به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

پرسش ۱	نام و نام خانوادگی	آرمین قاسمی
	شماره دانشجویی	۸۱۰۱۰۰۱۹۸
پرسش ۲	نام و نام خانوادگی	امیرحسین ثمودی
	شماره دانشجویی	۸۱۰۱۰۰۱۰۸
	مهلت ارسال پاسخ	۱۴۰۳.۱۲.۲۷

فهرست

پرسش ۱ – یادگیری بدون نظارت و انتقال دامنه با استفاده از GAN..... ۵

۲-۱: بخش نظری ۵

۱-۲-۱: ۵

۲-۲-۱: ۶

۳-۲-۱: ۶

۴-۲-۱: ۷

۵-۲-۱: ۷

۶-۲-۱: ۸

۳-۱: بخش عملی ۸

پیش‌پردازش داده‌ها ۸

آموزش مدل پایه و مشاهده **Domain Gap**: ۱۰

پیاده‌سازی معماری مدل: ۱۲

پیاده‌سازی توابع هزینه: ۱۵

آموزش مدل: ۱۷

نمایش نتایج: ۱۸

پرسش ۲ – بازسازی تصاویر پولیپ آندوسکوپی با EndoVAE ۲۲

۲-۱: پیش پردازش داده ها ۲۲

۲-۲: طراحی معماری EndoVAE ۲۵

۲-۳: تعریف توابع هزینه ۲۸

۲-۴: روند آموزش مدل ۲۹

۲-۵: تولید و بازسازی کیفی ۳۰

۲-۶: ارزیابی عددی ۳۳

۲-۷: تحلیل نتایج و بحث نهایی ۳۳

شکل‌ها

شکل ۱: تصاویر متناظر از مجموعه داده های MNIST و MNIST-M ۸
شکل ۲: معماری Classifier برای MNIST ۱۰
شکل ۳: جزئیات پیاده سازی Classifier ۱۱
شکل ۴: نمودار دقت و خطای مدل در طول آموزش ۱۲
شکل ۵: معماری کلی مدل ۱۳
شکل ۶: نمودار دقت و خطای مدل در طول آموزش ۱۸
شکل ۷: مقایسه تصاویر دامنه منبع و هدف و تصاویر تولید شده متناظرشان ۲۰
شکل ۱ سه نمونه از داده های نرمال ۲۲
شکل ۲ سه نمونه از داده های ناهنجار ۲۳
شکل ۳ مراحل پیش پردازش داده نمونه۲ ..... ۲۳
شکل ۴ مراحل پیش پردازش داده نمونه۱ ..... ۲۳
شکل ۵ مراحل پیش پرداز داده نمونه۳ ..... ۲۴
شکل ۶ سه نمونه تصویر به همراه تصاویر آگمنت شده از آنها ۲۴
شکل ۷ خروجی دیکودر در مود تولید ۳۰
شکل ۸ پنج نمونه از داده های پولیپ همراه با تصاویر بازسازی شده آنها توسط مدل ۳۱
شکل ۹ پنج مورد از داده های سالم دیتاست validation همراه با بازسازی آنها ۳۲

جدول‌ها

جدول ۱: مقایسه دقت **Classifier** قبل و بعد از اعمال **Domain-shift** ۲۰

پرسش ۱ – یادگیری بدون نظارت و انتقال دامنه با استفاده از GAN

۲-۱: بخش نظری

۱-۲-۱:

شبکه های GAN به خاطر تعامل رقابتی که بین Generator و Discriminator بوجود می‌آورند و اهداف متفاوتی که با هم دارند میتوانند در حین آموزش ناپایدار شوند. (در تقریبا همه مدل‌هایی که قبلا دیده بودیم، هدف آموزش مینیمم کردن تابع هزینه بود، حال آنکه در این شبکه‌ها بهینه سازی یک فرآیند MinMax می‌باشد و پیدا کردن نقطه تعادلی که هر دو شبکه عملکرد مطلوبی داشته باشند ساده به‌نظر نمی‌رسد.

یکی از عوامل اصلی بی‌ثباتی عدم همگرایی مدل های Generator و Discriminator می‌باشد، زیرا این دو شبکه در یک چرخه ای قرار میگیرند که دائما سعی در بهتر کردن عملکرد خود در ازای تضعیف عملکرد طرف مقابل اند. حال اگر در فرآیند آموزش به جای یادگیری عمیق و بهبود عملکرد تدریجی این بخش‌ها، وزن های شبکه به هر تغییر از طرف مقابل واکنش متقابل نشان دهند، مدل بجای آموزش عمیق تنها دچار نوسان میشود و به نتیجه خوبی نمیرسد. همچنین اگر سرعت یادگیری مدل ها تفاوت زیادی داشته باشد (برای مثال Discriminator در ابتدا بیش از حد قوی باشد و همه تصاویر مولد را تقلبی تشخیص دهد)، شبکه Generator دیگر فرصت این را ندارد که خود را با آن تطبیق بدهد.

عامل دوم ناپایداری را Mode Collapse می‌نامند، به این معنا که نمونه های تولیدی Generator تنها محدود به چند کلاس میشوند و نمونه هایی از کلاس های دیگر را تولید نمیکند که باعث میشود توزیع تصاویر تولید شده دیگر مانند توزیع ورودی نباشد. این اتفاق وقتی رخ میدهد که Generator با تولید نمونه هایی از یک سری کلاس های خاص، راحت‌تر Discriminator را فریب میدهد. در نتیجه خروجی اش را محدود به همان چند کلاس میکند و تولید خروجی هایی از کلاس های دیگر را به کلی کنار میگذارد و یاد نمیگیرد.

عامل سوم ناپایداری Vanishing Gradient می‌باشد. اگر Discriminator خیلی سریعتر از مولد آموزش داده شود و عملکرد بهتر پیدا کند، با اطمینان بالا همه تصاویر مولد را تقلبی تشخیص داده و خروجی اش خیلی نزدیک به صفر میشود. در نتیجه گرادیان تابع هزینه صفر شده و فرآیند آموزش مولد خیلی آهسته یا متوقف می‌شود.

یکی از مکانیزم‌ها برای مقابله با ناپایداری ها (بخصوص مورد اول)، اضافه کردن نویز به ورودی لایه های Discriminator و اعمال Dropout هست (که در این مقاله نیز انجام شده) تا از غالب شدن آن بر مولد جلوگیری کنیم. برای رفع مشکل Mode Collapse میتوان توابع هزینه ای را مشابه مقاله تعریف کرد (Task Classifier Loss) که مولد را مجبور کند هنگام تولید تصویر جدید محتوای آن را حفظ نماید تا کلاسی در فرآیند آموزش کنار گذاشته نشود. این توابع هزینه همچنین میتوانند مشکل Vanishing gradient را در ابتدای آموزش حل کنند و از آن جلوگیری کنند. همچنین برای پایداری گرادیان‌ها، به جای تولید برچسب های ۰ و ۱، از برچسب های نرم تری (مثلا ۰.۱ و ۰.۹) نیز میتوان استفاده نمود.

:۱-۲-۲

در این مقاله با استفاده از تصویر اصلی و نویز به عنوان ورودی Generaator، تصویر خروجی مولد را در حقیقت کنترل می‌کنیم. در شبکه های معمول GAN که تنها بردار نویز تصادفی به ورودی مولد داده می‌شود، دیگر کنترلی روی محتوای تصویر خروجی نداریم. ولی با دادن تصویر ورودی، مولد را مجبور میکنیم که تصویر تولید شده محتوای مشابهی با آن داشته باشد و تنها Style آن تصادفی و با نویز ورودی مشخص شود.

این کار باعث میشود که توزیع کلاس های تصاویر تولید شده مشابه تصاویر ورودی باشد (Mode Collapse رخ ندهد) و آموزش پایدارتر شود. همچنین فرآیند آموزش مولد نیز ساده‌تر می‌شود، زیرا در حالتی که ورودی نویز هست، شبکه باید توزیع های پیچیده ای را یاد بگیرد که از یک بردار تصادفی نویز قادر به تولید تصاویر فیک با کیفیت (و دارای محتوا) باشد، ولی در این حالت مولد تنها یک نوع نگاشت بین دامنه منبع و دامنه هدف (با تغییر ویژگی های ظاهری) را یاد می‌گیرد.

:۱-۲-۳

نقش Generator در این شبکه تولید تصاویر فیک از روی تصاویر دامنه منبع است، به گونه ای که با حفظ محتوا و تغییر ظاهر، آن ها را به تصاویر دامنه هدف مشابه کند. شاید بتوان گفت که این اصلی ترین بخش شبکه برای رسیدن به هدف تغییر دامنه Unsupervised هست، چون که باید تنها با استفاده از تصاویر لیبل دار منبع، تصاویر مصنوعی برای دامنه هدف تولید کنیم.

نقش Discriminator تشخیص تصاویر جعلی از واقعی است. این شبکه هست که مولد را مجبور میکند تصاویر واقع‌گرایانه‌تر و نزدیک تر به دامنه هدف تولید کند.

Classifier در انتها برای طبقه بندی تصاویر استفاده میشود و در حقیقت همان بخشی است که پس از فرآیند آموزش میخواهیم از آن استفاده کنیم و هدف نهایی آموزش مناسب آن برای عملکرد خوب روی

دامنه هدف است. ولی در حین آموزش با توجه به نحوه تعریف توابع هزینه، وظیفه نظارت بر خروجی های تولیدی مولد را نیز دارد تا محتوای تصاویر را تغییر نداده باشد.

با حذف Generator که عملا دیگر نمیتوان عملیات انتقال دامنه را انجام داد و مدل خاصیت خود را از دست میدهد. با حذف Discriminator مولد دیگر راهنمایی برای تولید تصاویر واقع‌گرایانه ندارد و عملکرد خود را نمیتواند بهبود دهد تا تصاویری با کیفیت مشابه دامنه هدف تولید کند. حذف Classifier در فرآیند آموزش هم همانگونه که گفته شد، نظارت بر محتوای خروجی های مولد را از بین میبرد و باعث Mode Collapse میتواند بشود.

۱-۲-۴:

از آنجایی که در این مسئله هدف ما آموزش طبقه‌بند روی داده های لیبل‌دار دامنه منبع که توسط مولد تغییر یافته اند (تا شبیه دامنه مقصد شوند) میباشد، پس برای حفظ تناظر بین لیبل تصویر ورودی و تصویر خروجی مولد، لازم است که محتوای اصلی تصویر منبع حفظ شود. درغیر اینصورت دیگر آموزش بدون نظارت قابل انجام نیست.

این مقاله برای حفظ محتوای تصاویر تولیدی مولد Content-similarity loss و Task specific loss را تعریف نموده است. اگرچه Task specific loss تا حدی باعث حفظ محتوای کلی مولد می‌شود ولی Content-similarity loss هست که با مقایسه پیکسل به پیکسل Foreground ، مولد را مقید به حفظ کامل محتوا میکند و با حذف آن انتظار میرود که جزئیات محتوای تصاویر را تغییر دهد. همچنین طبق جدول ۵ مقاله، حذف این Loss باعث ناپایداری آموزش و افزایش انحراف معیاری دقت شبکه میشود و بازتولید نتایج سختتر می‌شود.

۱-۲-۵:

در صورتی که طبقه‌بند را تنها روی تصاویر تولیدی آموزش دهیم، مدل آزاد است که لیبل کلاس ها را با هم جابجا کند در حالی که عملکرد خوبی هم در اپتیمایز کردن هدف آموزش دارد. به این صورت که در فرآیند آموزش، مولد و طبقه‌بند در تعامل با یکدیگر برای بهینه کردن سریعتر تابع هزینه، ممکن است یاد بگیرند که محتوای کلاس را تغییر دهند ولی طبقه بند برای مثال تصاویر با محتوای کلاس ۱ را به عنوان کلاس ۲ تشخیص دهد. ولی با آموزش همزمان روی تصاویر اصلی و تولیدی، دیگر طبقه بند و مولد نمیتوانند با تغییر کلاس ها هزینه را کاهش دهند و مولد مجبور به حفظ کلاس ورودی و طبقه بند مجبور به تشخیص درست کلاس میشود. همچنین طبق جدول ۵، به شدت پایداری مدل در آموزش افزایش می‌یابد. زیرا با محدود کردن مولد و طبقه بند به پایدار ماندن به محتوا، بخش زیادی از فضای جستجو

پارامترها را کاهش میدهد و مدل دیگر نمیتواند نقاط ایزوله و ناپایداری را در فضای پارامترها پیدا کند که عملکرد شکننده ای دارند و باید دنبال راه حل های پایدار بگردد.

