

به نام خدا



دانشگاه تهران



دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	مهرسا همتپناه – محمد هادی بابالو
شماره دانشجویی	۸۱۰۱۹۹۳۸۰ - ۸۱۰۱۹۹۵۸۴
تاریخ ارسال گزارش	۱۴۰۲، ۱۱، ۱۰

## فهرست

۵	پاسخ ۱. Control VAE
۵	۱-۱. مقدمه
۵	۲-۱. پیادهسازی VAE
۸	۳-۱. ارزیابی مدل VAE
۱۰	۴-۱. پیادهسازی Control VAE
۱۵	<b>پاسخ ۲ Generative Adversarial Networks (GANs) – ۲</b>
۱۵	۱-۲. آموزش مدل GAN بر روی دیتاست MNIST
۱۸	۱-۱-۲. پیادهسازی
۱۸	۲-۱-۲. PixelShuffle
۱۸	۳-۱-۲. هایپرپارامترهای مدل
۱۹	۴-۱-۲. ارزیابی مدل
۲۴	۲-۲. مدل Wasserstein GAN
۲۴	۲-۲-۱. بهبودهای مدل
۲۵	۲-۲-۲. ارزیابی مدل
۲۹	۳-۲. مدل Self-Supervised GAN
۲۹	۱-۳-۲. Generator
۳۱	۲-۳-۲. Discriminator
۳۵	۳-۳-۲. ارزیابی مدل

## شکل‌ها

- ۶ ..... شکل ۱ - نمونه داده‌های موجود در دیتاست dSprites
- ۶ ..... شکل ۲ - معماری بخش encoder
- ۷ ..... شکل ۳ - معماری بخش decoder
- ۷ ..... شکل ۴ - جریان داده در VAE
- ۸ ..... شکل ۵ - نمودار loss برای داده‌های آموزش
- ۸ ..... شکل ۶ - نمودار loss برای داده‌های ارزیابی
- ۹ ..... شکل ۷ - تعدادی از تصاویر تولید شده توسط مدل
- ۹ ..... شکل ۸ - فرمول محاسبه FID
- ۱۰ ..... شکل ۹ - استفاده از pytorch-fid برای محاسبه FID
- ۱۰ ..... شکل ۱۰ - الگوریتم PI
- ۱۱ ..... شکل ۱۱ - بخش Control VAE مدل PI Controller
- ۱۱ ..... شکل ۱۲ - نمودار loss برای CVAE با setpoint=8 روی داده آموزش
- ۱۲ ..... شکل ۱۳ - نمودار loss برای CVAE با setpoint=8 روی داده ارزیابی
- ۱۲ ..... شکل ۱۴ - معیار FID برای CVAE با setpoint=8
- ۱۲ ..... شکل ۱۵ - نمودار loss برای CVAE با setpoint=14 روی داده آموزش
- ۱۳ ..... شکل ۱۶ - نمودار loss برای CVAE با setpoint=14 روی داده تست
- ۱۳ ..... شکل ۱۷ - نمودار FID برای CVAE با setpoint=14
- ۱۳ ..... شکل ۱۸ - نمودار KL loss برای دو
- ۱۴ ..... شکل ۱۹ - نمودار Reconstruction loss برای دو
- ۱۵ ..... شکل ۲۰ - کد تابع خطای اشباع ناپذیر
- ۱۵ ..... شکل ۲۱ - کد تابع تولید نویز
- ۱۷ ..... شکل ۲۲ - کد معماری generator مدل GAN
- ۱۸ ..... شکل ۲۳ - کد معماری discriminator مدل GAN
- ۲۰ ..... شکل ۲۴ - نمودار تغییرات loss بخش generator و discriminator
- ۲۰ ..... شکل ۲۵ - محاسبه معیار FID
- ۲۱ ..... شکل ۲۶ - تصاویر تولید شده مدل در ابتدای آموزش
- ۲۲ ..... شکل ۲۷ - تصاویر تولید شده در میانه آموزش مدل

شکل ۲۸ - تصاویر تولید شده در انتهای فرایند آموزش توسط مدل	۲۳
شکل ۲۹ - معماری بخش critic مدل WGAN	۲۴
شکل ۳۰ - تابع loss مربوط به WGAN	۲۴
شکل ۳۱ - نمودار تغییرات loss بخش generator و discriminator	۲۵
شکل ۳۲ - محاسبه معیار FID	۲۶
شکل ۳۳ - تصاویر تولید شده در ابتدای فرایند آموزش مدل	۲۶
شکل ۳۴ - تصاویر تولید شده در میانه فرایند آموزش توسط مدل	۲۷
شکل ۳۵ - تصاویر تولید شده نهایی پس از اتمام فرایند آموزش مدل	۲۸
شکل ۳۶ - کد generator مدل SSGAN	۳۰
شکل ۳۷ - کد بلاک generator در residual SSGAN	۳۱
شکل ۳۸ - کد discriminator مدل SSGAN	۳۲
شکل ۳۹ - کد بلاک residual در discriminator SSGAN	۳۳
شکل ۴۰ - کد SSGAN loss	۳۴
شکل ۴۱ - کد rotation عکس ها برای مدل SSGAN	۳۵
شکل ۴۲ - نمودار تغییرات loss بخش generator و discriminator مدل SSGAN	۳۶

## جدول ها

- ۱۶ ..... جدول ۱ - معماری generator مدل GAN
- ۱۷ ..... جدول ۲ - معماری discriminator مدل GAN
- ۱۹ ..... جدول ۳ - پارامترهای مدل
- ۲۹ ..... جدول ۴ - معماری generator مدل SSGAN
- ۳۰ ..... جدول ۵ - بلاک generator residual در SSGAN
- ۳۱ ..... جدول ۶ - معماری discriminator مدل SSGAN
- ۳۳ ..... جدول ۷ - بلاک discriminator residual در SSGAN
- ۳۵ ..... جدول ۸ - پارامترهای مدل SSGAN

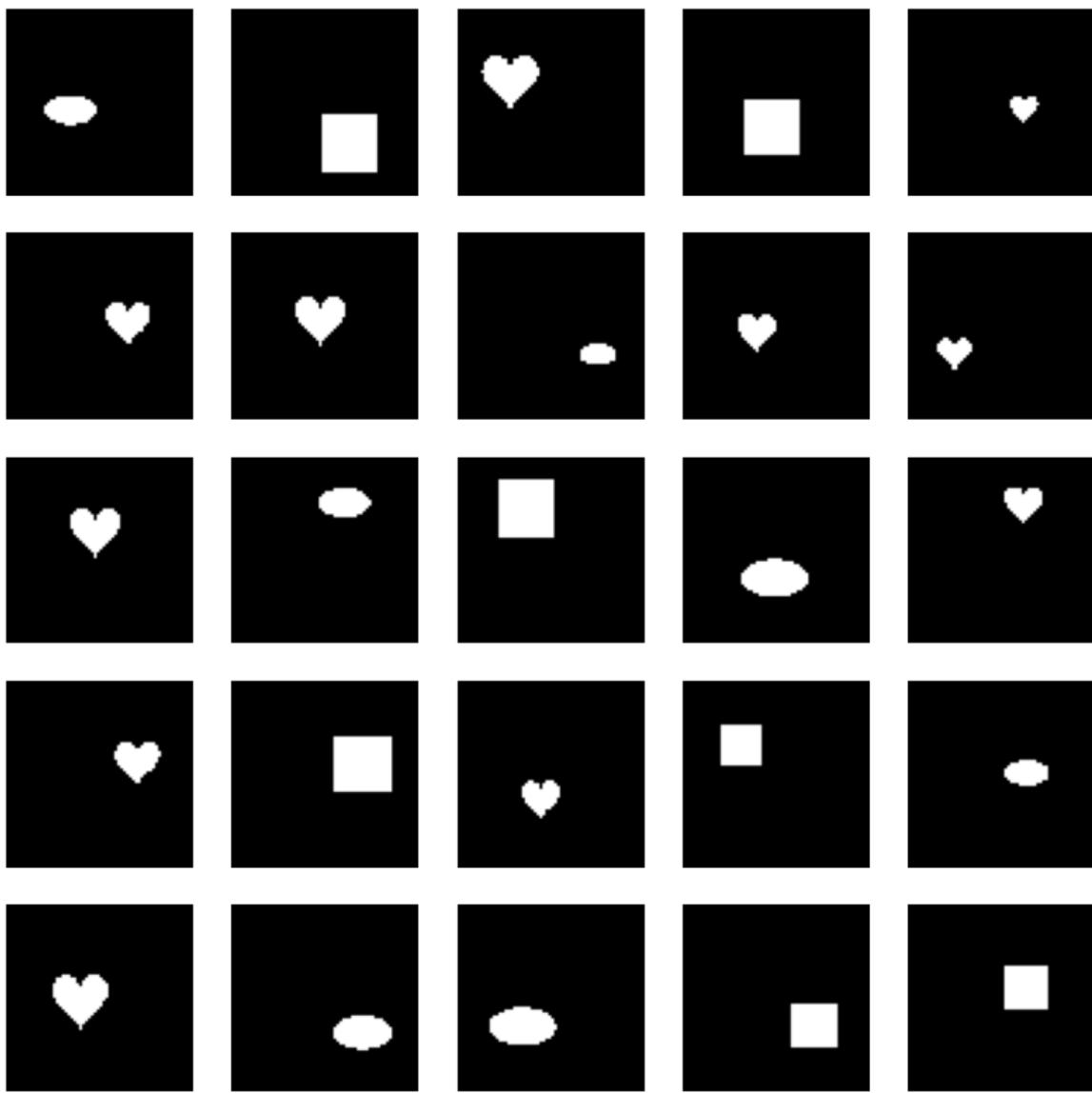
## پاسخ ۱. Control VAE

### ۱-۱. مقدمه

ساختار VAE ها شامل دو بخش کلی است. یک بخش encoder که ورودی مدل (در این تسك خاص عکس) را گرفته و به طور مثال در این معماری خاص با استفاده از لایه‌های convolutional ویژگی‌های مهم را از آن استخراج می‌کند. در واقع بخش encoder تلاش می‌کند که ورودی مدل را تبدیل به نمایش با ابعاد کمتر در فضای latent کند. این فضای latent در این معماری شامل دو بردار ویژگی با ابعاد dimension است که یکی شامل میانگین‌های توزیع آماری نرمال و همچنین انحراف از معیار آموزش دیده شده توسط مدل است. سپس در بخش میانی یک reparametrization انجام می‌شود و مدل از تابع توزیع latent space ساخته شده در latent space نمونه‌گیری انجام می‌دهد. سپس این sample های گرفته شده از latent space به بخش decoder داده می‌شود که دقیقاً عملکردی برعکس encoder ما دارد و قرار است که تصاویر خروجی generate شده توسط مدل را از روی بردار تولید شده از latent space تولید کند. در VAE ما متغیرهای کنترلی افزون بر مدل را داریم که به مدل اضافه شده‌اند و در هر دو بخش encoder و decoder به مدل داده می‌شوند. یعنی این متغیرهای کنترلی هم به عنوان ورودی و برای دادن اطلاعات بیشتر به مدل در بخش encoder استفاده می‌شوند و هم در بخش دادن سمپل‌ها به بخش decoder این اتفاق می‌افتد و متغیرهای کنترلی به ورودی decoder افزوده می‌شوند. افزودن این متغیرهای کنترلی باعث می‌شود که مدل هدایت شود که عکس‌های با ویژگی‌های مد نظر ما را تولید کند. در latent space ما به جهتی هدایت شود که loss regularizer term در محاسبه loss نهایی هستیم که مدل را تشویق می‌کند تا طوری رفتار کند که تغییر در control code باعث تغییر در ویژگی‌های تصاویر تولیدی مدل شود. از طرفی یک diversity loss term هم داریم که باعث می‌شود که مدل حتی در صورت دریافت یک control code خاص، باز هم تنوع تصاویر تولیدی خود را حفظ کرده و حتی بهبود ببخشد و از حفظ کردن ویژگی‌های مربوط به یک control code خاص توسط مدل جلوگیری کند. این افزودن متغیرهای کنترلی می‌تواند حتی بر روی ویژگی‌های سطح بالایی مثل سن، جنسیت، داشتن یا نداشتن عینک و ... با استفاده از attribute vector شود.

