به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

پرسش ۱ و ۲	نام و نام خانوادگی	علی خرم فر
	رایانامه	khoramfar@ut.ac.ir
پرسش ۱ و ۲	نام و نام خانوادگی	علی رمضانی
	رایانامه	Ali.ramezani.96@ut.ac.ir
	تاریخ ارسال پاسخ	۱۴۰۳/۰۴/۱۸

- هر دو نویسنده در هر دو پرسش همکاری داشته اند

- **فهرست**

فهرست تصاویر

فهرست جداول:

پرسش۱– Variational Auto–Encoder

برای پیاده‌سازی این تمرین از محیط Kaggle استفاده شد و دیتاست بر روی همین محیط ذخیره شد. در ادامه به بررسی کامل مدل‌ها خواهیم پرداخت.

۱-۱. پیش پردازش دیتاست

باتوجه به نکات اشاره شده در صورت تمرین ما ابتدا هردو دیتاست را دانلود کرده و در محیط Kaggle در یک پوشه مشترک ذخیره کردیم و یک کلاس دیتاست برای این تمرین نوشتیم که برای حالت‌های مختلف قابل استفاده باشد. در ادامه به بررسی این روند می‌پردازیم. دیتاست‌های مورد استفاده در این تمرین به شرح زیر هستند:

Anime Face Dataset که شامل تصاویر انیمه است.

Cartoon Faces Dataset که شامل تصاویر کارتونی است.

ابتدا دیتاست‌های Anime Faces و Cartoon Faces از منابع مربوطه دانلود و مسیرهای مربوط به این دیتاست‌ها مشخص و یک فولدر جدید برای ذخیره تصاویر ترکیبی از هر دو دیتاست ایجاد شد. تمامی تصاویر موجود در دیتاست Anime Faces به این فولدر جدید کپی شدند. سپس، **پنج** فولدر به صورت تصادفی از دیتاست Cartoon Faces انتخاب شده و تصاویر آن‌ها نیز به فولدر ترکیبی کپی شدند. در نهایت، تعداد کل فایل‌های موجود در دیتاست Anime Faces محاسبه و نمایش داده شد.

```
cartoon_folders = [str(i) for i in range(10)]

selected_cartoon_folders = random.sample(cartoon_folders, 5)
for folder in selected_cartoon_folders:
    folder_path = os.path.join(cartoon_faces_base_path, folder)
    copy_images_with_label(folder_path, combined_dataset_path, 'cartoon')
```

شکل ۱ انتخاب ۵ پوشه تصادفی از دیتاست **Cartoon** و اضافه کردن به پوشه مشترک

با اجرای این مراحل، یک فولدر ترکیبی شامل ۱۱۳۵۶۵ تصویر آماده شد که شامل ۵۰۰۰۰ تصویر کارتونی و ۶۳۵۶۵ تصویر انیمه می‌باشد. تا این مرحله به کمک Kaggle ذخیره‌سازی انجام شد. با اجرای دستور Commit فولدر مشترک به عنوان یک دیتاست در کگل ذخیره شد که به صورت یک فایل فشرده زیپ است.

برای استفاده از دیتاست، فولدر ترکیبی حاوی تصاویر از هر دو دیتاست را از فایل فشرده مربوطه استخراج می‌کنیم و تعداد تصاویر را برای تایید نهایی بررسی می‌کنیم:

Total number of files in Combined Folder: 113565

در ادامه هدف ما ایجاد یک کلاس دیتاست Custom است که بتواند تصاویر را از فولدر ترکیبی شامل دیتاست‌های Anime Faces و Cartoon Faces بارگذاری و پیش‌پردازش کند و برحسب مسئله بتوانیم از داده‌های یک نوع کلاس خاص و یا به صورت ترکیبی استفاده کنیم.

Create Custom Dataset

+ Code + Markdown

```
class CustomDataset(Dataset):
    def __init__(self, dataset_path, mode='combined', transform=None):
        self.dataset_path = dataset_path
        self.mode = mode
        self.transform = transform
        self.image_files = [f for f in os.listdir(dataset_path) if os.path.isfile(os.path.join(dataset_path, f))]

        if mode == 'anime':
            self.image_files = [f for f in self.image_files if 'anime' in f]
        elif mode == 'cartoon':
            self.image_files = [f for f in self.image_files if 'cartoon' in f]
        elif mode == 'combined':
            anime_files = [f for f in self.image_files if 'anime' in f]
            cartoon_files = [f for f in self.image_files if 'cartoon' in f]
            selected_anime_files = random.sample(anime_files, 50000)
            selected_cartoon_files = cartoon_files[:50000]
            self.image_files = selected_anime_files + selected_cartoon_files
```

شکل ۲ کلاس دیتاست **Custom**

در حالت combined، ۵۰۰۰۰ تصویر به صورت تصادفی از دسته انیمه و ۵۰۰۰۰ تصویر از دسته کارتونی انتخاب می‌شود. هر تصویر بر اساس نام فایل خود برچسب‌گذاری شده، تصاویر انیمه لیبل ۰ و تصاویر کارتونی لیبل ۱ دارند که در قسمت VAE از نوعConditional از این خاصیت دیتاست استفاده خواهیم کرد.

```
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.CenterCrop(128),
    transforms.ToTensor(),
   #transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # [-1, 1]
])
```

شکل ۳ پیش‌پردازش‌های انجام شده

برای پیش‌پردازش و آماده‌سازی تصاویر نیز از یک سری Transformation از کتابخانه پایتورچ استفاده شد. این ترانسفورمیشن‌ها به ترتیب شامل تغییر اندازه، برش مرکزی، تبدیل به تنسور و نرمال‌سازی هستند

که در نهایت برای نتیجه بهتر نرمال‌سازی را به همان تبدیل به تنسور بسنده کردیم که تصاویر را به نرمال ۰ و ۱ ببرد.

پس از آن دیتاست‌های Anime Faces، Cartoon Faces و Combinedیا ترکیبی را با استفاده از کلاس دیتاست Custom ایجاد می‌کنیم. سپس با استفاده از DataLoader، این دیتاست‌ها را با اندازه بچ‌سایز ۱۲۸ تایی بارگذاری می‌کنیم.

تعداد نمونه‌های موجود در هر دیتاست چاپ شد تا از صحت تعداد تصاویر در هر دسته اطمینان حاصل کنیم:

Anime Dataset: 63565

Cartoon Dataset: 50000

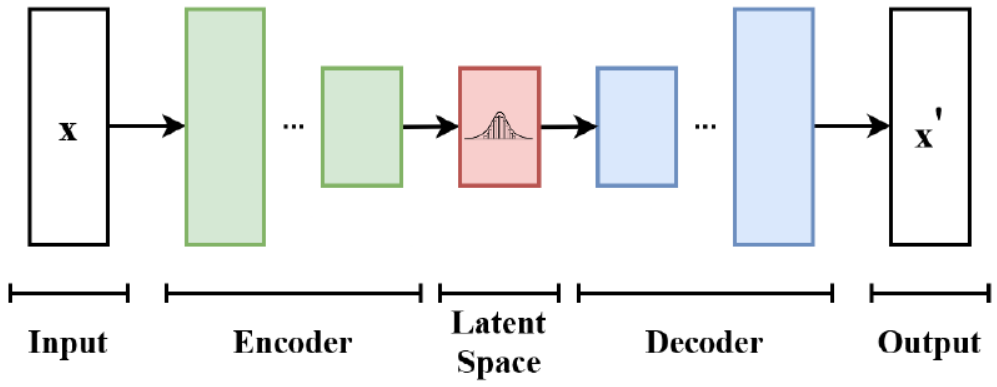
Combined Dataset: 100000 (۵۰۰۰۰ تصویر از هر دسته)

torch.Size([128, 3, 128, 128])

۱-۲. ساخت VAE روی دیتاست‌ها

توضیحات کلی در مورد VAE

Variational Autoencoder (VAE) یک نوع شبکه عصبی مولد است که برای تولید داده‌های جدید مشابه با داده‌های ورودی استفاده می‌شود. VAEها به صورت خاص برای تولید تصاویر، صداها و دیگر انواع داده‌های پیچیده کاربرد دارند. برخلاف Autoencoderهای معمولی، VAEها به جای یادگیری یک نگاشت مستقیم از ورودی به خروجی، یک توزیع احتمالی بر روی فضای Latent Space می‌گیرند. این ویژگی باعث می‌شود که VAEها قادر به تولید داده‌های جدید و متنوع باشند.



شکل ۴ معماری شبکه **VAE**

وظیفه Encoder، فشرده‌سازی ورودی به یک فضای پنهان یاLatent Space با ابعاد کمتر است. در VAE، به جای فشرده‌سازی مستقیم ورودی به یک بردار ثابت z، ورودی به دو بردار میانگین (μ) و انحراف معیار (σ) تبدیل می‌شود. از توزیع نرمال چندمتغیره یاد شده، نمونه‌گیری انجام می‌شود تا بردارهای فضای پنهان تولید شوند. این فرآیند از طریق ترفند reparameterization انجام می‌شود تا نمونه‌گیری به صورت قابل تمایز انجام شود. در نهایت وظیفه Decoder، بازسازی یا reconstruction ورودی اصلی از بردار فضای پنهان است.

Loss در VAE شامل دو قسمت اصلی است:

Reconstruction Loss: این قسمت از Loss نشان‌دهنده تفاوت بین داده ورودی و داده بازسازی شده است. هدف آن این است که بازسازی داده‌ها تا حد امکان به داده‌های اصلی نزدیک باشد. معمولاً از معیارهای مانند Mean Squared Error (MSE) یا Binary Cross-Entropy (BCE) برای محاسبه این لاس استفاده می‌شود.

Kullback-Leibler Divergence (KL Divergence): این Loss نشان‌دهنده تفاوت بین توزیع فضای پنهان یاد گرفته شده و یک توزیع نرمال استاندارد است. هدف این قسمت از لاس، نگه داشتن فضای پنهان به صورت یک توزیع نرمال استاندارد است.

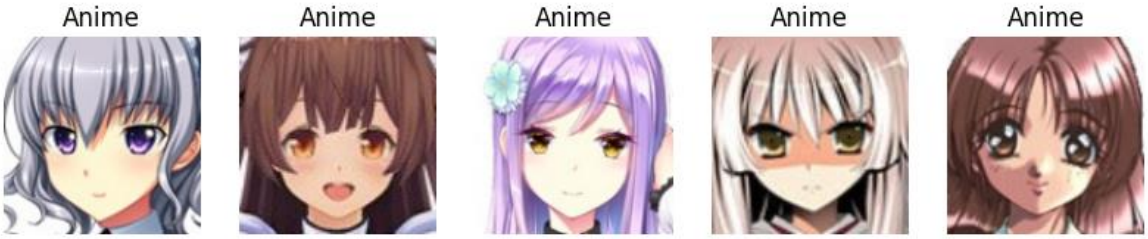
$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i \mid z)] + \mathbb{KL}(q_\theta(z \mid x_i) \mid\mid p(z))$$

شکل ۵ **Loss** به کار رفته در **VAE**

Loss function در VAE به منظور بهینه‌سازی دو هدف متضاد طراحی شده است: دقت بازسازی داده‌ها و نزدیکی توزیع احتمالی یادگیری‌شده به توزیع نرمال. این طراحی موجب می‌شود که VAEها توانایی تولید داده‌های جدیدی که مشابه داده‌های آموزش‌دیده هستند را داشته باشند، در حالی که خصوصیات آماری آن‌ها را نیز حفظ می‌کنند.

نمایش ۵ تصویر تصادفی از Anime Face

توضیحات مربوط به بارگذاری دیتاست در قسمت قبل به طور کامل گزارش شد. در فایل نوت‌بوک ۵ تصویر از هرکدام از دیتاست‌های ایجادشده خروجی گرفته‌شد. نتیجه ۵ تصویر تصادفی از دیتاست انیمه به صورت زیر است :



شکل ۶ نمایش ۵ تصویر تصادفی از **Anime Face**

همچنین باتوجه به اینکه دیتاست پیاده‌سازی شده هم امکان استفاده از CartoonFace و هم دیتاست ترکیبی را دارد از این نمونه‌ها هم خروجی گرفته شد:

شکل ۷ نمایش ۵ تصویر تصادفی از **Cartoon Face**

دیتاست ترکیبی نیز که از ۵۰ هزار تصویر Anime و ۵۰ هزار تصویر Cartoon تشکیل شده است:

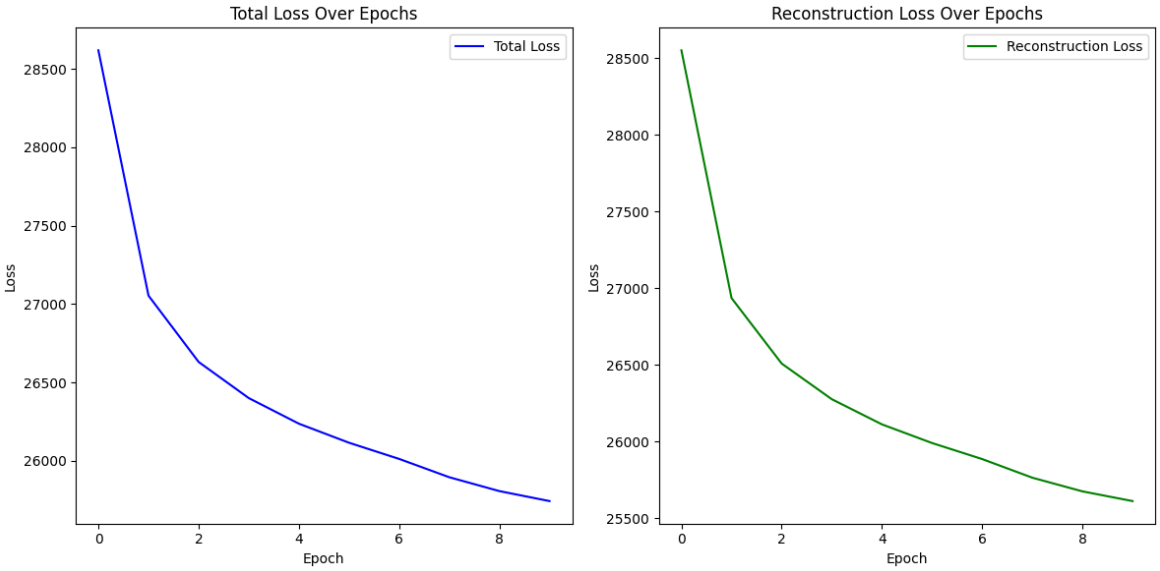
شکل ۸ نمایش ۵ تصویر تصادفی از **Combined**

آموزش مدل با معماری کانولوشن

لایه‌های کانولوشن با فیلترهای مختلف (۳۲، ۶۴، ۱۲۸، ۲۵۶، ۵۱۲) و اندازه‌های کرنل ۴ در ۴ به استفداه شدند.از تابع فعال‌سازی ReLU بعد از هر لایه کانولوشن استفاده شده است تا ویژگی‌های تصاویر استخراج شوند. خروجی آخرین لایه کانولوشن به یک بردار یک‌بعدی تبدیل و به لایه Fully Connected داده شد. یک لایه Fully Connected با ۱۰۲۴ نورون برای فشرده‌سازی بیشتر ویژگی‌ها به کار گرفته و دو لایه Fully Connected برای محاسبه میانگین و انحراف معیار Latent Spaceاستفاده شده است. همچنین از از reparameterization برای نمونه‌گیری از این توزیع و تولید بردار فضای پنهان استفاده شد. بخش Decoder نیز شامل لایه‌های Transpose Convolution است که وظیفه بازسازی تصاویر از فضای پنهان را دارند.

