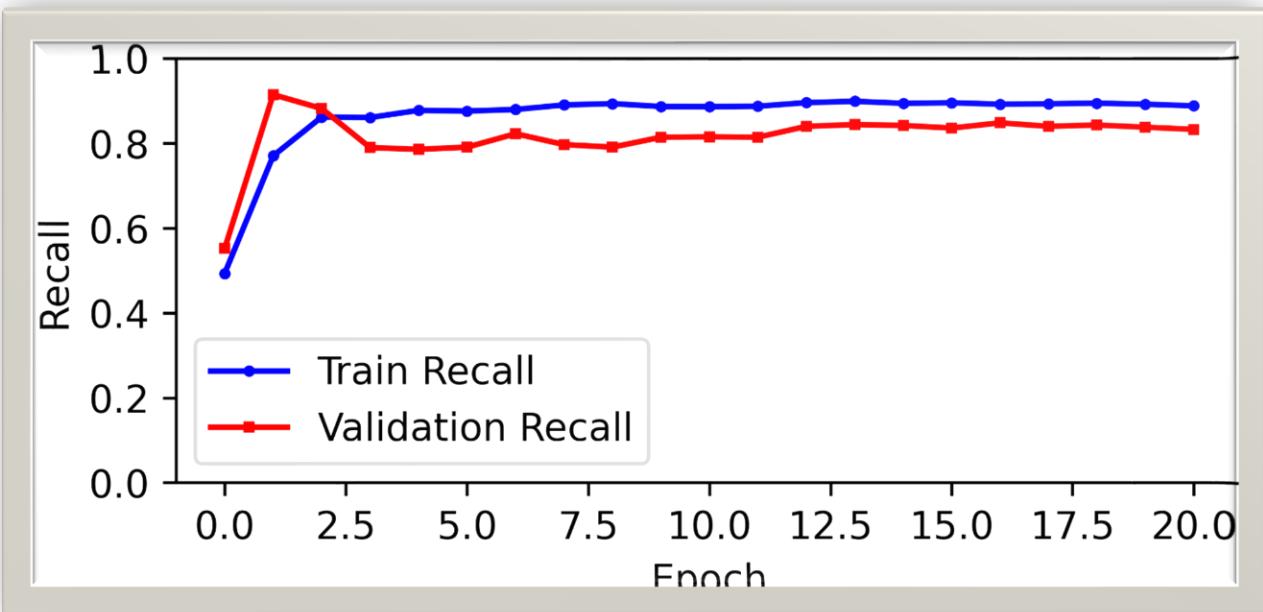
شکل ۲۴ – نمودار accuracy مدل (EfficientNetB7 + Feature extraction)



شكل ٢٥ - نمودار LOSS مدل ٢ (EfficientNetB7 + Feature Extraction)



شكل ٢٦ - نمودار Precision مدل ٢ (Feature Extraction + EfficientNetB7)



شکل ۲۷ – نمودار Recall مدل (EfficientNetB7 + feature extraction) ۲



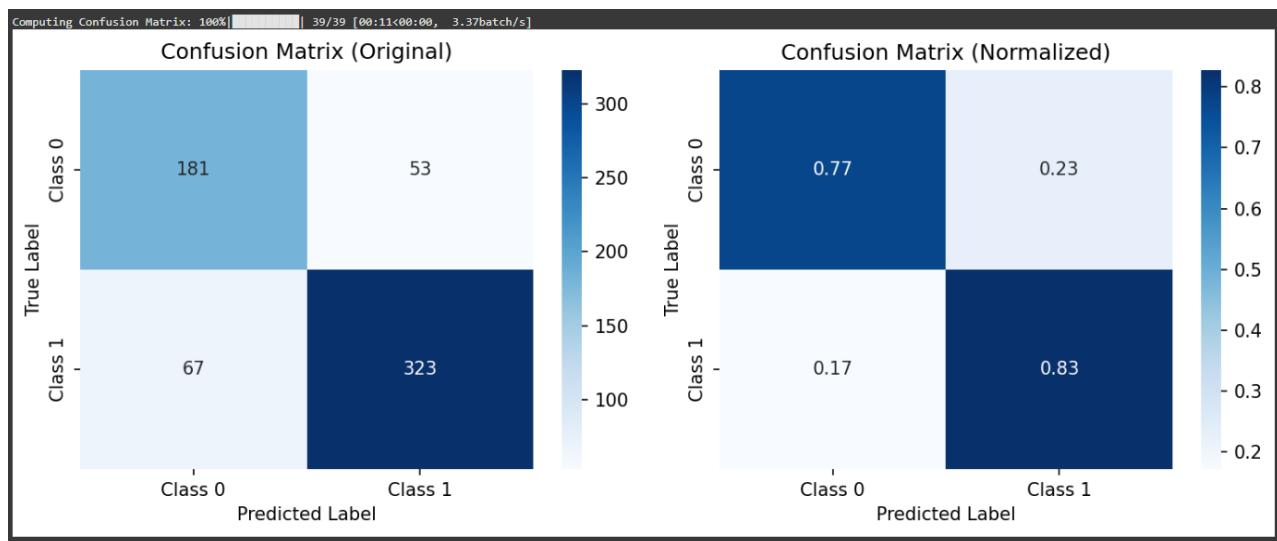
شکل ۲۸ – نمودار F1Score مدل (EfficientNetB7 + Feature Extraction) ۲

این نمودارها برای مدل ۲ برای بررسی همزمان Train و Validation بود. در حالت ارزیابی (Inference) دقیقاً همانند حالت مدل ۱، مدل بهینه که در طول فرآیند آموزش مدل ذخیره شده است (مدلی که دارای بیشترین accuracy بروی داده های validation است) فقط و یکبار بروی داده های test اعمال میشود و نتیجه نهایی نمایش داده میشود. این کار برای جلوگیری از نشت داده است.

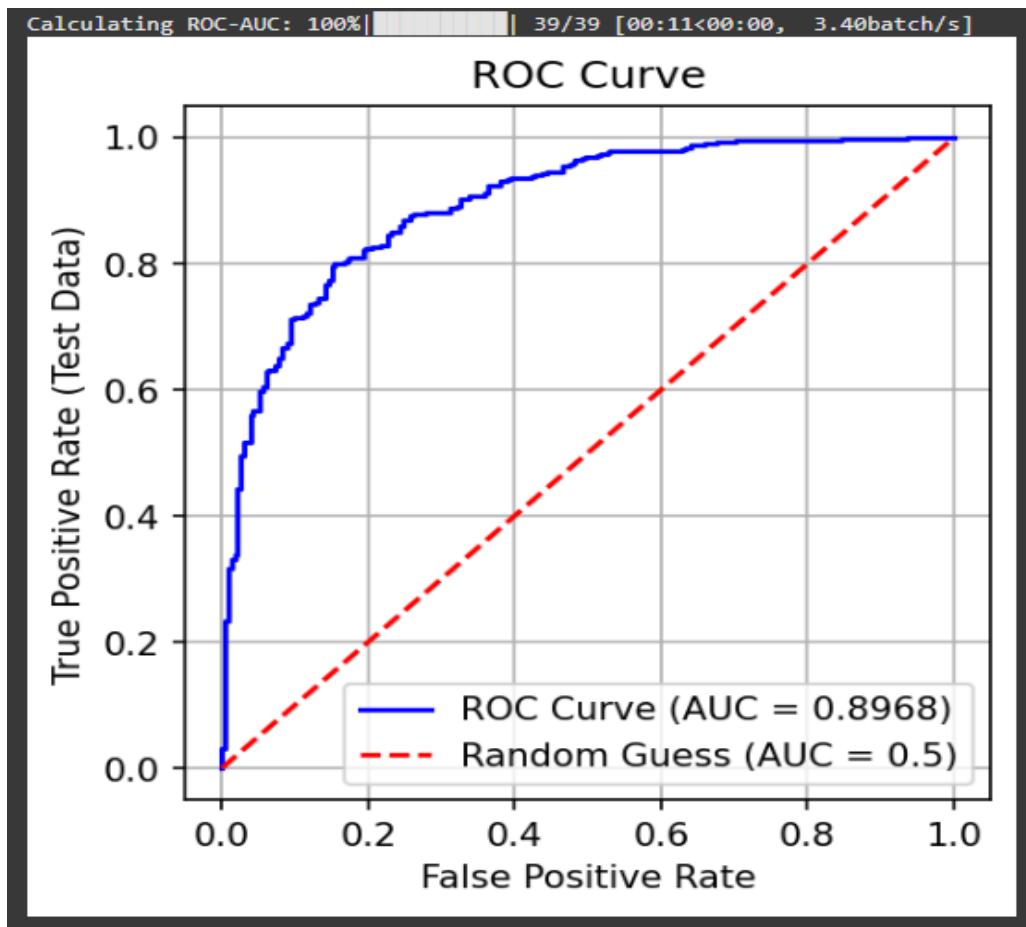
۱۹ - نتایج مدل ۲ برروی داده های دیده نشده (test / evaluation)

```
Evaluating on Test Data: 100% | 39/39 [00:11<00:00, 3.33batch/s]
== Final Evaluation Metrics ==
Evaluation Accuracy: 0.7772
Evaluation Precision: 0.7722
Evaluation Recall: 0.9128
Evaluation F1 Score: 0.8367
```

شکل ۲۹ - نتایج مدل ۲ برروی داده های تست دیده نشده



شکل ۳۰ - ماتریس درهم ریختگی برای داده های دیده نشده تست برای مدل ۲



شکل ۳۱ - ماتریس در هم ریختگی برای مدل ۲ بروی داده های دیده نشده تست (آزمون)

این بود نتایج مدل ۲ برای داده های ارزیابی (validation) و داده های دیده نشده آزمون (test). حال به سراغ مدل ۳ میرویم