### ۲-۱. پیاده‌سازی VAE

در این بخش به پیاده‌سازی مدل VAE ارائه شده در مقاله مذکور می‌پردازیم.



شکل ۱ - نمونه داده‌های موجود در دیتاست **dSprites**

بخش encoder این شبکه به صورت زیر است.

```

self.encoder = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=32, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=32, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=256, kernel_size=4, stride=1),
    nn.ReLU(),
)
self.fc_encoder = nn.Linear(256, 256)
self.fc_mu = nn.Linear(256, latent_dim)
self.fc_logvar = nn.Linear(256, latent_dim)

```

شکل ۲ - معماری بخش encoder

همانطور که می‌بینید، در انتهای بخش encoder ما شاهد یک لایه fully connected با `out_features=256` با هستیم که ویژگی‌های استخراج شده توسط لایه‌های convolutional در VAE را دریافت می‌کند. سپس این لایه به دو لایه fully connected دیگر که هر کدام ابعادی برابر با latent space dimension ما دارند متصل می‌شود که یکی از این دو لایه برای محاسبه میانگین‌های توزیع آماری و دیگری برای محاسبه انحراف از معیارهای توزیع آماری ما استفاده می‌شود.

به طور متناظر برای بخش decoder هم داریم:

```
self.fc_mu = nn.Linear(256, latent_dim)
self.fc_logvar = nn.Linear(256, latent_dim)
self.fc_decoder = nn.Linear(latent_dim, 256)
self.decoder = nn.Sequential(
    nn.ConvTranspose2d(in_channels=256, out_channels=64, kernel_size=4),
    nn.ReLU(),
    nn.ConvTranspose2d(in_channels=64, out_channels=64, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.ConvTranspose2d(in_channels=32, out_channels=32, kernel_size=4, stride=2, padding=1),
    nn.ReLU(),
    nn.ConvTranspose2d(in_channels=32, out_channels=1, kernel_size=4, stride=2, padding=1),
)

```

شكل ۳ - معماری بخش **decoder**

همانطور که در شکل زیر هم می‌بینید، پس از گذشتن داده‌های ورودی ما از بخش encoder و تبدیل شدن به ویژگی‌ها در latent space عمل نمونه‌گیری و reparameterization انجام می‌شود و نتیجه این بخش reparameterization داده شده تا تصاویر مدل نظر ما توسط مدل تولید شوند. در بخش decoder به دست آمده‌اند برای سmpl گرفتن استفاده می‌شود از میانگین و انحراف از معیارهایی که در latent space

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + std * eps

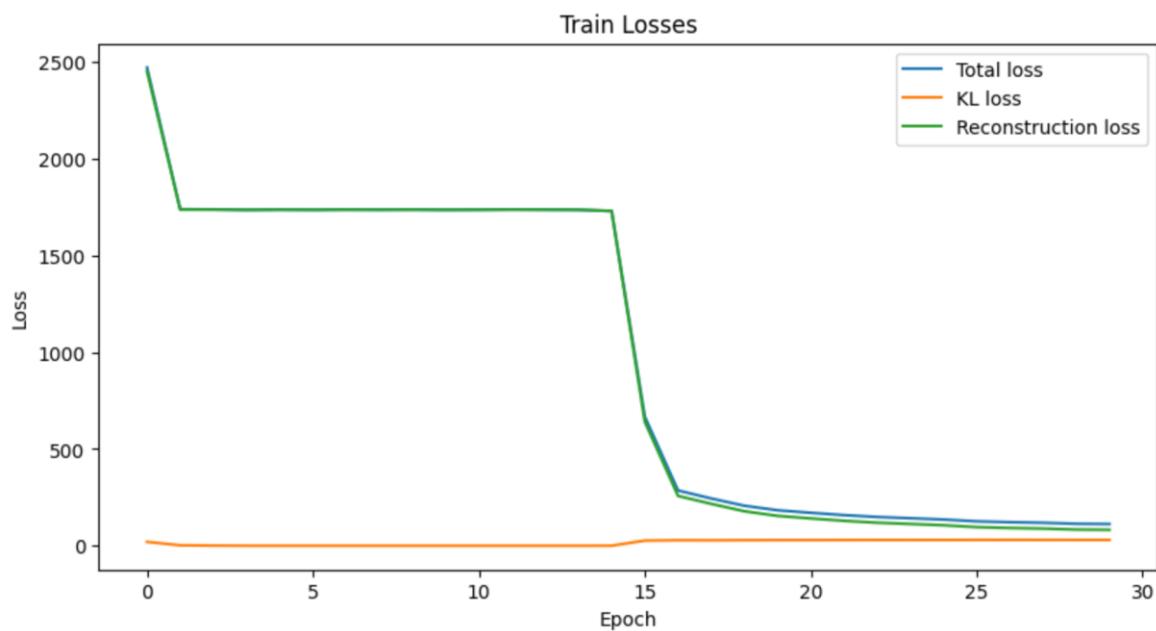
def encode(self, x):
    h = self.encoder(x)
    h = h.view(-1, 256)
    h = self.fc_encoder(h)
    mu = self.fc_mu(h)
    logvar = self.fc_logvar(h)
    z = self.reparameterize(mu, logvar)
    return z, mu, logvar

def decode(self, z):
    h = self.fc_decoder(z)
    h = h.view(-1, 256, 1, 1)
    x = self.decoder(h)
    x = self.sigmoid(x)
    return x
```

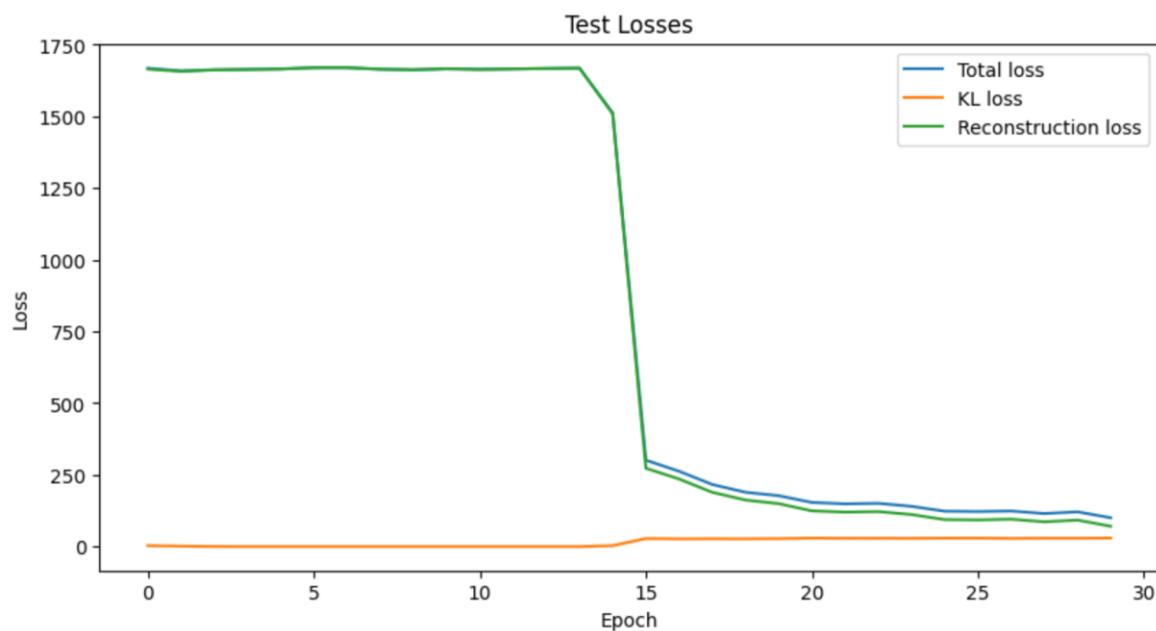
شكل ۴ - جریان داده در **VAE**

### ۱-۳. ارزیابی مدل VAE

در ابتدا نمودارهای loss خواسته شده را نمایش می‌دهیم.



شکل ۵ - نمودار loss برای داده‌های آموزش

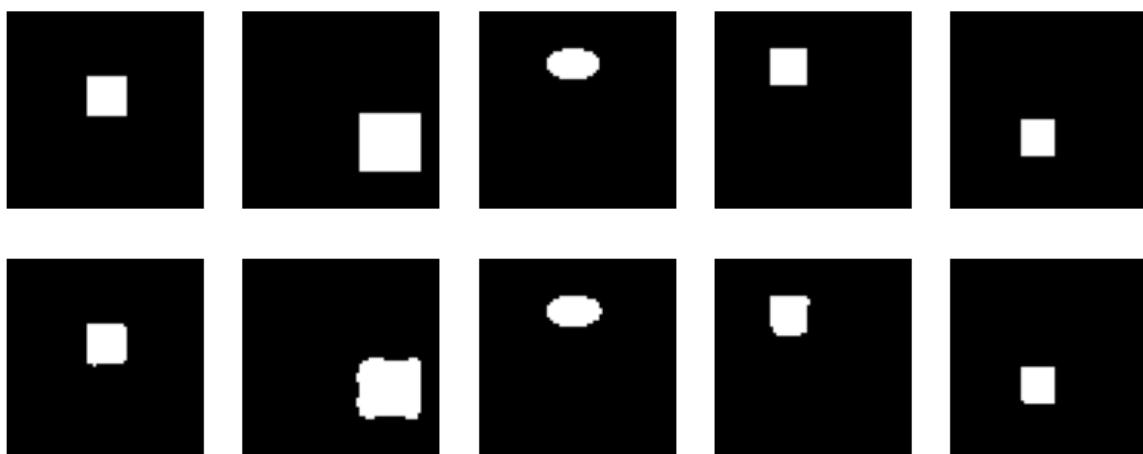


شکل ۶ - نمودار loss برای داده‌های ارزیابی

دو نکته در نمودارهای loss ما دیده می‌شود. اول اینکه در epoch های اولیه، مدل در یک local optimum افتاده است و نمی‌تواند به خوبی آموزش ببیند و loss خود را کاهش دهد. البته این مشکل با استفاده از تعریف مناسب loss function و وزن دار کردن آن و همچنین استفاده از یک optimizer مناسب حل شده

است و مدل توانسته که پس از چند epoch از این local optimum بیرون بیاید و فرایند آموزش خود را به خوبی ادامه دهد. نکته دوم این است که در epoch پانزدهم ما شاهد این هستیم که مقدار KL loss ما افزایش یافته است که این موضوع هم منطقی است. زیرا که هدف آموزش مدل کاهش loss مجموع است که در اینجا محقق شده است. در واقع درست است که KL loss ما شاهد افزایش کمی بوده است، ولی این افزایش همراه با کاهش Reconstruction loss ما بوده است که در مجموع باعث بهبود عملکرد مدل و آموزش آن شده است.

در شکل زیر می‌توانید تعدادی از تصاویر تولید شده توسط مدل را مشاهده کنید.



شکل ۷ - تعدادی از تصاویر تولید شده توسط مدل

معیار Fréchet Inception Distance (FID) یک روش محبوب برای اندازه‌گیری کیفیت و تنوع عکس‌های تولید شده توسط مدل‌های Generative است. این روش بر اساس Fréchet distance که معیاری برای اندازه‌گیری عدم شباهت بین دو توزیع آماری در feature space است عمل می‌کند. در این روش ابتدا از یک شبکه عصبی از قبل آموزش داده شده (که معمولاً یک CNN مثل inception است) استفاده می‌شود تا کار feature extraction را برای ما انجام دهد. سپس مولفه‌های آماری (مثل میانگین و کواریانس و این می‌شوند. همچنین در این روش فرض می‌شود که این feature representation ها از یک تابع توزیع چند متغیره پیروی می‌کنند. پس از محاسبه این مولفه‌های آماری، مقدار FID از فرمول زیر به دست می‌آید.

$$FID = \|\mu_r - \mu_g\|^2 + T_r(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2})$$

شکل ۸ - فرمول محاسبه **FID**

برای محاسبه FID در این پروژه از کتابخانه pytorch-fid استفاده شد.