در آموزش مدل از بهینه‌ساز Adam. نرخ یادگیری ۰.۰۰۱، اندازه Batch ۱۲۸ و تعداد ایپاک ۱۰ استفاده شد. لاس مدل شامل دو بخش اصلی بود: Reconstruction Loss که تفاوت بین تصاویر ورودی و بازسازی شده را با استفاده از Binary Cross-Entropy اندازه‌گیری می‌کند و KL Divergence که تفاوت بین توزیع فضای پنهان یاد گرفته شده و توزیع نرمال استاندارد را محاسبه می‌کند.

کاهش تدریجی این لاس‌ها در طول ایپاک‌ها نشان‌دهنده بهبود مدل در بازسازی تصاویر و یادگیری ویژگی‌های داده‌های ورودی بود.



شکل ۹ **Reconstruction Loss** و لاس کلی طی آموزش مدل با معماری کانولوشن

تولید و نمایش تصاویر

برای تولید تصاویر جدید، از بردارهای تصادفی در Latent Space استفاده می‌شود. این بردارها از توزیع نرمال استاندارد نمونه‌گیری می‌شوند و سپس توسط بخش Decoder مدل VAE به تصاویر تبدیل می‌شوند.برای تولید این نویز و تولید تصویر به کمک مدل از تابع زیر استفاده می‌کنیم:

z = torch.randn(num_images, latent_dim)

```python
def generate_and_save_images_vae(model, num_images=8, latent_dim=256, save_path='/kaggle/working/vae_generated_images'):
    os.makedirs(save_path, exist_ok=True)
    model.eval()
    with torch.no_grad():
        z = torch.randn(num_images, latent_dim).to(device)
        generated_images = model.decode(z)
        generated_images = generated_images.cpu()

        for i in range(num_images):
            save_image(generated_images[i], os.path.join(save_path, f'vae_generated_image_{i}.png'))

        fig, axes = plt.subplots(1, num_images, figsize=(20, 5))
        for i, ax in enumerate(axes):
            ax.imshow(generated_images[i].permute(1, 2, 0))
            ax.axis('off')
        plt.show()
```

شکل ۱۰ تابع تولید تصویر به کمک مدل **VAE**

همچنین تصاویر تولیدی را برای سوال آخر ذخیره می‌کنیم.

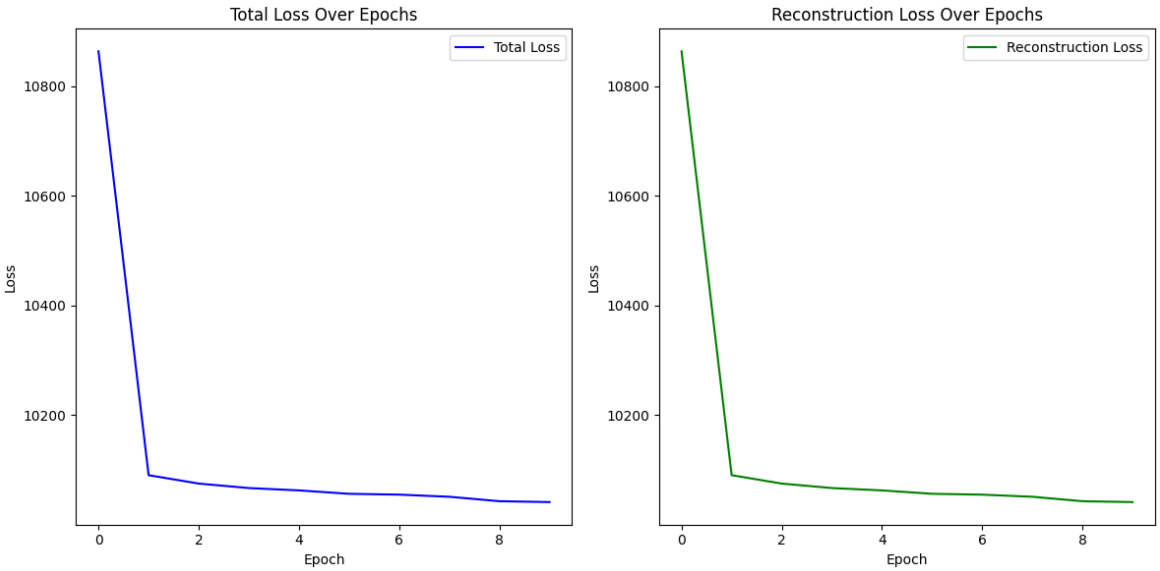
نتیجه تولید تصویر به کمک مدل آموزش دیده شده به صورت زیر است:

شکل ۱۱ تصاویر **Anime** تولیدشده توسط مدل با معماری کانولوشن

تصاویر تولیدی از کیفیت قابل قبولی برخوردارند و به مانند نمونه ارائه‌شده در صورت سوال هستند. این تصاویر نشان‌دهنده توانایی مدل در یادگیری ویژگی‌های پیچیده تصاویر و تولید داده‌های جدید است.

آموزش مدل با معماری کانولوشن بر دیتاست Cartoon Faces

در این مرحله هدف ما آموزش مدل VAE بر روی دیتاست Cartoon Faces است. نحوه انتخاب فولدرها و ساخت دیتاست در قسمت قبلی به طور کامل توضیح داده شد. همانند بخش قبل و با همان مدل و پارامترها، مدل VAE با استفاده از داده‌های جدید آموزش داده شد و روند کاهش Reconstruction Loss و Total Loss در طول ایپاک‌ها به صورت زیر است:



شکل ۱۲ **Reconstruction Loss** و لاس کلی طی آموزش بر روی دیتاست **Cartoon Faces**

همانطور که مشاهده می‌شود، Reconstruction Loss و Total Loss در طول ایپاک‌ها به تدریج کاهش یافته‌اند که نشان‌دهنده بهبود عملکرد مدل در بازسازی تصاویر کارتونی است. این نتایج مشابه با نتایج آموزش مدل بر روی دیتاست Anime Faces بوده و تأیید می‌کند که مدل VAE قادر به یادگیری و تولید تصاویر از هر دو نوع دیتاست می‌باشد. در شکل زیر نمونه تصاویر تولیدشده پس از آموزش این مدل بر روی دیتاست گفته شده را مشاهده می‌کنیم.

شکل ۱۳ تصاویر **Cartoon** تولیدشده توسط مدل با معماری کانولوشن

در تصاویر Cartoon Faces، تنوع بیشتری در ویژگی‌های چهره مانند رنگ پوست، مدل مو و حالت چهره مشاهده می‌شود. این نشان می‌دهد که مدل توانسته است به خوبی با تنوع موجود در دیتاست Cartoon Faces سازگار شود. در مقابل، تصاویر Anime Faces عمدتا دارای ویژگی‌های مشابهی هستند.

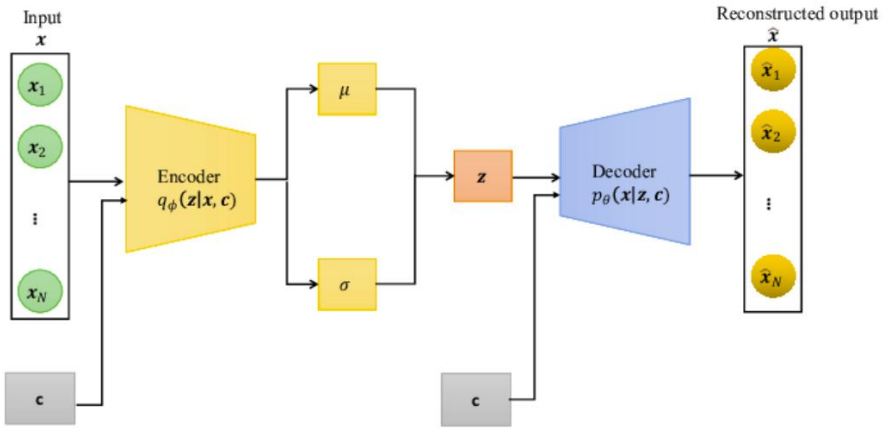
۱-۳. استفاده از یک مدل برای دو دیتاست

Conditional VAE (CVAE)

Conditional VAE (CVAE) یک نسخه پیشرفته از مدل VAE است که علاوه بر ورودی، از یک متغیر شرطی label برای تولید داده استفاده می‌کند. این مدل به ما امکان می‌دهد تا داده‌های تولیدی را به دسته‌های خاصی محدود کنیم، مانند تولید تصاویر از یک دسته خاص یا با ویژگی‌های مشخص.

تفاوت CVAE با VAE :

در CVAE، علاوه بر ورودی اصلی، یک label به مدل اضافه می‌شود که می‌تواند اطلاعاتی مانند دسته‌بندی تصاویر را به مدل بدهد. در VAE ساده، فقط ورودی اصلی به مدل داده می‌شود و هیچ اطلاعات اضافی به مدل داده نمی‌شود. در CVAE، هر دو بخش Encoder و Decoder علاوه بر ورودی اصلی، متغیر شرطی را نیز دریافت می‌کنند. این باعث می‌شود که فضای پنهان و بازسازی تصاویر به متغیر شرطی وابسته باشند در حالی که در VAE، فقط ورودی اصلی به Encoder داده می‌شود و بازسازی تصاویر نیز فقط بر اساس Latent Space انجام می‌شود.



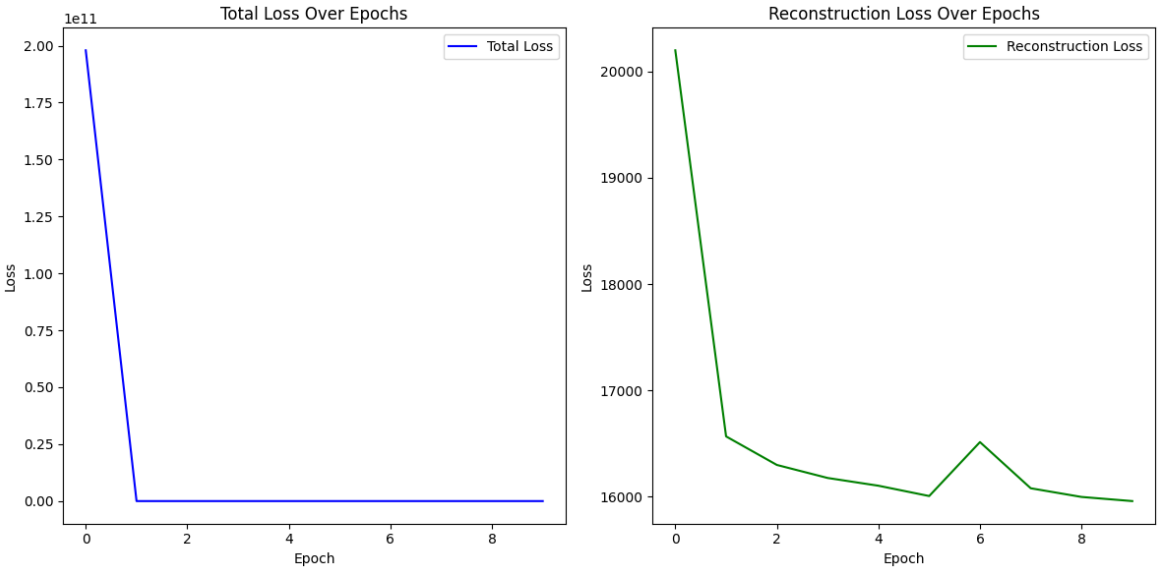
شکل ۱۴ معماری **CVAE**

این مدل نیز بر اساس انکودر و دیکودر است. انکودر از چندین لایه کانولوشنی تشکیل شده است که تصاویر ورودی به همراه برچسب‌ها را پردازش کرده و به یک بردار ویژگی فشرده تبدیل می‌کند. ابتدا، برچسب‌ها با اندازه تصاویر ورودی همسان شده و با تصاویر ترکیب می‌شوند. سپس، این ترکیب از طریق لایه‌های کانولوشنی عبور کرده و به یک بردار ویژگی تبدیل می‌شود. در نهایت، این بردار ویژگی به دو بردار مجزا برای میانگین (mu) و واریانس لگاریتمی (logvar) تقسیم می‌شود.

دیکودر نیز که وظیفه بازسازی تصاویر از فضای نهان را بر عهده دارد. ابتدا بردار ویژگی نهان به همراه لیبل‌ها ترکیب شده و از طریق لایه‌های Fully Connected و کانولوشن معکوس transpose convolution به یک تصویر بازسازی‌شده تبدیل می‌شود. در نهایت، خروجی دیکودر از تابع سیگموید عبور داده می‌شود تا مقادیر پیکسل‌ها در محدوده [۰, ۱] قرار گیرند.

برای این قسمت از دیتاست Combined استفاده کردیم که توضیحات آن در قسمت پیش‌پردازش دیتاست ارائه شد. این دیتاست شامل ۵۰۰۰۰ تصویر از هرکدام از دیتاست‌های اولیه است. و اندازه کل داده‌ها نیز برابر ۱۰۰۰۰۰ است. برای معماری نیز از لایه‌های کانولوشنی استفاده کردیم.

برای آموزش این مدل نیز از همان هایپرپارامترهای قبلی استفاده شد و مدل به خوبی آموزش داده‌شد. در تصویر زیر نمودار Loss را مشاهده می‌کنیم:

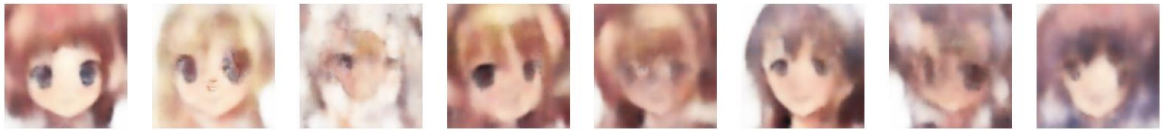


شکل ۱۵ **Reconstruction Loss** و لاس کلی طی آموزش مدل **CVAE**

در ابتدای آموزش، مقدار Total Loss بسیار بالا است و به تدریج با گذشت ایپاک‌ها کاهش می‌یابد. این کاهش نشان‌دهنده یادگیری تدریجی مدل و بهبود عملکرد آن است. نوسانی در در طول ایپاک‌ها هم قابل مشاهده است و احتمال دارد به دلیل تغییرات جزئی در داده‌ها و به‌روزرسانی وزن‌ها رخ دهد.