۲۰- مدل ۳

مدل ۳ نیز همان EfficientNetB7 است. با این تفاوت که این بار خود EfficientNetB7 در ۳۳ درصد پایانی آموزش پذیر است و در ۶۶ درصد ابتدایی فریز است. ما انتظار داریم با توجه به اینکه ۳۳ درصد انتهایی مدل که مسئول استخراج ویژگی های غنی تر است آموزش پذیر است، عملکرد بهتری نسبت به مدل ۲ نتیجه شود. بقیه اش همان است! کد نویسی مدل در شکل ۳۲ انجام شده است که نشان میدهد ۳۳ درصد انتهایی مدل همراه با لایه FC آموزش پذیر و ۶۶ درصد ابتدایی منجمد و غیرقابل آموزش هستند.

```
import torch
import torch.nn as nn
from torchvision.models import efficientnet_b7

# Load the pretrained EfficientNet-B7 model
model_3 = efficientnet_b7(weights='IMAGENET1K_V1') # Use the pretrained weights for ImageNet

# Extract all parameters from the model
all_params = list(model_3.parameters()) # List of all parameters in the model
total_params = len(all_params) # Total number of parameters

# Calculate the cutoff point for freezing 66% of parameters
freeze_params = int(total_params * 0.66) # 66% of total parameters

# Freeze the first 66% of parameters and unfreeze the remaining 33%
for idx, param in enumerate(all_params):
    if idx < freeze_params:
        param.requires_grad = False # Freeze the first 66% parameters
    else:
        param.requires_grad = True # Keep the last 33% trainable

# Modify the classifier for binary classification
model_3.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True), # Dropout layer for regularization
    nn.Linear(model_3.classifier[1].in_features, 1) # Binary classification: 1 output neuron for logit
)

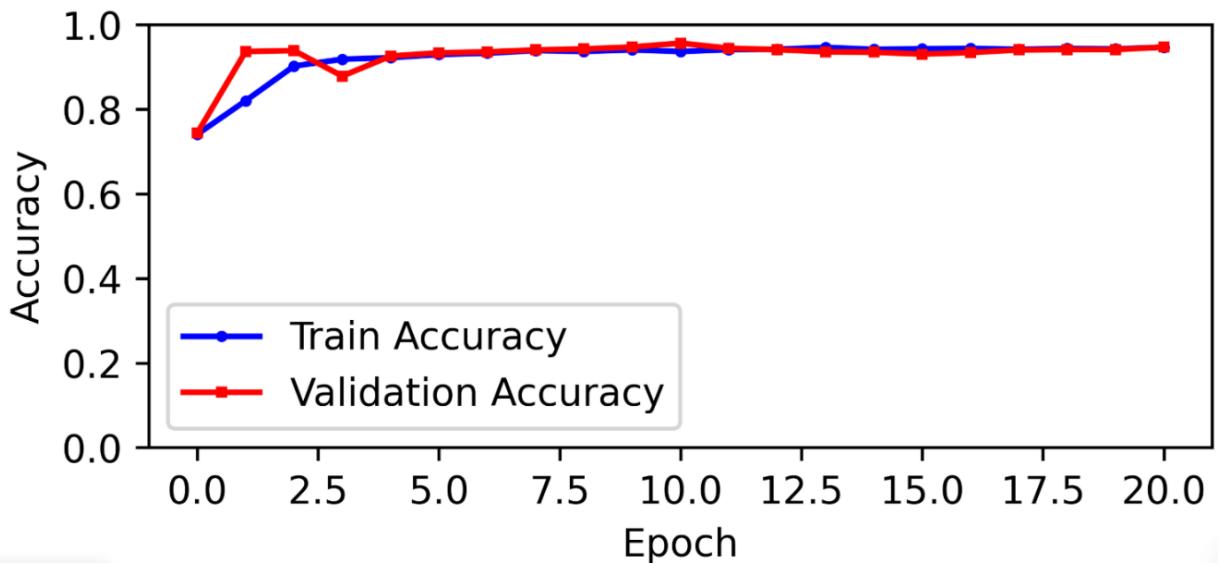
# Ensure the new classifier layer is trainable
for param in model_3.classifier.parameters():
    param.requires_grad = True

# Move the model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_3 = model_3.to(device)
```

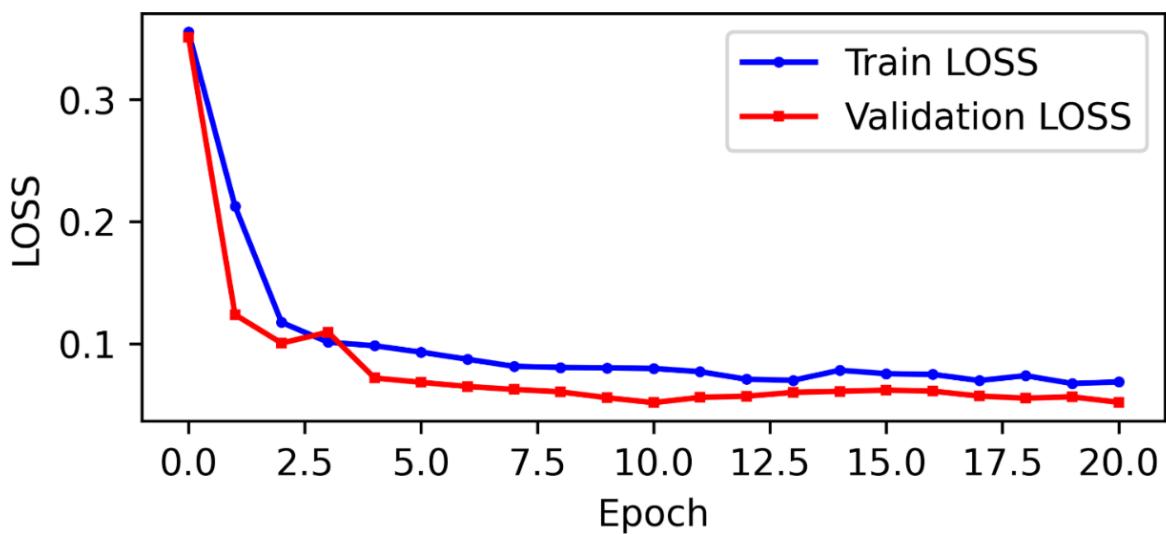
شکل ۳۲ - مدل ۳، EfficientNetB7 with 66% initials freeze and 33% deep trainable

۲۱- نتایج مدل برروی داده های آموزشی و ارزیابی

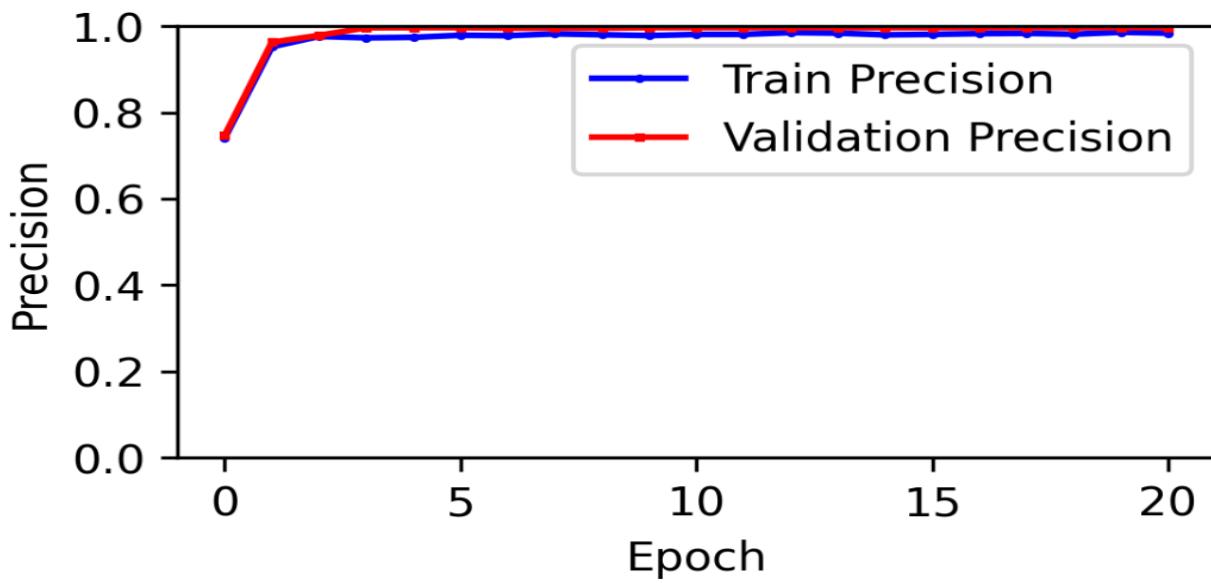
نتایج برای مدل شماره ۳، یعنی EfficientNetB7 همراه با ۳۳ درصد آموزش پذیر انتهایی و ۶۶ درصد منجمد در اینجا یعنی اشکال ۳۲، ۳۳، ۳۴ و ۳۵ آمده است.