```

fid_score = fid_score.calculate_fid_given_paths([ORIGINAL_DIR, GENERATED_DIR], batch_size=50, device=device, dims=2048)
print(fid_score)

100%|██████████| 100/100 [00:20<00:00,  4.99it/s]
100%|██████████| 100/100 [00:20<00:00,  4.83it/s]
15.552915793093746

```

شکل ۹ - استفاده از **pytorch-fid** برای محاسبه FID

همانطور که دیده می‌شود، این مدل توانسته به FID برابر با ۵۵.۱۵ در مقایسه تصاویر تولید شده با تصاویر ورودی برسد که مقدار قابل قبولی برای یک VAE روی این دیتاست به شمار می‌رود.

#### ۴-۴. پیاده‌سازی Control VAE

ماژول Control VAE که در Proportional-Integral (PI) Controller وجود دارد، توانایی کنترل کردن ویژگی‌ها و attribute های موجود در عکس‌های تولید شده توسط مدل را کنترل می‌کند. این ماژول یک control code مثل  $c$  را به عنوان ورودی دریافت می‌کند که درواقع ویژگی‌هایی که می‌خواهیم VAE ما تولید کند را encode می‌کند. این ماژول دو پارامتر  $kp$  و  $ki$  دارد که به ترتیب نشان‌دهنده proportional و integral gain هستند. همچنین نتیجه نهایی و خروجی نهایی این ماژول به صورت  $PI\ Output = kp \times c + ki \times \Sigma c$  محاسبه می‌شود. این خروجی به صورت element-wise posterior به میانگین توزیع  $kp \times c + ki \times \Sigma c$  افزوده می‌شود و سپس به decoder داده می‌شود. این موضوع باعث می‌شود که توزیع posterior ما به سمتی هدایت شود که تعیین می‌کند و در نتیجه عکس‌های تولیدی به سمت داشتن ویژگی‌های خواسته شده و مطلوب ما حرکت کنند. همچنین نیاز داریم که control loss را محاسبه کنیم و نماینده تفاوت بین latent representation فعلی و هدف ما باشد.

---

##### Algorithm 1 PI algorithm.

---

```

1: Input: desired KL  $v_{kl}$ , coefficients  $K_p$ ,  $K_i$ , max/min value
    $\beta_{max}$ ,  $\beta_{min}$ , iterations  $N$ 
2: Output: hyperparameter  $\beta(t)$  at training step  $t$ 
3: Initialization:  $I(0) = 0$ ,  $\beta(0) = 0$ 
4: for  $t = 1$  to  $N$  do
5:   Sample KL-divergence,  $\hat{v}_{kl}(t)$ 
6:    $e(t) \leftarrow v_{kl} - \hat{v}_{kl}(t)$ 
7:    $P(t) \leftarrow \frac{K_p}{1+\exp(e(t))}$ 
8:   if  $\beta_{min} \leq \beta(t-1) \leq \beta_{max}$  then
9:      $I(t) \leftarrow I(t-1) - K_i e(t)$ 
10:    else
11:       $I(t) = I(t-1)$  // Anti-windup
12:    end if
13:     $\beta(t) = P(t) + I(t) + \beta_{min}$ 
14:    if  $\beta(t) > \beta_{max}$  then
15:       $\beta(t) = \beta_{max}$ 
16:    end if
17:    if  $\beta(t) < \beta_{min}$  then
18:       $\beta(t) = \beta_{min}$ 
19:    end if
20:    Return  $\beta(t)$ 
21: end for

```

---

شکل ۱۰ - الگوریتم PI

```

class PIController:
    def __init__(self, setpoint, Kp=0.01, Ki=0.001, beta_min=0.0, beta_max=1.0):
        self.Kp = Kp
        self.Ki = Ki
        self.setpoint = setpoint
        self.beta_min = beta_min
        self.beta_max = beta_max
        self.cur_Ik = 0.0
        self.cur_Wk = 0.0

    def step(self, error):
        error_k = self.setpoint - error
        Pk = self.Kp * (1.0 / (1.0 + np.exp(error_k)))

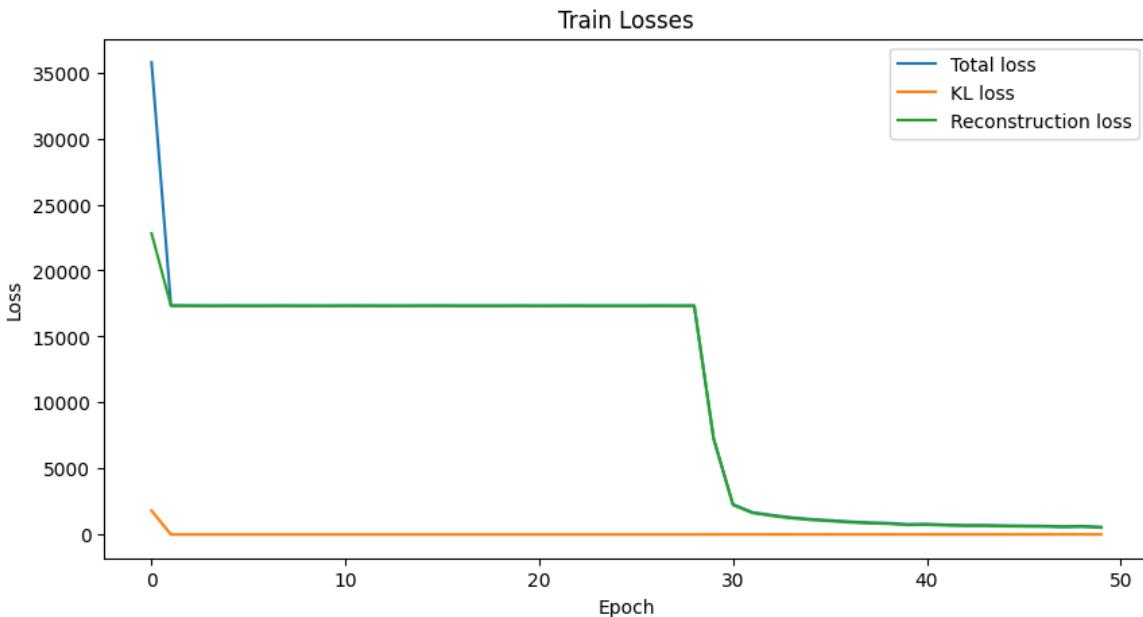
        Ik = self.cur_Ik
        if self.beta_min <= self.cur_Wk <= self.beta_max:
            Ik -= self.Ki * error_k

        Wk = Pk + Ik
        self.cur_Wk = Wk
        self.cur_Ik = Ik

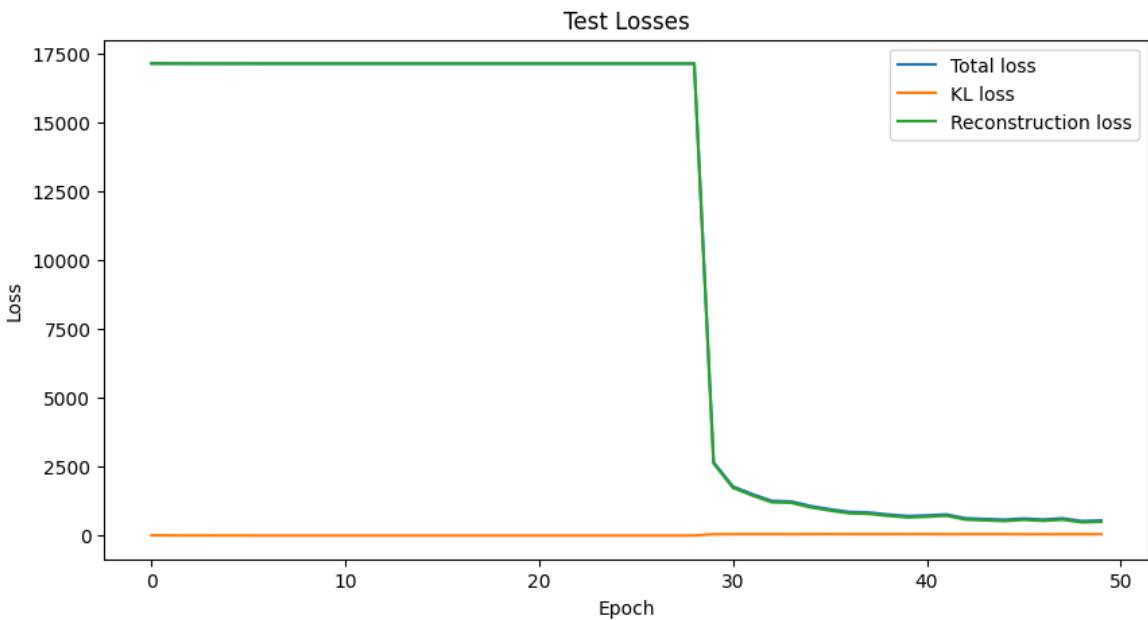
        Wk = np.clip(Wk, self.beta_min, self.beta_max)
        return Wk

```

شکل ۱۱ - بخش Control VAE مدل PI Contoller



شکل ۱۲ - نمودار loss برای CVAE با setpoint=8 روی داده آموزش

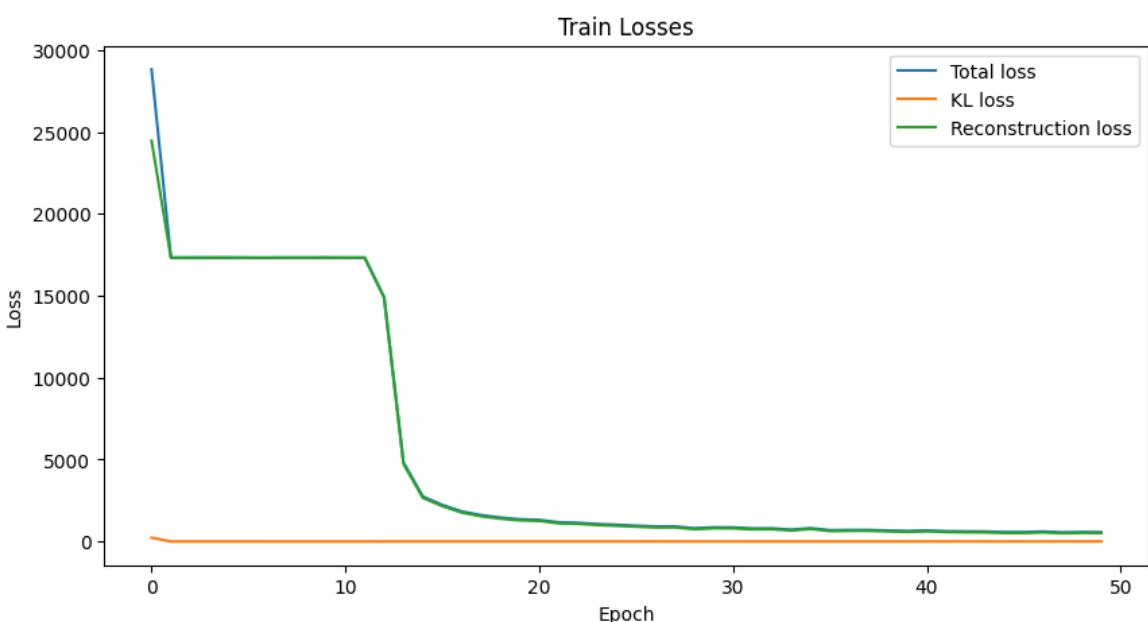


شکل ۱۳ - نمودار loss برای CVAE با setpoint=8 روی داده ارزیابی

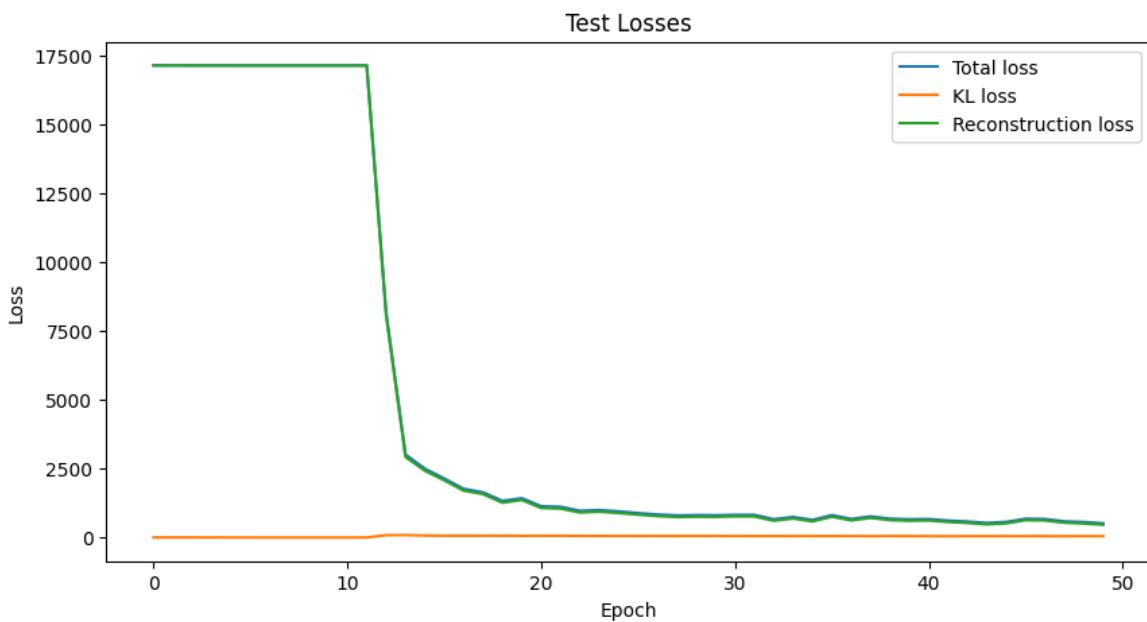
```
CVAE8_fid_score = fid_score.calculate_fid_given_paths([ORIGINAL_DIR, CVAE8_GENERATED_DIR], batch_size=50, device=device, dims=2048)
print(CVAE8_fid_score)