۶-۲-۱:

به نظر نمیرسد که این مدل در دامنه های با تفاوت معنایی عمیق بتواند موفق ظاهر شود. در مقاله اشاره شده است که مدل پیشنهادی که فرضشان این است که تفاوت میان دامنه ها Low-level هست (مانند نویز، رنگ، رزولوشن و ...) و تفاوت‌های High-level (مانند نوع اشیا یا ویژگی های هندسی متفاوت) ندارند.

برای مثال با تغییر معنایی گسترده دامنه مقصد (مانند اشیای متفاوت) دیگر نمیتوان از Classifier انتظار داشت که این دامنه جدید با کلاس های متفاوت را نیز به درستی طبقه بندی کند. همچنین به این نکته هم باید توجه داشت که هدف در اینجا آموزش بدون نظارت مدل برای دامنه هدف بود، پس تغییر محتوای عمیق تصاویر باعث بی معنی شدن لیبل آن در دامنه هدف شده و دیگر نمیتوان تصاویر تولید شده را به دامنه مقصد تعمیم داده و مدل را روی آن ها آموزش دهیم.

۳-۱: بخش عملی

پیش‌پردازش داده‌ها

در این تمرین از دو مجموعه داده MNIST و MNIST-M استفاده می‌کنیم. ابتدا دو مجموعه را دانلود کرده و ۵ نمونه متناظر از آن‌ها را با استفاده از اندیس‌های یکسان انتخاب کرده و نمایش می‌دهیم.

شکل ۱: تصاویر متناظر از مجموعه داده های **MNIST** و **MNIST-M**

همانطور که میبینیم محتوا و نحوه نوشتار اعداد در مجموعه MNIST-M کاملا مشابه MNIST هست و تنها تصویر پس‌زمینه ای به آن‌ها اضافه شده و تصاویر رنگی می‌باشند.

حال با تنظیمات ارائه شده در مقاله تصاویر را آماده سازی کرده و DataLoader ها را تشکیل میدهیم. مطابق مقاله، سایز همه تصاویر ۳۲ در ۳۲ باید باشد و مقادیر پیکسل ها را به بازه -۱ تا ۱ نرمال باید نمود. همچنین برای هماهنگی دو مجموعه داده، تصاویر تک کاناله MNIST را هم به تصاویر رنگی ۳ کاناله تبدیل می‌کنیم.

سایز بچ را برای همه DataLoader ها طبق مقاله ۳۲ در نظر می‌گیریم. همچنین به صورتی تصادفی ۲۰ درصد از هر دو مجموعه داده را انتخاب کرده و به عنوان داده های تست درنظر می‌گیریم و مابقی تصاویر داده های آموزش را تشکیل میدهند. (توجه شود که اندیس های تصاویر تست را به صورت رندوم یکبار انتخاب نموده و برای حفظ تناظر، هر دو مجموعه داده را با همین اندیس ها تقسیم میکنیم.

تعداد کل تصاویر موجود در هریک از این دیتاست ها ۷۰ هزار عکس میباشد که با تقسیم ذکر شده، ۵۶ هزار نمونه آموزشی و ۱۴ هزار نمونه تست خواهیم داشت.

ترنسفورم های منبع و هدف به صورت زیر تعریف می‌شوند:

```
source_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Grayscale(num_output_channels=3),
    transforms.Resize(IMG_SIZE),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])
target_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize(IMG_SIZE),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])
```

برای تطبیق داده ها با فریم‌ورک torch، یک کلاس سفارشی Dataset تعریف میکنیم و تصاویر و لیبل های بارگذاری شده از دو مجموعه داده را به دیتاست های جداگانه تبدیل میکنیم.

```
class CustomDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]
```

```
        if self.transform:
            image = self.transform(image)
        return image, torch.tensor(label, dtype=torch.long)
```

حال دیتاست ها را به بخش های آموزش و تست همانگونه که گفته شد تقسیم میکنیم. در نهایت ۴ دیتالودر خواهیم داشت که سایز آن ها به صورت زیر میباشد.

Number of batches in MNIST-Train: 1750

Number of batches in MNIST-Test: 438

Number of batches in MNIST-M-Train: 1750

Number of batches in MNIST-M-Test: 438

شکل تنسورهای دیتالودر های هر دو مجموعه داده را نیز بررسی میکنیم:

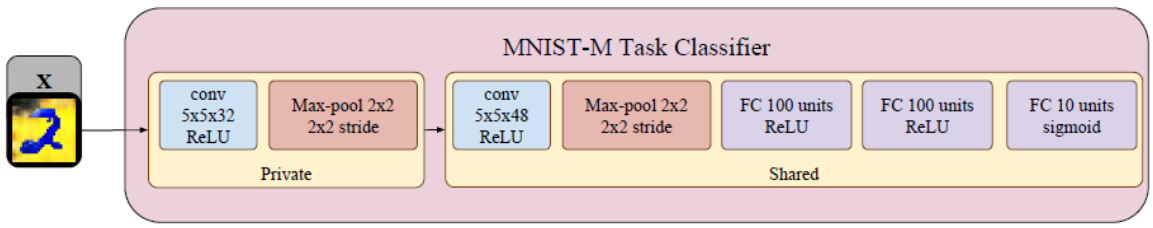
Tensor shape of source_train_loader: torch.Size([32, 3, 32, 32])

Tensor shape of traget_train_loader: torch.Size([32, 3, 32, 32])

همانطور که میبینیم مطابق انتظار همه تصاویر ۳ کاناله با ابعاد ۳۲ در ۳۲ هستند و سایز بچ نیز ۳۲ می‌باشد.

آموزش مدل پایه و مشاهده Domain Gap:

در این بخش Classifier ارائه شده توسط مقاله برای تسک طبقه‌بندی MNIST را پیاده‌سازی کرده و آن را روی داده های آموزشی MNIST (دامنه منبع) آموزش میدهیم. سپس عملکرد آن را روی داده های تست MNIST و کل داده های MNIST-M می‌سنجیم.



شکل ۲: معماری **Classifier** برای **MNIST**

جزئیات پیاده سازی لایه های مختلف به وضوح در شکل ۲ مشخص میباشد (برای مثال لایه اول یک کانولوشن با ۳۲ کانال خروجی و سایز کرنل ۵ در ۵ است که خروجی اش از تابع فعال‌سازی ReLU عبور می‌کند). ورودی این شبکه نیز تصاویر ۳ کاناله ۳۲ در ۳۲ هستند و خروجی آن احتمالات هر کلاس را نشان میدهد.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 32, 32]           2,432
              ReLU-2          [-1, 32, 32, 32]               0
         MaxPool2d-3          [-1, 32, 16, 16]               0
            Conv2d-4          [-1, 48, 16, 16]          38,448
              ReLU-5          [-1, 48, 16, 16]               0
         MaxPool2d-6            [-1, 48, 8, 8]               0
            Linear-7                 [-1, 100]         307,300
              ReLU-8                 [-1, 100]               0
            Linear-9                 [-1, 100]          10,100
             ReLU-10                 [-1, 100]               0
           Linear-11                  [-1, 10]           1,010
================================================================
Total params: 359,290
Trainable params: 359,290
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.78
Params size (MB): 1.37
Estimated Total Size (MB): 2.16
```

شکل ۳: جزئیات پیاده سازی **Classifier**

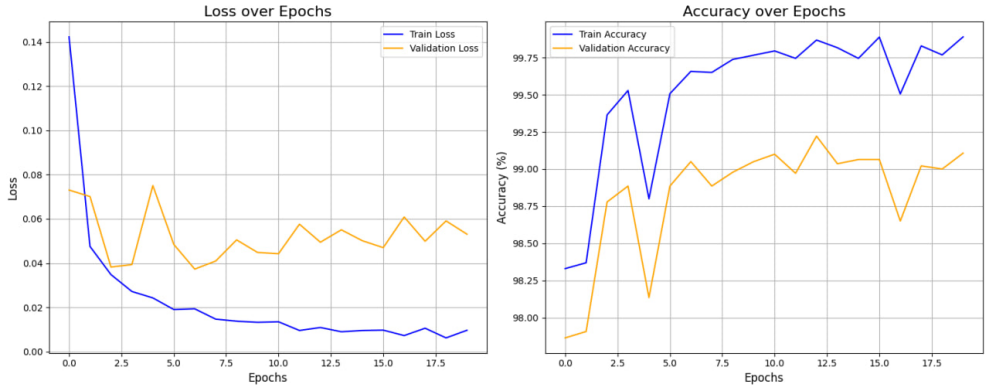
در شکل بالا نیز Summary مدل طبقه‌بند را پس از پیاده‌سازی مشاهده میکنیم که کاملا مطابق با شکل ۲ می‌باشد.

برای آموزش این مدل از تنظیمات کلی ارائه شده در بخش B مقاله استفاده میکنیم. وزن‌های اولیه شبکه را با توزیع گوسی با میانگین صفر و انحراف معیار ۰.۰۲ مقداردهی اولیه می‌کنیم. نرخ یادگیری را 1e-3 و Weight decay را 1e-5 در نظر می‌گیریم. همچنین از اپتیمایزر آدام با $\beta_1 = 0.5$ استفاده می‌کنیم. طبق گفته مقاله Learning rate decay با نسبت ۰.۹۵ در هر 20000 step را نیز اعمال می‌کنیم. تابع هزینه هم مطابق جنس مسئله که طبقه‌بندی است CrossEntropy در نظر می‌گیریم.

مدل را با تنظیمات ارائه شده برای ۲۰ ایپاک آموزش میدهیم (در هر مرحله بهترین مدل را ذخیره می‌کنیم). همچنین با ارزیابی مدل حین آموزش روی داده های تست MNIST نمودار خطا و دقت حین آموزش را نیز رسم می‌کنیم. لاگ آموزش مدل به صورت زیر بدست آمد:

```
Epoch [1/20] | Train Loss: 0.1423, Train Acc: 98.33% | Val Loss: 0.0730, Val Acc: 97.86%
Epoch [2/20] | Train Loss: 0.0476, Train Acc: 98.37% | Val Loss: 0.0701, Val Acc: 97.91%
Epoch [3/20] | Train Loss: 0.0348, Train Acc: 99.36% | Val Loss: 0.0382, Val Acc: 98.78%
Epoch [4/20] | Train Loss: 0.0272, Train Acc: 99.53% | Val Loss: 0.0394, Val Acc: 98.89%
Epoch [5/20] | Train Loss: 0.0243, Train Acc: 98.80% | Val Loss: 0.0751, Val Acc: 98.14%
Epoch [6/20] | Train Loss: 0.0190, Train Acc: 99.51% | Val Loss: 0.0485, Val Acc: 98.89%
Epoch [7/20] | Train Loss: 0.0194, Train Acc: 99.66% | Val Loss: 0.0373, Val Acc: 99.05%
Epoch [8/20] | Train Loss: 0.0147, Train Acc: 99.65% | Val Loss: 0.0409, Val Acc: 98.89%
Epoch [9/20] | Train Loss: 0.0138, Train Acc: 99.74% | Val Loss: 0.0505, Val Acc: 98.98%
Epoch [10/20] | Train Loss: 0.0133, Train Acc: 99.77% | Val Loss: 0.0448, Val Acc: 99.05%
Epoch [11/20] | Train Loss: 0.0135, Train Acc: 99.79% | Val Loss: 0.0443, Val Acc: 99.10%
Epoch [12/20] | Train Loss: 0.0096, Train Acc: 99.74% | Val Loss: 0.0576, Val Acc: 98.97%
Epoch [13/20] | Train Loss: 0.0109, Train Acc: 99.87% | Val Loss: 0.0495, Val Acc: 99.22%
Epoch [14/20] | Train Loss: 0.0090, Train Acc: 99.82% | Val Loss: 0.0551, Val Acc: 99.04%
Epoch [15/20] | Train Loss: 0.0096, Train Acc: 99.74% | Val Loss: 0.0502, Val Acc: 99.06%
Epoch [16/20] | Train Loss: 0.0097, Train Acc: 99.89% | Val Loss: 0.0470, Val Acc: 99.06%
Epoch [17/20] | Train Loss: 0.0073, Train Acc: 99.51% | Val Loss: 0.0609, Val Acc: 98.65%
Epoch [18/20] | Train Loss: 0.0106, Train Acc: 99.83% | Val Loss: 0.0499, Val Acc: 99.02%
Epoch [19/20] | Train Loss: 0.0062, Train Acc: 99.77% | Val Loss: 0.0591, Val Acc: 99.00%
```

