وزارت علوم، تحقیقات و فناوری
دانشگاه تحصیلات تکمیلی علوم پایه
گاوازنگ، زنجان

دانشکده علوم رایانه و فناوری اطلاعات

درس: یادگیری ژرف (Deep Learning)
تمرین ۴

استاد: دکتر رزاقی

دانشجو: امیرحسین صفری

شماره دانشجویی: ۱۴۰۱۴۱۲۱

بهار ۱۴۰۲

مشکلی اشاره شده در GAN ها (Generative Adversarial Networks) که به عنوان mode collapsing شناخته می شود، زمانی اتفاق می‌افتد که یک GAN نتواند کل توزیع داده‌های آموزشی را بفهمد و در عوض مجموعه محدودی از خروجی‌ها را به طور مکرر تولید می‌کند. این بدان معناست که مولد خروجی های مشابه یا یکسانی را بدون توجه به تغییرات موجود در داده های آموزشی تولید می کند.

mode collapsing ممکن است به دلایل مختلفی رخ دهد:

- قوی‌تر بودن تشخیص دهنده (Discriminator): در مراحل اولیه آموزش، تشخیص دهنده (Discriminator) ممکن است بیش از حد قدرتمند شود و نمونه های تولید شده را به دقت شناسایی کند. در نتیجه، مولد (generator) نمی تواند از بازخورد تشخیص دهنده (Discriminator) آموزش ببیند، که منجر به سقوط(collapse) در خروجی مولد می شود.
- مولد ضعیف: اگر شبکه مولد به اندازه کافی قدرتمند نباشد، ممکن است برای یاد گرفتن پیچیدگی توزیع داده ها دچار مشکل شود. این محدودیت می تواند باعث شود که مولد فقط چند نمونه نماینده را به طور مکرر تولید کند.
- متغییرهای آموزشی نامتعادل: GAN ها بر تعادل بین شبکه های مولد و تشخیص دهنده متکی هستند. اگر متغییرهای تمرین به درستی متعادل نباشد، می تواند منجر به فروپاشی حالت (mode collapsing) شود. به عنوان مثال، اگر تشخیص دهنده به طور قابل توجهی قوی تر از مولد باشد، مولد ممکن است برای بهبود و تولید نمونه های متنوع به مشکل بخورد.

حال به چند راه حل ممکن برای غلبه بر فروپاشی حالت در GAN ها را مورد بررسی قرار می‌دهیم:

1. اصلاح کردن معماری: اصلاح معماری GAN می تواند به کاهش فروپاشی حالت کمک کند. این شامل افزایش ظرفیت مولد، کاهش قدرت تشخیص دهنده، یا معرفی اجزای کمکی مانند تکنیک های منظم سازی (regularization) است. تنظیم معماری می تواند به حفظ تعادل بهتر بین مولد و تشخیص دهنده کمک کند.

2. اصلاح کردن تابع Loss: اصلاح تابع‌های Loss استفاده شده در آموزش GAN نیز می تواند فروپاشی حالت را کاهش دهد. تکنیک‌هایی مانند افزودن جریمه‌های گرادیان (مانند Wasserstein GAN با جریمه گرادیان که در سوال بعدی مطرح شده است) یا استفاده از معیارهای واگرایی جایگزین (به عنوان مثال، به حداقل رساندن واگرایی Jensen-Shannon) می‌تواند مولد را تشویق به تولید خروجی‌های متنوع‌تر کند.

3. ویژگی های Mini-batch: یک رویکرد برای کاهش فروپاشی حالت استفاده از ویژگی های Mini-batch است که در آن چندین نمونه از مجموعه آموزشی به طور همزمان در طول آموزش در نظر گرفته می شوند. این می تواند به تشخیص دهنده کمک کند تا بازخورد آموزنده تری را به مولد ارائه دهد و احتمال فروپاشی آن را کاهش دهد.