تولید تصاویر با مدل CVAE

در این قسمت هم برای تولید تصاویر، از بردارهای تصادفی در فضای پنهان استفاده شد. این بردارها از توزیع نرمال استاندارد نمونه‌گیری می‌شوند و سپس همراه با لیبل‌ها به بخش Decoder مدل CVAE داده می‌شوند تا تصاویر جدید تولید شوند که این مورد در تابع مربوط به تولید تصاویر و نمایش آن‌ها پیاده‌سازی شد. در تصاویر زیر تولیدات انجام شده توسط مدل CVAE را مشاهده می‌کنیم. باید توجه داشته باشیم که هردوی این تصاویر توسط ۱ مدل تولید شده‌اند ولی لیبل آن‌ها متفاوت است.



شکل ۱۶ تصاویر **Anime** تولیدشده توسط مدل **CVAE**

شکل ۱۷ تصاویر **Cartoon** تولیدشده توسط مدل **CVAE**

اگرچه کیفیت تصاویر تولیدشده توسط این مدل کمتر از قبلی است ولی باید توجه داشته باشیم که در آموزش این مدل از دیتاست ترکیبی استفاده شده است و بر اساس لیبل این عملیات صورت گرفته است. برخی تصاویر تولید شده دارای وضوح کمتری هستند که می‌توان با بهبود مدل و افزایش ایپاک‌ها این مشکل را برطرف کرد. باتوجه به زمان محدود و اینکه کیفیت ملاک این قسمت نیست، تلاش بیشتری برای آموزش بیشتر این مدل صورت نگرفت ولی یک مدل دیگر را با کمک مقاله زیر بر روی فقط ۱ کانال به تعداد ۱۰ ایپاک پیاده‌سازی و آموزش داده شد که در ادامه گزارش آن را بررسی خواهیم‌کرد.

Harvey, W., Naderiparizi, S., & Wood, F. (2021). Conditional image generation by conditioning variational auto-encoders. *arXiv preprint arXiv:2102.12037*.

پیاده‌سازی اضافه – مدل CVAE به صورت Hierarchical

در این قسمت، مدل Hierarchical Conditional Variational Autoencoder را پیاده‌سازی کرده‌ایم. این مدل مشابه مدل قبلی (CVAE) است اما با یک تفاوت اصلی: استفاده از معماری سلسله‌مراتبی برای فضای نهان (latent space).

در forward، انکودر ورودی را به چندین بردار نهان تبدیل می‌کند. سپس از تابع reparameterize برای نمونه‌گیری از این بردارها استفاده می‌شود و نتایج این نمونه‌گیری‌ها با هم ترکیب شده و به دیکودر داده می‌شود. دیکودر نیز تصویر بازسازی‌شده را تولید می‌کند. همچنین، بردارهای mu و logvar برای محاسبه تابع هزینه به خروجی برگردانده می‌شوند.

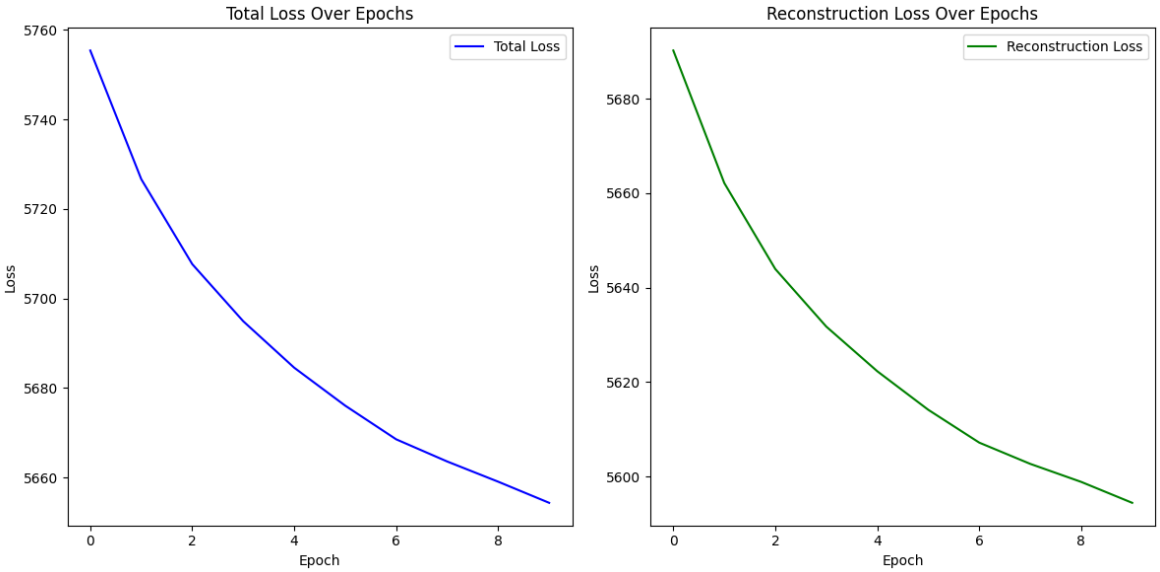
تفاوت با مدل قبلی (CVAE)

مدل Hierarchical CVAE از چندین لایه مستقل برای تولید بردارهای نهان استفاده می‌کند، در حالی که CVAE تنها از یک لایه برای تولید بردارهای نهان استفاده می‌کند. این باعث می‌شود که مدل بتواند اطلاعات پیچیده‌تری را در فضای نهان ذخیره کند. این تفاوت‌ها باعث می‌شود که Hierarchical CVAE در مقایسه با CVAE بتواند تصاویر با کیفیت‌تر و متنوع‌تری تولید کند، به ویژه در مواردی که داده‌ها دارای ساختارهای پیچیده هستند.

برای آموزش این مدل از پارامترهای زیر استفاده شد:

```
input_dim = 128 * 128
hidden_dim = 512
latent_dim = 20
num_layers = 2
n_labels = 2
num_epochs = 10
```

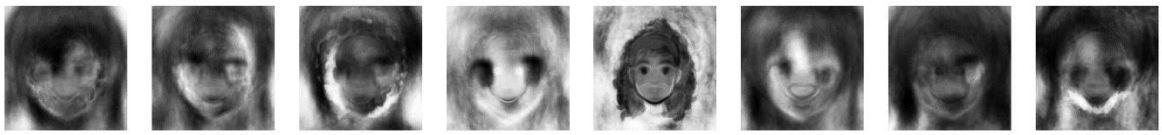
در تصویر زیر نمودار Loss را مشاهده می‌کنیم:



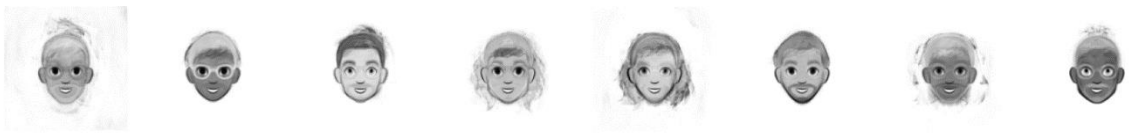
شکل ۱۸ **Reconstruction Loss** و لاس کلی طی آموزش مدل **Hierarchical CVAE**

در نهایت، مدل Hierarchical CVAE با موفقیت آموزش داده شد و توانستتصاویر ورودی را با دقت بالاتری بازسازی کند که این مورد را با توجه به نمودار Loss بالا مشاهده می‌کنیم. نتایج خروجی نیز به صورت زیر است:

باید توجه داشته باشیم که هردوی این تصاویر توسط ۱ مدل تولید شده‌اند ولی لیبل آن‌ها متفاوت است.



شکل ۱۹ تصاویر **Anime** تولیدشده توسط مدل **Hierarchical CVAE**



شکل ۲۰ تصاویر **Cartoon** تولیدشده توسط مدل **Hierarchical CVAE**

۱-۴. VQ-VAE

Vector Quantised-Variational AutoEncoder (VQ-VAE) یک مدل Generative جدید است که تفاوت‌های کلیدی با VAE دارد که عبارتند از:

فضای گسسته: برخلاف VAE که از فضای پیوسته استفاده می‌کند، VQ-VAE از فضای گسسته استفاده می‌کند. انکودر در VQ-VAE خروجی‌های گسسته تولید می‌کند که به وسیله جستجوی نزدیک‌ترین همسایه (nearest neighbor lookup) به یک بردار embedding vector نگاشت می‌شوند.

پیش‌آموزش :درVQ-VAE ، توزیع پیشین (prior) به صورت خودکار و با استفاده از مدل‌هایی مانند PixelCNNیا WaveNet یاد گرفته می‌شود، در حالی که درVAE ، توزیع پیشین معمولاً به صورت ثابت و گاوسی فرض می‌شود.

Posterior Collapse: یکی از مشکلات VAE در حضور دیکودرهای قوی، فروپاشی پسین است که در آن انکودر نقش کمتری در مدل ایفا می‌کند. فروپاشی پسین یک مشکل رایج در مدل‌های مولد مانند VAE است، به خصوص زمانی که از دیکودرهای بسیار پیچیده استفاده می‌شود. در این حالت، مدل یاد می‌گیرد که کاملاً به دیکودر وابسته شود و متغیرهای نهفته (Latent Variables) را نادیده بگیرد. به عبارت دیگر، انکودر دیگر اطلاعات مفیدی به متغیرهای نهفته نمی‌فرستد و دیکودر به تنهایی قادر به بازسازی ورودی‌ها می‌شود. VQ-VAE با استفاده از فضای گسسته و Quantization برداری این مشکل را برطرف می‌کند.

به طور کلی VQ-VAE قادر است ویژگی‌های مهم داده‌ها را به خوبی مدل کند و بر روی ویژگی‌های محلی و جزئیات بی‌اهمیت تمرکز نکند.

تفاوت‌ها با VAE ساده:

مدل VQ-VAE تفاوت‌های اساسی با VAE دارد که به طور عمده در نوع متغیرهای نهفته، روش کمیت‌گذاری و یادگیری توزیع پیشین آن دیده می‌شود. در VAE ساده، متغیرهای نهفته به صورت پیوسته و با توزیع نرمال مدل‌سازی می‌شوند و انکودر یک توزیع گاوسی را برای هر داده ورودی تولید می‌کند. اما در VQ-VAE، این متغیرها به صورت گسسته هستند و انکودر خروجی‌های خود را به نزدیک‌ترین کد در یک فضای Embedding برداری نگاشت می‌کند که این کدها از یک فضای گسسته انتخاب می‌شوند. این استفاده از Embedding برداری به مدل اجازه می‌دهد تا از یک فضای نمایشی گسسته استفاده کند که برای بسیاری از داده‌ها مناسب‌تر است.

علاوه بر این، در VAE ساده، توزیع پیشین به صورت ثابت و گاوسی فرض می‌شود، در حالی که در VQ-VAE، توزیع پیشین با استفاده از مدل‌هایی مانند PixelCNN یا WaveNet یادگرفته می‌شود. این توزیع پیشین قدرتمندتر است و می‌تواند وابستگی‌های پیچیده‌تری را بین متغیرهای نهفته مدل کند. این روش باعث می‌شود که مشکل فروپاشی پسین که در VAEهای ساده با دیکودرهای قدرتمند رایج است، در VQ-VAE به طور موثری کاهش یابد

معماری VQ-VAE:

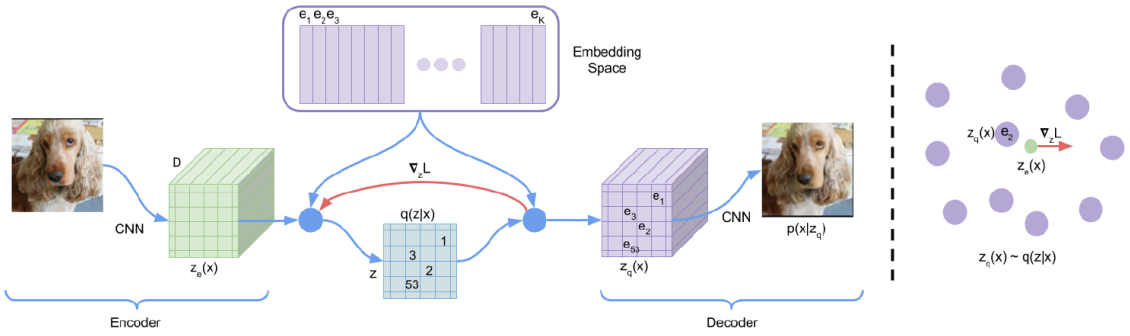
انکودر: انکودر ورودی را به یک فضای Embedding پیوسته نگاشت می‌کند و سپس با استفاده از جستجوی نزدیک‌ترین همسایه، آن را به کدهای گسسته تبدیل می‌کند.

دی‌کدر: دی‌کدر کدهای گسسته را به فضای ورودی Reconstruct می‌کند.

(Vector Quantization) : در این بخش، کدهای پیوسته خروجی انکودر به نزدیک‌ترین بردار Embedding نگاشت می‌شوند.

پیش‌آموزش Prior trained : این مدل با استفاده از مدل‌هایی مانند PixelCNN یا WaveNet توزیع پیشین را یاد می‌گیرد که امکان نمونه‌گیری از داده‌های جدید را فراهم می‌کند.

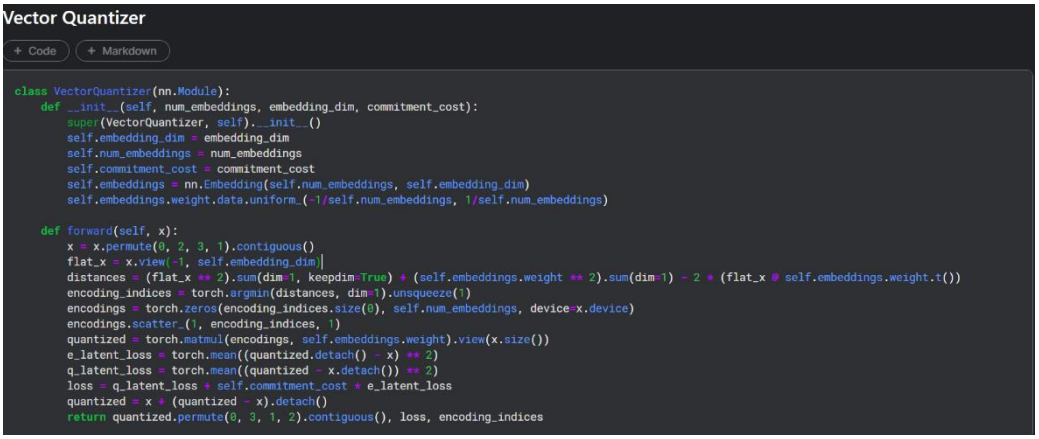
شکل معماری این مدل به صورت زیر است و فضای Embedding نیز در سمت راست تصویر قابل مشاهده است:



شکل ۲۱معماری **VQ-VAE**

پیاده‌سازی مدل

کلاس VectorQuantizer مسئول تبدیل ویژگی‌های استخراج شده از انکودر به بردارهای Embedding است.