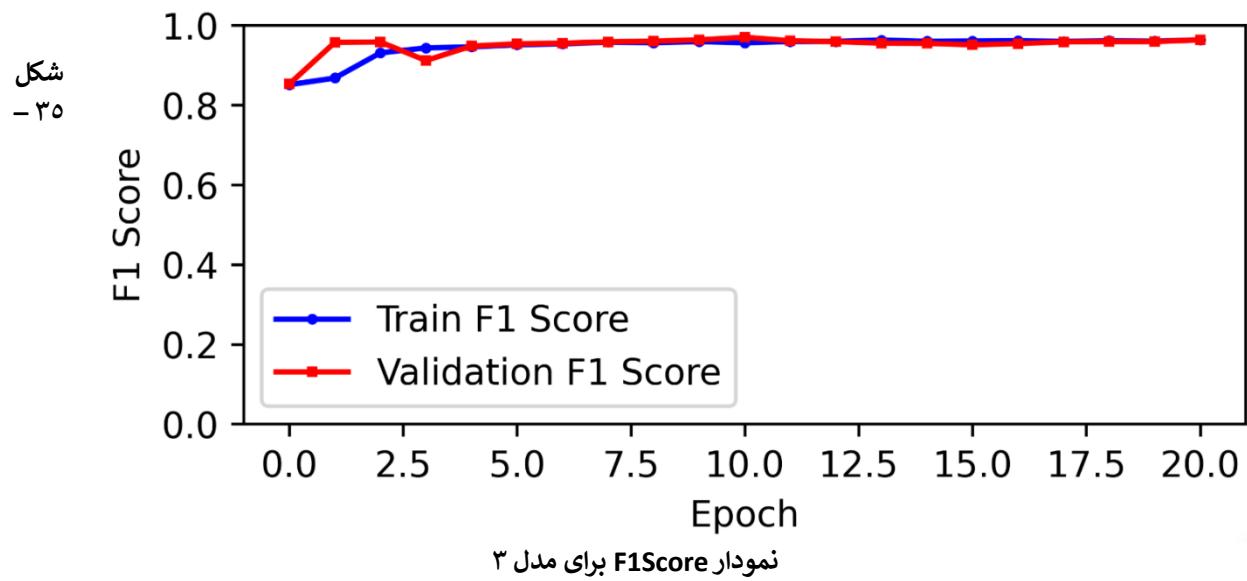
شکل ۳۲ - نمودار accuracy برای مدل ۳



شکل ۳۳ - نمودار LOSS برای مدل ۳



شکل ۳۴ – نمودار Precision برای مدل ۳

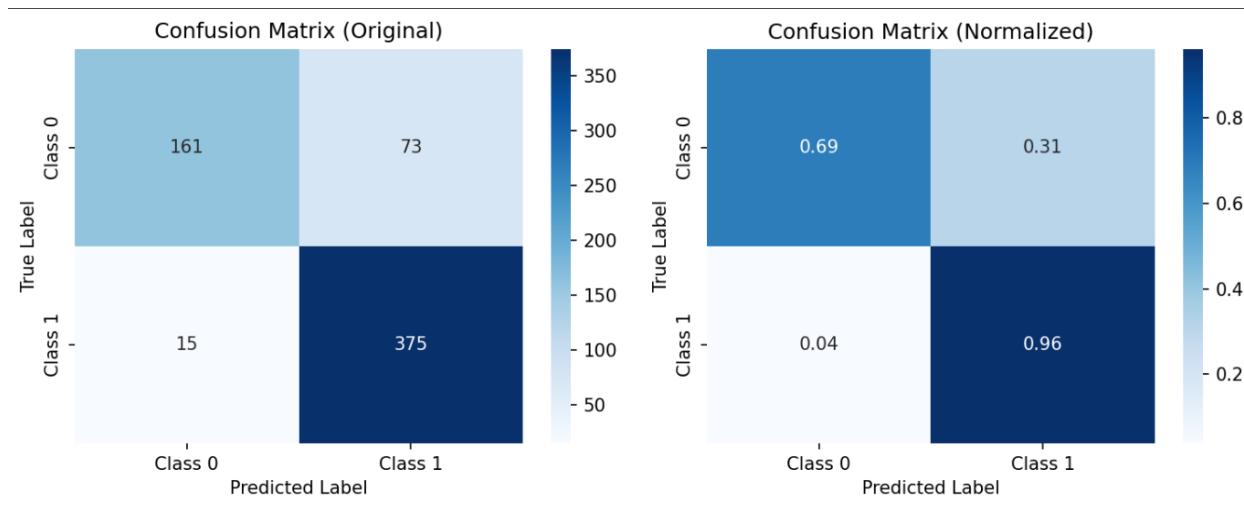


نمودار F1Score برای مدل ۳

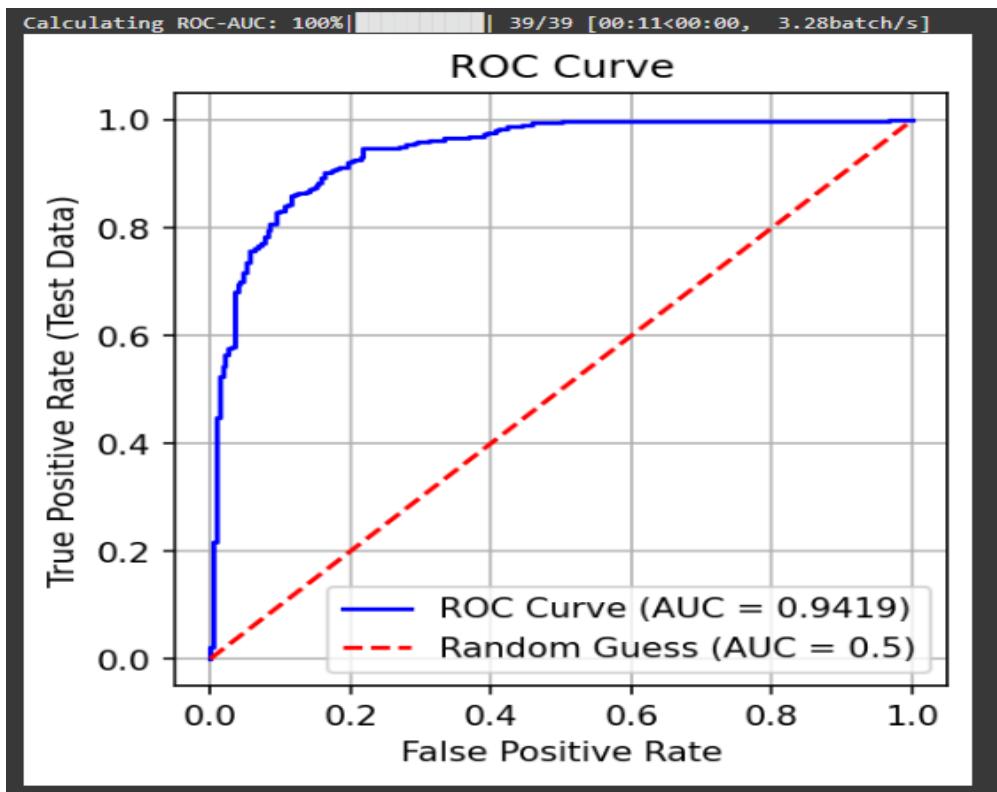
۲۲- نتایج مدل ۳ (test set) برروی داده های دیده نشده آزمون (EfficientNetB7 + Fine Tuning)

```
Evaluating on Test Data: 100%|██████████| 39/39 [00:11<00:00, 3.29batch/s]
== Final Evaluation Metrics ==
Evaluation Accuracy: 0.8590
Evaluation Precision: 0.8371
Evaluation Recall: 0.9615
Evaluation F1 Score: 0.8950
```

شکل ۳۶ - نتایج مدل ۳ برروی داده های آزمون



شکل ۳۷ - نتایج مدل ۳ در قالب ماتریس درهم ریختگی برروی داده های آزمون



شکل ۳۸ - منحنی ROC-AUC بر روی داده های آزمون (تست) برای مدل ۳

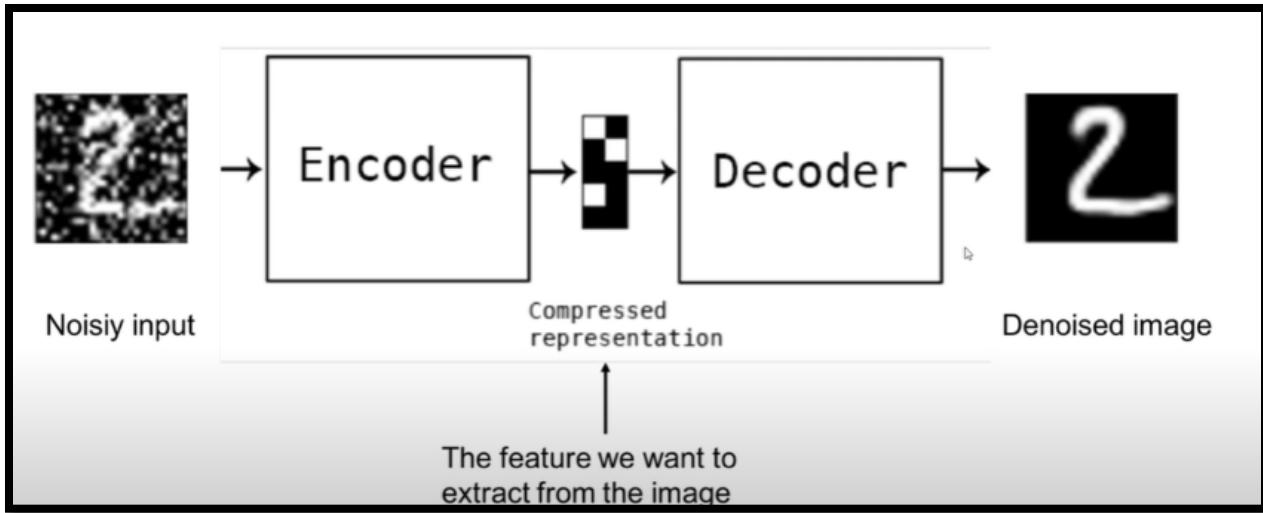
در اینجا وظایف محول شده برای بخش مدل های ۱ و ۲ و ۳ به پایان رسید. هر سه CNN طبق گفته ها و خواسته پروژه دقیق آموزش دیدند و نتایج نیز خوب بود.

بخش ۴ (شبکه عصبی GAN) و بخش ۵ (شبکه اتوانکودر) باقی مانده است. حال، ابتدا بخش ۵ (اتوانکودر) را می آوریم و سپس بخش ۴ را.

مدل پنجم، شبکه عصبی خودکدگذار (اتوانکودر)

(AutoEncoder) ۲۳ - معماری شبکه خودکدگذار

شماتیک معماری شبکه کدگذار (AutoEncoder) در شکل ۳۹ آمده است. ابتدا بخش Encoder داریم که مسئول استخراج ویژگی های عمیق است (بخش کدگذاری) و سپس بخش Decoder قرار دارد که کدگشایی می کند و این ویژگی های عمیق و سطح بالا را به داده اولیه (در این مثال تصاویر اولیه) تبدیل می کند. بنابراین شکل ۳۹ معماری اینها را نشان میدهند.



شکل ۳۹ - معماری اتو انکوڈر

۲۴ - نوع داده (برای شبکه اتوانکوڈر)

هم برای شبکه عصبی GAN و هم برای AutoEncoder باید یک نوع داده وارد شود (البته منظورم ساده ترین نوع GAN و AutoEncoder هاست). یعنی در اصل نمیشود که داده های تصاویر ریه نرمال (NORMAL) و ذات الریه (PNEUMONIA) را همزمان به آنها داد زیرا در این صورت شبکه گیج شده و بهینه ساز آن دچار خطأ میشود و ممکن است بخشی از داده های تولیدی ترکیبی از حالت نرمال و ذات الریه باشند. در این کار نظر به اینکه داده های کلاس ذات الریه تعداد بیشتری دارند و برای آموزش AutoEncoder مناسب تر هستند، از آنها، به طور دقیقتر داده های آموزشی کلاس ذات الریه، استفاده شده است. شکل ۴۰ کد نوشته شده برای مصورسازی تعداد ۹ عدد از این تصاویر را نشان میدهد. شایان ذکر است که سایز تصاویر نیز همچنان ۲۵۶ است.

```

# Preprocessing and Plot
# =====
# Preprocessing function to process images
img_size = 256
img_data = []

def preprocess_image(img_path):
    img = cv2.imread(img_path, 1)
    rgb_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    rgb_img = cv2.resize(rgb_img, (img_size, img_size))
    return img_to_array(rgb_img).astype('float32') / 255

# Load and preprocess images
for img_name in image_names: # Loop through all image names
    full_path = os.path.join(path_NORMAL_folder, img_name)
    img_data.append(preprocess_image(full_path))

# Convert to 4D tensor for model input
img_final = np.reshape(img_data, (len(img_data), img_size, img_size, 3))

num_shown_images = 9

# Plot some preprocessed images to visualize the data loader
fig, axes = plt.subplots(1, num_shown_images, figsize=(15, 5))
for i in range(num_shown_images):
    axes[i].imshow(img_final[i])
    axes[i].axis('off')
    axes[i].set_title(f"Image {i+1}")
plt.show()
# =====

Number of images: 3883


```

شکل ۴۰ - داده های ورودی به اتوانکودر (مدل ۴) شامل تصاویر ذات الریه به همراه مصورسازی ۹ عدد از آن ها

۲۵ - معماری Auto-Encoder

معماری اتوانکودر در این مطالعه، که کد نوشته شده برای آن و همچنین `model.summary` آن به طور کامل در شکل ۴۱ نشان داده شده است، شامل دو بخش اصلی انکودر (Encoder) و دیکودر (Decoder) است که هدف آن فشرده سازی دادهها و بازسازی آنها به شکل اولیه است. در بخش انکودر، مدل با استفاده از چندین لایه کانولوشن (Convolutional) و `MaxPooling` داده های ورودی را پردازش کرده و ابعاد ویژگی ها را به تدریج کاهش می دهد. ابتدا داده ورودی که شامل تصاویری با ابعاد مشخص و سه کanal رنگی است، وارد لایه کانولوشن می شود که از فیلترهایی با ابعاد $(3, 3)$ استفاده می کند و با فعال سازی `ReLU` ویژگی های پیچیده تر را استخراج می کند. در هر مرحله، لایه `MaxPooling` ابعاد مکانی داده ها را به نصف کاهش می دهد، که این امر باعث کاهش ابعاد و تمرکز بر ویژگی های اصلی تصویر می شود. این روند تا جایی ادامه پیدا می کند که ویژگی ها به شکل بسیار فشرده در انتهای انکودر ذخیره شوند.