4. تکنیک های منظم سازی(Regularization): روش های منظم سازی مانند dropout، کاهش وزن، یا نرمال سازی دسته ای (batch normalization) را می توان برای جلوگیری از فروپاشی حالت استفاده کرد. این تکنیک‌ها نویز یا محدودیت‌هایی را در طول آموزش ایجاد می‌کنند و تنوع در خروجی مولد را ارتقا می‌دهند.

5. تقویت داده ها: افزایش داده های آموزشی با ایجاد آشفتگی ها یا تبدیل های کوچک می تواند به مولد کمک کند تا طیف وسیع تری از تغییرات در توزیع داده ها را ثبت کند. این می‌تواند با قرار دادن مولد در معرض نمونه‌های متنوع‌تر، احتمال فروپاشی حالت را کاهش دهد.

B. معماری کلی و تابع هزینه مدل W-GAN را تشـــریح نمایید و تفاوتهای آن با مدل پایه GAN را توضـــیح دهید. آیا تابع هزینه این مدل کمکی به برطرف شـــدن مشـــکل Mode Collapse خواهد کرد؟توضیح دهید.

مدل W-GAN یا (Wasserstein GAN) توسعه‌ای از GAN سنتی است که تابع هزینه و هدف آموزشی متفاوتی را معرفی می‌کند. هدف آن غلبه بر برخی محدودیت‌های مدل اصلی GAN، از جمله فروپاشی حالت و بی‌ثباتی در طول تمرین است.

معماری کلی یک W-GAN مشابه یک GAN است که از یک مولد و یک تفکیک کننده (تشخیص دهنده) تشکیل شده است. مولد نویز تصادفی را به عنوان ورودی می گیرد و نمونه های مصنوعی تولید می کند، در حالی که تشخیص دهنده نمونه های تولید شده و نمونه های واقعی را از داده های آموزشی ارزیابی می کند. با این حال، تفاوت هایی در تابع هزینه و روش آموزش وجود دارد.

در W-GAN، تمرکز اصلی در به حداقل رساندن فاصله Wasserstein (همچنین به عنوان فاصله زمین حرکت دهنده (Earth Mover's distance)) بین توزیع داده های واقعی و تولید شده است. این فاصله عدم تشابه بین دو توزیع را اندازه گیری می کند و معیار معناداری و پایدارتری را برای آموزش GAN ها ارائه می دهد.

برای اعمال فاصله Wasserstein، تشخیص دهنده در W-GAN به عنوان یک شبکه انتقادی تغییر می‌کند که به جای ارائه یک طبقه بندی باینری (واقعی یا جعلی)، به نمونه‌ها امتیاز اختصاص می دهد یا کیفیت نمونه‌ها را تخمین می زند. هدف تشخیص دهنده/منتقد این است که نمرات بالا برای نمونه های واقعی و امتیازات پایین برای نمونه های تولید شده به دست آورد.

تابع هزینه مورد استفاده در W-GAN بر اساس مفهوم فاصله Wasserstein است. به جای استفاده از تلفات متقابل آنتروپی باینری سنتی مانند GAN اصلی، W-GAN از تفاوت بین میانگین امتیازات اختصاص داده شده توسط تشخیص دهنده/منتقد به نمونه های واقعی و تولید شده به عنوان هزینه استفاده می کند. این تفاوت تخمینی از فاصله Wasserstein را نشان می دهد.

در طول آموزش، مولد و منتقد به صورت متناوب به روز می شوند. ابتدا، وزن های منتقد با به حداکثر رساندن اختلاف امتیازات بین نمونه های واقعی و تولید شده به روز می شوند. سپس، وزن‌های مولد به‌روزرسانی می‌شوند تا امتیاز منتقد تخصیص یافته به نمونه‌های تولید شده به حداقل برسد. این روش آموزشی، مولد را تشویق می‌کند تا نمونه‌هایی را تولید کند که امتیاز بالاتری از منتقد دارند و آنها را به توزیع واقعی داده‌ها نزدیک‌تر کند.