```
Epoch [20/20]| Train Loss: 0.0097, Train Acc: 99.89% | Val Loss: 0.0531, Val Acc: 99.11%
```



شکل ۴: نمودار دقت و خطای مدل در طول آموزش

همانطور که میبینیم مدل در همان ایپاک های اول به دقت قابل قبولی روی داده های آموزشی و تست MNIST میرسد و با طی کردن ایپاک های بیشتر کمی افزایش دقت مشاهده میشود.

ارزیابی: با ارزیابی مدل روی داده های تست MNIST، دقت ۹۹.۲۲٪ به دست آمد. همچنین دقت مدل روی کل داده های MNIST-M (شامل آموزش و تست) به ۶۱.۱۹٪ میباشد.

این اختلاف دقت قابل توجه (حدود ۴۰ درصدی) کاملا مسئله ای که از ابتدا تعریف نمودیم یعنی مشکل Domain gap در مدل هایی که روی داده های دامنه منبع به خوبی آموزش دیده شده اند ولی در مواجهه با داده های جدید از دامنه دیگر (که تفاوت های جزئی مانند رنگ دارند و در نمونه های دنیای واقعی به وفور این تفاوت دامنه ها وجود دارند) عملکرد خوبی ندارند را نشان می‌دهد.

اگرچه تصاویر دامنه هدف شباهت محتوایی زیادی به دامنه منبع دارند و مدل هم تا حدی قادر به کلاس بندی نمونه های آن هست، ولی به دلیل اینکه مدل در فرآیند آموزشش تصاویری از این جنس را ندیده است و نحوه استخراج ویژگی بهینه و کلاس بندی آن ها را یاد نگرفته است دچار افت دقت محسوس می‌شود.

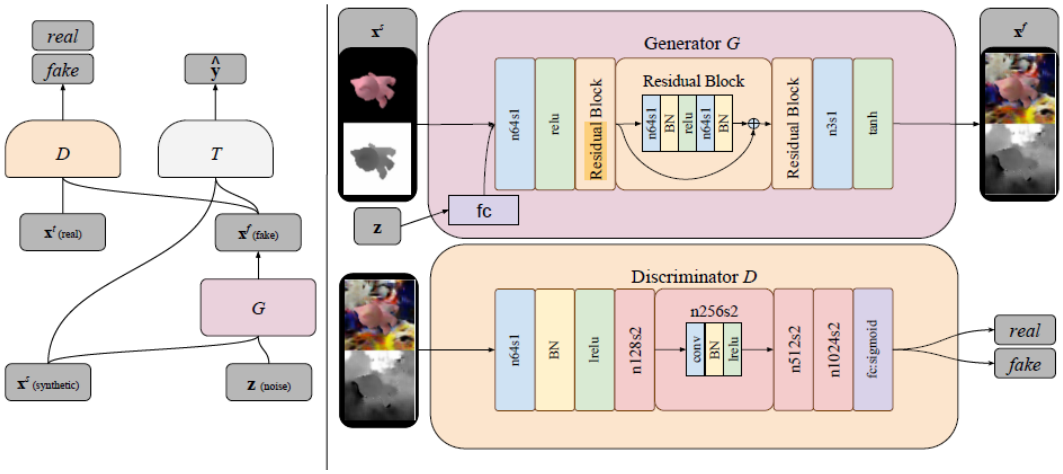
پیاده‌سازی معماری مدل:

در این بخش مدل کامل شامل Generator، Discriminator و Classifier را مطابق جزئیات ارائه شده در مقاله پیاده‌سازی می‌کنیم. در بخش قبل Classifier را پیاده‌سازی نمودیم.

Generator:

در شکل ۵ میتوانیم ساختار کلی مولد را ببینیم. همانطور که مشخص شده است، ابتدا بردار Z که از نویز با توزیع یکنواخت بین -۱ و ۱ نمونه برداری شده است را به یک لایه Fully-connected با سایز 32*32 میدهیم. خروجی این لایه را به صورت یک کانال از تصویر Reshape کرده و کنار تصویر ورودی

قرار میدهیم. تصویر جدید را ابتدا به یک لایه کانولوشن با سایز کرنل ۳ و تعداد کانال ۶۴ با تابع فعالسازی ReLU داده و در ادامه ۶ Residual Block قرار میدهیم (تعیین شده در بخش B.2 برای مسئله MNIST-M). در نهایت یک لایه کانولوشن با تعداد کانال ۳ و تابع فعال‌سازی Tanh قرار داده تا خروجی مولد به صورت یک تصویر ۳ کاناله در بیاید.



شکل ۵: معماری کلی مدل

بخش Residual Block از ۲ لایه کانولوشن و Batch Normalization مطابق شکل تشکیل شده است. همچنین یک Skip Connection در انتهای آن قرار دارد که ورودی را با خروجی شبکه جمع میکند و خروجی نهایی را میسازد. ساختار آن بعد از پیاده‌سازی به صورت زیر میشود:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Generator                                [32, 3, 32, 32]           --
├─Linear: 1-1                            [32, 1024]                11,264
├─Sequential: 1-2                        [32, 64, 32, 32]          --
│    └─Conv2d: 2-1                       [32, 64, 32, 32]          2,304
│    └─ReLU: 2-2                         [32, 64, 32, 32]          --
├─Sequential: 1-3                        [32, 64, 32, 32]          --
│    └─ResidualBlock: 2-3                [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-1                  [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-2             [32, 64, 32, 32]          128
│    │    └─ReLU: 3-3                    [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-4                  [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-5             [32, 64, 32, 32]          128
│    └─ResidualBlock: 2-4                [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-6                  [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-7             [32, 64, 32, 32]          128
│    │    └─ReLU: 3-8                    [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-9                  [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-10            [32, 64, 32, 32]          128
│    └─ResidualBlock: 2-5                [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-11                 [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-12            [32, 64, 32, 32]          128
│    │    └─ReLU: 3-13                   [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-14                 [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-15            [32, 64, 32, 32]          128
│    └─ResidualBlock: 2-6                [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-16                 [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-17            [32, 64, 32, 32]          128
│    │    └─ReLU: 3-18                   [32, 64, 32, 32]          --
```

```
│    │    └─Conv2d: 3-19                      [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-20                 [32, 64, 32, 32]          128
│    └─ResidualBlock: 2-7                     [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-21                      [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-22                 [32, 64, 32, 32]          128
│    │    └─ReLU: 3-23                        [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-24                      [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-25                 [32, 64, 32, 32]          128
│    └─ResidualBlock: 2-8                     [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-26                      [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-27                 [32, 64, 32, 32]          128
│    │    └─ReLU: 3-28                        [32, 64, 32, 32]          --
│    │    └─Conv2d: 3-29                      [32, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-30                 [32, 64, 32, 32]          128
├─Sequential: 1-4                            [32, 3, 32, 32]           --
│    └─Conv2d: 2-9                           [32, 3, 32, 32]           1,728
│    └─Tanh: 2-10                            [32, 3, 32, 32]           --
==========================================================================================
Total params: 459,200
Trainable params: 459,200
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 14.63
==========================================================================================
Input size (MB): 0.39
Forward/backward pass size (MB): 420.48
Params size (MB): 1.84
Estimated Total Size (MB): 422.71
==========================================================================================
```

:Discriminator

این شبکه طبق بخش B.2 باید شامل ۴ لایه کانولوشن با تعداد فیلترهای ۶۴، ۱۲۸، ۲۵۶ و ۵۱۲ (به ترتیب) باشد. ساختار آن مشابه شکل ۵ است، به این صورت که پس از هر لایه کانولوشن، یک لایه Batch normalization و تابع فعالسازی Leaky ReLU (با پارامتر Leakiness = 0.2) قرار دارند. همچنین بعد از هریک از این ۴ لایه طبق مقاله، یک نویز بدست آمده از توزیع گوسی با میانگین ۰ و انحراف معیار ۰.۲ به خروجی اضافه شده و Dropout با نرخ ۰.۱ اعمال میشود. (این دو عملیات برای آموزش پایدارتر Discriminator و تنظیم کردن آن تنها در هنگام آموزش اعمال میشوند)

ساختار این شبکه نیز پس از پیاده‌سازی به صورت زیر میباشد:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Discriminator                            [32, 1]                   --
├─Sequential: 1-1                        [32, 512, 4, 4]           --
│    └─Conv2d: 2-1                       [32, 64, 32, 32]          1,792
│    └─BatchNorm2d: 2-2                  [32, 64, 32, 32]          128
│    └─LeakyReLU: 2-3                    [32, 64, 32, 32]          --
│    └─Dropout: 2-4                      [32, 64, 32, 32]          --
│    └─GaussianNoise: 2-5                [32, 64, 32, 32]          --
│    └─Conv2d: 2-6                       [32, 128, 16, 16]         73,856
│    └─BatchNorm2d: 2-7                  [32, 128, 16, 16]         256
│    └─LeakyReLU: 2-8                    [32, 128, 16, 16]         --
│    └─Dropout: 2-9                      [32, 128, 16, 16]         --
│    └─GaussianNoise: 2-10               [32, 128, 16, 16]         --
│    └─Conv2d: 2-11                      [32, 256, 8, 8]           295,168
│    └─BatchNorm2d: 2-12                 [32, 256, 8, 8]           512
│    └─LeakyReLU: 2-13                   [32, 256, 8, 8]           --
│    └─Dropout: 2-14                     [32, 256, 8, 8]           --
│    └─GaussianNoise: 2-15               [32, 256, 8, 8]           --
│    └─Conv2d: 2-16                      [32, 512, 4, 4]           1,180,160
│    └─BatchNorm2d: 2-17                 [32, 512, 4, 4]           1,024
│    └─LeakyReLU: 2-18                   [32, 512, 4, 4]           --
│    └─Dropout: 2-19                     [32, 512, 4, 4]           --
│    └─GaussianNoise: 2-20               [32, 512, 4, 4]           --
```