ابتدا بردارهای Embedding با ابعاد مشخص تعریف و مقداردهی اولیه می‌شوند. سپس ویژگی‌های پیوسته خروجی انکودر به نزدیک‌ترین بردار تعبیه‌ای گسسته نگاشت می‌شوند. برای این کار، فاصله اقلیدسی بین ویژگی‌های پیوسته و بردارهای تعبیه‌ای محاسبه و نزدیک‌ترین بردار انتخاب می‌شود. برای بهبود فرایند یادگیری، دو نوع Loss محاسبه می‌شود: یکی برای ویژگی‌های انکودر و دیگری برای بردارهای Embedding. این Loss باعث می‌شوند که ویژگی‌های پیوسته به بردارهای Embedding نزدیک‌تر شوند.

کلاس VQVAE نیز ساختار کلی مدل را تعریف می‌کند و شامل سه بخش انکودر، pre-vq و دیکودر است. انکودر شامل چندین لایه کانولوشن است که وظیفه استخراج ویژگی‌های مهم از تصاویر ورودی را بر عهده دارند. لایه پیش از کوانتیزاسیون (pre-vq)نیز وظیفه تبدیل ویژگی‌های استخراج شده از انکودر به فضای بردارهای Embedding را دارد. در نهایت دیکودر شامل چندین لایه کانولوشن معکوس است که وظیفه بازسازی تصاویر از بردارهای Embedding گسسته را بر عهده دارد.

برای آموزش این مدل از هایپرپارامترهای زیر استفاده شد:

num_hiddens = 128: تعداد نورون‌های لایه‌های مخفی در انکودر و دیکودر

num_residual_hiddens = 32: تعداد نورون‌های residual layers

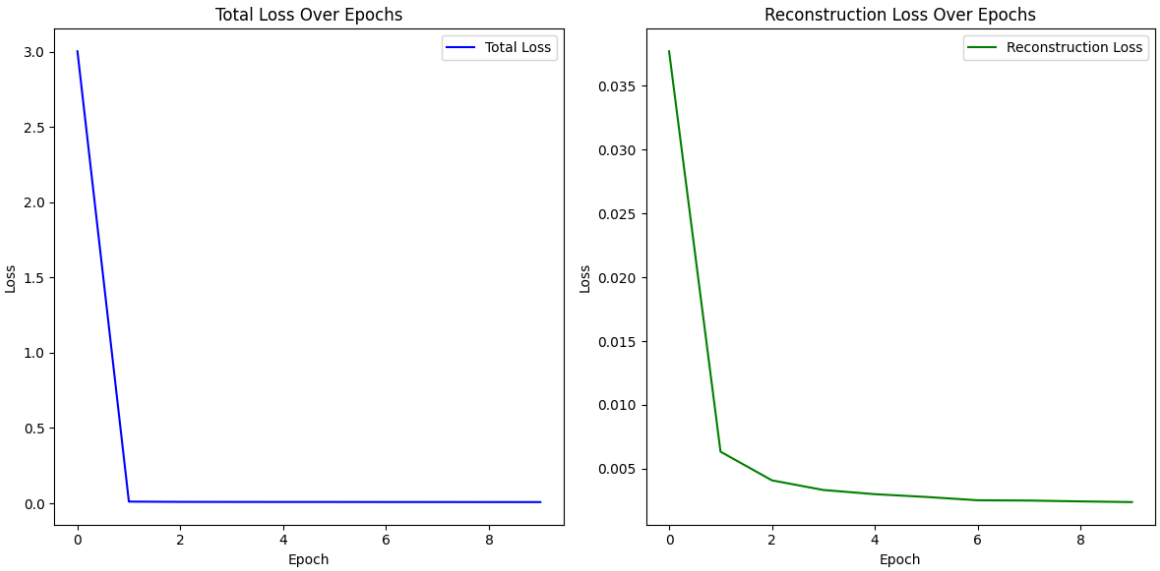
num_residual_layers = 2: تعداد لایه‌های residual layers در انکودر و دیکودر.

embedding_dim = 64: ابعاد بردارهای Embedding

num_embeddings = 512: تعداد بردارهای Embedding

commitment_cost = 0.25: هزینه commitmentکه باعث می‌شود که ویژگی‌های انکودر به بردارهای Embedding نزدیک‌تر شوند.

learning_rate = 1e-3: نرخ یادگیری



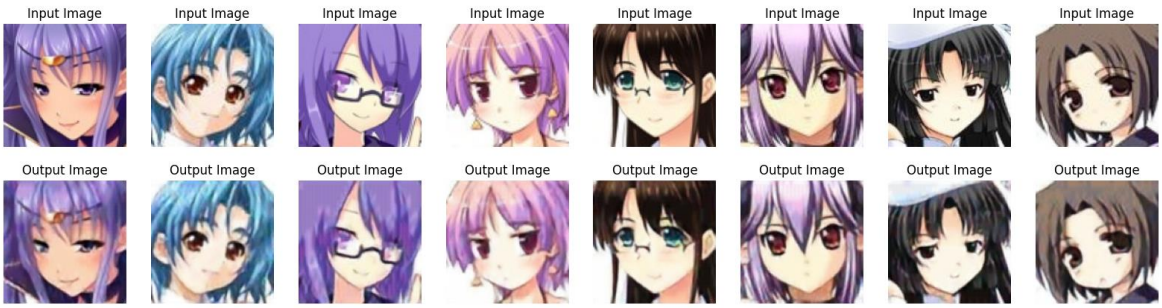
شکل ۲۲ **Reconstruction Loss** و لاس کلی طی آموزش مدل **VQ-VAE**

کاهش سریع و اولیه در Reconstruction Loss نشان‌دهنده بهبود کیفیت بازسازی تصاویر است. مقدار پایدار در ایپاک‌های بعدی نشان‌دهنده همگرایی مدل و توانایی آن در بازسازی تصاویر با کیفیت مناسب است.

در ادامه به بررسی تصاویر تولیدشده توسط این مدل می‌پردازیم:

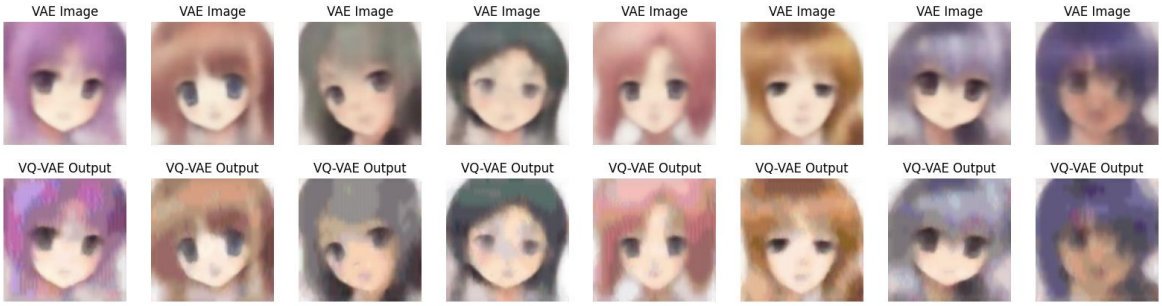
تولید تصاویر با مدل VQ-VAE

مدل VQ-VAE توانسته است به خوبی تصاویر ورودی از دیتاست **Anime Faces** را بازسازی کند. کیفیت و وضوح تصاویر بازسازی شده نشان‌دهنده عملکرد موفق مدل در یادگیری ویژگی‌های داده‌های ورودی و بازسازی آنها است.



شکل ۲۳ تصاویر بازسازی **Anime** توسط مدل **VQ-VAE**

مدل VQ-VAE با استفاده از بردارهای Embedding گسسته، قادر است جزئیات بیشتری از تصاویر را حفظ کرده و وضوح تصاویر را بهبود بخشد. در تصویر زیر، مقایسه‌ای بین تصاویر تولید شده توسط مدل‌های **VAE**و بازسازی شده توسط **VQ-VAE**ارائه شده است. تصاویر خروجی VAE از همان مدل پیاده‌سازی شده اند که آن‌ها را ذخیره کرده‌ایم.



شکل ۲۴تصاویر بازسازی خروجی **VAE** توسط مدل **VQ-VAE**

تصاویر بازسازی شده توسط مدل VQ-VAE نشان‌دهنده بهبود جزئیات و وضوح بیشتر نسبت به خروجی‌های VAE هستند.

۱-۵. VQ-VAE 2

این مقاله به بررسی استفاده از مدل‌های VQ-VAE برای تولید تصاویر در ابعاد بزرگ می‌پردازد. هدف اصلی مقاله این بوده که مدل‌های autoregressive را در VQ-VAE بهبود داده تا نمونه‌های مصنوعی با کیفیت بالاتر از مدل‌های قبلی تولید شوند. این مدل از شبکه‌های ساده encoder و decoder استفاده می‌کند. علاوه بر این، VQ-VAE تنها نیاز به نمونه‌گیری از مدل autoregressive در فضای نهان فشرده دارد که نسبت به نمونه‌گیری در فضای پیکسلی به ویژه برای تصاویر بزرگ بسیار سریع‌تر است.

همانطور که در بخش قبلی تمری بررسی کردیم، مدل VQ-VAE شامل یک encoder است که ورودی را به یک توالی از متغیرهای نهان گسسته نگاشت می‌کند و یک decoder که مشاهدات را از این متغیرهای گسسته بازسازی می‌کند.

این مقاله دو مرحله اصلی برای بهبود عملکرد مدل‌های VQ-VAE را معرفی می‌کند.

مرحله ۱: یادگیری کدهای نهفته سلسله‌مراتبی (Hierarchical Latent Codes)

هدف این مرحله مدل‌سازی اطلاعات Local و Global تصاویر به صورت جداگانه است. این کار باعث می‌شود مدل بتواند جزئیات دقیق‌تری را حفظ کرده و بازسازی کند.

در این مرحله، مدل VQ-VAE به صورت سلسله‌مراتبی طراحی شده است. تصاویر ورودی ابتدا به یک نمای ۶۴×۶۴ تبدیل می‌شوند و سپس به یک نمای بالاتر ۳۲×۳۲ مقیاس داده می‌شوند. هر دو نمای به صورت مجزا Quantize شده و به کدهای نهفته Latent Codes تبدیل می‌شوند. این کار باعث می‌شود مدل بتواند اطلاعات مختلف را در سطوح مختلف ذخیره کند.

این رویکرد باعث می‌شود که مدل بتواند بازسازی‌های دقیق‌تر و با جزئیات بیشتری داشته باشد.

مرحله ۲: یادگیری توزیع‌های پیشین (Prior Distributions) بر روی کدهای نهفته

هدف این مرحله فشرده‌سازی بیشتر تصاویر و تولید نمونه‌های جدید از مدل آموزش دیده است

در این مرحله، از مدل‌های Autoregressive مانند PixelCNN برای یادگیری توزیع‌های اولیه بر روی کدهای نهفته استفاده می‌شود. این مدل‌ها با استفاده از شبکه‌های عصبی قدرتمند، توزیع‌های پیچیده‌تری را مدل‌سازی می‌کنند که منجر به بازسازی‌های دقیق‌تر می‌شود. استفاده از این روش باعث می‌شود که مدل بتواند تصاویر با کیفیت بالا و تنوع زیاد تولید کند.

مزایا و نتایج مقاله

کیفیت و تنوع تصاویر: مدل VQ-VAE-2 توانسته است تصاویری با کیفیت و تنوع بالا تولید کند که قابل مقایسه با روش‌های پیشرفته مانند GANها است. این مدل برخلاف GANها، مشکلاتی مانند Mode Collapse و فقدان تنوع را ندارد.

سرعت نمونه‌گیری: یکی از مزایای مهم این مدل، سرعت بالای نمونه‌گیری است. با فشرده‌سازی تصاویر به فضای نهفته گسسته، نمونه‌گیری از مدل بسیار سریع‌تر از نمونه‌گیری در فضای پیکسلی انجام می‌شود.

کاربردها: این مدل می‌تواند در کاربردهای مختلفی مانند بهبود وضوح تصاویر، ویرایش دامنه‌ها و تولید تصاویر هنری به کار گرفته شود.

به طور کلی مقاله نشان می‌دهد که با استفاده از رویکرد سلسله‌مراتبی و مدل‌های Autoregressive، می‌توان تصاویری با کیفیت بالا و تنوع زیاد تولید کرد. مدل VQ-VAE-2 با حفظ سادگی و کارایی، به عنوان یک راهکار جذاب برای کاربردهایی که نیاز به کدگذاری و کدگشایی سریع تصاویر دارند، مطرح شده است.

تفاوت با VAE ساده

کوانتیزاسیون برداری (Vector Quantization): برخلاف VAE ساده که از توزیع‌های پیوسته استفاده می‌کند، VQ-VAE از کوانتیزاسیون برداری و کدهای گسسته استفاده می‌کند که باعث فشرده‌سازی بهتر و بازسازی دقیق‌تر می‌شود.

مدل‌سازی سلسله‌مراتبی: VQ-VAE-2 با استفاده از مدل‌سازی سلسله‌مراتبی، اطلاعات Local و Global را به صورت جداگانه مدل‌سازی می‌کند که باعث بهبود بازسازی تصاویر می‌شود.

سرعت نمونه‌گیری (Sampling Speed): با فشرده‌سازی تصاویر به فضای نهفته گسسته، سرعت نمونه‌گیری در VQ-VAE بسیار بالاتر از VAE ساده است.