عملکرد هزینه W-GAN می تواند تا حدی به کاهش فروپاشی حالت کمک کند. با بهینه‌سازی مستقیم فاصله Wasserstein، مولد را تشویق می‌کند تا محدوده وسیع‌تری از توزیع داده‌ها را بررسی کند و نمونه‌های متنوعی تولید کند. سیگنال‌های گرادیان ارائه‌شده توسط نمرات منتقد اغلب قوی‌تر و آموزنده‌تر از طبقه‌بندی باینری تشخیص‌دهنده در GANهای سنتی هستند، که می‌تواند به هدایت مولد به سمت تولید نمونه‌های متنوع‌تر و واقعی‌تر کمک کند.

با این حال، در حالی که چارچوب W-GAN می تواند فروپاشی حالت را کاهش دهد، مشکل را به تنهایی حل نمی کند. فروپاشی حالت همچنان می تواند در W-GAN ها رخ دهد، به خصوص زمانی که معماری شبکه مولد یا منتقد به اندازه کافی قدرتمند نباشد یا دینامیک آموزش نامتعادل باشد. تکنیک‌های اضافی، مانند اصلاح کردن معماری، روش‌های منظم‌سازی، یا تنظیم دقیق هایپرپارامترها (hyperparameter)، ممکن است برای رفع کردن بیشتر فروپاشی حالت در W-GAN ضروری باشد.

الگوریتم WGAN

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

تفاوت تابع هزینه (cost function) در GAN و WGAN

	Discriminator/Critic	**Generator**
GAN	$\nabla_{\theta_d} \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m}\sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m}\sum_{i=1}^{m}\left[f\left(x^{(i)}\right) - f\left(G\left(z^{(i)}\right)\right)\right]$	$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f\left(G\left(z^{(i)}\right)\right)$

سوال ۲ : (عملی)

هدف این سوال طراحی ساده شبکه GAN میبا شد که بتواند تصاویر دیتا ست MNIST را تولید نماید. فایل GAN.ipynb را بر ا ساس موارد خوا سته شده تکمیل نمایید.با توجه به نکات تمرین ، ساختار شبکه و تابع خطا و پارامترهای دیگر را طوری طراحی و انتخاب نمایید که در نهایت تصاویر قابل قبولی توسط Generator تولید شود. در iteration و epoch های مختلف مقدار loss چه تغییراتی میکند و این نو سان ها چه معنی ای میدهند؟ آیا همگرایی (convergence) قابل م شاهده ا ست؟ تغییر نرخ یادگیری چه اثری دارد؟(تحلیل کنید) فایل تکمیل شده را بعلاوه فایل پارامترهای شبکه Generator ای که آموزش داده اید به همراه پا سخ سوال های طرح شده قبلی ارسال نمایید.

توضیح قسمت های کامل شده:

Batch_size یا تعداد نمونه های تمرینی در یک پاس رو به جلو/عقب(forward/backward pass) را برابر با ۱۲۸ قرار می‌دهیم.

d_lr که میزان یادگیری برای تشخیص دهنده می‌باشد و g_lr که نرخ یادگیری برای مولد است را ابتدا برابر با 0.0002 قرار می‌دهیم.

n_epochs که تعداد دفعاتی که کل مجموعه داده آموزشی در طول آموزش از طریق شبکه به جلو و عقب(forward and backward pass) منتقل می شود را برابر با ۱۰۰ تنظیم کردیم.

```
[3]  # Device configuration
     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

     # [x] ################ Problem 01 (5 pts) ################

     # define hyper parameters
     batch_size = 128
     d_lr = 0.0002
     g_lr = 0.0002
     n_epochs = 100
     ######################## End ########################
     z_dim = 100
```

در این قسمت train_loader مسئول بارگذاری مجموعه داده های آموزشی به صورت دسته‌ای(batch) است، در حالی که test_loader مسئول بارگذاری مجموعه داده های آزمایشی است. (که تعریف شده بود:)) پارامتر suffle برای دیتای test به صورت false در نظر گرفته شده تا ترتیب دیتای test بهم نخورد.

```
[14]
0s
    ############### Problem 02 (5 pts) ###############
    # Define Dataloaders
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
    ###################### End ######################
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

در کلاس Discriminator، یک شبکه متوالی را با استفاده از لایه های کانولوشن تعریف شده است. معماری شبکه شامل لایه‌های کانولوشن، leaky ReLU activations ، نرمال‌سازی دسته‌ای(batch normalization) و یک sigmoid activation نهایی است. این معماری به تثبیت روند آموزش کمک می کند و از مشکلاتی مانند فروپاشی حالت(mode collapse) و ناپدید شدن گرادیان ها(vanishing gradients) جلوگیری می کند.