```
├─Sequential: 1-2                        [32, 1]                   --
│    └─Linear: 2-21                      [32, 1]                   8,193
│    └─Sigmoid: 2-22                     [32, 1]                   --
==========================================================================================
Total params: 1,561,089
Trainable params: 1,561,089
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.87
==========================================================================================
Input size (MB): 0.39
Forward/backward pass size (MB): 62.91
Params size (MB): 6.24
Estimated Total Size (MB): 69.55
==========================================================================================
```

مسیر داده ها و نحوه تعامل این سه بخش شبکه را در فرآیند آموزش مشخص میکنیم. در شکل ۵ سمت چپ جزئیات این کار تا حدی مشخص است ولی در بخش آموزش مدل، با دقت بیشتری آن را بررسی می‌کنیم.

پیاده‌سازی توابع هزینه:

یکی از مهمترین گام ها در آموزش این مدل، تعریف توابع هزینه معرفی شده در مقاله، درک کارکرد و پیاده سازی آن ها می‌باشد. مطابق بخش ۴.۱ مقاله، برای این مسئله از رابطه خطای معرفی شده در فرمول ۱ مقاله که مطابق زیر است باید استفاده کنیم:

$$\min_{\boldsymbol{\theta}_G,\boldsymbol{\theta}_T} \max_{\boldsymbol{\theta}_D} \alpha \mathcal{L}_d(D,G) + \beta \mathcal{L}_t(G,T)$$

این رابطه یک بهینه سازی متقابل بیشینه کمینه را نشان میدهد. بخش اول آن Adversarial loss میباشد که تعامل رقابتی بین Generator و Discriminator را بهینه کرده و بخش دوم آن Classification loss هست که هم عملکرد طبقه بندی را ارزیابی و بهینه کرده و هم Generator را در تولید تصاویر با حفظ محتوای تصویر ورودی راهنمایی میکند.

جزئیات هر یک از این دو بخش به صورت زیر است (فرمول ۲ و ۳ مقاله):

$$\begin{aligned}
\mathcal{L}_d(D,G) &= \mathbb{E}_{\mathbf{x}^t}[\log D(\mathbf{x}^t;\boldsymbol{\theta}_D)] + \\
&\quad \mathbb{E}_{\mathbf{x}^s,\mathbf{z}}[\log(1 - D(G(\mathbf{x}^s,\mathbf{z};\boldsymbol{\theta}_G);\boldsymbol{\theta}_D))] \\
\mathcal{L}_t(G,T) &= \mathbb{E}_{\mathbf{x}^s,\mathbf{y}^s,\mathbf{z}}\big[-\mathbf{y}^{s\top} \log T\left(G(\mathbf{x}^s,\mathbf{z};\boldsymbol{\theta}_G);\boldsymbol{\theta}_T\right) \\
&\quad - \mathbf{y}^{s\top} \log T(\mathbf{x}^s);\boldsymbol{\theta}_T\big] \qquad (3)
\end{aligned}$$

برای محاسبه Loss های مربوط به هریک از بخش های Generator، Discriminator و Classifier باید نحوه آموزش مدل را ابتدا تعریف کنیم: مطابق مقاله ۲ گام بهینه سازی G و D را باید در هر مرحله طی کنیم: در گام D پارامترهای Discriminator و Classifier آپدیت شده در حالی که پارامتر های Generator فریز میشوند. در گام G نیز پارامتر های Generator آپدیت میشوند.

تابع هزینه Discriminator در گام D از همان رابطه ۲ مقاله بدست می‌آید. ضریب آن نیز طبق بخش B.2 برابر ۰.۱۳ میباشد. بخش اول این تابع هزینه مدل را تشویق میکند که تصاویر واقعی از دامنه هدف را با به عنوان تصویر واقعی برچسب بزند و بخش دوم آن مدل را تشویق میکند که تصاویر تولید شده توسط مولد را برچسب فیک بزند تا تابع هزینه کمینه شود. هر دو نیز از جنس یک مسئله Binary-CrossEntroy میباشند

تابع هزینه Classifier نیز مطابق رابطه ۳ مقاله میباشد و در گام D محاسبه میشود. طبقه بند هم روی تصاویر اصلی دامنه منبع و هم روی تصاویر تولید شده توسط مولد از روی آن ها تولید میشود (با تابع هزینه CrossEntropy) پیاده سازی میشوند. ضریبی برای این تابع هزینه مشخص نشده است پس ضریب آن را همان یک درنظر میگیریم.

اما تابع هزینه Generator که باید در گام G محاسبه شود از دو بخش تشکیل میشود. بخش اول آن همان بخش دوم در تابع هزینه Discriminator (که پیشبینی Discriminator از فیک یا اصل بودن تصاویر تولیدی مولد است) میباشد. ولی اینبار برخلاف آموزش Discriminator، هدف این است که Discriminator به اشتباه به تصاویر فیک مولد برچسب اصل بزند. ضریب این Loss در بخش B.2 مقدار ۰.۰۱۱ تعیین شده است. بخش دوم این تابع هزینه نیز بخش اول رابطه ۳ مقاله مربوط به تابع هزینه Classifier هست. این Loss تطابق لیبل پیشبینی شده برای تصویر تولیدی مولد را با لیبل تصویر اورجینال میسنجد و مولد را مجاب به حفظ محتوای تصویر میکند. این Loss هم طبق مقاله ضریب 0.01 را دارد.

در نهایت این ۳ رابطه را به صورت زیر پیاده سازی میکنیم:

```
adversarial_loss_fn = nn.BCELoss()
task_loss_fn = nn.CrossEntropyLoss()
def calculate_losses(G, D, T, source_images, source_labels, target_images):

    real_labels = torch.ones(BATCH_SIZE, 1, device=device)
    noise = torch.randn(BATCH_SIZE, 10, device=device)
    fake_images = G(source_images, noise)
    # Step D Losses (D and T)
    real_d_output = D(target_images)
    loss_d_real = adversarial_loss_fn(real_d_output, real_labels)
    fake_d_output = D(fake_images.detach())
    loss_d_fake = adversarial_loss_fn(1 - fake_d_output, real_labels)
    total_loss_D = (loss_d_real + loss_d_fake) * loss_weight_D

    loss_t_source = task_loss_fn(T(source_images), source_labels)
    loss_t_fake = task_loss_fn(T(fake_images.detach()), source_labels)
    total_loss_T = loss_t_source + loss_t_fake
    # Step G Losses (G) ---
```

```
    d_output_for_g = D(fake_images)
    loss_g_adv = adversarial_loss_fn(d_output_for_g, real_labels) * loss_weight_G_adv
    t_output_for_g = T(fake_images)
    loss_g_task = task_loss_fn(t_output_for_g, source_labels) * loss_weight_G_task
    total_loss_G = loss_g_adv + loss_g_task