100%|██████████| 100/100 [00:18<00:00,  5.28it/s]
100%|██████████| 100/100 [00:19<00:00,  5.15it/s]
7.958554652797815
```

شکل ۱۴ - معیار FID برای CVAE با setpoint=8



شکل ۱۵ - نمودار loss برای CVAE با setpoint=14 روی داده آموزش

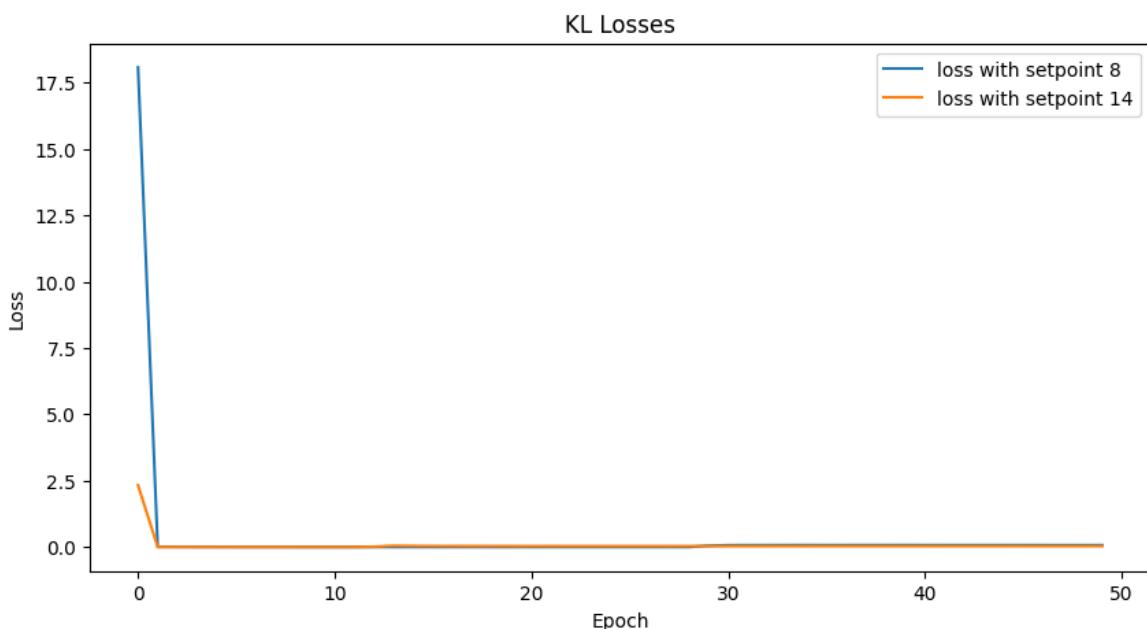


شکل ۱۶ - نمودار loss برای CVAE با setpoint=14 روی داده تست

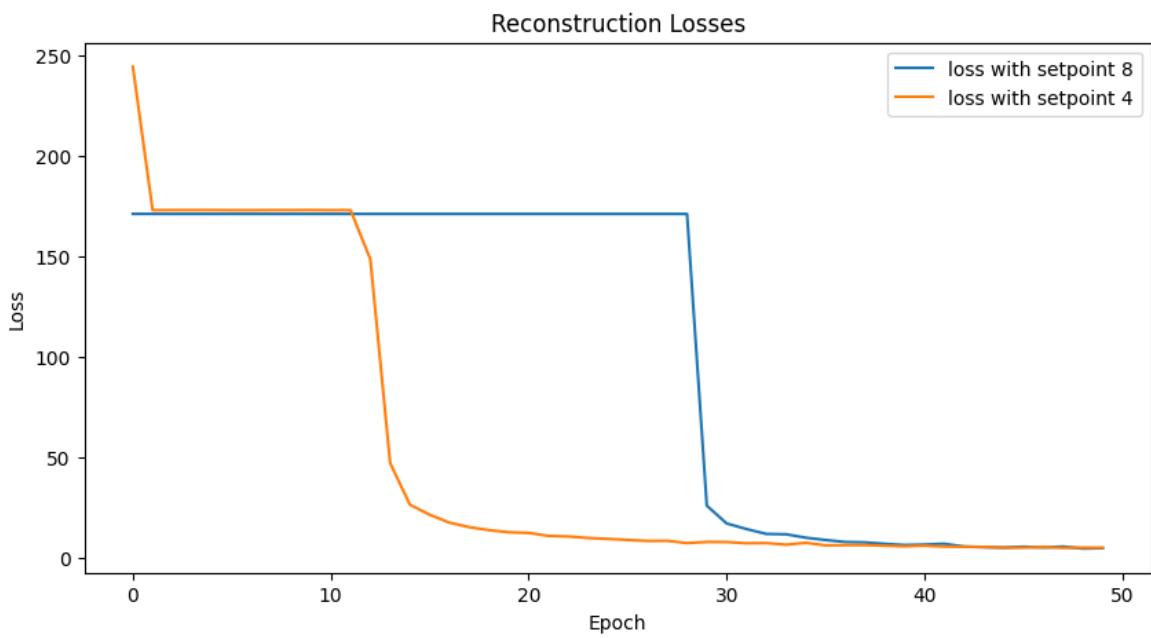
```
CVAE14_fid_score = fid_score.calculate_fid_given_paths([ORIGINAL_DIR, CVAE14_GENERATED_DIR], batch_size=50, device=device, dims=2048)
print(CVAE14_fid_score)

100%|██████████| 100/100 [00:18<00:00,  5.27it/s]
100%|██████████| 100/100 [00:18<00:00,  5.28it/s]
7.958554652797815
```

شکل ۱۷ - نمودار FID برای CVAE با setpoint=14



شکل ۱۸ - نمودار KL loss برای دو setpoint



شکل ۱۹ - نمودار Reconstruction loss برای دو setpoint

## پاسخ ۲ - Generative Adversarial Networks (GANs)

### ۱-۱. آموزش مدل GAN بر روی دیتاست MNIST

پس از دانلود کردن داده های MNIST، شبکه های generator و discriminator و سپس تابع loss را طبق فرمول های داده شده در صورت پروژه پیاده سازی می کنیم.

$$L_{\text{discriminator}}(\phi; \theta) \approx -\frac{1}{m} \sum_{i=1}^m \log D_\phi(\mathbf{x}^{(i)}) - \frac{1}{m} \sum_{i=1}^m \log (1 - D_\phi(G_\theta(\mathbf{z}^{(i)})))$$
$$L_{\text{generator}}^{\text{ns}}(\phi; \theta) \approx -\frac{1}{m} \sum_{i=1}^m \log D_\phi(G_\theta(\mathbf{z}^{(i)}))$$

for batch-size  $m$ , and batches of *real-data*  $\mathbf{x}^{(i)} \sim p_{\text{data}}(\mathbf{x})$  and *fake-data*  $\mathbf{z}^{(i)} \sim \mathcal{N}(0, I)$

```
class NSLosses:
    def generator_loss(self, outputs):
        return -torch.mean(torch.log(outputs))

    def discriminator_loss(self, outputs_real, outputs_fake):
        loss_real = -torch.mean(torch.log(outputs_real))
        loss_fake = -torch.mean(torch.log(1.0 - outputs_fake))
        return loss_real + loss_fake
```

شکل ۲۰ - کد تابع خطای اشباع ناپذیر

شکل زیر نشان دهنده تابع تولید نویز برای تولید داده های مصنوعی را نشان می دهد. طبق خواسته سوال این نویز از توزیه نورمال بدست می آید.