```
[15] class Discriminator(nn.Module):
0s      def __init__(self):
            super().__init__()

            self.discriminator = nn.Sequential(
                ################ Problem 03 (15 pts) ################
                # use linear or convolutional layer
                # use arbitrary techniques to stabilize training
                nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),
                nn.LeakyReLU(0.2),
                nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
                nn.BatchNorm2d(128),
                nn.LeakyReLU(0.2),
                nn.Flatten(),
                nn.Linear(128 * 7 * 7, 1),
                nn.Sigmoid()

                ###################### End ######################
            )

        def forward(self, x):
            return self.discriminator(x)
```

در کلاس Generator یک شبکه ترتیبی با استفاده از لایه های خطی و کانولوشن تعریف شده است. معماری شبکه شامل لایه‌های خطی، نرمال‌سازی دسته‌ای، فعال‌سازی ReLU و لایه‌های کانولوشن انتقالی(transpose convolutional layers) است. لایه نهایی از فعال سازی Tanh برای تولید نمونه های خروجی استفاده می کند. این معماری به تثبیت فرآیند آموزش و بهبود کیفیت نمونه های تولید شده کمک می کند.

```python
class Generator(nn.Module):
    def __init__(self):
        super().__init__()

        self.generator = nn.Sequential(
            ################ Problem 04 (15 pts) ################
            # use linear or convolutional layer
            # use arbitrary techniques to stabilize training
            nn.Linear(100, 128 * 7 * 7),
            nn.BatchNorm1d(128 * 7 * 7),
            nn.ReLU(),
            nn.Unflatten(1, (128, 7, 7)),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),
            nn.Tanh()

            ######################## End ########################
        )

    def forward(self, z):
        return self.generator(z)
```

در این کد، نمونه(instance) هایی از ماژول های Discriminator و Generator را ایجاد شده است و نمونه مدل ها در device مورد استفاده قرار داده شده‌اند.

```python
[16] ################ Problem 05 (5 pts) ################
    # Create instances of modules (discriminator and generator)
    # don't forget to put your models on device
    discriminator = Discriminator().to(device)
    generator = Generator().to(device)
    ######################## End ########################
```

خروجی مدل‌های تشخیص دهنده(discriminator) و مولد(generator)

```
[19] print(discriminator)

Discriminator(
  (discriminator): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Flatten(start_dim=1, end_dim=-1)
    (6): Linear(in_features=6272, out_features=1, bias=True)
    (7): Sigmoid()
  )
)

print(generator)

Generator(
  (generator): Sequential(
    (0): Linear(in_features=100, out_features=6272, bias=True)
    (1): BatchNorm1d(6272, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Unflatten(dim=1, unflattened_size=(128, 7, 7))
    (4): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (8): Tanh()
  )
)
```

در این کد با استفاده از بهینه ساز optim.Adam دو بهینه ساز تعریف شده است. d_optimizer با پارامترهای ماژول discriminator.parameters() و نرخ یادگیری d_lr مقداردهی اولیه می شود. به همین ترتیب برای g_optimizer پارامترهای ماژول و نرخ یادگیری مقداردهی شده است.

```
[17] ################ Problem 06 (5 pts) ################
     # Define two optimizer for discriminator and generator
     d_optimizer = optim.Adam(discriminator.parameters(), lr=d_lr)
     g_optimizer = optim.Adam(generator.parameters(), lr=g_lr)
     ######################### End ##########################
```

ورودی هایی مانند تصاویر و برچسب‌ها(لیبل‌ها) با استفاده از to.(device) به دستگاه (device) منتقل می شوند. بردار نویز z نیز بر روی دستگاه(device) تولید می شود.