    return total_loss_D, total_loss_T, total_loss_G
```

حال که توابع هزینه را به مشخصا برای هرمدل به تفکیک گام های آموزش مشخص نموده ایم میتوان مدل را آموزش داد.

آموزش مدل:

برای آموزش مدل همانطور که گفته شد در هر ایپاک ۲ فاز داریم: در گام D، Loss های مربوط به Discriminator و Classifier را محاسبه نموده، با هم جمع کرده و با روش گرادیان پارامتر های آن ها را آپدیت میکنیم. در گام G نیز Loss مربوط به Generator را محاسبه کرده و گرادیان زده و تنها پارامترهای خودش را آپدیت میکنیم. پس ۲ اپتیمایزر جداگانه هر کدام مختص به یکی از دو گام تعریف میکنیم.

هایپرپارامتر های آموزش را کاملا مطابق با مقاله (همانگونه که در بخش قبل داشتیم) انتخاب میکنیم.

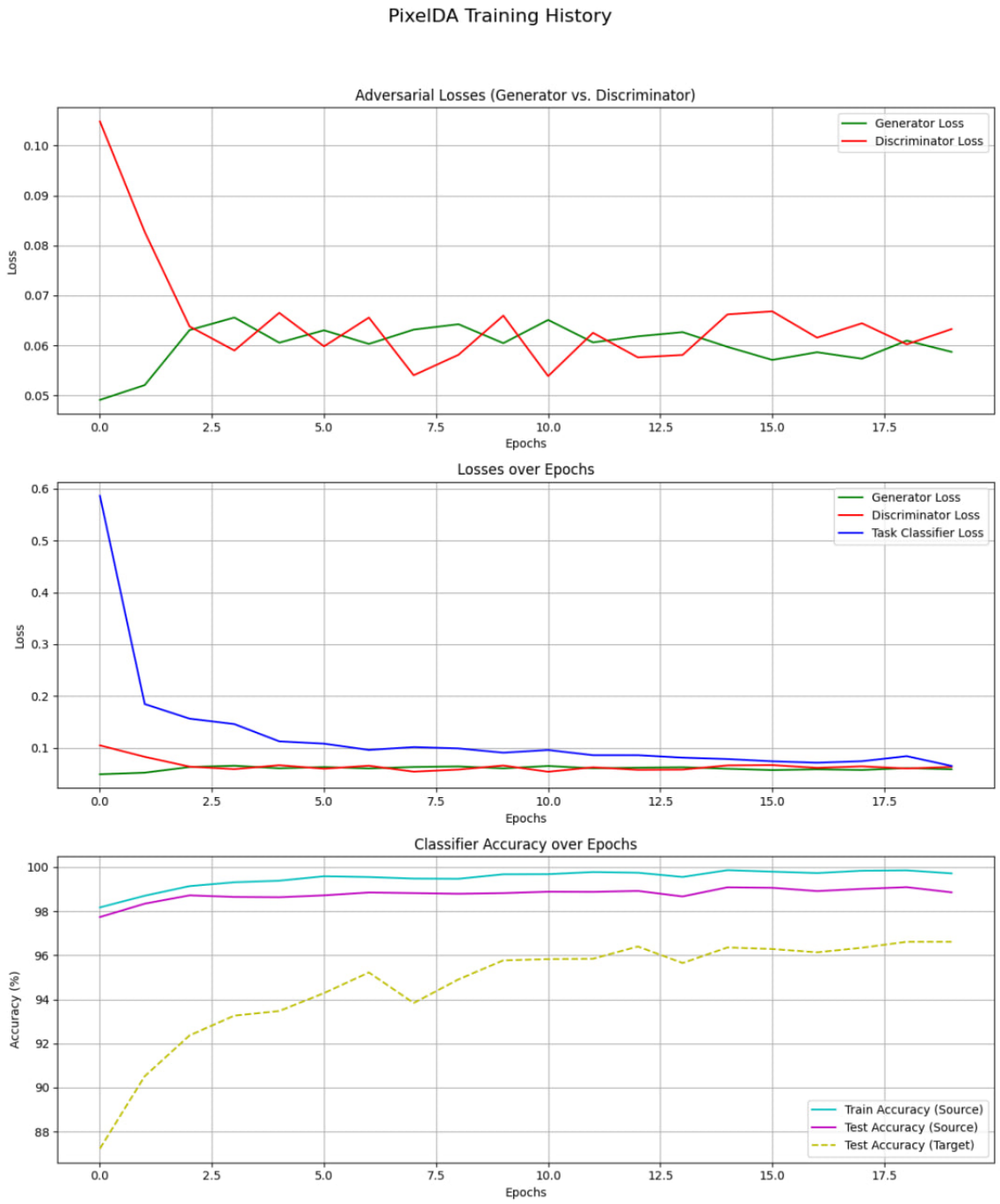
وزن‌های اولیه شبکه را با توزیع گوسی با میانگین صفر و انحراف معیار ۰.۰۲ مقداردهی اولیه می‌کنیم. نرخ یادگیری را 1e-3 و Weight decay را 1e-5 در نظر می‌گیریم. همچنین از اپتیمایزر آدام با $\beta_1 = 0.5$ استفاده می‌کنیم. طبق گفته مقاله Learning rate decay با نسبت ۰.۹۵ در هر 20000 step را نیز اعمال می‌کنیم. تابع هزینه هم مطابق جنس مسئله که طبقه‌بندی است CrossEntropy در نظر می‌گیریم.

آموزش را برای ۲۰ ایپاک انجام میدهیم. در بخش آموزش هر ایپاک ابتدا گام D طی میشود که در اول آن، تصاویر فیک توسط مولد از روی تصاویر اصلی تولید میشوند، سپس با اعمال تصاویر فیک تولید شده و تصاویر **بدون لیبل** آموزشی دیتاست مقصد (MNIST-M) به Discriminator، Loss آن را محاسبه میکنیم. سپس با اعمال تصاویر اصلی لیبل دار منبع و تصاویر تولید شده (با همان لیبل ها) به Classifier، Loss آن را نیز محاسبه میکنیم. در گام G نیز تصاویر فیک تولید شده، Loss مولد محاسبه و پارامترهایش محاسبه میشود.

در بخش ارزیابی ایپاک، دقت طبقه بند را روی تصاویر آموزشی منبع، تصاویر تست منبع و تصاویر تست مقصد محاسبه میکنیم. دقت شود که تنها جایی که از برچسب داده های MNIST-M استفاده میشود در همین فاز ارزیابی است و در هیچ بخش از فرآیند آموزش از داده های برچسب دار این مجموعه استفاده نمیشود تا اصل آموزش بدون نظارت نقض نشود.

نمایش نتایج:

در این بخش نتایج Loss و Accuracy را که در هنگام آموزش ذخیره کرده بودیم رسم میکنیم تا بتوانیم نتایج را تحلیل کنیم. نهایتا نیز دقت مدل را ارزیابی میکنیم.



شکل ۶: نمودار دقت و خطای مدل در طول آموزش

تحلیل نتایج: در شکل ۶ (نمودار وسط) خطای هر ۳ بخش مدل در کنار یکدیگر مقایسه شده است. همانطور که انتظار داشتیم بخش Classifier که در نهایت نیز بخشی از مدل است که از آن استفاده

میکنیم، با طی کردن ایپاک ها در حال یادگیری بوده و خطای خود را کاهش داده است. این بخش که هم روی داده های اصلی و هم داده های تولید شده (مانند دامنه هدف) آموزش میبیند در اصل دارد عملکرد خود را در طبقه بندی این تصاویر بهبود میدهد. ولی در این نمودار مشخص است که خطاهای Discriminator و Generator پس از طی کردن ایپاک های اولیه به یک نقطه تعادل میرسند و تغییر زیادی از آنجا نمیکنند. نمودار بالایی شکل ۶ این دو Loss و نحوه تعاملشان را بهتر نشان میدهد. همانطور که گفته بودیم، در شبکه های GAN هدف یک بهینه سازی MinMax است و این دو بخش در حال رقابت با یکدیگرند. پس هدف این است به یک نقطه تعادل برسیم که هم Discriminator قدرت خوبی در تشخیص موارد جعلی داشته باشد (خروجی آن بالانس باشد و روی ۰.۵ به تعادل برسد) و هم Generator بتواند Discriminator را به خوبی فریب دهد. در این نقطه به یک تعادل میرسیم و خطاها با آهنگ متضاد نوسان میکنند: به این معنا که با کمی کاهش خطایGenerator، خطای Discriminator افزایش می‌یابد (چون مولد قوی تر شده و راحتتر فریبش میدهد). سپس Discriminator از روی این خطا یاد میگیرد و عملکرد خود را بهبود میدهد و در عین حال باعث خطای بیشتر مولد، به دلیل تشخیص بهتر می‌شود.

البته نباید برداشت اشتباه کرد که به دلیل خطای تقریبا ثابت Generator، با طی کردن ایپاک ها عملکرد آن بهتر نمیشود، بلکه به خاطر قوی تر شدن همزمان Discriminator و Generator هست که این توازن برقرار شده و با بررسی دقت مدل به وضوح قوی تر شدن تصاویر فیک تولیدی را میتوانیم ببینیم.

نمودار پایینی شکل ۶ نیز دقت طبقه بند را روی داده های آموزش و تست MNIST و تست MNIST-M نشان میدهد. همانطور که میبنیم دقت آن روی داده های MNIST در همان ایپاک های اول به مقدار بسیار بالایی میرسد که طبیعی است، زیرا طبقه بند تصاویر آموزشی لیبل‌دار دامنه منبع را کامل میبینید و به سرعت آن ها را یاد میگیرد. ولی دقت طبقه بند روی داده های تست دامنه مقصد به تدریج بهتر میشود. این امر نشان دهنده آموزش بهتر Generator در طول آموزش است که دارد تصاویری مشابه‌تری به تصاویر دامنه مقصد تولید میکند، برای همین عملکرد طبقه بند روی داده های تست مقصد بهتر میشود (با اینکه مدل هیچ داده لیبل داری از دامنه مقصد را نمیبیند).

دقت مدل نهایی روی داده های تست منبع (MNIST) به ۹۹.۰۲٪، روی داده های تست مقصد (MNIST-M) به ۹۶.۶۱٪ و روی کل داده های MNIST-M به ۹۷.۳۵٪ می‌رسد. در مقایسه با دقت مدل پایه بخش های اول بهبود قابل توجهی داشتیم و همانطور که میبینیم عملکرد طبقه بند روی هر دو دامنه به مقدار قابل قبولی رسیده و توانستیم با یادگیری بدون نظارت، عملکرد طبقه بند را به دامنه جدیدی تعمیم بدهیم.

در ادامه چند نمونه تصویر از تصاویر تست MNIST انتخاب میکنیم و با اعمال آن ها (همراه با بردار نویز) به Generator، تصاویر تولیدی آن ها را ساخته و آن ها را در کنار تصویر متناظر از MNIST-M نشان میدهیم (هر یک از تصاویر را به طبقه بندی نیز اعمال کرده و لیبل اصلی اش را در کنار لیبل پیشبینی شده نمایش میدهیم).

شکل ۷: مقایسه تصاویر دامنه منبع و هدف و تصاویر تولید شده متناظرشان

همانطور که میبینیم تصاویر تولید شده (ردیف دوم) کاملا محتوای تصویر اول را حفظ کرده اند. از طرفی با مقایسه تصاویر ردیف ۲ و ۳ میبینیم که علارغم موفقیت مولد در تولید تصاویر مشابه دامنه هدف، به وضوح جزئیات پس‌زمینه آن ها فرق دارد. پس مدل عمیقا یادگرفته از روی بردار نویز این تصاویر را تولید کند و اینگونه نیست که تصاویر مقصد را حفظ کرده باشد. همچنین همگی این نمونه ها و تصاویر تولید شده به درستی توسط طبقه بند کلاس‌بندی شده‌اند.

جدول ۱: مقایسه دقت **Classifier** قبل و بعد از اعمال **Domain-shift**

Method	MNIST Accuracy (testset)	MNIST-M Accuracy (Whole dataset)
Before Domain-shift	99.22	61.19
After Domain-shift	99.02	97.35

در جدول ۱ نیز بهبود عملکرد و بسته شدن Domain gap را مشاهده میکنیم. دقت بدست آمده توسط مقاله در جدول ۱ مقاله ۹۸.۲٪ بود که تقریبا توانستیم به آن برسیم. (دقت پیاده‌سازی ما ۹۷.۳۵٪ بدست آمد)

پرسش ۲ – بازسازی تصاویر پولیپ آندوسکوپی باEndoVAE

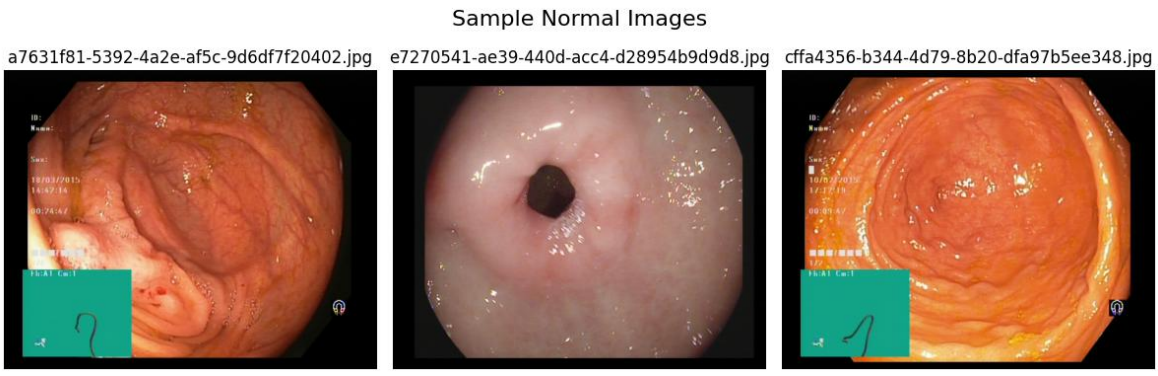
۲-۱: پیش پردازش داده ها

در این سوال از تمرین می خواهیم از یک Variational Autoencoder استفاده کنیم و یک سری تصاویر پزشکی مربوط به دستگاه گوارش انسان را بازسازی کنیم. در ابتدا از Kaggle دیتاست را دانلود میکنیم، تصاویر سالم را از پوشه های normal-z-line و normal-pylorus و normal-cecum جدا میکنیم و تصاویر ناهنجار را از پوشه polyps جدا میکنیم. سپس یک خلاصه آماری از تعداد تصاویر و اندازه آنها بدست می آوریم و تصاویر از هر دسته نرمال و ناهنجار سه نمونه از تصاویر را نشان میدهیم:

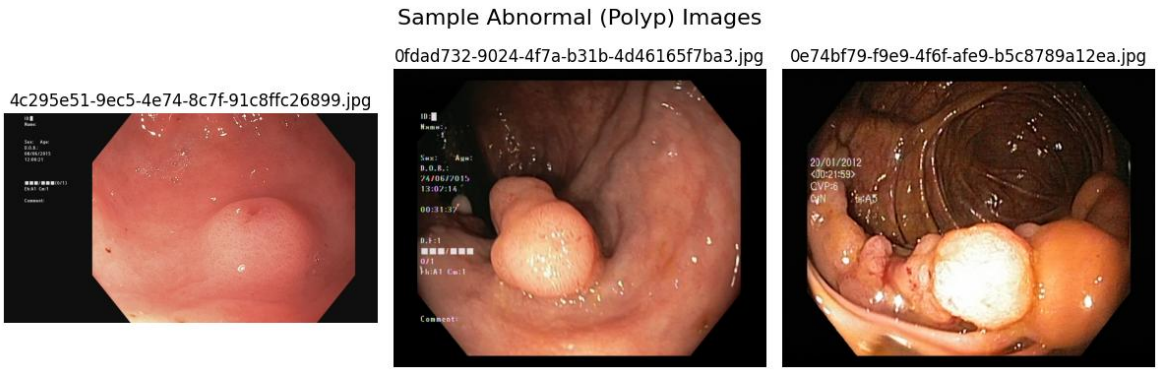
```
--- Statistical Summary ---
Total Normal Images: 1500
Total Abnormal Images: 500
--------------------------
Image dimensions (width, height) in 'normal' folder:
  - (1920, 1072): 226 images
  - (1280, 1024): 682 images
  - (720, 576): 592 images