پرسش۲ – Image Translation

۲-۱.آشنایی با Image Translation و معماری Pix2Pix

سوال اول – تفاوت مدل ذکر شده با مدل GAN ساده:

مدل PIX2PIX در اصل یک cGAN یا به عبارتی یک Conditional GAN است و بهتر است ابتدا این تفاوت را برجسته کنیم:

- مدل GAN: در این مدل ساده Generator از یک نویز، خروجی مورد نظر را میسازد و discriminator آن خروجی را به یکی از برچسب های real, fake نظیر میکند:

$$G: \{z\} \rightarrow y.$$

شکل ۲۵ - مدل ساده **GAN**

در حالی که در مدل cGAN، ورودی علاوه بر نویز ورودی، به یک ورودی مشخص شرطی شده است، به طوری که خروجی باید یک خروجی Stochastic اما شرطی شده بر اساس ورودی x باشد.

$$G : \{x, z\} \rightarrow y.$$

شکل ۲۶ - مدل **cGAN**

این تفاوت در نحوه تعریف loss نیز خودش را به خوبی نشان میدهد، به طوری که در مدل cGAN مقدار احتمال خروجی در تولید کننده وابسته به مقدار شرطی ورودی است:

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))]$$

شکل ۲۷ - تابع هزینه در **cGAN**

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_{y}[\log D(y)] + \mathbb{E}_{x,z}[\log(1 - D(G(x, z))]$$

شکل ۲۸ - تابع هزینه در **GAN**

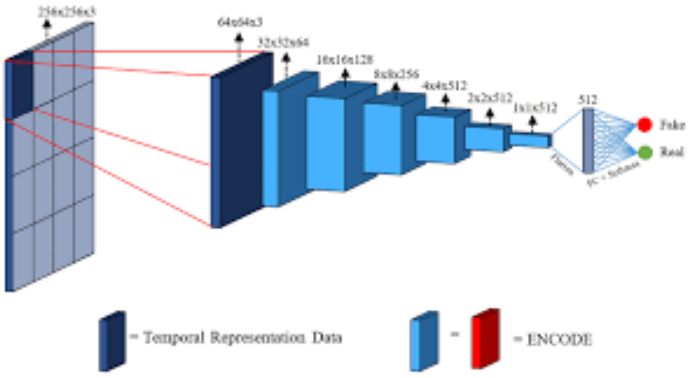
این تفاوت البته در معماری نیز بسیار مشهود است که در آن هدف Generator، نگاشت ورودی از یک دامنه به دامنه دیگری است، و در Discriminator نیز ما در حال ایجاد تغییرات سطحی در تصویر ورودی هستیم و تفاوت های زمینه ای را محدود میکنیم.

از این ایده در حوزه پزشکی میتوان استفاده های بسیاری برد، مثلا میتوان تصاویر CT-SCAN را به MRI نظیر کرد، این ایده در مقالاتی مانند Ambient-Pix2PixGAN for Translating Medical Images from Noisy Data استفاده شده است.

سوال دوم – تفاوت بخش Discriminator:

در یک مدل ساده GAN، بخش Discriminator بسیار ساده است، این بخش یک شبکه طبقه بند ساده است، که از یک یا چند لایه کانوولوشنی و در نهایت یک لایه خطی(Dense or Linear) تشکیل شده است که به طبقه بندی تصویر ورودی به کلاس های real و fake میپردازد، در حالی که در معماری این مدل از PatchGAN به عنوان جداکننده استفاده شده است برای بررسی دقیق این دو بگذارید یک مثال بزنیم:

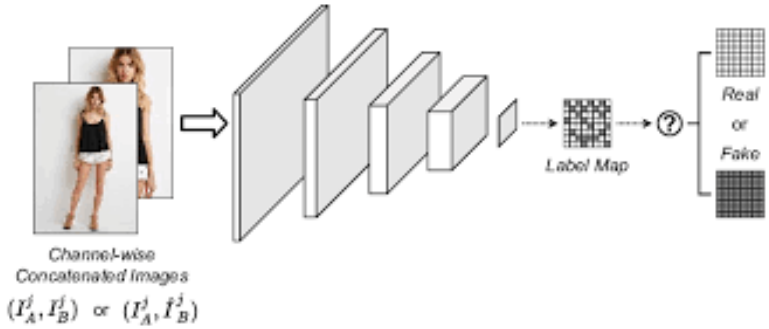
- یک تصویر ورودی ۲۵۶ در ۲۵۶ در یک معماری ساده GAN، ابتدا از چند لایه کانوولوشنی میگذرد، سپس Flat شده و در نهایت در یک لایه خطی با یک نورون، یک عدد اسکالر در خروجی تولید میکند، این عدد احتمال real یا fake بودن کل تصویر ۲۵۶ در ۲۵۶ است.



شکل ۲۹ - ساختار **Discriminator** یک مدل **GAN**

- مدل PatchGAN صرفا از لایه های کانوولوشنی تشکیل شده است، یعنی لایه Flat را در لایه پایانی ندارد، پس خروجی نیز یک تنسور دو بعدی N * N است، فرض کنید نام این تنسور X باشد، پس هر خانه ij در این تنسور، به یک بخشی از تصویر ۲۵۶ در ۲۵۶ ورودی اشاره میکند، این بخش که همان receptive filed است، در این معماری برابر با

یک بخش overlapping با ابعاد ۷۰ در ۷۰ است. بسیار عالی، اما هر یک از ij چه چیزی را نشان میدهد؟ هر عدد در هر خانه ij معین میکند که آن بخش ۷۰ در ۷۰ معادل در تصویر اولیه real است یا fake، و سپس با میانگین گیری روی همه این بخش های یک response واحد برای خروجی تولید میشود، سوال این است که خب چرا این به اصطلاح patch ها به اندازه کل تصویر نباشد؟ کوچک کردن patch یعنی پارامتر کمتر و اجرای سریعتر مدل هم در آموزش و هم در تست.

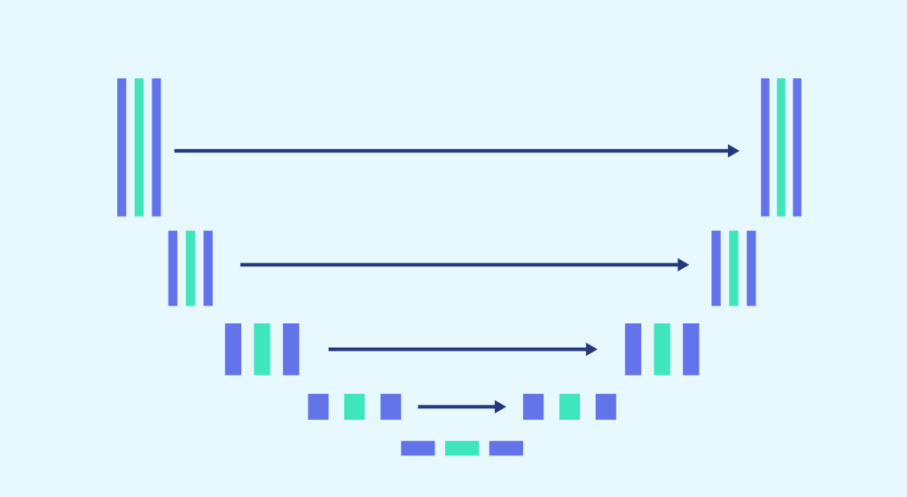


شکل ۳۰ - ساختار **Discriminator** در **PatchGAN**

سوال این است که تغییر معماری چطور باعث بهبود عملکرد مدل میشود؟ به اینکار ما مدل را به دلیل انحراف از بخش های کوچک در ورودی Penalize میکنیم، در نتیجه مدل سعی میکند ویژگی های پرتکرار high frequency features را در نظر بگیرد، این نوع معماری در مقالات به عنوان Markov random field شناخته میشود که در حقیقت یک تابع هزینه برای کمینه کردن تفاوت style/texture است.

سوال سوم - تفاوت بخش Generator:

در این مدل بخش Generator از معماری UNet استفاده میکند، این معماری با توجه به نامش، شبیه U است، در حقیقت شامل یک بخش Encoder و یکی بخش Decoder است، با این تفاوت که برخلاف معماری encoder- decoder شامل skip connection هایی که است خروجی هر لایه نظیر در encoder با به لایه decoder، اضافه میکند.



شکل ۳۱ - معماری **UNet**

برای بررسی هر بخش میتوانیم موارد را به شکل زیر بررسی کنیم:

هر مرحله از بخش down sample یک لایه کانوولوشنی با اندازه فیلتر 4 در ۴ است و stride 2 است که از padding = same استفاده میکند، همچنین به جز لایه اول، همه لایه ها دارای لایه bathc normalization بوده که و دارای تابع فعالساز leaky Relu هستند، برای بررسی ابعاد در این بخش میتوانیم روند زیر را در نظر بگیریم:

Down sample layer	Size
Input size	256,256,3
Conv(64) + leaky Relu	128,128,64
Conv(128) + BN + leaky Relu	64,64,128
Conv(256) + BN + leaky Relu	32,32,256
Conv(512) + BN + leaky Relu	16,16,512
Conv(512) + BN + leaky Relu	8,8,512
Conv(512) + BN + leaky Relu	4,4,512
Conv(512) + BN + leaky Relu	2,2,512
Conv(512) + BN + leaky Relu	1,1,512

جدول ۱ - بخش **Encoder** در **UNET**

سپس در بخش Decoder (Up sample) این روند را برعکس میکنیم، لایه های Upsample دارای یک لایه Conv2DTranspose هستند که دارای stride 2 و padding = same است، همه آن ها دارای لایه Batch Norm هستند، که در انتها نیز دارای تابع فعالساز Relu هستند، در سه لایه اول از فرآیند Up

sample از droput با احتمال ۰.۵ استفاده میکنیم. دلیل اضافه شده این لایه Dropout را در ادامه بررسی میکنیم:

- اضافه کردن Dropout در لایه های ابتدایی Generator: دلیل این موضوع به خاصیت استفاده از ساختار cGAN بر میگردد، بخاطر داریم که گفتیم در یک Conditional GAN ما یک بردار ورودی به صورت $G: \{x, z\} \rightarrow y$ را که در حقیقت z نویز ورودی به مدل است، را شکل میدهیم، واقعیت این است که این مدل ها بدون استفاده از بردار z نیز قابلیت یادگیری دارند، اما در این شرایط خروجی کاملا deterministic میشود، و به جز یک تابع دلتا که در یک نقطه پیک میزند، نمیتوانند هیچ توزیعی از ورودی را یادبگیرند.

این موضوع در GAN های اولیه نیز شناخته شده و آن ها به افزودن بردار Z به مدل اقدام کردند، اما در این مقاله، نویسندگان ادعا میکنند با اینکار فایده ی زیاد ندیده اند، چراکه Generator تنها اقدام به نادیده گرفتن نویز کرده است، پس اقدام به اضافه کردن این نویز به صورت Dropout به مدل کرده اند تا بتوانند خروجی های Stochastic از مدل تولید کنند، اما در نهایت هم بهبود کمی مشاهده کرده اند، در نهایت نویسنده تولید خروجی های تماما Stochastic از مدل های cGAN را یک Open question مهم میدانند.

our nets. Designing conditional GANs that produce highly stochastic output, and thereby capture the full entropy of the conditional distributions they model, is an important question left open by the present work.

شکل ۳۲- بخش جالب مقاله درباره نویز در مدل های **cGAN**

اما در ادامه به بررسی هر لایه در Up sample و ابعادش میپردازیم:

Up sample layer	Size
Input size	2,2,1024
ConvTrans(512) + BN + Dropout + Relu	4,4,1024
ConvTrans(512) + BN+ Dropout + Relu	8,8,1024
ConvTrans(512)+ BN + Dropout + Relu	16,16,1024
ConvTrans(512) + BN + Relu	32,32,512
ConvTrans(256) + BN + Relu	64,64,256
ConvTrans(128)+ BN + Relu	128,128,128

جدول ۲ - **decoder** در **UNET**

و در نهایت در لایه خروجی:

Output layer	Size
ConvTrans(3) + tanh	256,256,3

جدول ۳ - لایه خروجی در **UNET**

بدیهی است که نحوه کارکرد این بخش، در ابتدا شامل یک down sample برای رسیدن به یک feature map است، سپس با up sample و skip connection بازسازی تصویر اولیه، به صورتی که ویژگی های سطح بالا از طریق skip connection و ویژگی های سطح پایین از طریق feature map ساخته شده بدست می آید، نویز نیز از طریق لایه های dropout به مدل تزریق میشود.

سوال چهارم – تابع هزینه نهایی مدل:

تابع هزینه نهایی مدل به شکل زیر است، که به معرفی هر بخش میپردازیم:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

شکل ۳۳ - تابع هزینه نهایی مدل

البته خوب است همین ابتدا ذکر کنم، که مقاله با استناد به تغییر مقاله اصلی GAN، سعی میکند Generator را نیز max کند که در ادامه توضیح آن را خواهیم داد:

- بررسی تابع L_cGAN:

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_y[\log D(y)] + \mathbb{E}_{x,z}[\log(1 - D(G(x, z))].$$

شکل ۳۴- تابع هزینه **cGAN**

این تابع شامل دو بخش است، در بخش اول، احتمال تشخیص real به ورودی های درست، بر اساس تابع D(y) به طور متوسط روی همه نمونه های آموزش Max میشود، این بخش در حقیقت احتمال اینکه تصاویر واقعی توسط Discriminator تشخیص داده شوند را max میکند.

در بخش دوم، G(x,z) خروجی های حاصل از Generator است و D(G(x,z)) احتمال این است که Discriminator این خروجی ها را به عنوان real تشخیص دهد، در نتیجه هدف ما این است که احتمال مکمل این مقدار یعنی 1 – D(G(x,z)) که یعنی Discriminator این تصاویر را fake تشخیص دهد کمینه شود.

اما خب از اصول احتمال میدانیم که رابطه زیر برقرار است و نتیجه بهتری نیز میدهد، زیر کمینه کردن احتمال توسط Generator یعنی Generator سعی کند، Discriminator را گول بزند، اما تلاش بیشتری نمیکند، یعنی به محض اینکه موفق شد Discriminator را گول بزند، دیگر تلاشی برای بهتر شدن نمیکند، و یادگیری اش کند میشود، اما max شدن، یعنی بهترین خود را ارائه کند، مثل این است که برای یک امتحان انقدری تلاش کنیم که از پس آن امتحان بربیاییم، و یا به قدری تلاش کنیم، که از پس هر امتحان دیگری نیز بربیاییم.

$$\min E\left[\log\left(1 - D\left(G(x,z)\right)\right)\right] = maxE\left[\log\left(D\left(G(x,z)\right)\right)\right]$$

بخش دوم تابع هزینه یک، فاصله L1 است:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1]$$

شکل ۳۵ - فاصله هندسی **L1**

این ترم تابع هزینه، در بخش Discriminator کار خاصی نمیکند، اما در بخش Generator سبب میشود که خروجی Generator به در هر Patch به خروجی های واقعی نزدیک باشد، در حقیقت از این نوع loss ها در توابع VAE برای بازتولید تصویر استفاده میشود، این ترم در حقیقت سبب میشود، که خروجی Blurry نباشد و نزدیک به مقدار اصلی تصویر باشد.

در مورد بروزرسانی وزن ها نیز به به ازای هر اپدیت در Generator یک آپدیت نیز در Discriminator انجام میشود اما، از سوی دیگر مقدار loss برای optimize کردن Discriminator بر ۲ تقسیم میشود، تا نرخ یادگیری Discriminator در مقایسه با Generator آهسته باشد.