```
plot_frequency = 10

for epoch in range(n_epochs):
    for i, (images, labels) in enumerate(train_loader):

        ################ Problem 07 (15 pts) ################
        # Put your inputs on device
        images = images.to(device)
        labels = labels.to(device)
        z = torch.randn(images.size(0), z_dim).to(device)

        ######################## End ########################

        ################ Problem 08 (10 pts) ################
```

در این قسمت، برچسب‌های واقعی روی ۱ تنظیم شده‌اند، زیرا این تصاویر واقعی هستند که تشخیص دهنده باید به درستی طبقه‌بندی کند. همینطور برای تصاویر غیر واقعی، برچسب‌های جعلی روی 0 تنظیم شده‌اند، زیرا این تصاویر جعلی هستند که توسط مولد تولید می‌شوند و تشخیص دهنده باید به درستی آن‌ها را به عنوان جعلی طبقه‌بندی کند.

تشخیص دهنده تصاویر واقعی را از طریق شبکه خود پردازش می کند و خروجی تشخیص دهنده را برای این تصاویر واقعی به دست می آورد. loss واقعی با استفاده از تابع loss متقاطع باینری(binary cross-entropy loss function) محاسبه می شود. این loss اندازه‌گیری می‌کند که تشخیص‌دهنده چقدر می‌تواند تصاویر واقعی را به عنوان واقعی طبقه‌بندی کند.

مولد نویز تصادفی (z) را به عنوان ورودی می گیرد و تصاویر جعلی (fake_images) را با استفاده از شبکه خود تولید می کند. تصاویر جعلی از طریق شبکه تشخیص دهنده منتقل می شوند، اما گرادیان ها نسبت به مولد محاسبه نمی شوند و گرادیان های تشخیص‌دهنده در طول انتشار پس زمینه(backpropagation) بروز می شوند.

همینطور برای fake loss نیز با استفاده از تابع loss متقاطع باینری محاسبه می شود. این loss اندازه‌گیری می‌کند که تشخیص‌دهنده چقدر می‌تواند تصاویر جعلی تولید شده توسط مولد را به عنوان جعلی طبقه‌بندی کند.

ضرر تشخیص‌دهنده(discriminator loss) مجموع loss واقعی و loss جعلی است. این loss کلی نشان‌دهنده عملکرد کلی تشخیص‌دهنده در تشخیص تصاویر واقعی و جعلی است.

گرادیان discriminator loss با توجه به پارامترهای تشخیص دهنده با استفاده از الگوریتم پس انتشار(backpropagation) محاسبه می شود. این گرادیان ها برای به روز رسانی پارامترهای تشخیص‌دهنده در طول بهینه سازی استفاده خواهند شد.

سپس از بهینه ساز (d_optimizer) برای به روز رسانی پارامترهای تشخیص دهنده بر اساس گرادیان های محاسبه شده استفاده می شود. این مرحله به تشخیص دهنده کمک می‌کند تا طبقه‌بندی بهتر تصاویر واقعی و جعلی را بیاموزد و تشخیص بین این دو را قوی‌تر می‌کند.

```
################ Problem 08 (10 pts) ################
# Calculate discriminator loss and update it
discriminator.zero_grad()

real_labels = torch.ones(images.size(0), 1).to(device)
fake_labels = torch.zeros(images.size(0), 1).to(device)

real_output = discriminator(images)
real_loss = F.binary_cross_entropy(real_output, real_labels)

fake_images = generator(z)
fake_output = discriminator(fake_images.detach())
fake_loss = F.binary_cross_entropy(fake_output, fake_labels)

d_loss = real_loss + fake_loss
d_loss.backward()
d_optimizer.step()