```
def generate_noise(batch_size, latent_dim=64):
    return torch.randn(batch_size, latent_dim).to(device)
```

شکل ۲۱ - کد تابع تولید نویز

شکل های زیر نشان دهنده نسخه کامل شده و کد پیاده سازی معماری generator است. با توجه به اینکه لایه بعدی linear و لایه relu لایه pixel Shuffle است و این لایه به ورودی با  $4 \times 4$  بعد نیاز دارد مقدار خروجی لایه linear را به گونه ای تنظیم کنیم که بتوان آن را به شکل زیر reshape کرد.

(batch\_size, channels, height \* upscale\_factor, width \* upscale\_factor)

از طرفی upscale\_factor برابر با ۲ است و ورودی لایه بعدی دارای channels برابر با ۱۶ است. در نتیجه مقدار ورودی لایه pixel Shuffle باید دارای channels برابر با  $16 \times 2^2$  باشد تا زمانی که upscale\_factor از آن تقسیم میشود، خروجی دارای channels برابر با ۱۶ باشد. در نتیجه مقدار خروجی لایه linear را برابر با  $16 \times 2^2 \times 7 \times 7$  قرار می دهیم. همچنین چونکه ورودی این لایه Linear، خروجی به دست آمده از لایه convolutional قبل از آن است که ۶۴ کانال دارد که هر کدام دارای ابعاد ۷ در ۷ هستند.

همچنین چون خروجی لایه آخر generator یک تصویر با اندازه برابر با تصویر های اصلی دیتا است تعداد channels خروجی لایه آخر را برابر با تعداد channels های عکس های دیتا است که ۱ است قرار می دهیم.

**جدول ۱ – معماری GAN generator مدل**

	Name	In	Out	Batch Norm., Stride, Padding
1	Linear	64	512	BN
	ReLU	-	-	-
2	Linear	512	$16 \times 4 \times 7 \times 7$	BN
	ReLU	-	-	-
3	PixelShuffle	-	-	-
4	Conv 3*3	16	32	BN, p=1
	ReLU	-	-	-
5	PixelShuffle	-	-	-
6	Conv 3*3	8	1	p=1

```

class Generator(nn.Module):
    def __init__(self, latent_dim=64):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),

            nn.Linear(512, 64*7*7),
            nn.BatchNorm1d(64*7*7),
            nn.ReLU(),

            Reshape((-1, 16*4, 7, 7)), # Reshape to be used in PixelShuffle
            nn.PixelShuffle(upscale_factor=2),

            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),

            nn.PixelShuffle(upscale_factor=2),

            nn.Conv2d(8, 1, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        z = self.model(z)
        return z

```

شکل ۲۲ - کد معماری GAN generator مدل

شکل های زیر نشان دهنده نسخه کامل شده و کد پیاده سازی معماری discriminator است.

همان طور که می توان مشاهده کرد تعداد channel های لایه قبل از linear برابر با ۶۴ است. همچنین طول و عرض تصاویر برابر با ۲۸ است که با توجه به stride=2 برای دو لایه قبل کانولوشنی، داریم در نتیجه مقدار ورودی لایه linear برابر با  $28/(2*2)=7$  می شود.

جدول ۲ - معماری GAN discriminator مدل

	Name	In	Out	Stride, Padding
1	Conv 4*4	1	32	S=2, p=1
	ReLU	-	-	-
2	Conv 4*4	32	64	S=2, p=1
	ReLU	-	-	-
3	Linear	64*7*7	512	-
	ReLU	-	-	-
4	Linear	512	1	-

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

```

شکل ۲۳ - کد معماری **GAN discriminator** مدل

### ۱-۱. پیاده‌سازی

### ۲-۱-۲. PixelShuffle

این عملگر در واقع برای upsample کردن بهینه در بخش generator شبکه GAN مورد استفاده قرار می‌گیرد. در واقع این عملگر ورودی خود را طوری rearrange و reshape می‌کند تا آن را افزایش دهد. به طور دقیق‌تر این عملگر با گرفتن یک tensor با ابعاد  $(N, C \times r^2, H, W)$  آن را تبدیل به tensor دیگری با ابعاد  $(N, C, H \times r, W \times r)$  می‌کند که N همان batch size، C تعداد چنل‌ها، r نشان‌دهنده upsampling factor و W و H هم طول و عرض هستند.

در واقع هدف و تاثیر این عملگر در بخش GAN generator یک upsampling feature بر روی map های ما پس از لایه‌های convolutional است که در حین اینکه checkboard artifact هایی که ممکن است در حین فرایند upsampling رخ بدene (مخصوصاً در صورت استفاده از روش‌های کلاسیک‌تر مثل nearest neighbor و یا bilinear interpolation) را کاهش می‌دهد، باعث افزایش spatial resolution از لحاظ بصری ما هم شود. در واقع استفاده از PixelShuffle باعث می‌شود که تصاویر upsample شده ما از لحاظ بصری نرم‌تر و راضی‌کننده‌تر باشند. همچنین PixelShuffle از لحاظ محاسباتی نیز غالباً سبک‌تر و به صرفه‌تر از روش‌های سنتی عمل می‌کند.

### ۳-۱-۲. هایپرپارامترهای مدل

در این بخش هایپرپارامترهای متفاوتی برای آموزش مدل مورد بررسی و ارزیابی قرار گرفت تا در نهایت هایپرپارامترهایی که با استفاده از آن‌ها مدل عملکرد بهتری از خود به جا می‌گذارد به دست بیایند.

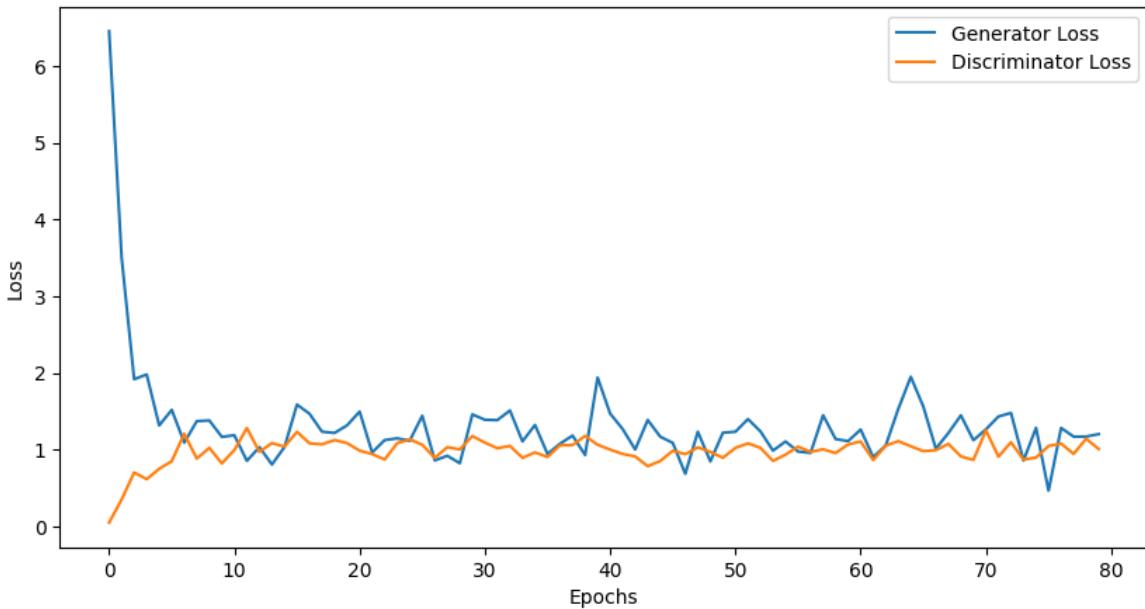
های متفاوتی برای Adam optimizer استفاده شده در آموزش دو مدل Learning rate و Generative مورد استفاده قرار گرفت که از بین مقادیر مختلف بین  $1 \times 10^{-5}$  تا  $1 \times 10^{-1}$ ، مقدار Discriminator learning rate، مقادیر مختلف برای پارامترهای  $\beta_1$  و  $\beta_2$  مدل در نظر گرفته شد که در نهایت مقدار پیشفرض خود optimizer عملکرد بهتری از خود به نمایش می‌داد. همچنین علاوه بر data loader است که مقادیر مختلفی مثل ۳۲، ۶۴ و ۱۲۸ مورد بررسی قرار گرفته و در نهایت مقدار ۶۴ برای این متغیر در نظر گرفته شد. همچنین تعداد epoch های مورد نیاز برای آموزش مدل هم تغییر داده شد و در نهایت مشاهده شد که مدل با حدود ۸۰ اپیک به عملکرد و آموزش خوبی می‌رسد.

جدول ۳ - پارامترهای مدل

پارامتر	مقدار
<b>Learning Rate</b>	<b>0.00005</b>
<b># Epochs</b>	<b>64</b>
<b>Optimizer</b>	<b>Adam</b>
<b>Loss Function</b>	<b>Non-saturating loss</b>
<b>Metrics</b>	<b>FID</b>

#### ۴-۱-۲. ارزیابی مدل

در ابتدا نمودار تغییرات loss را برای دو بخش generator و discriminator مدل GAN مشاهده می‌کنیم.



شکل ۲۴ - نمودار تغییرات loss بخش discriminator و generator

همانطور که دیده می‌شود، این دو بخش در حال رقابت با هم هستند و هر دو در تلاشند که مقدار loss خود را کاهش دهند و پس از طی شدن تعداد کافی قدم از آموزش، این دو بخش تقریباً به ثبات می‌رسند و آموزش مدل به وضعیت پایداری می‌رسد.

در این بخش می‌توانیم تصاویر تولید شده توسط بخش generator مدل را در مراحل مختلف آموزش مشاهده کنیم. عکس اول متعلق به تصاویر تولید شده در ابتدای آموزش، تصویر دوم متعلق به تصاویر تولید شده مدل در میانه آموزش و تصویر سوم هم متعلق به تصاویر تولید شده در انتهای آموزش مدل می‌باشد.

در نهایت نیز معیار FID که در سوال اول توضیحات مربوط به آن داده شده است با استفاده از کتابخانه pytorch-fid محاسبه و گزارش شده است.

```
from pytorch_fid import fid_score

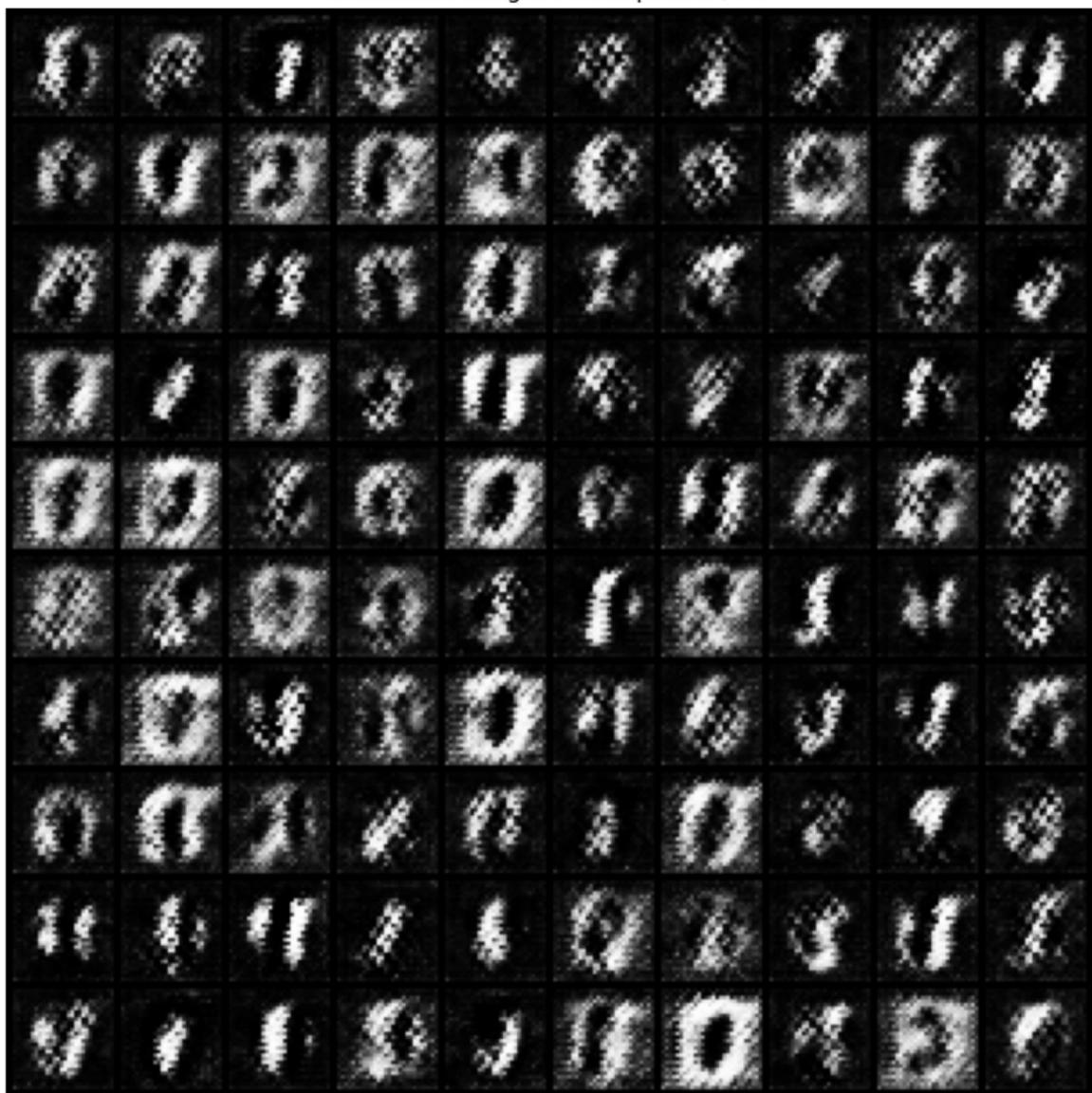
fid_value = fid_score.calculate_fid_given_paths([real_images_path, fake_images_path], batch_size=batch_size, dims=2048, device=device)
print(f"FID Score: {fid_value}")

Downloading: "https://github.com/mseitzer/pytorch-fid/releases/download/fid_weights/pt_inception-2015-12-05-6726825d.pth" to /root/.cache/torchaudio/checkpoints/pt_inception-2015-12-05-6726825d.pth
100%|██████████| 91.2M/91.2M [00:00<00:00, 154MB/s]
100%|██████████| 157/157 [00:41<00:00,  3.81it/s]
100%|██████████| 156/156 [00:40<00:00,  3.82it/s]
FID Score: 17.99406885381657
```