سوال پنجم – در این بخش دو مقاله بهبود در PIX2PIX را بررسی میکنیم:

- مقاله اول:
- Enhancing Pix2pix for Image Dehazing

شبکه پیشرفته از بین بردن مه(haze) بر پایه معماری پیکسل به پیکسل (EPDN) روشی است که در یک مقاله تحقیقاتی پیشنهاد شده است تا عملکرد شبکه (Pix2pix GAN)را برای وظیفه از بین بردن مه در تصاویر بهبود بخشد.

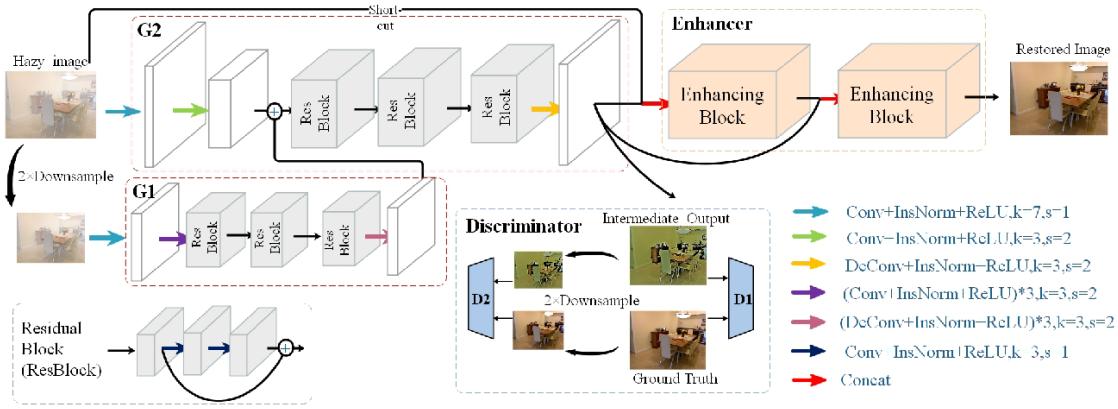
ایده اصلی مقاله استفاده از روش image translate در تسک کاهش مه نور در تصاویر است و بر اساس معماری و ترکیب توابع هزینه مختلف که در ادامه توضیح میدهیم، این روش را بهبود میبخشد.

نکات EPDN :

۱. **معماری مبتنی بر** EPDN: بر پایه چارچوب پیکسل به پیکسل ساخته شده است که شامل یک مولد و یک تشخیص دهنده است. مولد سعی می کند تصویری عاری از مه تولید کند، در حالی که تشخیص دهنده تلاش می کند تصویر تولید شده را از تصاویر واقعی عاری از مه تشخیص دهد.
۲. **ماژول تقویت کننده** :علاوه بر اجزایGAN ، EPDNیک ماژول تقویت کننده طراحی شده به خوبی را شامل می شود. تقویت کننده دارای دو "بلوک تقویت کننده" است که با استفاده از یک مدل میدان پذیرش، ویژگی های چند مقیاسی را استخراج می کند. این امر به تقویت اثر از بین بردن مه هم در رنگ و هم در جزئیات کمک می کند.
۳. **استخراج ویژگی های چند مقیاسی** :بلوک های تقویت کننده در تقویت کننده EPDN با نمونه برداری پایین ویژگی های ورودی، ویژگی ها را در چندین مقیاس استخراج می کنند. این امر به

دستگاه امکان می دهد اطلاعات جهانی و محلی را نیز ثبت کند، که منجر به نتایج بهتر از بین بردن مه در مقایسه با پیکسل به پیکسل اصلی می شود.

۴. **توابع هزینه تخصصی** EPDN :از ترکیبی از توابع هزینه از جمله هزینه رقابتی Adversarial loss، هزینه تطابق ویژگی.Feature matching loss ، هزینه ادراکیPerceptual loss و هزینه وفاداریFidelity loss استفاده می کند. این هزینه ها با هم کار می کنند تا مولد را هدایت کنند تا تصاویر واقع گرایانه و عاری از مه را با حفظ جزئیات مهم تولید کند.



شکل ۳۶ - شبکه **EPDN**

- مقاله دوم:

- **An Enhanced pix2pix Dehazing Network with Guided Filter Layer**

این مقاله یک شبکه پیشرفته پیکسل به پیکسل برای از بین بردن مه در تصاویر را پیشنهاد می دهد ایده این مقاله استفاده از یک لایه فیلتر هدایت شده برای بهبود عملکرد شبکه Pix2Pix در این تسک است.

برای مقایسه روش دیگری در مقایسه با مقاله اول، در این بخش نیز در همین زمینه dehazing مقاله ای انتخاب کردم.

ویژگی های اصلی:

۱. شبکه پیکسل به پیکسل بهبود یافته :این روش از یک شبکه پیکسل به پیکسل بهبود یافته استفاده می کند که بر اساس تابع زیان ادراکی طراحی شده است.

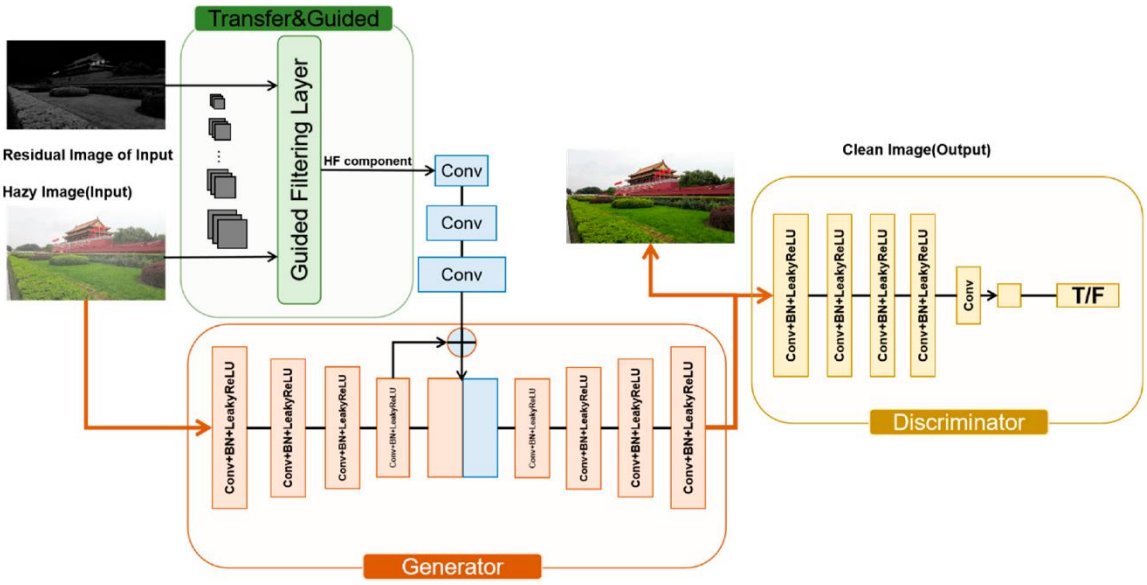
۲. لایه فیلتر هدایت شده :یک لایه فیلتر هدایت شده طراحی شده است که به طور موثر اطلاعات کانتور تصویر مه آلود را به دست می آورد و آن را با شبکه پیکسل به پیکسل بهبود یافته ترکیب می کند.

۳. حفظ جزئیات کلی :این روش یک خط لوله برای نگاشت اطلاعات کانتور به ویژگی های با ابعاد بالاتر ارائه می دهد که هدف آن محافظت از اطلاعات ویژگی های جزئیات کلی در برابر ویژگی های محلی است.

۴. عدم وابستگی به مدل پراکندگی فیزیکی :این شبکه تصاویر واضح را بدون اتکا به یک مدل پراکندگی فیزیکی تولید می کند.

اجزای اصلی:

- ماژول انتقال و هدایت :این ماژول شامل عملیات انتقال و فیلتر هدایت شده است که برای استخراج اجزای فرکانس بالا استفاده می شود.
- مولد :مولد شبکه از معماری U-Net استفاده می کند و شامل لایه های کانولوشن، نرمال سازی دسته ای و ReLU است.
- تشخیص دهنده :تشخیص دهنده از یک ساختار PatchGAN استفاده می کند.



شکل ۳۷ - تصویر شبکه

- مقاله سوم:

[High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs]

مقاله سوم را نیز که در صورت سوال ذکر شده را بررسی کردیم.

نکات اصلی:

- معرفی یک چارچوب استفاده ازGAN های شرطی برای تولید تصاویر با وضوح بالا و دستکاری معنایی.
- استفاده از یک شبکه مولد با ساختار شبکه باقیمانده و چندین مقیاس تفکیک‌کننده برای تولید تصاویر با کیفیت بالا.
- نمایش کاربردهایی مانند تولید تصاویر واقعی از نقشه‌های برچسب معنایی و ویرایش تصاویر با استفاده از اطلاعات معنایی.
- تاکید بر اهمیت حفظ همگنی مکانی و جزئیات بافت در تصاویر تولید شده.

مزایا:

- تولید تصاویر با وضوح بالا و جزئیات دقیق.
- موثر برای وظایفی که نیاز به خروجی‌های با کیفیت بالا دارند، مانند تولید تصاویر واقع‌گرایانه و ویرایش دقیق تصاویر.

معماری:

مقاله "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs" از یک شبکه مولد با ساختار شبکه باقیمانده (Residual Network) استفاده می‌کند. این شبکه شامل چندین بلوک باقیمانده (Residual Blocks) است که کمک می‌کنند جزئیات بیشتری حفظ شود. همچنین، از یک تفکیک‌کننده چندمقیاس (Multi-Scale Discriminator) بهره می‌برد که به بهبود کیفیت تصاویر تولید شده کمک می‌کند.

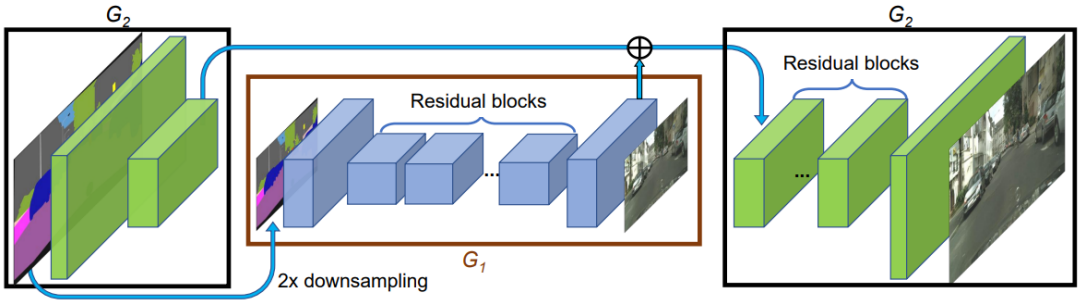
تابع هزینه:

تابع هزینه (Loss Function) ترکیبی از دو نوع هزینه است:

هزینه تقابلی (Adversarial Loss): این هزینه بر اساس تفاوت بین تصاویر واقعی و تصاویر تولید شده توسط شبکه مولد محاسبه می‌شود و هدف آن بهبود واقع‌گرایی تصاویر تولید شده است.

هزینه بازسازی (Reconstruction Loss): این هزینه بر اساس تفاوت بین تصاویر اصلی و تصاویر بازسازی شده توسط مولد محاسبه می‌شود و هدف آن حفظ شباهت و جزئیات در تصاویر بازسازی شده است.

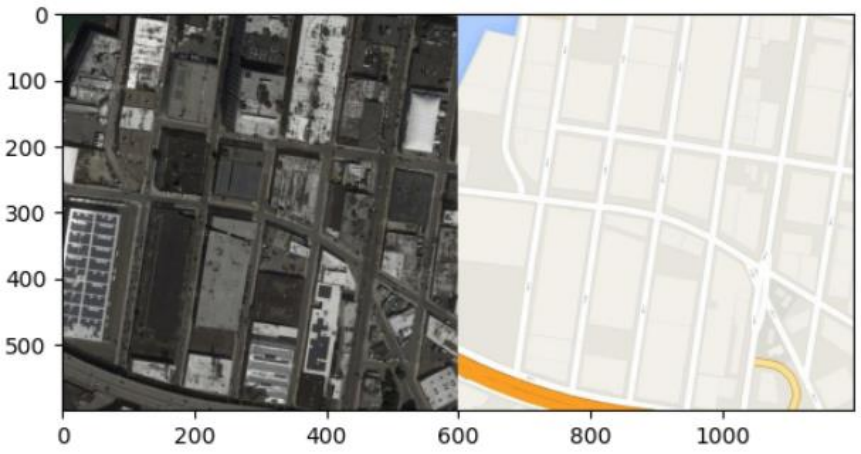
این ترکیب هزینه‌ها کمک می‌کند که شبکه مولد بتواند تصاویر با وضوح و جزئیات بالا تولید کند که هم از نظر بصری واقع‌گرایانه باشند و هم اطلاعات معنایی را حفظ کنند.



شکل ۳۸- معماری **Generator pix2pixHd**

۲-۲ – پیاده سازی معماری Pix2Pix:

۱ - در این بخش ابتدا به دانلود داده های مورد نظر پرداختیم، و آن ها در دو مجموعه آموزش و ارزیابی تقسیم بندی کردیم، همچنین یکی از این تصاویر را نیز به عنوان نمونه نمایش دادیم:



شکل ۳۹ - تصاویر خام در دیتاست

در ادامه تصاویر را برای استفاده در مدل طبق خواسته مقاله preprocess کردیم:

در قدم اول با توجه به اینکه هر تصویر و لیبل آن از عرض به هم چسبیده اند، می بایست تصاویر را از هم جدا میکردیم، که در شکل زیر میبینیم اینکار چطور انجام شده و در نهایت به تنسور تبدیل شده است:

```
def preprocess_images(image_dir, crop_image=True, random_flip=True):

    image = tf.io.read_file(image_dir)
    image = tf.io.decode_jpeg(image)

    w = tf.shape(image)[1]
    w = w // 2
    input_image = image[:, :w, :]
    real_image = image[:, w:, :]
    input_image = tf.cast(input_image, tf.float32)
    real_image = tf.cast(real_image, tf.float32)
```

شکل ۴۰ - جدا کردن تصویر و برچسب آن

سپس تصاویر تغییر اندازه داده شد به طوری که ابتدا تصاویر به اندازه ۲۸۶ در ۲۸۶ تغییر اندازه داده شدند و سپس از آن تصاویر یک تصویر ۲۵۶ در ۲۵۶ بریده(Crop) شد و بدست آمد، همچنین طبق خواسته مقاله با احتمال ۵۰ درصد، تصاویر را flip-left-right نیز کردیم، در نهایت با نرمال کردن تصاویر به بازه - ۱ و ۱، پیش پردازش این بخش به پایان رسید.