######################## End ########################
```

مولد با عبور دادن نویز تصادفی (z) از شبکه خود، تصاویر جعلی (fake_images) تولید می کند. سپس این تصاویر جعلی به تشخیص دهنده وارد می شوند تا خروجی تشخیص دهنده برای این تصاویر تولید شده به دست آید.

loss مولد با استفاده از تابع loss متقاطع باینری (F.binary_cross_entropy) محاسبه می شود. سپس برچسب‌های هدف(target) برای مولد روی برچسب‌های واقعی (real_labels) تنظیم می‌شوند، زیرا هدف مولد تولید تصاویری است که تشخیص دهنده را فریب می‌دهد تا آنها را به عنوان واقعی طبقه‌بندی کند.

در g_loss.backward گرادیان loss مولد با توجه به پارامترهای مولد با استفاده از پس انتشار (backpropagation) محاسبه می شود.

سپس از بهینه ساز (g_optimizer) برای به روز رسانی پارامترهای مولد بر اساس گرادیان های محاسبه شده استفاده می شود. این مرحله به مولد کمک می کند تا بیاموزد تصاویر واقعی تری تولید کند که بهتر می تواند تشخیص دهنده را فریب دهد.

```
################ Problem 09 (10 pts) ################
# Calculate generator loss and update it
generator.zero_grad()

fake_output = discriminator(fake_images)
g_loss = F.binary_cross_entropy(fake_output, real_labels)

g_loss.backward()
g_optimizer.step()