در بخش دیکودر، هدف بازسازی داده های اصلی از ویژگی های فشرده شده است. این کار با استفاده از لایه های کانولوشن و `UpSampling` انجام می شود. لایه های `UpSampling` ابعاد مکانی را به تدریج افزایش می دهند و لایه های کانولوشن دوباره ویژگی ها را بازسازی می کنند. هر لایه کانولوشن در این بخش با فعال سازی `ReLU` به مدل کمک می کند تا جزئیات دقیق تری را بازسازی کند. معماری دیکودر به صورت متقاضی با انکودر طراحی شده است، که این امر تطابق دقیق بین مراحل کاهش و بازسازی ابعاد را تضمین می کند. در نهایت، یک لایه کانولوشن خروجی برای بازسازی داده اصلی با ابعاد اولیه و سه کanal استفاده شده است.

این اتوانکودر ازتابع هزینه `Mean Squared Error` برای به حداقل رساندن اختلاف بین داده های بازسازی شده و داده های اصلی استفاده می کند و از `Adam` به عنوان بهینه ساز برای تنظیم وزن ها بهره می گیرد. ساختار مدل نشان می دهد که تعداد کل پارامترها به دقت محاسبه شده

و تمامی پارامترها قابل آموزش هستند. طراحی این معماری به گونه‌ای انجام شده که دقت بازسازی را بهینه کرده و عملکرد مطلوبی در بازسازی داده‌ها را ارائه دهد.

```
# Define the AutoEncoder model
model = Sequential()

# Encoder
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(img_size, img_size, 3)))
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2), padding='same'))
|
# Decoder
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(3, (3, 3), activation='relu', padding='same'))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])
model.summary()

/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: D
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"

```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 64)	1,792
max_pooling2d (MaxPooling2D)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 128, 128, 128)	73,856
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 128)	0
conv2d_2 (Conv2D)	(None, 64, 64, 256)	295,168
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 256)	0
conv2d_3 (Conv2D)	(None, 32, 32, 256)	590,080
up_sampling2d (UpSampling2D)	(None, 64, 64, 256)	0
conv2d_4 (Conv2D)	(None, 64, 64, 128)	295,040
up_sampling2d_1 (UpSampling2D)	(None, 128, 128, 128)	0
conv2d_5 (Conv2D)	(None, 128, 128, 64)	73,792
up_sampling2d_2 (UpSampling2D)	(None, 256, 256, 64)	0
conv2d_6 (Conv2D)	(None, 256, 256, 3)	1,731

```
Total params: 1,331,459 (5.08 MB)
Trainable params: 1,331,459 (5.08 MB)
Non-trainable params: 0 (0.00 B)
```

شکل ۴۱ – معماری و ساختار مدل AutoEncoder

۲۶- آموزش مدل

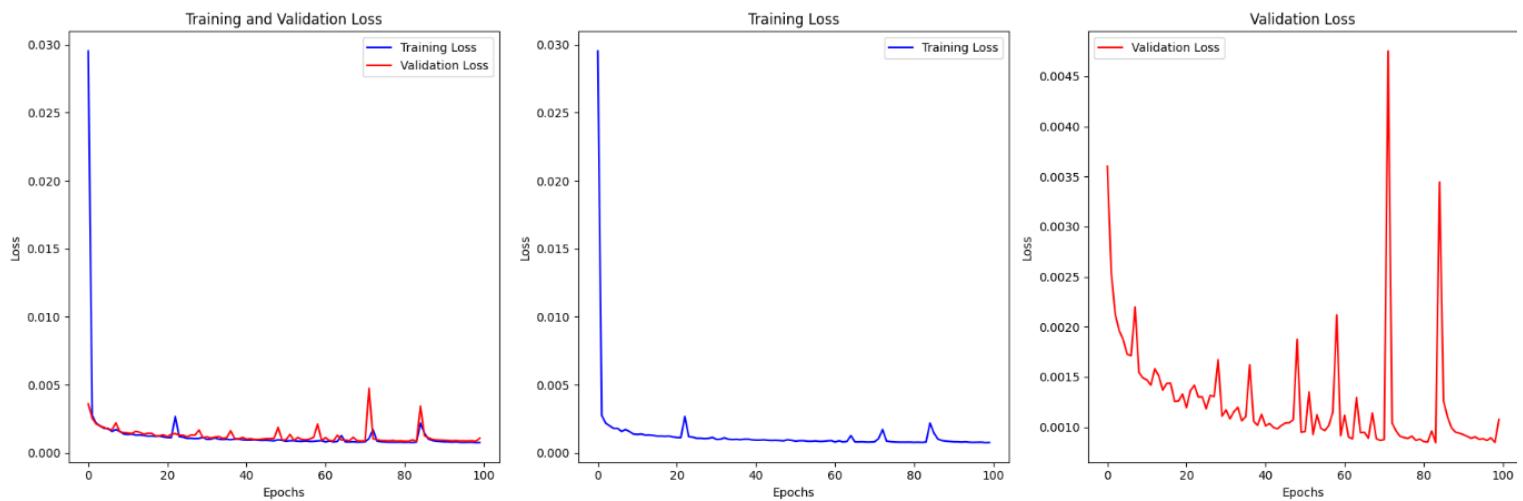
در شکل ۴۲ نشان داده شده است که سایز تصویر ۲۵۶ است. تعداد ایپاک های آموزش ۱۰۰ عدد است. داده ها به صورت رندوم و نامرتب می آیند. پروسه آموزش چاپ میشود و از کل $3900 + 20$ تصویری که برای آموزش درنظر گرفته شده است ۲۰ درصد برای ارزیابی و ۸۰ درصد برای آموزش مدل انتخاب میشوند. مقدار سایز برای هر بچ نیز ۲ در نظر گرفته شده است. حال مدل نوشته شده ببروی اینها `fit` می شود. و در بخش بعدی `predict` را داریم

```
▶ # Train the model
history = model.fit(
    img_final, img_final, # The input data (img_final) is the same as the target data since AutoEncoders learn to
    epochs=100,           # Number of complete passes over the entire dataset during training.
    shuffle=True,         # Shuffle the training data before each epoch to improve learning stability.
    verbose=1,            # Display detailed progress of training, including loss and metrics at each epoch.
    validation_split=0.2, # Use 20% of the data as validation set to evaluate the model during training.
    batch_size = 2        # The default batch size in Keras is 32.
)
```

شکل ۴۲ - آماده سازی برای آموزش مدل ۴ (اتوانکوادر)

۲۷- نتایج مدل

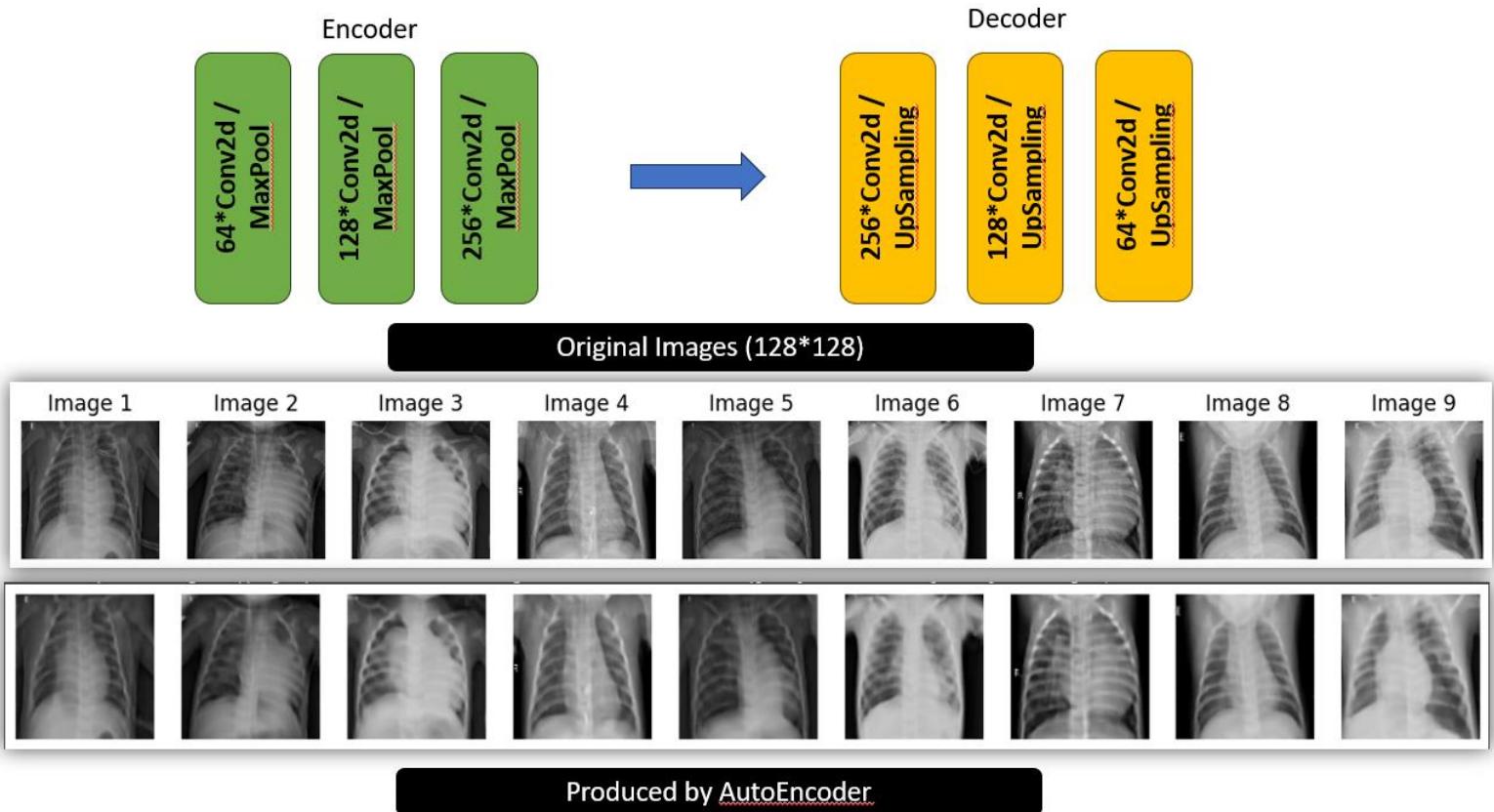
مدل به خوبی آموزش دید و نمودار مربوط به کاهش LOSS در شکل ۴۳ رسم شده است. برای بخش Validation مقداری نوسان (fluctuation) مشاهده میشود که در حوزه یادگیری عمیق، منوط به ایکه روند کلی LOSS در حال کاهش باشد، که اینجا نیز بدینگونه است طبیعی است. بنابراین نمودارهای LOSS به طور کلی قانع کننده است. شایان ذکر است که تمامی آموزش مدل ها در کل این پژوهش با T4 GPU حاوی ۱۶ گیگ حافظه انجام شد.