بخش مورد نظر برای انجام این کار را میتوانید در ادامه ببینید:

```
if crop_image:
    input_image = tf.image.resize(input_image, [286, 286],
                                  method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(real_image, [286, 286],
                                 method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    stacked_image = tf.stack([input_image, real_image], axis=0)
    cropped_image = tf.image.random_crop(stacked_image, size=[2, 256, 256, 3])
    input_image, real_image = cropped_image[0], cropped_image[1]
else:
    input_image = tf.image.resize(input_image, [256, 256],
                                  method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(real_image, [256, 256],
                                 method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

if random_flip:
    if tf.random.uniform(()) > 0.5:
        input_image = tf.image.flip_left_right(input_image)
        real_image = tf.image.flip_left_right(real_image)

input_image = (input_image / 127.5) - 1
real_image = (real_image / 127.5) - 1

return input_image, real_image
```

شکل ۴۱ - تابع تغییر اندازه و نرمال کردن تصاویر

سپس دیتاست های دریافت شده را با توابع موجود نگاشت کردیم، و دیتاست های نهایی برای مدل را ساختیم، در این قسمت قابل ذکر است که طبق تاکید مقاله تعداد batch size را برابر ۱ گرفتیم.

```
BUFFER_SIZE = 400
BATCH_SIZE = 1

train_dataset = tf.data.Dataset.list_files(train_list)
train_dataset = train_dataset.map(lambda x: preprocess_images(x, crop_image=True, random_flip=True),
                                  num_parallel_calls=tf.data.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

val_dataset = tf.data.Dataset.list_files(val_list)
val_dataset = val_dataset.map(lambda x: preprocess_images(x, crop_image=False, random_flip=False),
                                  num_parallel_calls=tf.data.AUTOTUNE)
val_dataset = val_dataset.batch(BATCH_SIZE)
```

شکل ۴۲ - ساخت دیتاست های لازم

در پایان این بخش، ۳ تصویر خروجی از دیتاست آموزش را برای اطمینان از درستی روند بررسی کردیم:

شکل ۴۳- تصاویر موجود در دیتاست مدل

۲ – در این بخش به پیاده سازی بخش های شبکه میپردازیم، شبکه متشکل از توابع Discriminator، Generator و همچنین Up sample و down sample می باشد:

- تابع down: گفتیم که این تابع در بخش Encode در معماری UNet کاربرد دارد، طبق توضیحات مقاله اولا وزن های همه بخش های مدل براساس یک توزیع گاوسی با مقادیر میانگین صفر و انحراف از معیار ۰.۰۲ مقدار دهی شده اند، همچنین در بخش Down sample، توابع فعالساز Leaky Relu بوده و به جز لایه اول در سایر بخش ها از Batch Norm استفاده شده است:

```
def down(inputs ,filters, batchnorm=True):

    init = tf.random_normal_initializer(0., 0.02)
    x = tf.keras.layers.Conv2D(filters, (4,4), strides=2, padding='same', kernel_initializer=init, use_bias=False)(inputs)

    if batchnorm:
        x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.LeakyReLU()(x)
    return x
```

شکل ۴۴ - تابع **down sample**

در بخش بعدی به پیاده سازی up sample میپردازیم، این تابع شامل Transpose Convolution است که اولا با استفاده از همان تابع گوسی گفته شده وزن های اولیه مقداردهی شده اند، همچنین در تمامی بخش ها دارا Batch Normalization بوده و همچنین تنها در سه لایه اول(پایین) در بخش Decoder مقادیر Dropout لحاظ شده اند.

```python
def up(inputs ,filters, dropout=False):

    init = tf.random_normal_initializer(0., 0.02)
    x = tf.keras.layers.Conv2DTranspose(filters, (4, 4), strides=2, padding='same', kernel_initializer=init, use_bias=False)(inputs)
    x = tf.keras.layers.BatchNormalization()(x)
    if dropout:
        x = tf.keras.layers.Dropout(0.5)(x)
    x = tf.keras.layers.ReLU()(x)
    return x
```

شکل ۴۵ - تابع **Up sample** در **Decoder** در معماری **UNet**

در نهایت از این توابع در ساخت Generator استفاده کردیم، دقت میکنیم که Generator علاوه بر توابع بالا شامل Concat شدن هر لایه i با لایه n-i است به طوری که n تعداد کل لایه هاست و شماره های i در Encoder و شماره های n-i در decoder هستند، همچنین یک لایه نهایی برای تولید خروجی در ۳ کلاس با فعالساز Tanh نیز وجود دارد:

```python
def Generator():

    inputs = tf.keras.layers.Input(shape=[256, 256, 3])
    init = tf.random_normal_initializer(0., 0.02)

    x1 = down(inputs, 64, batchnorm=False)
    x2 = down(x1, 128)
    x3 = down(x2, 256)
    x4 = down(x3, 512)
    x5 = down(x4, 512)
    x6 = down(x5, 512)
    x7 = down(x6, 512)
    x8 = down(x7, 512)

    x = up(x8, 512, dropout=True)
    x = tf.keras.layers.Concatenate()([x, x7])

    x = up(x, 512, dropout=True)
    x = tf.keras.layers.Concatenate()([x, x6])

    x = up(x, 512, dropout=True)
    x = tf.keras.layers.Concatenate()([x, x5])

    x = up(x, 512)
    x = tf.keras.layers.Concatenate()([x, x4])

    x = up(x, 256)
    x = tf.keras.layers.Concatenate()([x, x3])

    x = up(x, 128)
    x = tf.keras.layers.Concatenate()([x, x2])

    x = up(x, 64)
    x = tf.keras.layers.Concatenate()([x, x1])

    last = tf.keras.layers.Conv2DTranspose(3, 4, strides=2, padding='same', kernel_initializer=init, activation='tanh')(x)

    return tf.keras.Model(inputs=[inputs], outputs=[last])
```


شکل ۴۶ - شبکه **Generator**

در نهایت تعداد پارامتر های Generator به شکل زیر است:

```
Total params: 54,425,859 (207.62 MB)
Trainable params: 54,414,979 (207.58 MB)
Non-trainable params: 10,880 (42.50 KB)
```

شکل ۴۷ - پارامتر های **Generator**

سپس به تعریف کلاس Discriminator پرداختیم، این شبکه نیز با همان تابع گاوسی دارای وزن های اولیه میشود، این بخش نیز ورودی و خروجی مدل را به صورت Concat شده در ووردی میگیرد، و آن را بعد از سه مرحله استفاده از Convolution در نهایت با استفاده از یک لایه Conv 1 * 1، یک فیچرمپ براساس patch های در نظر گرفته شده میسازد و بر اساس آن کار جدا کنندگی را انجام میدهد، این تابع در ادامه مشخص است:

```python
def Discriminator():

    init = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')
    x = tf.keras.layers.concatenate([inp, tar])

    down1 = down(x, 64, False)
    down2 = down(down1, 128)
    down3 = down(down2, 256)

    layers = tf.keras.Sequential([
        tf.keras.layers.ZeroPadding2D(),
        tf.keras.layers.Conv2D(512, 4, strides=1, kernel_initializer=init, use_bias=False),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LeakyReLU(),
        tf.keras.layers.ZeroPadding2D(),
        tf.keras.layers.Conv2D(1, 4, strides=1, kernel_initializer=init)
    ])


    return tf.keras.Model(inputs=[inp, tar], outputs=layers(down3))
```

شکل ۴۸ - شبکه **Discriminator**

تعداد پارامتر های یادگیری Discriminator نیز در ادامه قابل مشاهده است:

```
Total params: 2,770,433 (10.57 MB)
Trainable params: 2,768,641 (10.56 MB)
Non-trainable params: 1,792 (7.00 KB)
```

شکل ۴۹ - پارامتر های شبکه **Discriminator**

سپس به ترکیب این دوشبکه در یک کلاس PIX2PIX میپردازیم که بتوانیم آن را به صورت End-to-END آموزش دهیم:

در این کلاس ابتدا دو شبکه و Optimizer و توابع هزینه هر کدام را دریافت میکنیم، و یک step آموزش را Overwrite میکنیم، ورودی ابتدا به Generator داده میشود، و خروجی آن نیز توسط Discriminator دریافت میشود، بعد از دریافت خروجی های مورد نظر محاسبه تابع هزینه، این تابع هزینه به ترتیب روی دو شبکه لحاظ شده و پارامتر های آن ها را بهبود میدهد:

```python
class Pix2Pix(tf.keras.Model):
    def __init__(self, generator, discriminator):
        super(Pix2Pix, self).__init__()
        self.generator = generator
        self.discriminator = discriminator

    def compile(self, generator_optimizer, discriminator_optimizer, gen_loss_fn, disc_loss_fn):
        super(Pix2Pix, self).compile()
        self.generator_optimizer = generator_optimizer
        self.discriminator_optimizer = discriminator_optimizer
        self.gen_loss_fn = gen_loss_fn
        self.disc_loss_fn = disc_loss_fn

    def train_step(self, data):
        real_images, target = data

        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_images = self.generator(real_images, training=True)

            real_output = self.discriminator([real_images, target], training=True)
            fake_output = self.discriminator([real_images, fake_images], training=True)

            gen_total_loss, gen_gan_loss, gen_l1_loss = self.gen_loss_fn(fake_output, fake_images, target)
            disc_loss = self.disc_loss_fn(real_output, fake_output)

        generator_gradients = gen_tape.gradient(gen_total_loss, self.generator.trainable_variables)
        discriminator_gradients = disc_tape.gradient(disc_loss, self.discriminator.trainable_variables)

        self.generator_optimizer.apply_gradients(zip(generator_gradients, self.generator.trainable_variables))
        self.discriminator_optimizer.apply_gradients(zip(discriminator_gradients, self.discriminator.trainable_variables))

        return {"gen_loss": gen_total_loss, "disc_loss": disc_loss}
```

شکل ۵۰ - ساختار نهایی شبکه **PIX2PIX**

تعداد کل پارامتر های شبکه:

```
Total params: 57,196,292 (218.19 MB)
Trainable params: 57,183,620 (218.14 MB)
Non-trainable params: 12,672 (49.50 KB)
```

شکل ۵۱ - تعداد کل پارامتر های شبکه

پیاده سازی و آموزش مدل در نقشه های شهری:

۴ – در این بخش به پیاده سازی توابع هزینه لازم میپردازیم و مدل را آموزش میدهیم:

همانطور که قبلا بحث شد، شبکه Generator شامل یک تابع هزینه است که سعی میکند خروجی هایش همگی به عنوان Real دسته بندی شوند و یک L1-loss نیز در کنارش هست، ترکیب این تابع هزینه با وزن ها LAMBDA انجام میشود که مقدار آن ۱۰۰ پیشنهاد شده است

```
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def generator_loss(disc_generated_output, gen_output, target, LAMBDA = 100):

    gan_loss = loss_fn(tf.ones_like(disc_generated_output), disc_generated_output)
    l1_loss = tf.reduce_mean(tf.abs(target - gen_output))
    total_gen_loss = gan_loss + (LAMBDA * l1_loss)
    return total_gen_loss, gan_loss, l1_loss
```

شکل ۵۲ - تابع هزینه **Generator**

تابع هزینه Discriminator اما، سعی میکند اولا خروجی های واقعی را به یک و خروجی های Generator را به صفر نزدیک کند و مجموع آن را به عنوان هزینه حساب میکند:

```
def discriminator_loss(disc_real_output, disc_generated_output):

    real_loss = loss_fn(tf.ones_like(disc_real_output), disc_real_output)
    generated_loss = loss_fn(tf.zeros_like(disc_generated_output), disc_generated_output)
    total_disc_loss = real_loss + generated_loss
    return total_disc_loss
```

شکل ۵۳ - تابع هزینه **discriminator**

آپتیمایزر هر دو مدل نیز به شکل زیر انتخاب شد:

```
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5, beta_2=0.999)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5, beta_2=0.999)
```

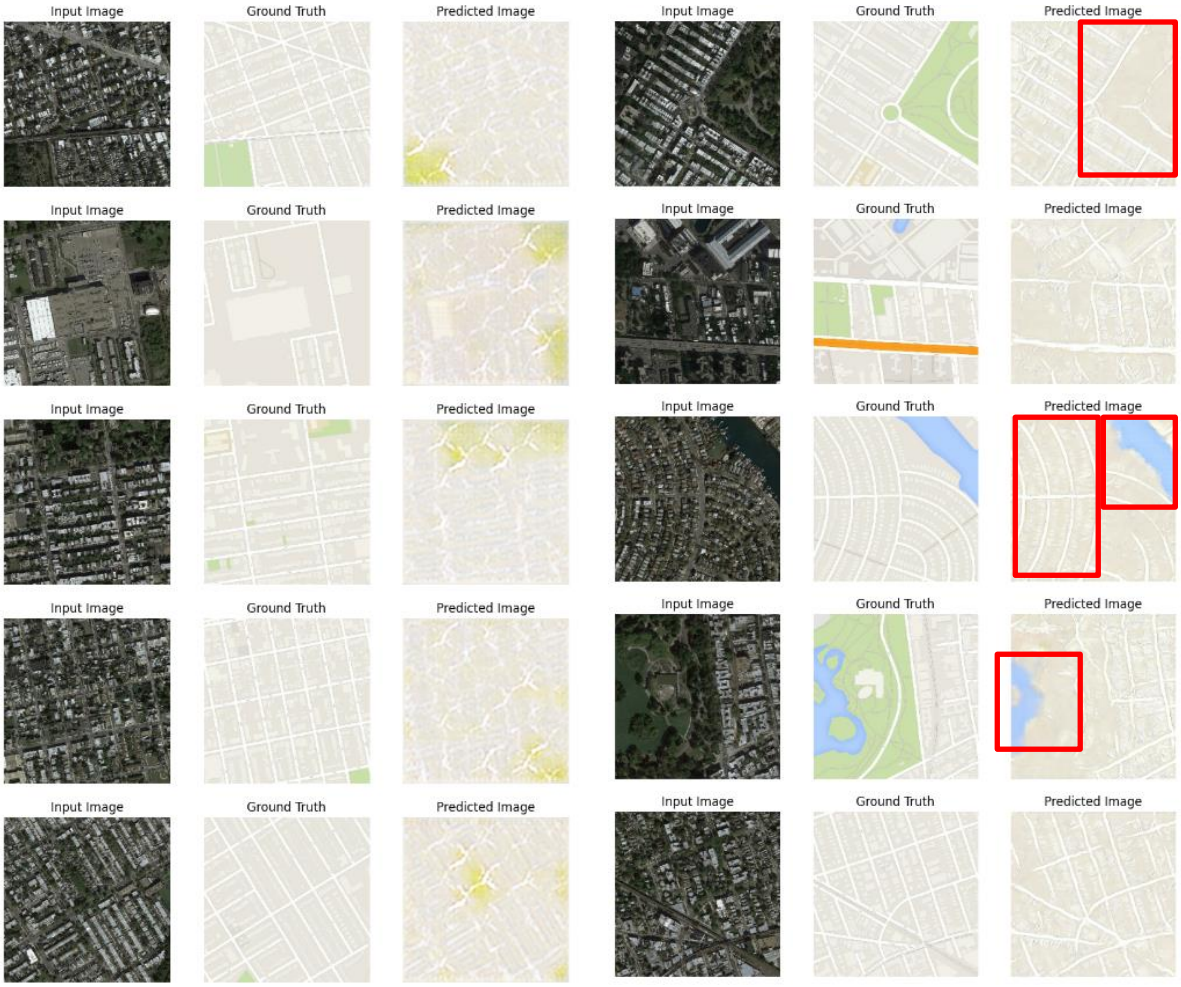
شکل ۵۴ - آپتیامایزر های دو شبکه

همچنین در این بخش برای نمایش ۵ تصویر به صورت رندوم حین فرایند آموزش، یک Callback برای مانیتور کردن روند آموزش نوشتیم به طوری که در انتهای هر ایپاک، ۵ تصویر از مجموعه ارزیابی گرفته، و آن ها را به Generator میدهد، سپس تصویر اصلی، prediction مدل و target را نیز نمایش میدهد:

```python
class Pix2PixMonitor(tf.keras.callbacks.Callback):

    def __init__(self, num_img=5):
        self.num_img = num_img

    def on_epoch_end(self, epoch, logs=None):

        for img, tar in val_dataset.take(self.num_img):

            prediction = self.model.generator(img, training=True)

            display_list = [img[0], tar[0], prediction[0]]
            title = ['Input Image', 'Ground Truth', 'Predicted Image']

            plt.figure(figsize=(10, 3))
            for i in range(3):
                plt.subplot(1, 3, i+1)
                plt.title(title[i])
                plt.imshow(display_list[i] * 0.5 + 0.5)
                plt.axis('off')
            plt.show()
```