شکل ۲۵ - محاسبه معیار FID

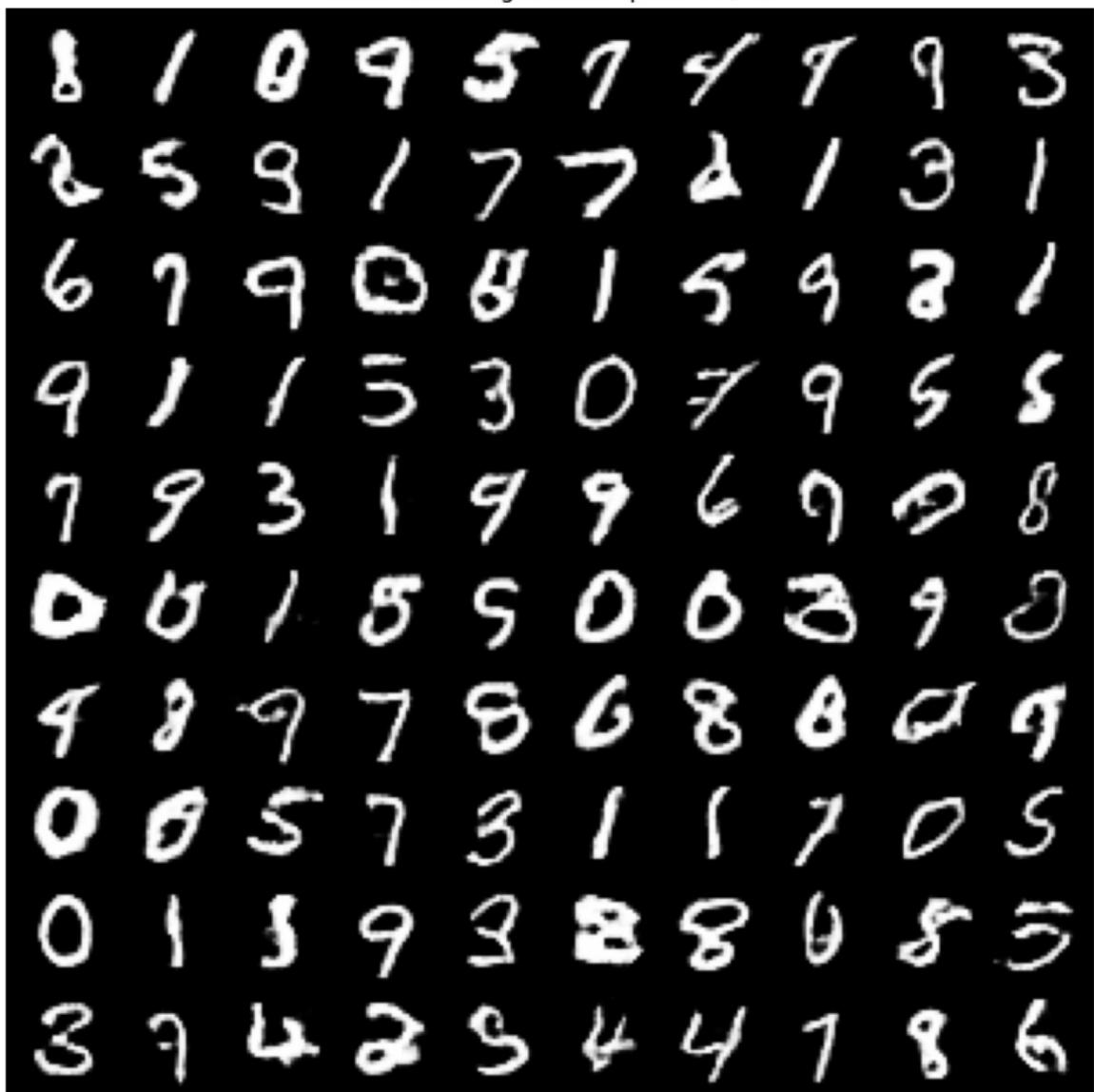
همانطور که دیده می‌شود، کیفیت تصاویر خروجی و تولید شده توسط مدل کاملاً قابل قبول است و معیار FID هم این موضوع را تایید می‌کند، چرا که مقدار FID در حدود ۱۸ است که برای دیتاست MNIST و با استفاده از این معماری GAN که معماری نسبتاً ساده‌ای هم هست، نتیجه خوب و قابل قبولی است.

Generated Images From Epoch 1 / 80



شکل ۲۶ - تصاویر تولید شده مدل در ابتدای آموزش

Generated Images From Epoch 41 / 80



شکل ۲۷ - تصاویر تولید شده در میانه آموزش مدل

Generated Images From Epoch 80 / 80



شکل ۲۸ - تصاویر تولید شده در انتهای فرایند آموزش توسط مدل

## Wasserstein GAN ۲-۲. مدل

```
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 1),
        )
```

شکل ۲۹ - معماری بخش **critic** مدل WGAN

$$g_w \leftarrow \nabla_w \left[ \frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)})) \right]$$
$$g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$$

```
class WGANLosses:
    def critic_loss(self, outputs_real, outputs_fake):
        return -(torch.mean(outputs_real) - torch.mean(outputs_fake))

    def generator_loss(self, outputs):
        return -torch.mean(outputs)
```

شکل ۳۰ - تابع **loss** مربوط به WGAN

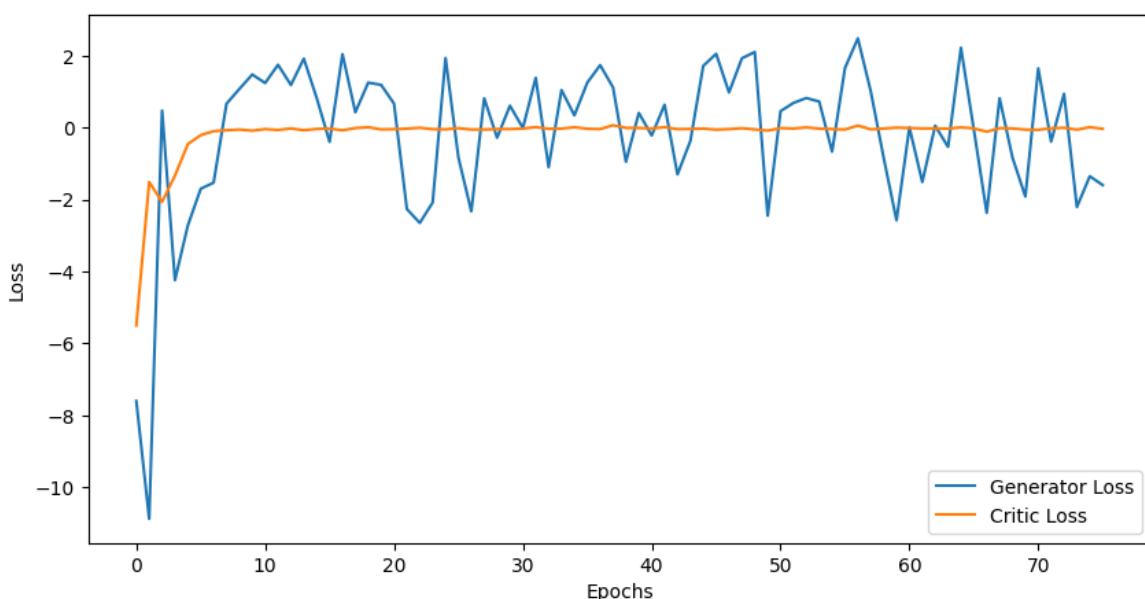
## ۲-۱. بهبودهای مدل

این مدل در جهت بهبود چند ضعف موجود در مدل GAN اولیه عرضه شد. یکی از این ضعف‌ها، مشکل mode collapse بود، یعنی اینکه GAN نمی‌توانست که تمام تنوع داده‌ها را شناسایی و استفاده کند و در عوض صرفاً می‌توانست بخش محدودی از تنوع داده‌ها را تولید کند. این مشکل منجر به تنوع ضعیف سempل‌های تولید شده و پوشش ندادن تمام انواع داده توسط مدل می‌شود. در WGAN ما با استفاده از تعریف کردن Wasserstein distance که با نام Earth Mover's distance هم شناخته می‌شود، معیار بهتر و با معنی تری نسبت به معیار Jensen-Shannon divergence که در نسخه‌های اولیه GAN استفاده می‌شود برای مقایسه دو تابع توزیع خواهیم داشت که این مشکل را بهبود می‌بخشد. از طرف دیگر فرایند آموزش GAN‌های اولیه بسیار غیرپایدار و غیرقابل پیش‌بینی است که در WGAN‌ها با تشویق مدل به ساختن

gradient های نرم‌تر این مشکل هم بهبود یافته است. همچنین در مدل‌های اولیه GAN، ما با مشکل gradient saturation و vanishing gradients مواجهیم (مخصوصاً در مراحل ابتدایی آموزش و اگر که بخش discriminator از پیش‌بینی‌های خود زیادی مطمئن باشد) که این مشکل باعث اخلال در فرایند یادگیری مدل و به خصوص بخش generator می‌شود که این مشکل هم تا حدی در WGAN‌ها به خاطر استفاده از معیاری متفاوت تا حدی رفع شده و بهبود یافته است.

## ۲-۲-۲. ارزیابی مدل

در ابتدا نمودار تغییرات loss را برای دو بخش generator و discriminator مدل WGAN مشاهده می‌کنیم.