شکل ۴۳ - Training and validation LOSS for Auto-Encoder training

علاوه بر این، اشکال حاصل شده مدل نیز ذخیره شدند. برای قیاس و اینکه ببینم تصاویر حاصل شده چقدر به تصاویر اولیه شباهت داشتند، که خود معیاری از عملکرد خوب مدل است، ۹ تصویر مصورسازی شده اصلی در ابتدا در اینجا با تصاویر تولید شده برای آنها قیاس شدند. این امر در

شکل ۴۴ بازتاب داده شده است. تصاویر اصلی در ریف بالا و تصاویر تولیدشده با استفاده از Auto Encoder در ریف پایین امده اند که چون بسیار شبیه به هم هستند نوید از عملکرد خوب مدل دارد.



شکل ۴۴ – تصاویر اصلی (ریف بالا) و تصاویر تولیدشده (ریف پایین) با استفاده از Auto Encoder

مدل چهارم، شبکه عصبی مولد تخاصمی (GAN)

نکته: از آنجایی که کد مربوط به شبکه عصبی گن جداگانه زده شده است (نسبت به چهار وظیفه قبلی که در یک فایل پایتونی هستند) نکاتی مجددًا توضیح داده میشود.

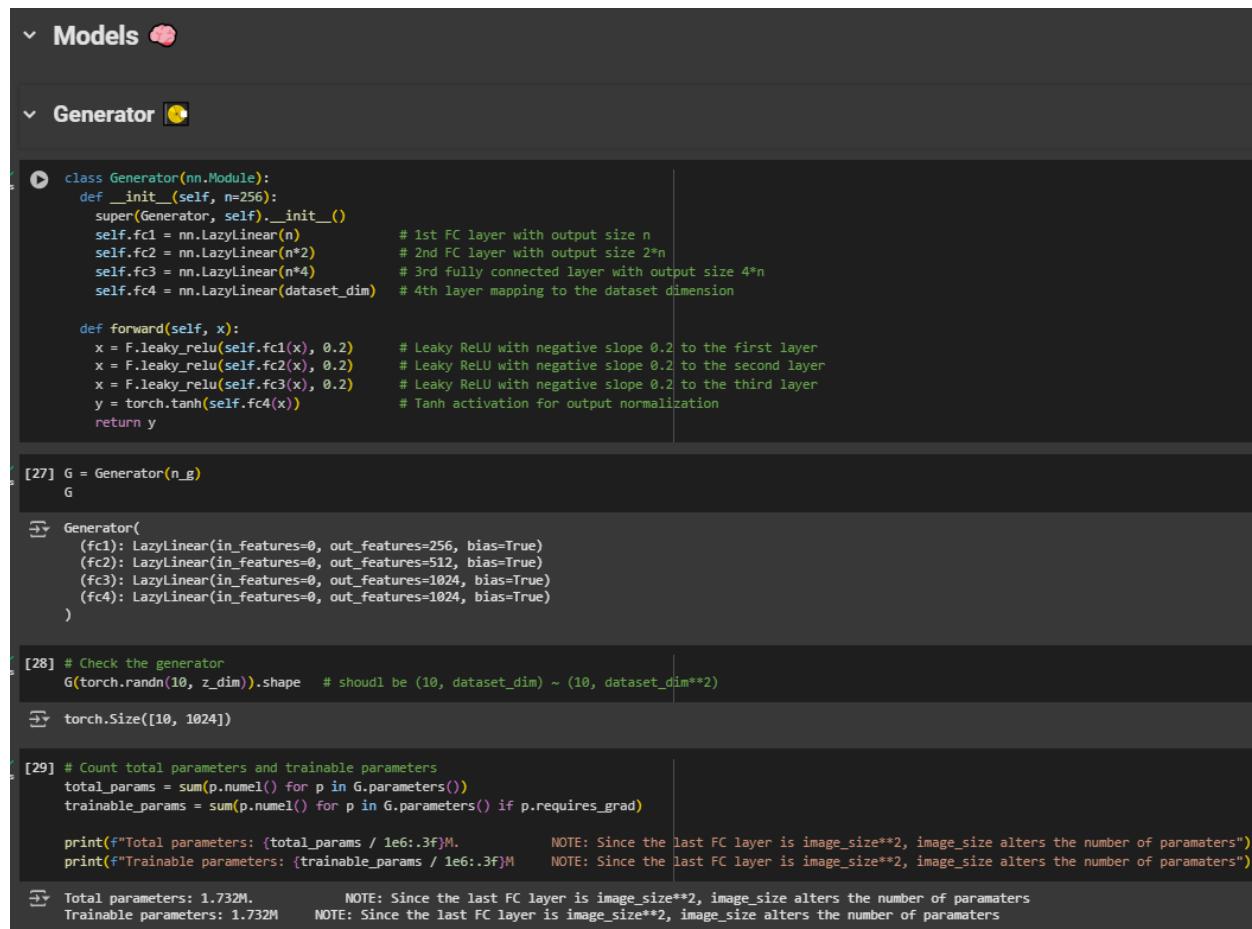
-۲۸- معماری GAN (مولد، ممیز و ترکیب آنها)

بخش مولد این شبکه در شکل ۴۵ نشان داده شده است. رویکرد اقتباسی ما پیاده سازی ساده ترین GAN بود که برای اولین بار طراحی شده بود. در این معماری در بخش مولد ایشان از لایه های FC استفاده شده بود. به طور دقیقتر، این امر برای سایزهای تصویر نه چندان بزرگ مناسب است (حداکثر ۲۵۶ با سخت افزار قوی و سایزهای ۳۲ و ۶۴ با سخت افزار و GPU های متداول). بنابراین داده های ما، هم برای

آموزش موثر (بهتر است بگوییم نسبتاً موثر) و هم برای تعادل قوی بین سرعت و دقت (تنها و تنها در حد کاری آموزشی) به سایز ۳۲ در ۳۲ که سایزی کوچک محسوب می‌شود اما برای کارهای آموزشی مناسب است، تغییر سایز یافتند. در هر حال، شکل ۴۵ معماری مدل پیشنهادی را نشان می‌دهد. چنانچه نشان داده شده است از لایه‌های FC استفاده شده است.

تصویر ورودی در هر سایز که باشد flat می‌شود. برای مثال تصویر ۳۲ در ۳۲ به برداری به طول ۱۰۲۴ تبدیل می‌شود و مشابه کارهای MLP به داخل این شبکه ورود پیدا می‌کند. ضمناً در تصویر مقابل پارامتر n که تعداد نورون‌های اول را نشان میدهد، و تعداد نورون‌های سایر لایه‌ها نیز ضریبی از آن هستند برابر با ۲۵۶ در نظر گرفته شده است. بنابراین میتوان گفت (طبق متن پرینت شده پس از ران کردن مولد)، لایه اول دارای ۲۵۶ نورون، لایه دوم دارای ۵۱۲ نورون و لایه‌های سوم و چهارم هر یک دارای ۱۰۲۴ نورون هستند.

به عبارتی دیگر بردار حاصل از تصویر ورودی که دارای سایز ۱۰۲۴ است ابتدا به ۲۵۶ تقلیل می‌یابد. سپس ۲۵۶ به ۵۱۲ منبسط می‌شود. پس از آن نیز لایه ۵۱۲ به ۱۰۲۴ منبسط شده و یک لایه ۱۰۲۴ دیگر نیز برای ترکیب ویژگی‌های محلی ایجاد شده در لایه‌های پیشین و ایجاد ویژگی‌های عمیق‌تر و سطح بالاتر گذاشته می‌شود. این دو لایه ۱۰۲۴ عملای مسئول استخراج ویژگی‌های عمیق‌تر هستند. در این حالت، تعداد کل پارامترهای مولد نیز حدوداً ۱.۷ میلیون محاسبه شد که عدد بزرگی نیست.



```

Models 🧠
Generator 🎛

class Generator(nn.Module):
    def __init__(self, n=256):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(n)
        self.fc2 = nn.Linear(n**2)
        self.fc3 = nn.Linear(n**4)
        self.fc4 = nn.Linear(dataset_dim)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = F.leaky_relu(self.fc3(x), 0.2)
        y = torch.tanh(self.fc4(x))
        return y

[27] G = Generator(n_g)
G

Generator(
  (fc1): Linear(in_features=0, out_features=256, bias=True)
  (fc2): Linear(in_features=0, out_features=512, bias=True)
  (fc3): Linear(in_features=0, out_features=1024, bias=True)
  (fc4): Linear(in_features=0, out_features=1024, bias=True)
)

[28] # Check the generator
G(torch.randn(10, z_dim)).shape # should be (10, dataset_dim) ~ (10, dataset_dim**2)

torch.Size([10, 1024])

[29] # Count total parameters and trainable parameters
total_params = sum(p.numel() for p in G.parameters())
trainable_params = sum(p.numel() for p in G.parameters() if p.requires_grad)

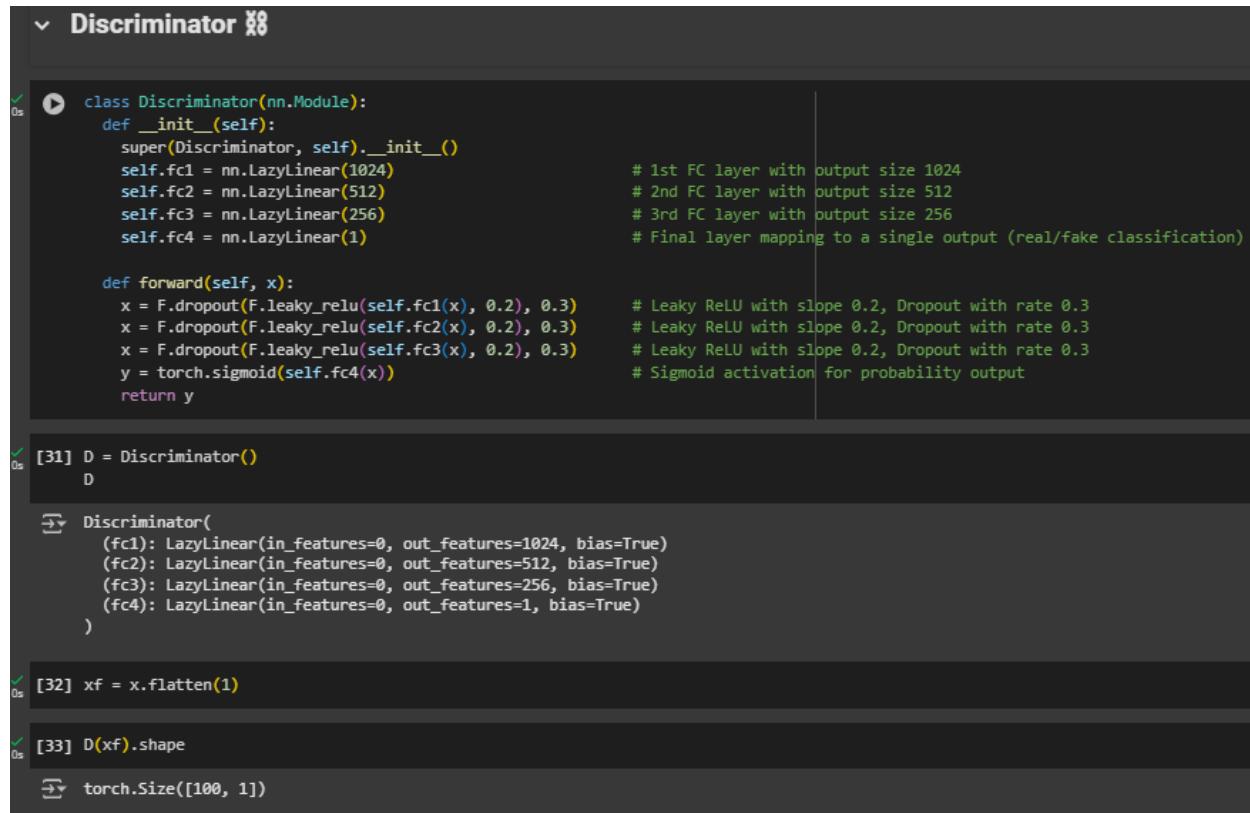
print(f"Total parameters: {total_params / 1e6:.3f}M.          NOTE: Since the last FC layer is image_size**2, image_size alters the number of parameters")
print(f"Trainable parameters: {trainable_params / 1e6:.3f}M      NOTE: Since the last FC layer is image_size**2, image_size alters the number of parameters")

Total parameters: 1.732M.          NOTE: Since the last FC layer is image_size**2, image_size alters the number of parameters
Trainable parameters: 1.732M      NOTE: Since the last FC layer is image_size**2, image_size alters the number of parameters