Image dimensions (width, height) in 'abnormal' folder:
  - (720, 576): 477 images
  - (1920, 1072): 16 images
  - (1280, 1024): 7 images
```

با توجه به این آمار ها تعداد تصاویر ناهنجار نسبت به تصاویر نرمال کمتر است و در بقیه پوشه های دیتاست بازهم تصاویر ناهنجار و جود دارد و می توانیم از آنها استفاده کنیم ولی با توجه به اینکه در دیتاست موردا استفاده مقاله هم همین نسبت وجود داشت از آن داده ها استفاده نمیکنیم. چن نمونه از داده های نرمال و ناهنجار در زیر آورده شده.

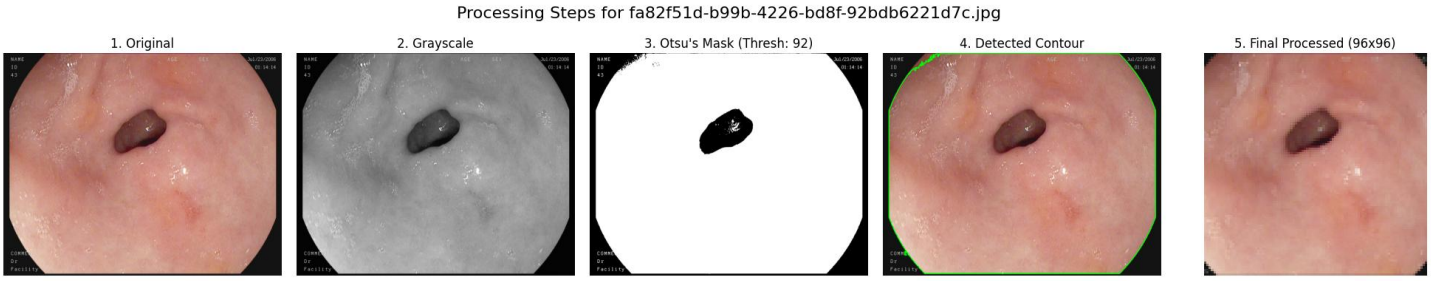


شکل ۸ سه نمونه از داده های نرمال

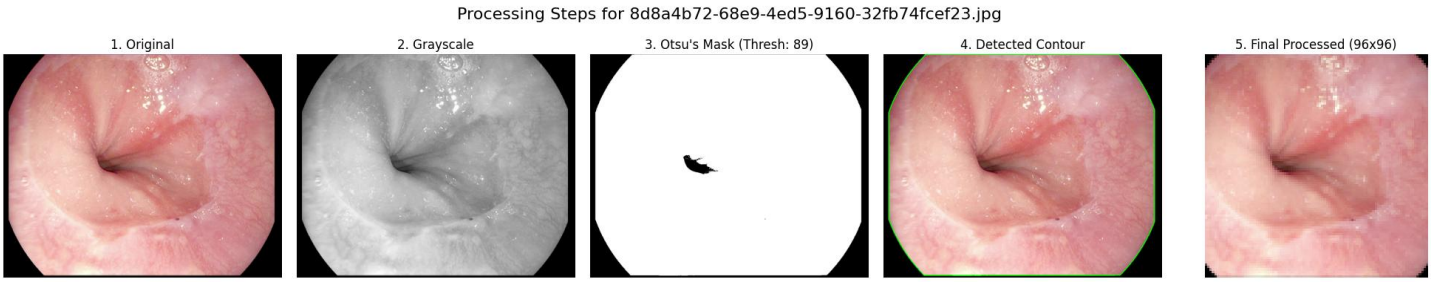


شکل ۹ سه نمونه از داده های ناهنجار

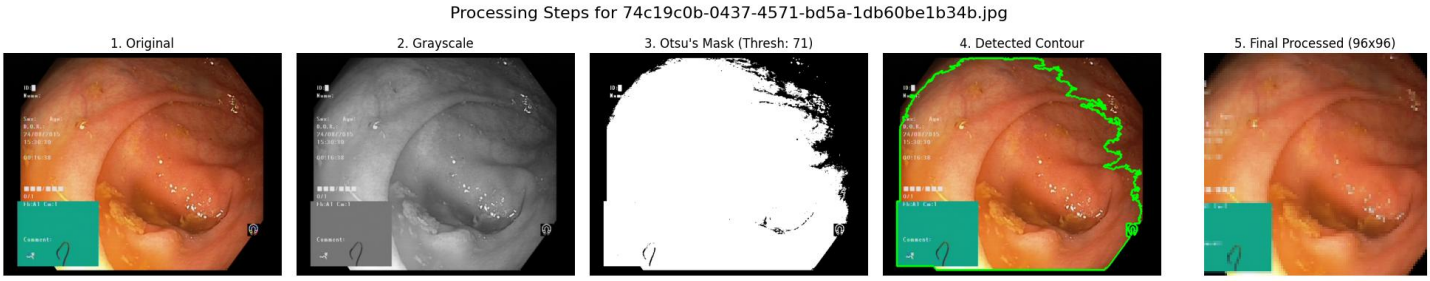
برای پیش پردازش تصاویر در مرحله اول باید حاشیه تصاویر را crop کنیم و سپس تصاویر را به شکل مربع برش دهیم، به اندازه ۹۶ در ۹۶ resize کنیم و نهایتا مقدار پیکسل ها را بین ۰و۱ نرمالایز کنیم. چالش برانگیز ترین بخش این فرایند بخش crop کردن تصویر در ابتدای کار است زیرا تصاویر در اندازه های مختلف هستند و حاشیه آنها نیز متفاوت است. برای اینکه از Contours در کتابخانه OpenCV استفاده میکنیم. در این الگوریتم ابتدا تصویر به یک تصویر سیاه و سفید تبدیل می شود و سپس با توجه به شکل تصاویر انگار یک لکه سفید با حاشیه مشکی باقی میماند. حالا ما مرز این لکه سفید و حاشیه را پیدا میکنیم و کوچکترین مستطیلی که این لکه سفید در آن جا شود را برش میدهیم. پس از این کار مراحل برش یک مربع از وسط تصویر، resize کردن تصویر به اندازه ۹۶ در ۹۶ و نهایتا نرمالایز کردن تصویر را اجرا میکنیم. مراحل این فرایند در زیر آورده شده.



شکل ۱۱ مراحل پیش پردازش داده نمونه۱



شکل ۱۰ مراحل پیش پردازش داده نمونه۲

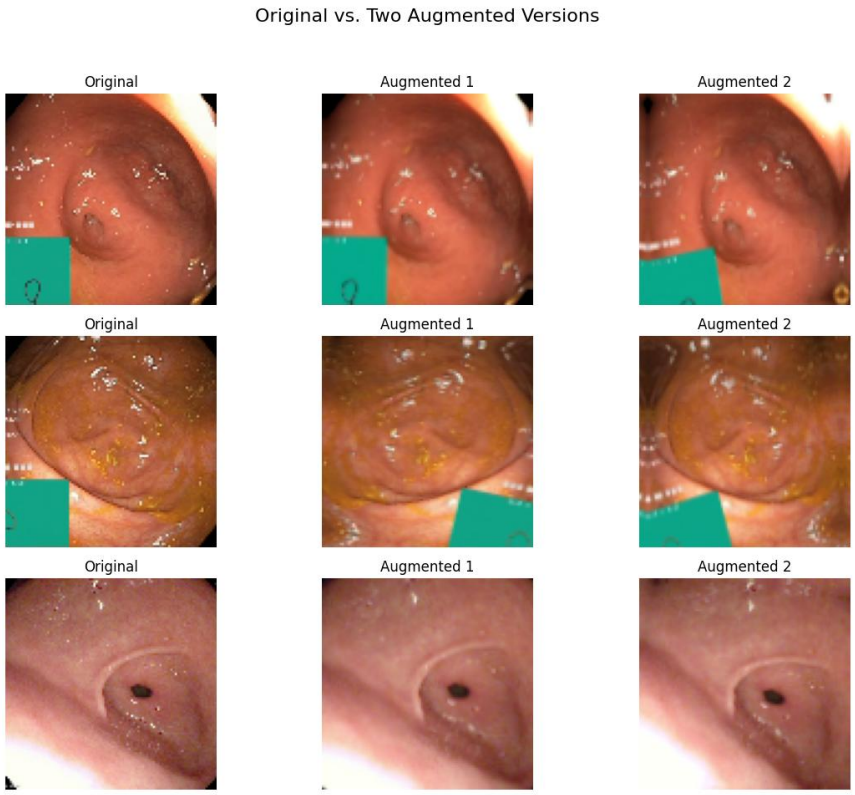


شکل ۱۲ مراحل پیش پرداز داده نمونه۳

همه تصاویر سالم و ناهنجار به همین روش پردازش شده و در پوشه های متناظر قرار گرفتند و روی گوگل درایو کپی شدند. پس از این مرحله به مرحله data Augmentation می رویم. در این مرحله تغییرات را به صورت زیر به تصاویر سالم اعمال میکنیم و تعداد آنها را از ۱۵۰۰ تصویر به ۴۵۰۰ تصویر افزایش میدهیم:

```
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.05),
        layers.RandomZoom(height_factor=0.1, width_factor=0.1),
        layers.RandomContrast(factor=0.1),
```

همانطور که مشاهده میشود در این کد از قرینه کردن افقی تصویر، چرخش تصادفی به مقدار کم و تغییر زوم و Contrast استفاده شده. سه نمونه تصویر پس از اعمال آگمنتیشن در زیر آورده شده:



شکل ۱۳ سه نمونه تصویر به همراه تصاویر آگمنت شده از آنها

۲-۲: طراحی معماری EndoVAE

در این بخش می خواهیم مدل مورد نظر و بررسی شده در مقاله را پیاده سازی کنیم. قدم به قدم این کار را انجام میدهیم.

a. Encoder :

این بخش از مدل وظیفه دارد تصویر ورودی ما را به یک بردار فلت تبدیل کند. بعد تر در آموزش مدل با تعریف توابع هزینه ما این بردار را مجاب میکنیم تا نشان دهنده میانگین و واریانس یک توزیع گوشی نرمال باشد. لایه های encoder به این شکل هستند:

- لایه ۱ Conv : با ۱۶فیلتر و Stride=1
- لایه ۲ Conv : با ۳۲فیلتر و Stride=1
- لایه ۳ Conv : با ۳۲فیلتر و Stride=2 (ابعاد نصف می‌شود ← 48x48)
- لایه ۴ Conv :با ۶۴ فیلتر و Stride=2 (ابعاد نصف می‌شود ← 24x24)
- لایه ۵ Conv :با ۱۲۸ فیلتر وStride=1
- لایه ۶ Conv :با ۲۵۶ فیلتر و Stride=2 (ابعاد نصف می‌شود ← 12x12)

تمامی این فیلتر ها ۳در۳ و تابع فعال ساز آنها Relu است.

نهایتا خروجی لایه آخر فلت میشود و وارد یک لایه دنس با ۲۵۶ نرون میشود.

b. Latent Space :

در این بخش از ۲۵۶ خروجی encoder دو بردار مجزای ۶ بعدی توسط دو شبکه دنس که در هرکدام ۶ نرون وجود دارد، ساخته می شود که این دو بردار همان میانگین و واریانس ما هستند. البته در مقاله جمله زیر آمده است:

These fully connected layers consist of six neurons with for estimating the distribution parameters μ, σ of the latent space.

می توان این برداشت را کرد که برای هرکدام از پارامتر های میانگین و واریانس سه نرون و در مجموع ۶ نرون اختصاص داده شده است ولی با توجه به اینکه عدد ۶ منطقی تر است، ما نیز در نظر میگیریم که میانگین و واریانس هر یک ۶ بعدی هستند. البته دقت شود که بجای واریانس از لگارتیم آن استفاده میکنیم. یک علت این کار این است که طبق تعریف ریاضی واریانس باید حتما عدد مثبت باشد ولی خروجی یک شبکه عصبی ممی تواند هر عددی باشد. علت دیگر این کار نیز

پایداری در محاسبات است زیرا واریانس می تواند عددی بسیار کوچک باشد و در محاسبات کامپیوتری مشکل ساز شود.

نهایتا از روی لگاریتم واریانس انحراف معیار محاشبه میشود و با استفاده از فرمول زیر یک نمونه(z) را می سازیم و به decoder می دهیم.

$$z=\mu+\epsilon\cdot\sigma$$

در این فرمول ϵ یک بردار هم بعد با میانگین و انحراف معیار است و از یک توزیع نرمال استاندارد می آید.

c. Decoder :

ورودی این بخش همان بردار 6 بعدی z است. در ابتدای کار یک لایه دنس این ورودی ۶ بعدی را به ۳۶۸۶۴ مپ میکند. سپس خروجی این لایه reshape می شود و به یک تنسور ۱۲ در ۱۲ در ۲۵۶ می شود. البته در مقاله به صراحت گفته نشده که این ۳۶۸۶۴ نرون به این ابعاد reshape می شوند. مثلا می توانستند به ابعاد ۶ در ۶ در ۱۰۲۴ هم reshape شوند. ولی برای حفظ قرینه بودن encoder و decoder ما این ابعاد را در نظر میگیریم.

از اینجا به بعد، یک سری از لایه‌های Transposed Convolution (برای بزرگ کردن ابعاد تصویر) و Convolution (برای پالایش ویژگی‌ها) به کار می‌روند. تمام لایه‌ها از فیلترهای 3×3استفاده می‌کنند:

- لایه۱ : Transposed Conv با ۲۵۶ فیلتر و Stride=2 ابعاد را به 24x24 افزایش می‌دهد.
- لایه۲ : Conv با ۱۲۸ فیلتر و Stride=1
- لایه ۳: Transposed Conv با ۶۴ فیلتر و Stride=2 ابعاد را به 48x48 افزایش می‌دهد.
- لایه ۴: Conv با ۶۴ فیلتر و Stride=1
- لایه ۵: Transposed Conv با ۳۲ فیلتر و Stride=2 ابعاد را به 96x96 افزایش می‌دهد.
- لایه ۶: Conv با ۳۲ فیلتر وStride=1

نهایتا یک لایه کانولوشنی نهایی با سه فیلتر تصویر نهایی ۹۶ در ۹۶ را بازسازی میکند. تابع فعال سازی لایه آخر Sigmoid است که تضمین می کند خروجی ما بین ۰ و ۱ بماند.

نهایتا مدل را با این مشخصات پیاده سازی میکنیم و summary دیکودر و انکودر را میگیریم:

Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to
input_layer_20 (InputLayer)	(None, 96, 96, 3)	0	-
conv2d_80 (Conv2D)	(None, 96, 96, 16)	448	input_layer_20[0…
conv2d_81 (Conv2D)	(None, 96, 96, 32)	4,640	conv2d_80[0][0]
conv2d_82 (Conv2D)	(None, 48, 48, 32)	9,248	conv2d_81[0][0]
conv2d_83 (Conv2D)	(None, 24, 24, 64)	18,496	conv2d_82[0][0]
conv2d_84 (Conv2D)	(None, 24, 24, 128)	73,856	conv2d_83[0][0]
conv2d_85 (Conv2D)	(None, 12, 12, 256)	295,168	conv2d_84[0][0]
flatten_8 (Flatten)	(None, 36864)	0	conv2d_85[0][0]
dense_16 (Dense)	(None, 256)	9,437,440	flatten_8[0][0]
latent_mu (Dense)	(None, 6)	1,542	dense_16[0][0]
latent_log_var (Dense)	(None, 6)	1,542	dense_16[0][0]

Total params: 9,842,380 (37.55 MB)
Trainable params: 9,842,380 (37.55 MB)
Non-trainable params: 0 (0.00 B)

Model: "vae_4"

Layer (type)	Output Shape	Param #
encoder (Functional)	((None, 6), (None, 6))	9,842,380
decoder (Functional)	(None, 96, 96, 3)	1,282,467
sampling_6 (Sampling)	?	0 (unbuilt)

Total params: 11,124,847 (42.44 MB)
Trainable params: 11,124,847 (42.44 MB)
Non-trainable params: 0 (0.00 B)

Model: "decoder"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 6)	0
dense_1 (Dense)	(None, 36864)	258,048
reshape (Reshape)	(None, 12, 12, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 24, 24, 256)	590,080
conv2d_6 (Conv2D)	(None, 24, 24, 128)	295,040
conv2d_transpose_1 (Conv2DTranspose)	(None, 48, 48, 64)	73,792
conv2d_7 (Conv2D)	(None, 48, 48, 64)	36,928
conv2d_transpose_2 (Conv2DTranspose)	(None, 96, 96, 32)	18,464
conv2d_8 (Conv2D)	(None, 96, 96, 32)	9,248
conv2d_9 (Conv2D)	(None, 96, 96, 3)	867

Total params: 1,282,467 (4.89 MB)
Trainable params: 1,282,467 (4.89 MB)
Non-trainable params: 0 (0.00 B)

۲-۳: تعریف توابع هزینه

برای تعریف تابع خطای این مدل، ما دو ترم اصلی داریم و می خواهیم سعی کنیم دو ترم خطا را مینیمم کنیم. تک تک این ترم ها را شرح میدهیم:

a. خطای بازسازی (Reconstruction Loss) :

با این خطا از قبل نیز آشنایی داریم. این خطا را برای بازسازی دقیق تصاویر ورودی به کار میبریم. انگار یک خروجی 96x96x3 داریم که می خواهیم تک تک این پیکسل ها به مقادیر پیکسل های تصویر مرجع برسند. برای این کار از این خطا استفاده میکنیم تا مقدار خطای بین پیکسل های تولیدی و پیکسل های تصویر مرجع را کاهش دهیم. در واقع این خطا تعریف شده که مدل را به سمت دقت سوق دهد. برای تصاویری که مقدار پیکسل های آنها بین ۰و۱ نرمالایز شده معمولا از BCE استفاده می شود و ما هم از همین تابع خطا استفاده میکنیم. فرمول این تابع خطا به این شکل است:

$$-[y\log(\hat{y})+(1-y)\log(1-\hat{y})]$$

در این تابع y مقدار پیکسل در تصویر مرجع و $\hat{y}$ مقدار پیکسلی است که مدل تولید کرده. ما این مقدار را برای تک تک پیکسل ها محاسبه و با هم جمع میکنیم.

b. Kullback-Leibler Divergence :

این بخش از تابع خطا بخش جدید و جذاب در VAE ها است. هدف آن پاسخ به این سوال است که آیا انکودر، توزیع آماری تصویر ورودی را در یک جای منظم و قابل پیش‌بینی در فضای پنهان قرار داده است که بعدا بشود برا تولید تصاویر از آنها استفاده کرد؟ این خطا، توزیع هر تصویر ورودی $N(\mu, \sigma^2)$ را با توزیع مرجع نرمال استاندارد $N(0, 1)$ مقایسه کرده و اختلاف آن‌ها را به عنوان یک جریمه محاسبه می‌کند. برای مقایسه یک توزیع گوشی با توزیع نرمال این تابع خطا به این شکل در می آید:

$$-0.5\times\sum(1+\log(\sigma^2)-\mu^2-\sigma^2)$$

می توان این مقدار را برای تک تک ابعاد محاسبه کرد و با یکدیگر جمع کرد.

در نهایت این دو خطا با هم جمع می شوند و بهینه ساز می خواهد که هر دوی این خطا ها را بهینه کند. می توان گفت که قدرت تعمیم و تولید VAE ها از تقابل بین این دوخطا بدست می آید. Reconstruction Loss می خواهد هر نمونه را با دقت بالا بازسازی کند و برای این کار نیاز دارد که هر یکی از نمونه ها را در جایی از فضای ویژگی بگذارد که از بقیه نمونه ها دور باشد و بتوان آن را تشخیص داد. از طرفی Kullback-Leibler Divergence می خواهد به هر یک از داده ها یک توزیع گوسی نسبت دهد و همه را در مرکز جمع کند. در واقع این دو خطا با یکدیگر در تقابل هستند و نقطه بهینه جایی است که این دو خطا به تعادل برسند.

هر دوی این خطا ها در کد خود مدل و در تابع train_step پیاده سازی شده است.

۲-۴: روند آموزش مدل

هایپر پارامتر ها را مطابق مقاله تنظیم میکنیم ولی به علت محدودیت سخت افزار و زمان تعداد ایپاک ها را به ۱۰۰۰ ایپاک کاهش میدهیم. همچنین لرنینگ ریت را مطابق مقاله ۰.۰۰۱ قرار میدهیم ولی بعد از هر ۱۰ ایپاک که مقدار loss روی داده های ولیدیشن کم نشد آن را نصف میکنیم. مقدار learning rate نمی تواند از ۰.۰۰۰۰۰۱ کمتر شود. مکانیزم early stop را نیز برای مدل تعبیه میکنیم که اگر برای پنجاه ایپاک متوالی وضعیت مدل بهتر نشد، یادگیری مدل متوقف شود. چک پوینت را نیز برای ذخیره بهترین مدل تعبیه میکنیم. نهایتا بهترین مدل با مشخصات زیر در ایپاک ۲۲۰ بدست می آید:

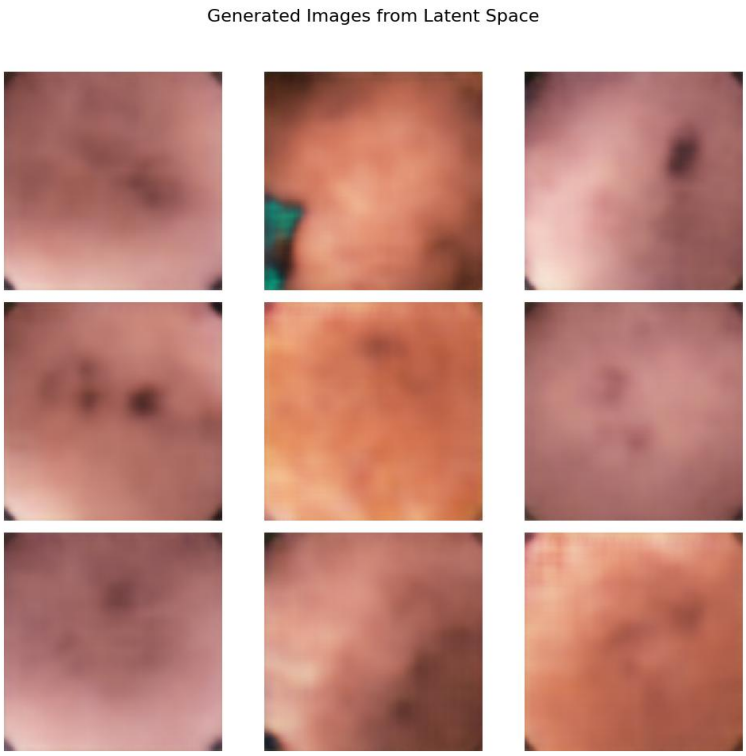
```
Epoch 220/1000
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 219ms/step - kl_loss: 19.5514 -
reconstruction_loss: 5228.8262 - total_loss: 5248.3779
Epoch 220: val_total_loss improved from 4252.67236 to 4046.91748, saving model to
/content/drive/MyDrive/EndoVAE_Project/best_model_final.keras
32/32 ━━━━━━━━━━━━━━━━━━━━ 11s 271ms/step - kl_loss: 19.5539 -
reconstruction_loss: 5228.8975 - total_loss: 5248.4512 - val_kl_loss: 20.0282 -
val_reconstruction_loss: 4026.8889 - val_total_loss: 4046.9175 - learning_rate: 1.0000e-06
```

مقادیر خطای بازسازی و kl در دو محدوده متفاوت هستند. بهتر بود با یک ضریب این دو loss را در یک محدود بیاوریم تا تعادل بین آنها به درستی شکل گیرد ولی به دلیل پایبندی به مقاله از همان فرمول های موجود در مقاله استفاده میکنیم. وقتی خطای بازسازی در حدود ۵۰۰۰ و خطای kl در حدود ۱۵ است، واضح است که بهینه ساز همواره تلاش میکند که خطای بازسازی را کم کند زیرا خطای kl تاثیر چندانی روی خطای کل ندارد. همچنین با توجه به تقابلی که بین این دو خطا وجود دارد و در بخش قبل مطرح کردیم بهینه ساز می توند با کمی افزایش خطای kl خطای بازسازی را به مقدار زیادی کاهش دهد و این مورد باعث کاهش توانایی مدل در generate کردن تصاویر میشود اگرچه می تواند در بازسازی به خوبی عمل کند.

نکته ای که وجود داشت این بود که مقاله از ترکیب داده های سالم و ناهنجار برای آموزش مدل استفاده کرده بود ولی در صورت تمرین گفته شده بود که از تصاویر سالم برای آموزش مدل استفاده کنیم. این نکته کمی مدل را در بازسازی تصاویر ناهنجار ضعیف کرده که در بخش تولید و بازسازی کیفی درباره آن صحبت میکنیم.

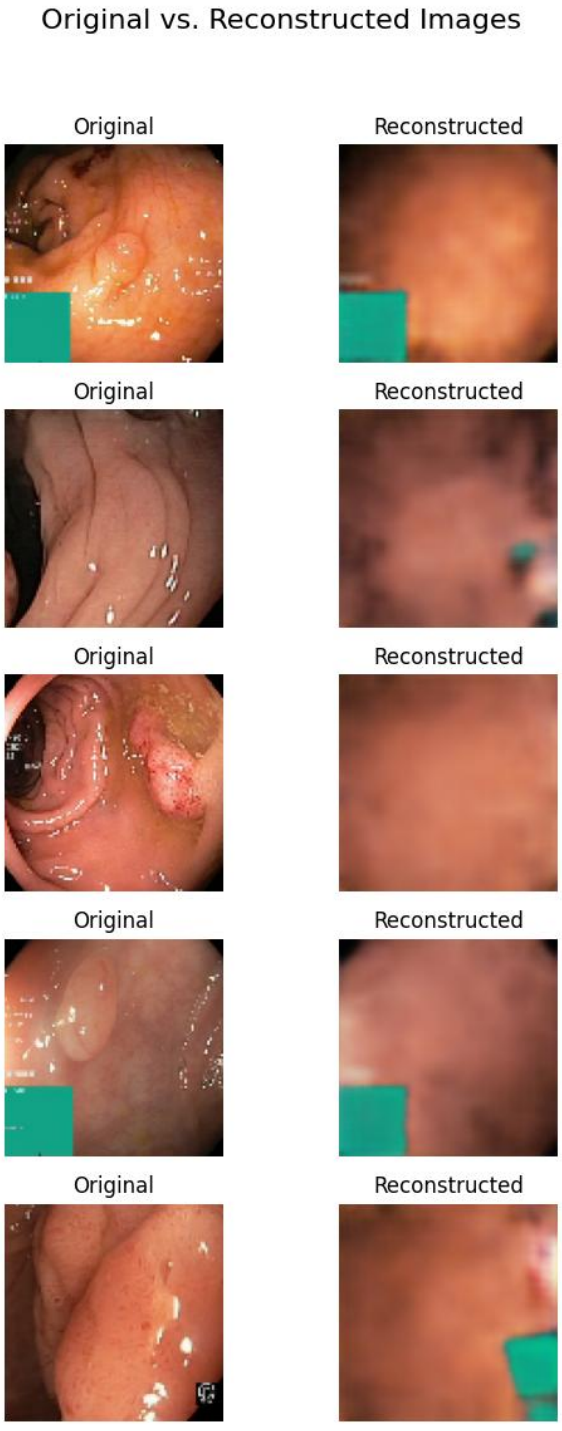
۲-۵: تولید و بازسازی کیفی

با توجه به دستور کار ۹ بردار با مقادیر تصادفی به دیکودر میدهیم و خروجی های آن را نمایش میدهیم.

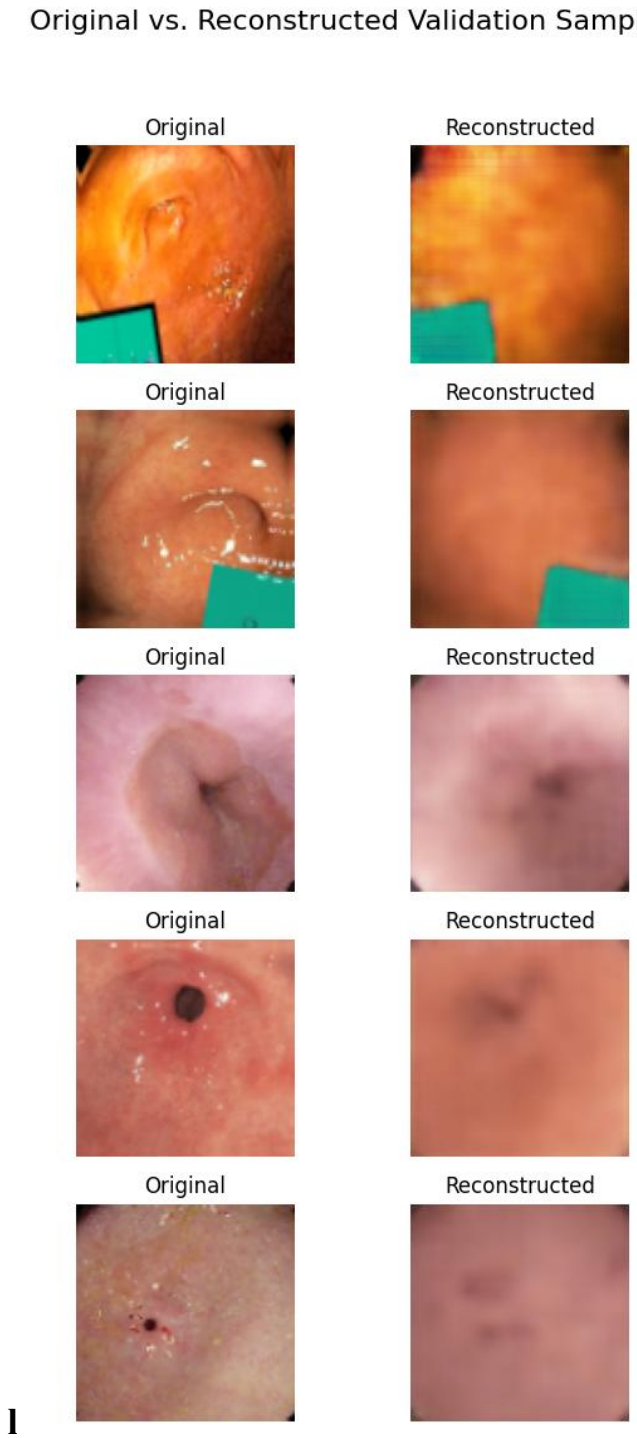


شکل ۱۴ خروجی دیکودر در مود تولید

همچنین ۵ مورد از تصاویر پولیپ را به مدل می دهیم و آنها را بازسازی میکنیم. بجز این تصاویر، پنج مورد از تصاویر داده های ولیدیشن که مدل آنها را ندیده را هم به مدل میدهیم تا مدل را ارزیابی کنیم.



شکل ۱۵ پنج نمونه از داده های پولیپ همراه با تصاویر بازسازی شده آنها توسط مدل



l

شکل ۱۶ پنج مورد از داده های سالم دیتاست **validation** همراه با بازسازی آنها

با توجه به این تصاویر میتوان گفت در حالت کلی مدل عملکرد بدی نداشته ولی همچنان تاری و وضوح نداشتن تصاویر چالش برانگیز است. نکته قابل توجه دیگر این است که تصاویر بازسازی شده داده های validation نسبت به تصاویر بازسازی شده داده های پولیپ کیفیت بهتری دارند اگرچه هیچ کدام از این دو دسته تصویر روی پارامتر های مدل تاثیر نداشته اند.

۲-۶: ارزیابی عددی

ابتدا معیار های بررسی شده را تعریف میکنیم و سپس آنها را بدست می آوریم.

a. PSNR : در این معیار ما فرض میکنیم تصویر بازسازی شده همان تصویر اصلی است که یک نویز روی آن سوار شده است. انگار PSNR نسبت وضوح تصویر اصلی به وضوح نویز را اندازه گیریر میکند و هرچقدر این معیار بیشتر باشد یعنی نویز کمتر است و کیفیت بازسازی بیشتر است.
b. SSIM : این معیار سعی میکند کیفیت بصری که تصویر برای چشم انسان دارد را بسنجد. SSIM روی پنجره های کوچک از تصویر حرکت می کند و روشنایی، کنتراست و بافت تصاویر را مقایسه میکند. این معیار بین -۱ و ۱ است و هرچه به ۱ نزدیک تر باشد بازسازی بهتر انجام شده.

پس از بررسی این موارد حالا مقادیر این معیار ها را نمایش میدهیم:

```
--- Evaluation Results ---
Metric      Mean  Std Dev
  SSIM  0.424002 0.099547
  PSNR 17.162870 2.372506
```

در بخش مربوط به تحلیل کمی بخش بعد این مقادیر را تحلیل می کنیم.

۲-۷: تحلیل نتایج و بحث نهایی

- تصاویر تولیدشده تا چه حد واقع گرایانه‌اند؟

با توجه به تصاویر رنگ و شدت روشنایی و بافت تصاویر تولید شده (چه در بازسازی و چه در تولید) دور از انتظار و غیر منطقی نیستند و صرفا کمی نا واضح و تار هستند. به نحوی که اگر این تصاویر را به یک شخص عادی نشان دهیم احتمالا خواهد گفت که این تصاویر مربوط به تصاویر اندوسکپی هستند ولی کمی تار هستند.

- بازسازی پولیپ چه جزئیاتی را حفظ کرده و چه بخش‌هایی محو شده‌اند؟

باز سازی ها اکثر حفره ها یا لکه های سیاه موجود روی عکس، لبیل های سبز رنگ تصاویر اندوسکپی، بافت کلی رنگ تصویر و شدت روشنایی تصاویر را حفظ کرده است ولی جزئیات تصویر مانند چین خوردگی ها، بعضا مویرگ ها، براق بودن ترشحات، بخش های مربوط به استخوان ها و غزروف ها (مانند غزروف های حلقوی) و به طور کلی جزئیات تصویر محو شده اند.

- مقادیر PSNR و SSIM چه نکاتی درباره کیفیت بازسازی نشان میدهند؟

در بخش مربوط به ارزیابی عددی این موارد را تعریف کردیم و گفتیم چه چیز هایی درباره بازسازی را نشان میدهند. به طور کلی می توان گفت PNSR یک معیار خطای ریاضی است که سعی میکند

اختلاف پیکسل به پیکسل تصویر بازسازی شده و تصویر اصلی را اندازه گیری کند در حالی که SSIM تلاش می کند کیفیت بصری درک شده توسط انسان را اندازه گیری کند.

- اگر مقادیر PSNR و SSIM پایین هستند، احتمالاً مشکل در کدام بخش (معماری، هایپرپارامتر یا پیش‌پردازش) است؟

بنظر من در این مسئله خاص اگر متریک ها و معیار ها به حد قابل قبول نرسیدند دلیل عمده آن دیتاست است. مسئله ای که وجود دارد این است که دیتا ست ما بسیار کوچک بود و ما از همین دیتاست کوچک هم به طور کامل استفاده نکردیم! این تعداد داده برای اینکه یک مسئله طبقه بندی باینری ساده به یک دقت معقول برسد هم کافی نیست چه برسد به مسئله بازسازی و تولید تصاویر که ما باید به ازای یک تصویر ورودی یک تصویر خروجی تولید کنیم. نکته دیگر درباره دیتاست اندازه تصاویر آن است یکسری از تصاویر اندازه های بسیار بزرگی داشتند که ووقتی ما آنها را resize کردیم، کیفیت آنها از بین رفت. نکته دیگر نیز برچسب های سبز رنگ کنار تصاویر هستند. اگر به تصاویر دقت کنید مدل این برچسب ها را به خوبی بازسازی میکند و احتمالا چند میلیون پارامتر خرج یادگیری آنها کرده درصورتی که به هیچ وجه ضروری نیست! جدا کردن این برچسب ها در مرحله پیش پردازش کار دشواری است ولی اگر یکسری داده مشخصا به منظور ترین کردن شبکه های عصبی از تصاویر اندوسکپی جمع شوند، بهتر است که این برچسب ها را نداشته باشند.

یک نکته دیگر درباره معماری مدل است. به summary مربوط به انکدر دقت کنید. حدود ۹ میلیون پارامتر ما فقط در یک لایه دنس این انکدر استفاده شده در حالی که کل مدل ۱۱ میلیون پارامتر دارد. این افزونگی می توانست در جا های بهتری مصرف شود. میشد تعداد لایه ها را بیشتر کرد یا حتی از مکانیزم های توجه استفاده کرد ولی تمام این ۹ میلیون پارامتر در یک لایه دنس تجمع کرده اند!