######################## End ########################
```

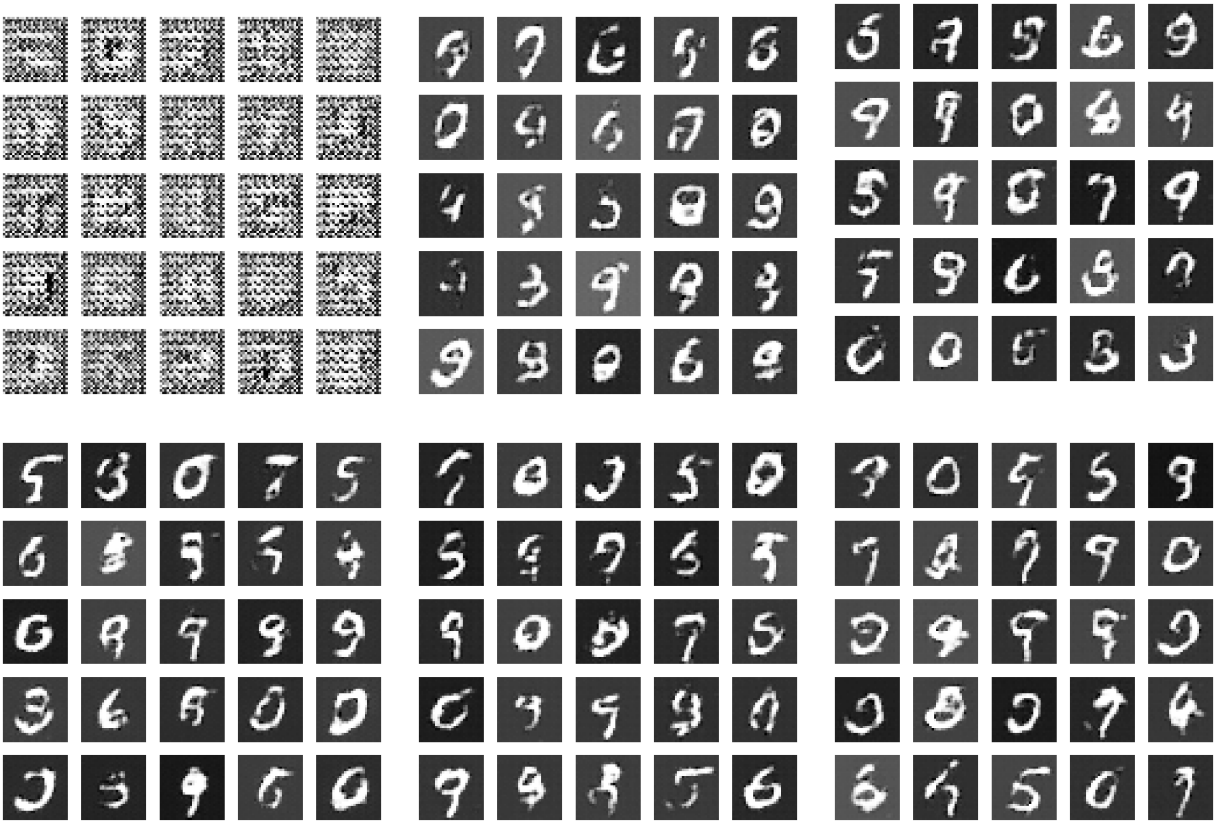
اگر epoch فعلی مضربی از plot_frequency باشد، مجموعه ای از تصاویر تولید شده با استفاده از matplotlib.pyplot.imshow رسم می شوند.

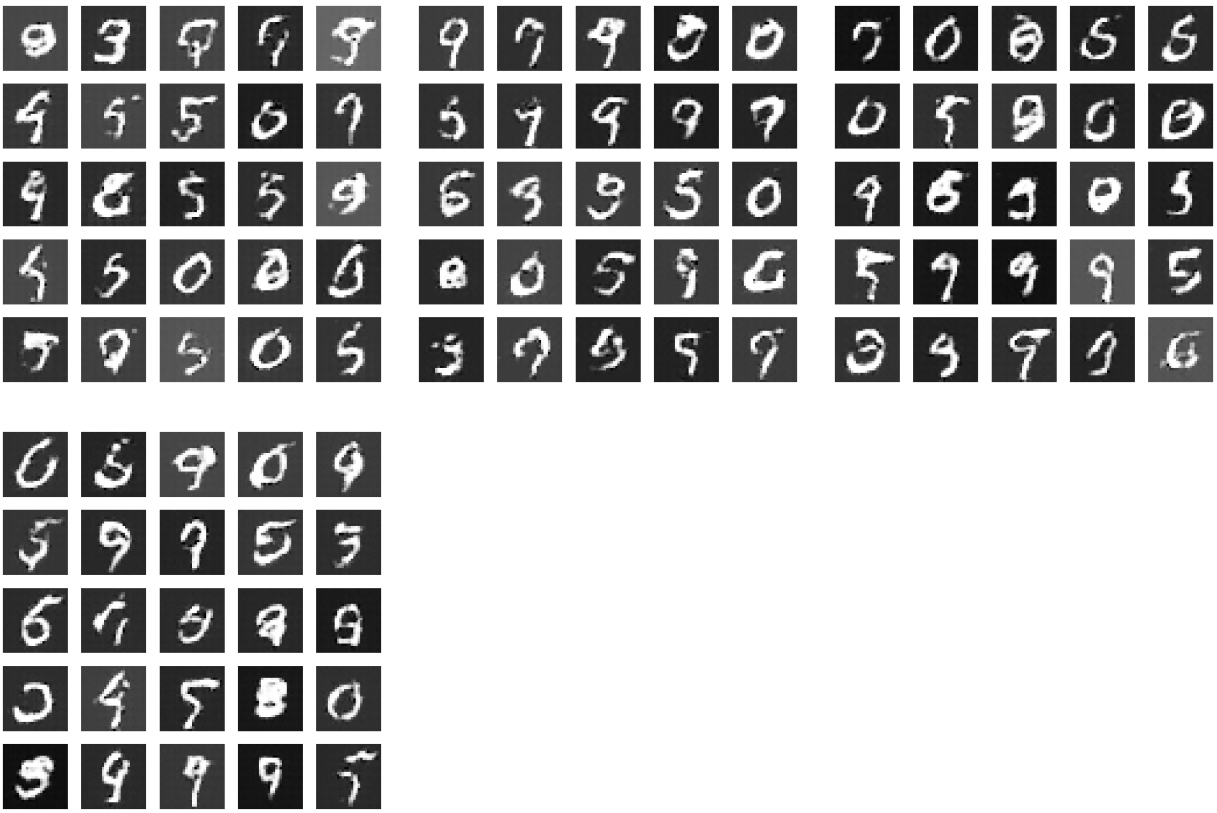
```
################ Problem 10 (10 pts) ################
# Plot some of the generated pictures based on plot frequency variable
if (epoch % plot_frequency == 0):
    generated_images = generator(torch.randn(25, z_dim).to(device)).detach()
    generated_images = generated_images.view(-1, 28, 28).cpu()