```

شکل ۴۵ – معماری و جزئیات مولد در شبکه عصبی گن

در تصویر ۴۶ معماری ممیز نشان داده شده است. وظیفه ممیز در اینجا دسته بندی داده وارد شده است. چونکه مسئله طبقه بندی دو کلاسه است در انتهای فقط و فقط باید یک نورون داشته باشیم. چنانچه در شکل ۴۶ نشان داده شده است، در لایه های مختلف ممیز به ترتیب ۱۰۲۴، ۵۱۲ و ۲۵۶ نورون داریم و سپس یک نورون برای طبقه بندی نهایی.



```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(1024) # 1st FC layer with output size 1024
        self.fc2 = nn.Linear(512) # 2nd FC layer with output size 512
        self.fc3 = nn.Linear(256) # 3rd FC layer with output size 256
        self.fc4 = nn.Linear(1) # Final layer mapping to a single output (real/fake classification)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.2, 0.3) # Leaky ReLU with slope 0.2, Dropout with rate 0.3
        x = F.leaky_relu(self.fc2(x), 0.2, 0.3) # Leaky ReLU with slope 0.2, Dropout with rate 0.3
        x = F.leaky_relu(self.fc3(x), 0.2, 0.3) # Leaky ReLU with slope 0.2, Dropout with rate 0.3
        y = torch.sigmoid(self.fc4(x)) # Sigmoid activation for probability output
        return y

D = Discriminator()
D

Discriminator(
  (fc1): Linear(in_features=1024, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=1, bias=True)
)

```

```

x = x.flatten(1)
D(x).shape
torch.Size([100, 1])

```

شکل ۴۶ - معماری و جزئیات ممیز شبکه عصبی گن

در نهایت نیز، مطابق شکل ۴۷، هر دو به دستگاه منتقل شدند و یکبار دیگر چک شدند.

```

Build Models

In [34]: G = Generator().to(device)
          G

In [35]: D = Discriminator().to(device)
          D

In [36]: Discriminator(
              (fc1): LazyLinear(in_features=0, out_features=1024, bias=True)
              (fc2): LazyLinear(in_features=0, out_features=512, bias=True)
              (fc3): LazyLinear(in_features=0, out_features=256, bias=True)
              (fc4): LazyLinear(in_features=0, out_features=1, bias=True)
          )

```

شکل ۴۷ – بردن مولد و ممیز به دستگاه و تست آنها

۲۹- تابع خطأ و بهینه ساز

تابع خطأ BCE LOSS و بهینه ساز نیز Adam انتخاب شده است شایان ذکر است که هم LOSS و هم Optimizer برای Generator و Discriminator باید جداگانه تعیین شوند. دلیل استفاده از Adam نیز همگرایی سریع و به اصطلاح good enough بودن بهتری و نیاز به بهینه سازی ابرمولفه (هایپرپارامترهای) کمتری است.

۳۰- هایپرپارامترها

توضیحات و تعریف هایپرپارامترهای GAN (که در شکل ۴۸ آورده شده اند)

هایپرپارامترهای تنظیم شده در این بخش به منظور آموزش مدل GAN تعریف شده‌اند. هر یک از این پارامترها نقش مهمی در عملکرد نهایی مدل دارند و در ادامه به تفصیل توضیح داده شده‌اند:

۱. **Device**: این پارامتر تعیین می‌کند که مدل روی کدام سختافزار اجرا شود. اگر GPU در دسترس باشد، مدل بر روی آن اجرا خواهد شد (برای افزایش سرعت محاسبات)، در غیر این صورت روی CPU اجرا می‌شود. در این کار، مدل به قدری سریع بود که با سایز تصویر ۳۲ در ۳۲ بر روی CPU هر ایپاک حدود ۰.۳۲ دقیقه طول می‌کشید.

۲. **batch_size**: ایندازه‌ی هر دسته از نمونه‌هایی که به طور همزمان به مدل ارائه می‌شوند، برابر با ۱۰۰ تنظیم شده است. این پارامتر تعیین می‌کند که در هر مرحله از آموزش، چند داده به مدل وارد شود و مستقیماً بر روی حافظه و سرعت اجرا تأثیرگذار است.

۳. **z_dim**: ایندازه‌ی فضای نویز ورودی (latent space) است که برابر با ۱۰۰ تنظیم شده است. این مقدار تعداد ویژگی‌هایی را تعیین می‌کند که به عنوان ورودی برای شبکه تولیدکننده (Generator) استفاده می‌شود.

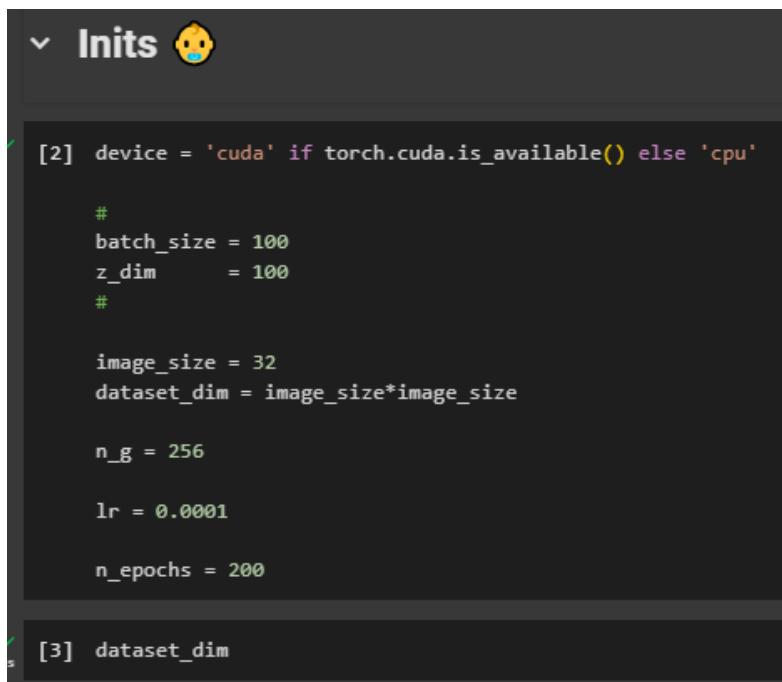
۴. اندازه‌ی هر تصویر ورودی (ارتفاع و عرض) است که برابر با ۳۲ پیکسل در نظر گرفته شده است.

۵. تعداد کل پیکسل‌های هر تصویر را نشان می‌دهد. این مقدار برابر است با $\text{image_size} * \text{image_size}$ و در اینجا برابر با $1024 * 1024$ پیکسل محاسبه شده است. به صورت Soft Code نوشته شده است تا در صورت تغییر سایز تصویر، این نیز تغییر کند.

۶. تعداد نورون‌های لایه‌ی اول شبکه‌ی تولیدکننده (Generator) است که برابر با ۲۵۶ در نظر گرفته شده است. این مقدار تأثیر مستقیمی بر ظرفیت شبکه‌ی تولیدکننده دارد.

۷. **n_g:** نرخ یادگیری که برابر با 0.0001 تنظیم شده است. این مقدار سرعت تنظیم وزن‌ها را در طی فرآیند بهروزرسانی مشخص می‌کند. نرخ یادگیری پایین‌تر معمولاً به نتایج پایدارتر اما آموزش طولانی‌تر منجر می‌شود.

۸. **n_epochs:** تعداد کل دوره‌های آموزشی یا به عبارتی تعداد دفعات عبور از کل مجموعه داده است که برابر با 200 دوره تعیین شده است.



```
[2] device = 'cuda' if torch.cuda.is_available() else 'cpu'

#
batch_size = 100
z_dim      = 100
#

image_size = 32
dataset_dim = image_size*image_size

n_g = 256

lr = 0.0001

n_epochs = 200

[3] dataset_dim
```

شکل ۴۸ – هایپرپارامترهای گن

نکته دیگر حائز اهمیت در رابطه با هایپرپارامترهای GAN: با مرور ادبیات دریافته شد که برای GAN، تقریباً با هر معماری که باشد، بهتر است از تابع فعالسازی Leaky Relu و با شبیب بخش منفی استفاده شود. ضمناً این شبکه، با هر معماری معمولاً با نرخ یادگیری بسیار کوچک (در حد 1×10^{-4}) و با تعداد ایپاک بالا (در حد 5000) آموزش داده می‌شود. کدهای بررسی شده در GitHub و Kaggle معمولاً بدینگونه بود.