شکل ۳۱ - نمودار تغییرات loss بخش generator و discriminator

همانطور که دیده می‌شود، این دو بخش در حال رقابت با هم هستند و هر دو در تلاشند که مقدار loss خود را کاهش دهند و پس از طی شدن تعداد کافی قدم از آموزش، این دو بخش تقریباً به ثبات می‌رسند و آموزش مدل به وضعیت پایداری می‌رسد.

در این بخش می‌توانیم تصاویر تولید شده توسط بخش generator مدل را در مراحل مختلف آموزش مشاهده کنیم. عکس اول متعلق به تصاویر تولید شده در ابتدای آموزش، تصویر دوم متعلق به تصاویر تولید شده مدل در میانه آموزش و تصویر سوم هم متعلق به تصاویر تولید شده در انتهای آموزش مدل می‌باشد. در نهایت نیز معیار FID که در سوال اول توضیحات مربوط به آن داده شده است با استفاده از کتابخانه pytorch-fid محاسبه و گزارش شده است.

```

from pytorch_fid import fid_score

fid_value = fid_score.calculate_fid_given_paths([real_images_path, fake_images_path], batch_size=batch_size, dims=2048, device=device)
print(f"FID Score: {fid_value}")

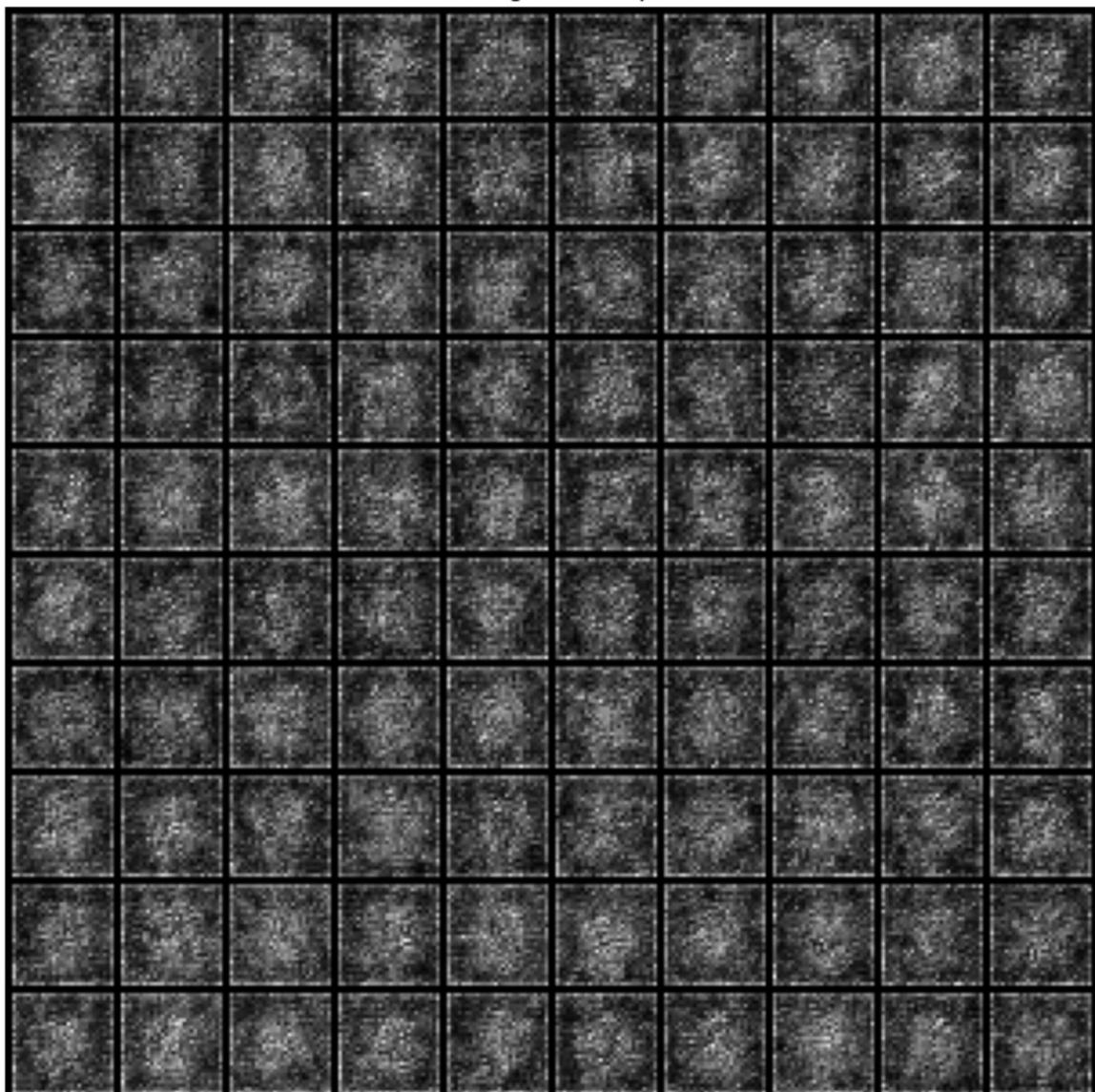
Downloading: "https://github.com/mseitzer/pytorch-fid/releases/download/fid_weights/pt_inception-2015-12-05-6726825d.pth" to /root/.cache/torch/fid_weights/pt_inception-2015-12-05-6726825d.pth
100%|██████████| 91.2M/91.2M [00:00<00:00, 296MB/s]
100%|██████████| 157/157 [00:43<00:00,  3.61it/s]
100%|██████████| 156/156 [00:42<00:00,  3.69it/s]
FID Score: 37.977727046721895

```

شکل ۳۲ - محاسبه معیار **FID**

همانطور که دیده می‌شود، کیفیت تصاویر خروجی و تولید شده توسط مدل کاملاً قابل قبول است و معیار FID هم این موضوع را تایید می‌کند، چرا که مقدار FID در حدود ۳۸ است که برای دیتابست MNIST با استفاده از این معماری WGAN، نتیجه خوب و قابل قبولی است.

Generated Images From Epoch 1 / 76



شکل ۳۳ - تصاویر تولید شده در ابتدای فرایند آموزش مدل

Generated Images From Epoch 39 / 76



شکل ۳۴ - تصاویر تولید شده در میانه فرایند آموزش توسط مدل

Generated Images From Epoch 76 / 76



شکل ۳۵ - تصاویر تولید شده نهایی پس از اتمام فرایند آموزش مدل

## ۳-۲. مدل Self-Supervised GAN

در این قسمت از تمرین یک مدل SSGAN را با استفاده از مقاله و شبکه های پیشنهادی در صورت سوال پیاده سازی می کنیم.

در مقاله از دو تکنیک محبوب یادگیری بدون ناظارت، آموزش self-supervision و adversarial استفاده شده است. از این تکنیک ها برای رفع اشکال اصلی GAN های شرطی یعنی ضرورت استفاده از داده های labelled استفاده شده است. نقش self-supervision در این زمینه، تشویق شبکه discriminator به یادگیری بازنمایی ویژگی های معنادار است که در طول آموزش فراموش نمی شوند. استفاده از بلاک های residual و تصویر های چرخش داده شده این امر را ممکن می سازد.

ابتدا داده آموزش MNIST را دانلود کرده و برای استفاده از آن dataloader برای آن می سازیم. با توجه به سنگین بودن و محدودیت استفاده از GPU T4 مدل تنها batch\_size قابل استفاده عدد ۲ است. همچنین برای کوتاه شدن زمان آموزش تنها ۱۰۰۰۰ داده از داده های آموزش مورد استفاده قرار گرفته شده است. شکل های زیر معماری پیشنهادی در صورت سوال و پیاده سازی انجام شده از آن معماری را نشان میدهند.

## Generator .۱-۳-۲

جدول ۴- معماری generator مدل SSGAN

	Name	In	Out	Batch Norm., Stride, Padding, Up-sampling
1	Linear	128	256*4*4	BN
2	Residual	256	256	UpS=T
3	Residual	256	256	UpS=T
4	Residual	256	256	UpS=T
5	ReLU	-	-	-
6	Conv 3*3	256	1	S=1, p=1
7	Tanh	-	-	-

```

class Generator(nn.Module):
    def __init__(self, latent_dim=128):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256*4*4),
            Reshape((-1, 256, 4, 4)), # Reshape to be in 4D
            nn.BatchNorm2d(256),

            GeneratorResidual(256, 256),
            GeneratorResidual(256, 256),
            GeneratorResidual(256, 256),

            nn.ReLU(),
            nn.Conv2d(256, 1, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        z = self.model(z)
        return z

```

شکل ۳۶ - کد SSGAN مدل generator

همان طور که در شکل های بالا مشخص است ابتدا با یک لایه fully connected شروع می کنیم و قبل از ارسال خروجی این لایه به لایه residual قرار می دهیم تا خروجی را برای وارد شدن به این لایه به ۴ dimension تبدیل کند.

جدول ۵ – بلاک generator در residual مدل SSGAN

	Name	In	Out	args
Block1	1 Batchnorm2D	in_channels	-	-
	2 ReLU	-	-	-
	3 Upsample	-	-	scale_factor=2 mode=nearest
	4 conv2D 3*3	in_channels	out_channels	strid=1 padding=1
	5 Batchnorm2D	num_features=out_channel	-	-
	6 ReLU	-	-	-
	7 conv2D 3*3	out_channels	out_channels	strid=1 padding=1
	8 Upsample	-	-	scale_factor=2 mode=nearest
output		Block1 + Layer Input		

```

class GeneratorResidual(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(GeneratorResidual, self).__init__()

        self.block1 = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(),
            nn.Upsample(scale_factor=2, mode='nearest'),
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),

            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1),
            nn.Upsample(scale_factor=2, mode='nearest')
        )

        self.adjust_channels = nn.Upsample(scale_factor=4, mode='nearest')

    def forward(self, z):
        block1 = self.block1(z)
        layerInput = self.adjust_channels(z)
        return block1 + layerInput # Skip Connection

```

شکل ۳۷ – کد بلاک **generator residual** در **SSGAN**

شکل های بالا نیز پیاده سازی بلاک residual را نشان میدهند. خروجی این بلاک دارای skip connection است که باعث می شود اطلاعات داده ورودی نیز در نظر گرفته شود. برای یکسان شدن سایز ورودی و خروجی لایه های بلاک برای انجام عمل جمع، روی ورودی نیز عمل Up-sampling را انجام می دهیم.

## Discriminator .۲-۳-۲

جدول ۶ - معماری **discriminator** مدل **SSGAN**

Name	In	Out	Batch Norm., Stride, Padding, Down-sampling
1	Residual	1	128
2	Residual	128	128
3	Residual	128	128
4	Residual	128	128
5	Linear	128	1
6	Linear	128	4

```

class Discriminator(nn.Module):
    def __init__(self, ssup=True, num_rotation = 4):
        super(Discriminator, self).__init__()
        self.ssup = ssup

        self.residualBlocks = nn.Sequential(
            DiscriminatorResidual(1, 128, first_block=True, down_sampling=True),
            DiscriminatorResidual(128, 128, down_sampling=True),
            DiscriminatorResidual(128, 128),
            DiscriminatorResidual(128, 128),
        )

        self.adversarialLayer = nn.Sequential(
            nn.Linear(128, 1),
        )
        self.auxiliaryLayer = nn.Sequential(
            nn.Linear(128, num_rotation)
        )
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        features = self.residualBlocks(x)
        features = torch.sum(features, dim=(2, 3))
        adv_out = self.adversarialLayer(features)
        if self.ssup:
            rot_out = self.auxiliaryLayer(features)
            rot_prob = self.softmax(rot_out)
            return self.sigmoid(adv_out), adv_out, rot_out, rot_prob
        else:
            return self.sigmoid(adv_out), adv_out

```

شکل ۳۸ - کد مدل **discriminator** SSGAN

در معماری generator برای همه لایه ها قرار گرفته پس این پارامتر را به عنوان ورودی در نظر نگرفته ایم. اما در پیاده سازی بلاک residual discriminator در معماری Up-sampling بعنوان ورودی در نظر گرفته ایم زیرا برای همه لایه ها مقدار یکسانی ندارد. همچنین Down-sampling به عنوان ورودی در نظر گرفته ایم زیرا برای همه لایه ها مقدار یکسانی ندارد. همچنین این مدل دارای دو لایه fully connected خروجی است یکی با یک خروجی برای تشخیص واقعی یا مصنوعی بودن داده و دیگری با ۴ خروجی برای تشخیص زاویه تصویر. همان طور که در مقاله نیز به آن اشاره شده است، تصاویر واقعی و مصنوعی در ۰، ۹۰، ۱۸۰ و ۲۷۰ درجه rotate می شوند.