شکل ۵۵ - مانیتور کردن
آموزش در هر ایپاک

سپس فرآیند اموزش برای ۲۰
ایپاک شروع شد، که تصاویر حدس شده توسط مدل در اولین و آخرین ایپاک به شکل زیر بود:

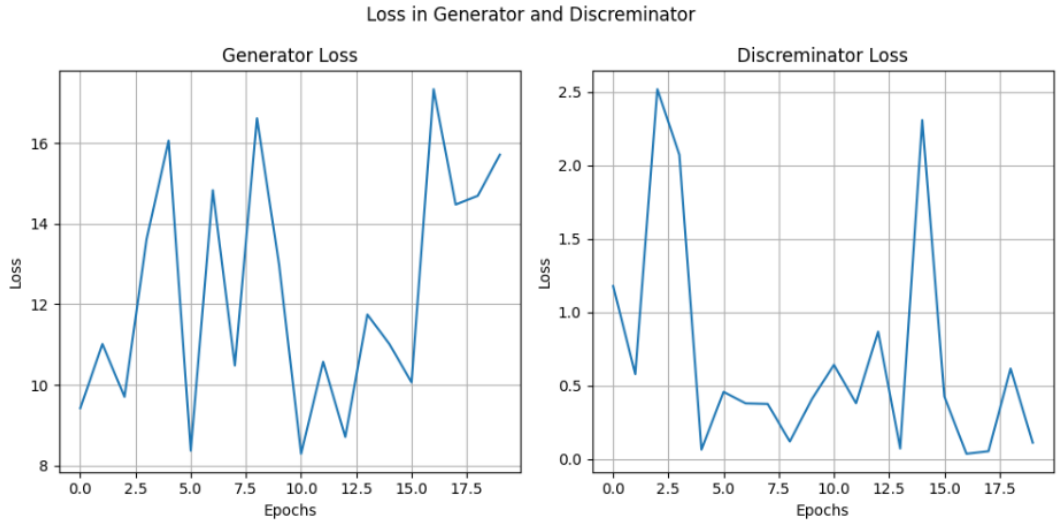


شکل ۵۶ - تصاویر ایپاک ۲۰

شکل ۵۷ - تصاویر ایپاک اول

ت بسیار واضح است، میبینیم که در تصاویر اولیه، تصاویر کاملا مبهم است، و احتمالا تنها به دلیل L1- loss میتوانیم یک Texture کلی شبیه به خروجی تولید کنیم، اما در ایپاک ۲۰ میبینم که تصاویر پس زمینه که بسیار شفاف شده اند، بخش آبی نیز در موارد زیاد خوب تشخیص داده شده اند و ساختار کلی نقشه بدسه آمده است، سوال این است که چرا نتایج هنوز به خوبی مقاله نیست؟ واضح است، مقاله برای ۲۰۰ ایپاک آموزش داده شده است، یعنی ۱۰ برابر بیشتر از این چیزی که ما به آن رسیده ایم، در تصاویر بالا با کادر های قرمز، تشخیص رنگ یا خطوط منحنی نقشه را بولد کرده ام، این نشان از یادگیری خوب در مدل دارد.

در نهایت توابع هزینه را نیز نمایش دادیم:

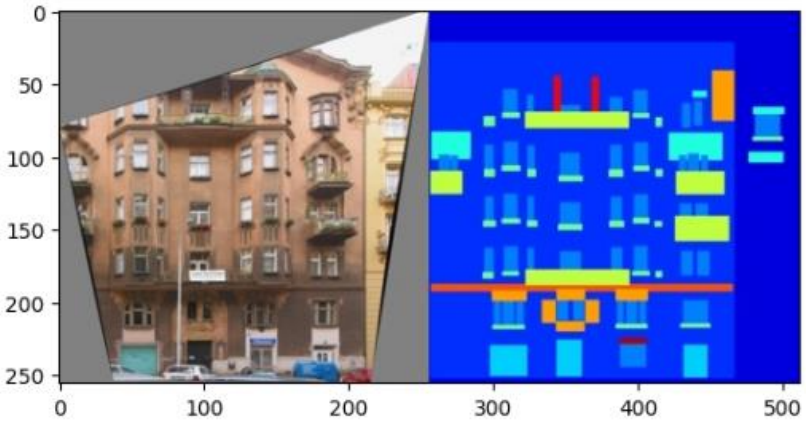


شکل ۵۸ - توابع هزینه در مدل **PIX2PIX**

در تصویر مشخص است که کاهش هزینه در Discriminator اتفاق افتاده است، تابع Discriminator بر اساس تسک ساده تر و تعداد پارامتر های بسیار کمتری که دارد، سریعتر همگرا شده است اما، در Generator نوسان شدیدی مشاهده میشود، این امر به این دلیل است که معمولا این مدل ها طی ایپاک های بیشتر از ۱۰۰ به تعادل میرسند، از طرفی، به دلیل استفاده از Dropout با نرخ ۵۰ درصد در سه لایه از Generator انتظار رفتار نوسانی و در تابع هزینه میرود.

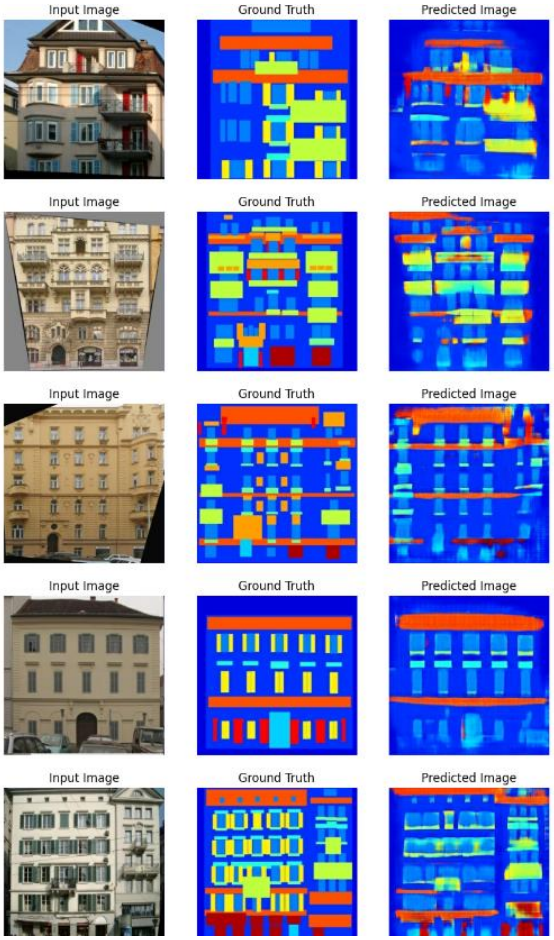
پیاده سازی و آموزش مدل در نمای ساختمان:

در این بخش نیز ابتدا بعد از دانلود دیتاست مورد نظر، یکی از تصاویر را بررسی کردیم:

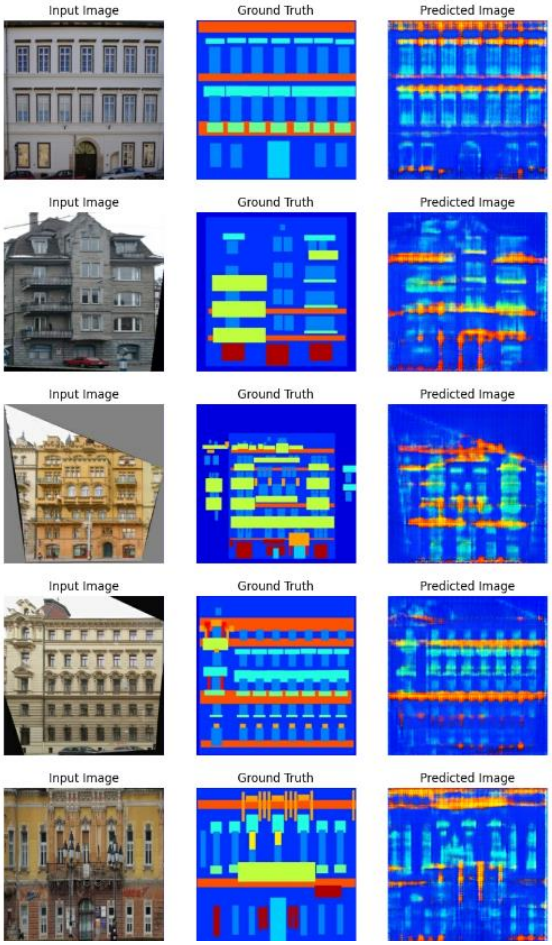


شکل ۵۹ - تصاویر نمونه از دیتاست

سپس بدون تغییر پارامتر به آموزش مدل به اندازه ۲۰ ایپاک پرداختیم، تصاویر خروجی ایپاک اول و آخر را مشاهده میکنید:



شکل ۶۰ - تصاویر ایپاک ۲۰

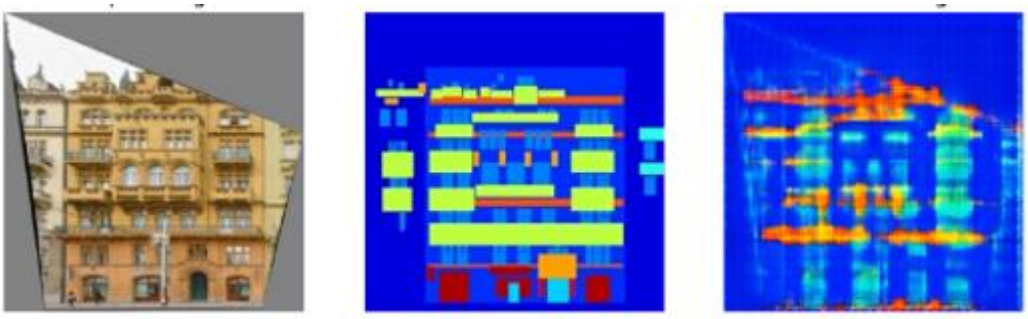


شکل ۶۱ - تصاویر نخستین ایپاک

در این بخش نیز میبینم که تصاویر بسیار بهتر شده اند، یادگیری رنگ ها، جانمایی آن ها، و در هم فرونرفتن رنگ ها بسیار مهم است، مثلا در مورد بهم ریختگی در رنگ ها، این تصویر نمونه بسیار خوبی است:

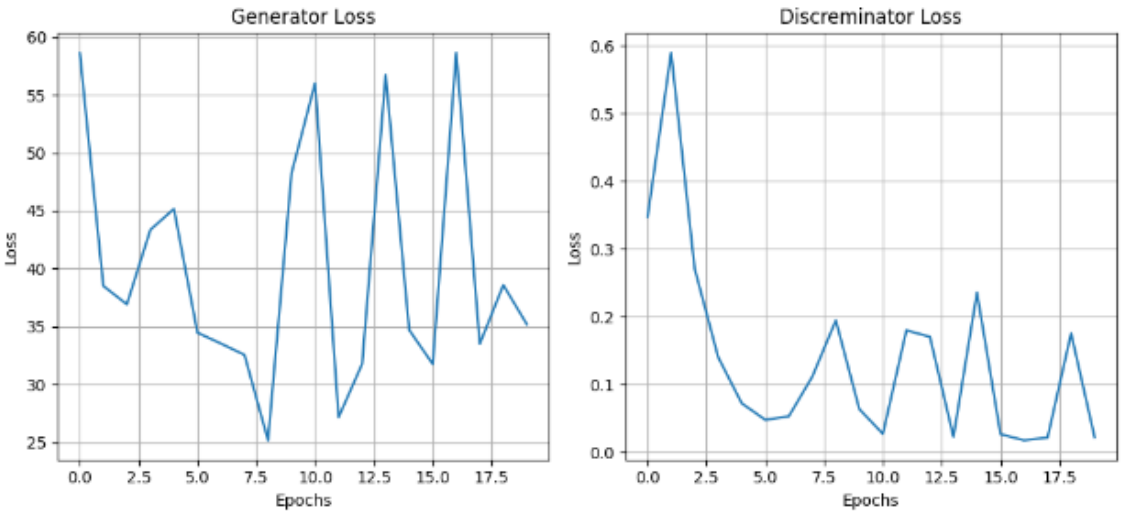
شکل ۶۲ - تصویر نمونه جداشدن رنگ ها

در حالی که در تصاویر اولیه، رنگ ها مرز های شفافی ندارند:



شکل ۶۳ - تصویر با مرز های بهم ریخته

در نهایت به رسم نمودار توابع هزینه این بخش پرداختیم:



شکل ۶۴ - توابع هزینه در دیتاست **facades**

این نمودار نیز دقیقا مانند قبل است، بدیهی است که تابع Discriminator بر اساس تسک ساده تر و تعداد پارامتر های بسیار کمتری که دارد، سریعتر همگرا شده است، اما در مورد Generator این موضوع هنوز اتفاق نیفتاده است و نیاز است تا آموزش بیشتر از این ادامه پیدا کند، از سوی دیگر، سه لایه Dropout با نرخ ۵۰ درصد، سبب ایجاد نوسان های بسیار جدی در loss میشود که آن را در این قسمت نیز شاهد هستیم.