۳۱- چرخه آموزش مدل:

```
loss_D_hist, loss_G_hist = [], []  
*  
# import os  
  
# # Create the images directory if it doesn't exist  
# os.makedirs('./images', exist_ok=True)  
  
import os  
import shutil  
  
# Create the images directory if it doesn't exist  
os.makedirs('./images', exist_ok=True)  
  
# Clear the images directory if it contains any files  
for filename in os.listdir('./images'):  
    file_path = os.path.join('./images', filename)  
    try:  
        if os.path.isfile(file_path) or os.path.islink(file_path): # Check if it's a file or symbolic link  
            os.unlink(file_path) # Remove file or symbolic link  
        elif os.path.isdir(file_path): # Check if it's a directory  
            shutil.rmtree(file_path) # Remove directory and its contents  
    except Exception as e:  
        print(f'Failed to delete {file_path}. Reason: {e}')  
*  
import time # Import the time library for measuring epoch duration  
  
# Initialize variables to track the minimum Generator loss and corresponding epoch  
min_G_loss = float('inf') # Set initial minimum Generator loss to infinity
```

```
min_G_loss_epoch = -1    # Initialize the epoch of the minimum loss

for epoch in range(n_epochs):
    # Start timing the epoch
    epoch_start_time = time.time()                                # Record the start time of the epoch

    # Initialize average meters to track loss for Discriminator and Generator
    loss_D_avg, loss_G_avg = AverageMeter(), AverageMeter()

    for x in dataloader:
        # -----
        # Discriminator
        # -----
        D.zero_grad()                                              # Reset gradients for Discriminator

        # Process real data
        x_real = x.flatten(1).to(device)                            # Flatten and move real data to device
        y_real = torch.ones(x_real.size(0), 1).to(device)          # Dynamically create real labels based
        on batch size

        # Forward pass real data through Discriminator
        D_output = D(x_real)                                      # Discriminator's output on real data
        D_loss = loss_fn(D_output, y_real)                          # Compute loss for real data

        # Generate fake data
        z = torch.randn(x_real.size(0), z_dim).to(device)          # Dynamically sample noise
        y_fake = torch.zeros(x_real.size(0), 1).to(device)          # Dynamically create fake labels

        # Generate fake data using Generator
```

```
x_fake = G(z)                                # Pass noise through Generator

# Forward pass fake data through Discriminator
G_output = D(x_fake)                          # Discriminator's output on fake data
G_loss = loss_fn(G_output, y_fake)            # Compute loss for fake data

# Total Discriminator loss
D_loss = G_loss + D_loss                      # Combine real and fake losses

# Backpropagation and optimization for Discriminator
D_loss.backward()                            # Backpropagate
D_optimizer.step()                           # Update Discriminator weights

# -----
# Generator
# -----
G.zero_grad()                               # Reset gradients for Generator

# Generate noise and labels for Generator training
z = torch.randn(batch_size, z_dim).to(device)      # Sample noise
y_real = torch.ones(z.size(0), 1).to(device)        # Dynamically create real labels for
Generator

# Forward pass: noise -> Generator -> Discriminator
output = D(G(z))                            # Discriminator's output on fake data from
Generator

# Compute Generator loss
G_loss = loss_fn(output, y_real)             # Loss for Generator (fooling Discriminator)
```

```

# Backpropagation and optimization for Generator

G_loss.backward()                                # Backpropagate
G_optimizer.step()                               # Update Generator weights

# Update loss trackers
loss_D_avg.update(D_loss.item())                # Update average Discriminator loss
loss_G_avg.update(G_loss.item())                # Update average Generator loss

# Save losses for each epoch
loss_D_hist.append(loss_D_avg.avg)              # Save average Discriminator loss
loss_G_hist.append(loss_G_avg.avg)              # Save average Generator loss

# Update the minimum Generator loss and its epoch
if loss_G_avg.avg < min_G_loss:                  # Check if the current Generator loss is lower
    min_G_loss = loss_G_avg.avg                  # Update the minimum loss
    min_G_loss_epoch = epoch                     # Update the epoch of the minimum loss

# Generate and save sample images
with torch.no_grad():                           # Disable gradients for evaluation
    bs = 16                                     # Batch size for sample generation
    z_test = torch.randn(bs, z_dim).to(device)    # Sample noise
    outputs = G(z_test)                          # Generate fake images

# Create a grid of images in 4x4 format
grid_img = make_grid(outputs.view(bs, 1, image_size, image_size), nrow=4, normalize=True) # nrow=4 for 4 rows and 4 columns

# Save the grid with larger figure size to a file

```

```

plt.figure(figsize=(12, 12))          # Set figure size to make it larger
plt.imshow(grid_img.permute(1, 2, 0).cpu().numpy(), cmap="gray")    # Display the grid of images
plt.axis('off')                      # Turn off axes
plt.savefig(f'./images/gan-{epoch:03d}.png', bbox_inches='tight')   # Save the plot with larger size
plt.close()                          # Close the plot to free memory

# Calculate and display epoch duration
epoch_duration = (time.time() - epoch_start_time) / 60           # Calculate epoch duration in minutes
print(f'Epoch {epoch}/{n_epochs} Duration: {epoch_duration:.2f} minutes') # Print epoch duration in minutes

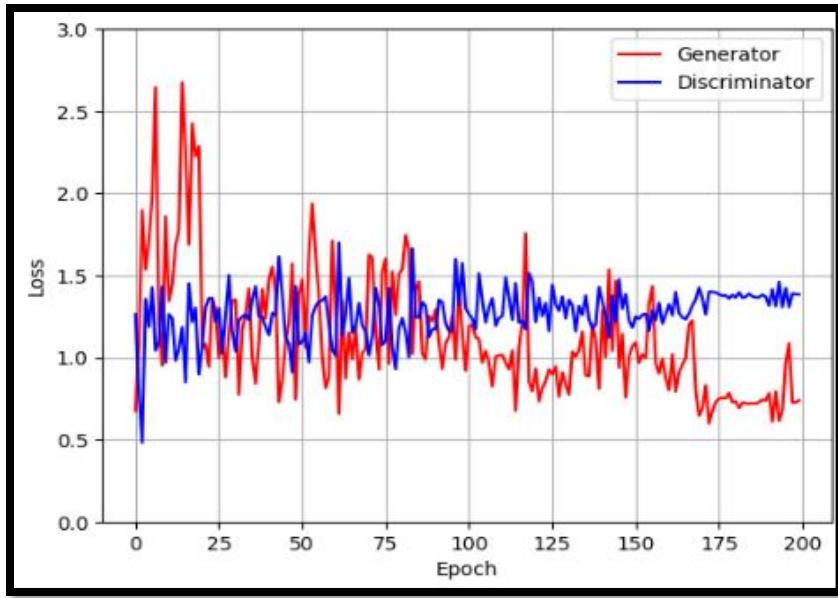
# Print training progress
print(f'D Loss: {loss_D_avg.avg:.4f}, G Loss: {loss_G_avg.avg:.4f}')
print("-----")

# Print two empty lines
print("\n\n")

# Print the minimum Generator loss and its epoch
print(f'Minimum Generator Loss: {min_G_loss:.4f} at Epoch: {min_G_loss_epoch}')
*

```

کد مربوط به آموزش مدل در بالا آورده شده است. این کد در ۴-۳ سلول ران شده است که برای جداسازی آنها از رنگ زرد و ستاره استفاده شده است. نهایتا نتایج حاصل شده در قالب نمودار تابع خطا در شکل ۴۹ نشان داده شده است.



شکل ۴۹ – نمودار تابع زیان برای GAN. هم برای مولد و هم برای ممیز

نکته حائز اهمیت آنکه هدف اصلی شبکه عصبی GAN، تولید داده هایی مشابه است. برای این منظور باید مولدی قوی داشته باشیم. یعنی هدف غایی شبکه عصبی گن کاهش خطای مولد است که بالطبع خطای ممیز افزایش می یابد. با مطالعه شبکه های گن دریافتہ شده که هدف اصلی کاهش هرچه بیشتر زیان مولد است که در این بین افزایش زیان شبکه ممیز اجتناب ناپذیر است. اگر یک گن ایده آل و با دسترسی به سخت افزارهای قوی آموزش داده شود، و بالطبع با سایز تصویر مناسب (حداقل ۲۵۶)، انتظار میرود که نمودارها بسیار کم نوسان باشند ولی زیان مولد رو به کاهش و زیان ممیز رو به افزایش. لکن چون در اینجا اصلا داخل گن (چه مولد و چه ممیز) از Conv2d و MaxPool استفاده نشده است و ساده ترین نوع GAN آنهم با لایه های FC بوده است و بدتر از آن با سایز تصویر ۳۲ که بسیار بسیار کوچک است، نمودار تابع زیان مذکور به نظر اینجانب بسی خوب و شایان است. خط قرمزرنگ یا همان خطای مولد رو به کاهش و خطای ممیز یا همان خط آبی رنگ رو به افزایش است، حتی شده به صورت کلی. ضمنا معمولا GAN را ۵۰۰۰-۶۰۰۰ ایپاک آموزش می دهند اما در اینجا تنها ۲۰۰ ایپاک آموزش داده است که بیانگر محدودیت های این تحقیق است.

متن بالا توسط اینجانب نوشته شده بود، متن زیر با توجه به شکل توضیح دقیق تری از ChatGPT است. 😊

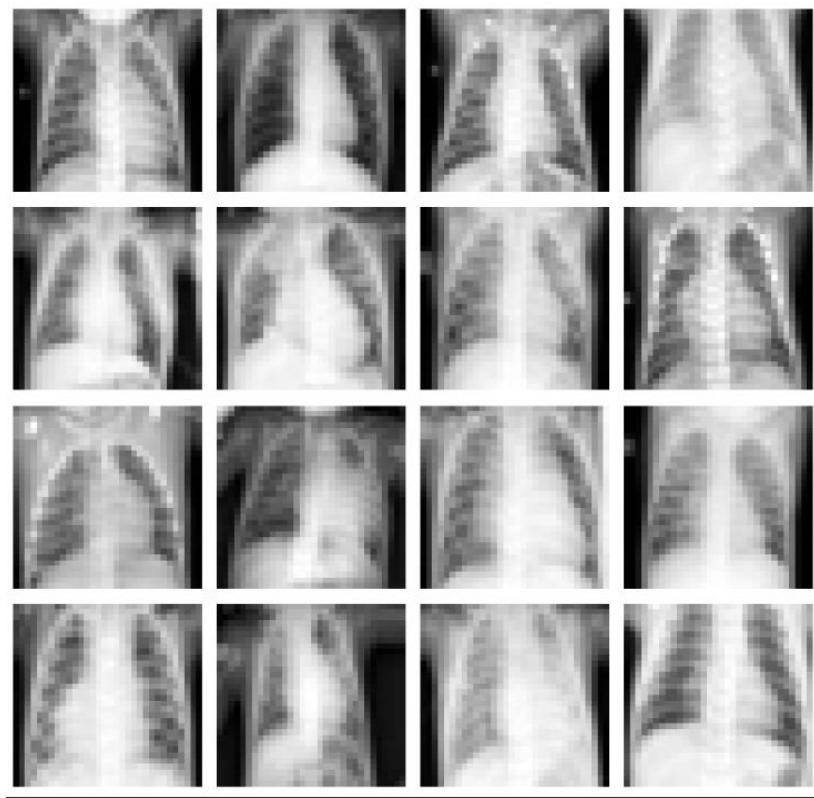
نمودار ارائه شده نشان دهندهی زیان مولد و ممیز در طی ۲۰۰ دوره آموزش شبکه GAN است. همان طور که مشاهده می شود، خطای مولد (خط قرمز) روندی نزولی دارد که نشان دهنده بهبود عملکرد مولد در تولید داده های واقعی نما است. در مقابل، خطای ممیز (خط آبی) به صورت کلی افزایش یافته است، که این رفتار نیز منطقی و مطابق با اهداف شبکه GAN می باشد. هدف اصلی در چنین شبکه ای کاهش زیان مولد است، چرا