## جدول ۷ - بلاک residual در discriminator مدل SSGAN

	Name	In	Out	args
Block1	1	ReLU* (not for the first Res Block)	-	-
	2	covn2D 3*3	in_channels	out_channles strid=1 padding=1
	3	specteral_norm	-	-
	4	ReLU	-	-
	5	conv2D 3*3	out_channels	out_channles strid=1 padding=1
	6	specteral_norm	-	-
	7	AvgPool2D 2*2	-	- strid=2 padding=1
Block2	1	AvgPool2D 2*2	-	- strid=2 padding=1
	2	conv2D 1*1	out_channels	out_channles strid=1 padding=0
<b>output</b>		Block1 + Block2		

```

class DiscriminatorResidual(nn.Module):
    def __init__(self, in_channels, out_channels, first_block=False, down_sampling=False):
        super(DiscriminatorResidual, self).__init__()

        self.first_block = first_block
        self.down_sampling = down_sampling

        self.relu = nn.ReLU()
        self.conv1b1 = spectral_norm(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1))
        self.conv2b1 = spectral_norm(nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1))
        self.avgpoolb1 = nn.AvgPool2d(kernel_size=2, stride=2, padding=1)

        self.avgpoolb2 = nn.AvgPool2d(kernel_size=2, stride=2, padding=1)
        self.conv3b2 = spectral_norm(nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0))

    def forward(self, x):
        b2in = x
        if self.first_block:
            block1 = self.relu(self.conv1b1(x))
            block1 = self.conv2b1(block1)
            if self.down_sampling:
                block1 = self.avgpoolb1(block1)
        else:
            block1 = self.relu(x)
            block1 = self.relu(self.conv1b1(block1))
            block1 = self.conv2b1(block1)
            if self.down_sampling:
                block1 = self.avgpoolb1(block1)

        if self.down_sampling:
            b2in = self.avgpoolb2(b2in)
            block2 = self.conv3b2(b2in)

        return block1 + block2

```

شکل ۳۹ - کد بلاک residual در discriminator مدل SSGAN

همان طور که در بالاتر نیز اشاره شد این بلاک مقدار Down-sampling را نیز به عنوان ورودی دریافت می کند و تنها زمانی AvgPool را انجام می دهد که مقدار این پارامتر True باشد. همچنین طبق معماری داده شده برای بلاک اول از residual لایه اول  $\text{relu}$  را نداریم.

در ادامه به توضیح پیاده سازی loss function می پردازیم. برای این کار از معادله داخل مقاله استفاده کرده ایم. در مقاله adversarial loss و rotation-based loss با هم ترکیب شده اند. ورودی rotation-based loss تشخیص شبکه discriminator از واقعی یا مصنوعی بودن عکس و ورودی loss  $\text{discriminator}$  از زاویه چرخش عکس است.

$$V(G, D) = \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}(\mathbf{x})} [\log P_D(S = 1 \mid \mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim P_G(\mathbf{x})} [\log(1 - P_D(S = 0 \mid \mathbf{x}))]$$

$$L_G = -V(G, D) - \alpha \mathbb{E}_{\mathbf{x} \sim P_G} \mathbb{E}_{r \sim \mathcal{R}} [\log Q_D(R = r \mid \mathbf{x}^r)],$$

$$L_D = V(G, D) - \beta \mathbb{E}_{\mathbf{x} \sim P_{\text{data}}} \mathbb{E}_{r \sim \mathcal{R}} [\log Q_D(R = r \mid \mathbf{x}^r)],$$

```
class SSGANLoss:
    def __init__(self, a=0.5, b=1.0):
        self.a = a
        self.b = b

    def rotation_loss(self, y_true, y_pred):
        criterion = nn.CrossEntropyLoss()
        return criterion(y_pred, y_true)

    def generator_loss(self, D_fake, D_rot, rot_labels):
        # Adversarial Loss for generator
        G_loss_adversarial = torch.mean(F.softplus(-D_fake))
        # Rotation Loss for generator
        G_loss_rotation = self.rotation_loss(rot_labels, D_rot)
        # Total Generator Loss with coefficients
        G_loss = -G_loss_adversarial + self.a * G_loss_rotation
        return G_loss

    def discriminator_loss(self, D_real, D_fake, D_rot, rot_labels):
        # Adversarial Loss for real and fake images
        D_loss_real = torch.mean(D_real)
        D_loss_fake = torch.mean(F.softplus(-D_fake))
        # Total Adversarial Loss
        D_loss = D_loss_real + D_loss_fake
        # Auxiliary Rotation Loss
        D_rot_loss = self.rotation_loss(rot_labels, D_rot)
        # Total Loss
        loss = D_loss + self.b * D_rot_loss
        return loss
```

SSGAN loss – ۴۰ کد

batch\_size پارامتر های مدل مشابه مقاله انتخاب شده اند. بعضی موارد مانند تعداد ایپاک ها و به دلیل محدودیت منابع و crash کردن برنامه کوچک انتخاب شده اند.

جدول ۸ - پارامتر های مدل SSGAN

پارامتر	مقدار
Learning Rate	<b>2e-4</b>
Beta1	<b>0.0</b>
Beta2	<b>0.99</b>
# Train Samples	<b>10000</b>
# Test Samples	<b>1000</b>
Batch Size	<b>2</b>
# Epochs	<b>10</b>
Optimizer	<b>Adam</b>
Loss Function	<b>SSGAN(custom loss)</b>

در حین آموزش مدل، داده های موجود در هر batch و داده های مصنوعی تولید شده توسط generator در هر ۴ زاویه ۰، ۹۰، ۱۸۰ و ۲۷۰ درجه می شوند.

```
# Rotate generated and real images in 4 angles
x = real_images
x_real = x
x_90, x_180, x_270 = x.transpose(2,3), x.flip(2,3), x.transpose(2,3).flip(2,3)
real_images = torch.cat((x, x_90, x_180, x_270), 0)
real_images = Variable(real_images)

z = generate_noise(size)
fake_images = generator(z)
x = fake_images
x_fake = x
x_90, x_180, x_270 = x.transpose(2,3), x.flip(2,3), x.transpose(2,3).flip(2,3)
fake_images = torch.cat((x, x_90, x_180, x_270), 0)
```

شکل ۴۱ - کد rotation عکس ها برای مدل SSGAN

### ۳-۳-۳. ارزیابی مدل

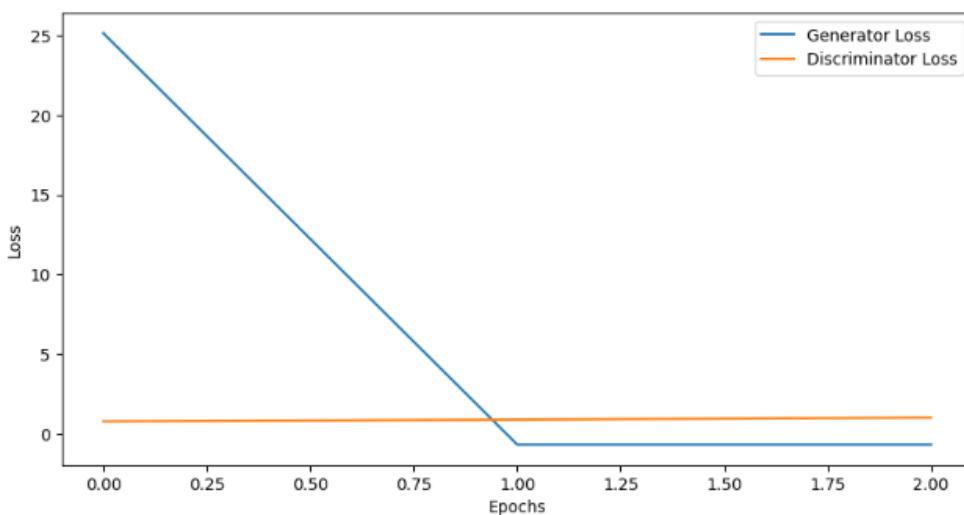
نتایج نهایی بدلیل نبود منابع برای آموزش مدل و crash کردن کد چندان ایده آل، بر خلاف دو مدل قبل نشد.

همان طور که در اعداد گزارش شده و نمودار خطای رزیر قابل مشاهده است، با افزایش ایپاک ها مقدار خطای با کمی نوسان کاهش می یابد. همچنین تصاویر نیز تا حدود کمی پیشرفت می کنند اما بدلیل کم

بودن تعداد ایپاک ها و استفاده از تنها ۱۰۰۰۰ داده از ۶۰۰۰۰ موجود نتایج چندان پیشرفت نمی کنند. در نتیجه مقدار fid نیز بزرگ می باشد.

نتایج از ران کردن مدل با ۱۵ ایپاک بدست آمده اند اما چون مدل crash کرده است، باقی نتایج قابل مشاهده نیست. به همین دلیل دو روزن از این مدل اپلود شده. مدلی که تنها ۳ ایپاک ران شده و مدلی که ۱۵ ایپاک ران شده اما به مشکل کمبود ریسورس خورده است.

Epoch [1/15], D Loss: 0.705377995967865, G Loss: -0.6931471824645996  
Epoch [2/15], D Loss: 0.7000482082366943, G Loss: -0.6931471824645996  
Epoch [3/15], D Loss: 0.7070268988609314, G Loss: -0.6931471824645996  
Epoch [4/15], D Loss: 0.7087118625640869, G Loss: 0.07497292757034302  
Epoch [5/15], D Loss: 0.3134720027446747, G Loss: -0.3132617175579071  
Epoch [6/15], D Loss: 0.3164346516132355, G Loss: -0.3132617175579071  
Epoch [7/15], D Loss: 0.5262981057167053, G Loss: -0.3132617175579071  
Epoch [8/15], D Loss: 0.31490227580070496, G Loss: -0.3132617175579071  
Epoch [9/15], D Loss: 0.31522849202156067, G Loss: -0.3132617175579071  
Epoch [10/15], D Loss: 0.3135070204734802, G Loss: -0.3132617175579071  
Epoch [11/15], D Loss: 0.32216817140579224, G Loss: -0.3132617175579071  
Epoch [12/15], D Loss: 0.31378620862960815, G Loss: -0.3132617175579071  
Epoch [13/15], D Loss: 0.3282022476196289, G Loss: -0.3132617175579071  
Epoch [14/15], D Loss: 0.31577637791633606, G Loss: -0.3132617175579071  
Epoch [15/15], D Loss: 0.5534244775772095, G Loss: -0.3132617175579071



شکل ۴۲ - نمودار تغییرات loss بخش discriminator و generator مدل SSGAN