که مولد با تولید داده‌های مشابه داده‌های واقعی ممیز را در تشخیص داده‌های جعلی به چالش می‌کشد. البته نمودار دارای نوسانات قابل توجهی است که بخشی از آن به دلیل ساختار ساده‌ی شبکه، یعنی استفاده از لایه‌های کاملاً متصل (FC) و بعد کوچک تصاویر (32×32)، بازمی‌گردد. این موارد محدودیت‌هایی را در استخراج ویژگی‌های دقیق و عمیق ایجاد کرده و بر پایداری عملکرد مدل تأثیرگذار بوده است.

علاوه بر این، تعداد ایپاک‌های محدود (۲۰۰ دوره) نیز یکی دیگر از عوامل تأثیرگذار بر نتایج بوده است. آموزش GAN‌ها معمولاً در دوره‌های بسیار طولانی‌تر (۵۰۰۰ تا ۶۰۰۰ ایپاک) انجام می‌شود تا مدل فرصت کافی برای همگرای و کاهش نوسانات داشته باشد. با این حال، روند کلی کاهش خطای مولد و افزایش خطای ممیز، حتی در این شرایط، نشان‌دهنده‌ی حرکت مدل در مسیر صحیح است. اگرچه این نتایج با محدودیت‌های یادشده همراه است، اما می‌توان آن را عملکردی مطلوب تلقی کرد، به ویژه با توجه به پیچیدگی محاسباتی کمتر و شرایط ساده‌شده این پژوهش. به طور کلی، نمودار زیان تولیدکننده و ممیز، با وجود نوسانات و محدودیت‌ها، عملکرد مناسبی را از خود نشان داده و نشانگر تحقق نسبی اهداف آموزش شبکه است.

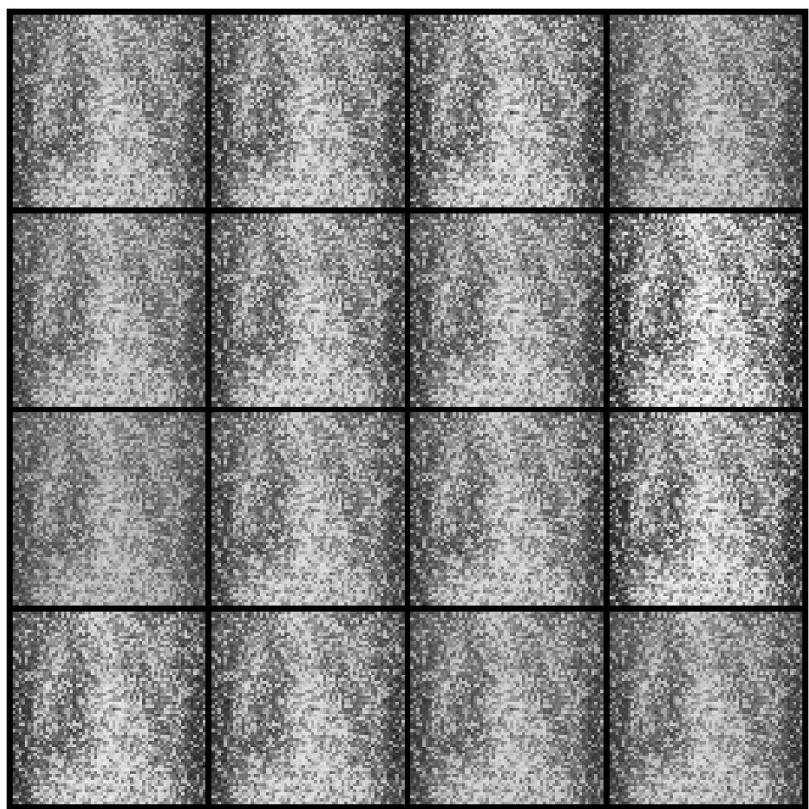
۳۲- تصاویر تولیدی توسط GAN و قیاس آن با تصاویر ورودی

شکل ۵۰ تصاویر ورودی به مدل را نشان میدهد. سایز این تصاویر ۳۲ در ۳۲ می‌باشد که سبک باشند و به راحتی قابل اجرا باشند. از طرفی دیگر تصاویر کوچک معمولاً نتایج خوبی را به بار نمی‌آورند. بنابراین انتظار تصاویری با کیفیت بالا را نداریم. صرفاً فرآیند آموزش مدل مهم است و اینکه نتایج مربوط به کاهش خطای مولد منطقی باشد که در شکل ۴۹ به آن دست یافته شده است.

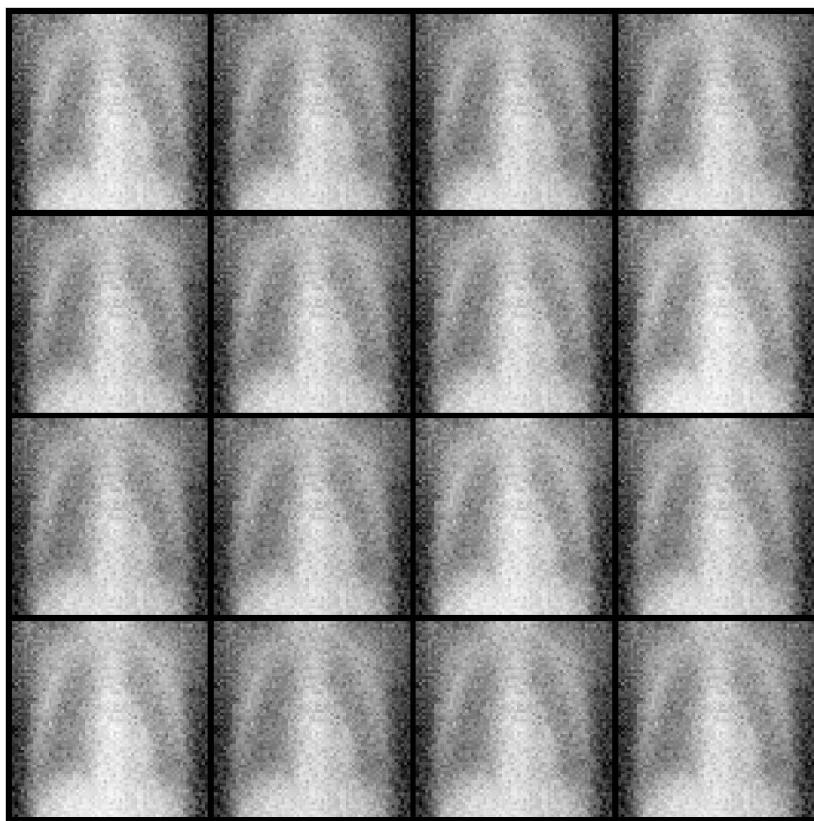


شکل ۵۰ - تصاویر ورودی به مدل، دارای سایز ۳۲

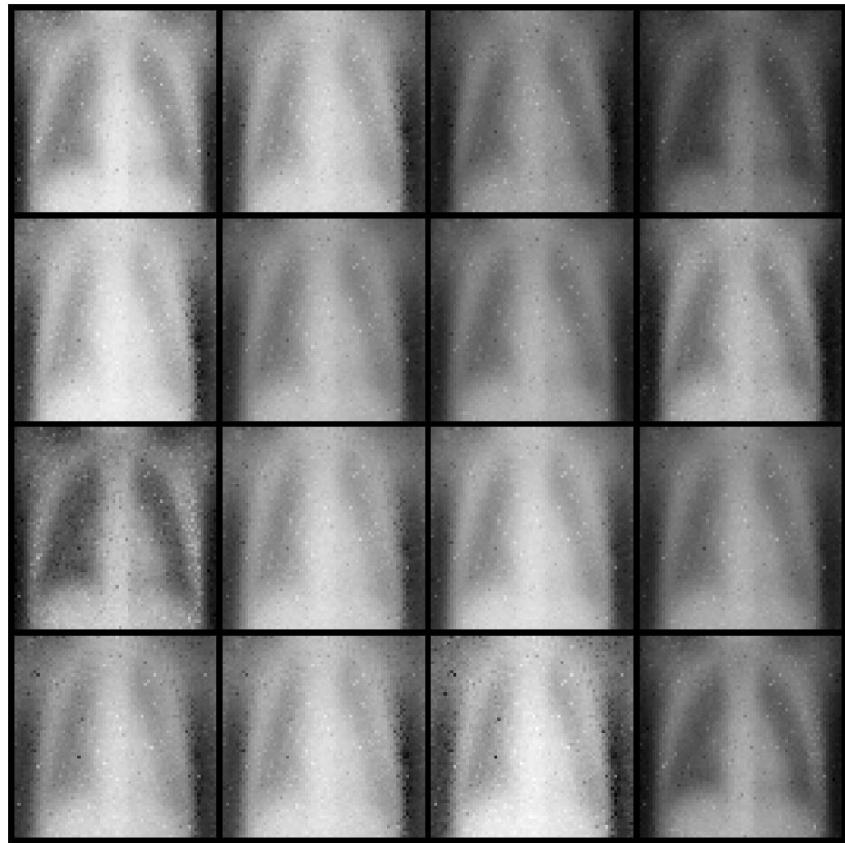
اشکال ۵۱، ۵۲، ۵۳ و ۵۴ تصاویر تولیدی مدل در طول فرآیند آموزش را نشان می‌دهند. روند کلی فرآیند آموزش مبنی بر اینکه در طول فرآیند بهبود می‌یابند نیز درست است، هرچند که بهینه نیستند!



شکل ۵۱ - تصویر تولیدشده در ایپاک ۱ (فقط یک ایپاک)



شکل ۵۲ – تصویر تولیدشده در ایپاک ۱۰۰



شکل ۵۳ – تصویر تولیدشده در ایپاک ۲۰۰

به پایان رسید دفتر، حکایت همچنان باقی است!

با تشکر فراوان از زحمات فراوان خانم دکتر خطیبی.