    plt.figure(figsize=(10, 10))
    for j in range(25):
        plt.subplot(5, 5, j+1)
        plt.imshow(generated_images[j], cmap='gray')
        plt.axis('off')
    plt.show()

######################## End ########################
```

- تصاویر ایجاد شده بعد از آموزش در هر ۱۰ epoch (plot_frequency) (به ترتیب از سمت چپ به راست) با **Learning rate** های تشخیص دهنده و مولد یکسان و برابر با **0.0002**:





همانطور که مشاهده می‌شود در ابتدا تنها تصویر نویزی‌ای ایجاد شده است و در هر ۱۰ epoch خروجی تصویر بهبود یافته است.

میزان loss تشخیص دهنده و مولد در epochهای برابر با [۰و ۱۰۰] با learning rate تشخیص دهنده و مولد یکسان و برابر با **0.0002**:

epoch: 1 discriminator last batch loss: 0.0029837926849722862 generator last batch loss: 7.605355262756348

epoch: 2 discriminator last batch loss: 0.01441839151084423 generator last batch loss: 6.070727348327637

epoch: 3 discriminator last batch loss: 0.0029650009237229824 generator last batch loss: 6.932927131652832

epoch: 4 discriminator last batch loss: 0.01589801535010338 generator last batch loss: 5.226722717285156

epoch: 5 discriminator last batch loss: 0.010050502605736256 generator last batch loss: 5.905343532562256

epoch: 6 discriminator last batch loss: 0.006498701870441437 generator last batch loss: 5.9208502769470215

epoch: 7 discriminator last batch loss: 0.004850407131016254 generator last batch loss: 6.64449405670166

epoch: 8 discriminator last batch loss: 0.003982572350651026 generator last batch loss: 6.7194366455078125

epoch: 9 discriminator last batch loss: 0.004345082677900791 generator last batch loss: 7.655231475830078

epoch: 10 discriminator last batch loss: 0.001680827233940363 generator last batch loss: 7.754276275634766

. . .

. . .

epoch: 90	discriminator last batch loss: 0.0007421550690196455	generator last batch loss: 12.20050048828125
epoch: 91	discriminator last batch loss: 0.00017420342192053795	generator last batch loss: 10.912463188171387
epoch: 92	discriminator last batch loss: 0.0003238495555706322	generator last batch loss: 9.659568786621094
epoch: 93	discriminator last batch loss: 9.881787991616875e-05	generator last batch loss: 10.53780746459961
epoch: 94	discriminator last batch loss: 0.0003507834335323423	generator last batch loss: 9.169036865234375
epoch: 95	discriminator last batch loss: 0.000907691428437829	generator last batch loss: 8.585773468017578
epoch: 96	discriminator last batch loss: 0.10910326987504959	generator last batch loss: 18.2074031829834
epoch: 97	discriminator last batch loss: 0.0017032496398314834	generator last batch loss: 8.342453002929688
epoch: 98	discriminator last batch loss: 0.001363197690807283	generator last batch loss: 8.479537010192871
epoch: 99	discriminator last batch loss: 0.0012668648269027472	generator last batch loss: 8.488889694213867
epoch: 100	discriminator last batch loss: 0.0003480447048787027 6	generator last batch loss: 11.688682556152344

برای رسم نمودار تغییرات loss (در هر plot_frequency تا epoch) مقادیر loss تشخیص دهنده و مولد را ذخیره می‌کنیم.

```
[26] plot_frequency = 10

     d_losses = []
     g_losses = []
```

```
    d_losses.append(d_loss.item())
    g_losses.append(g_loss.item())
```

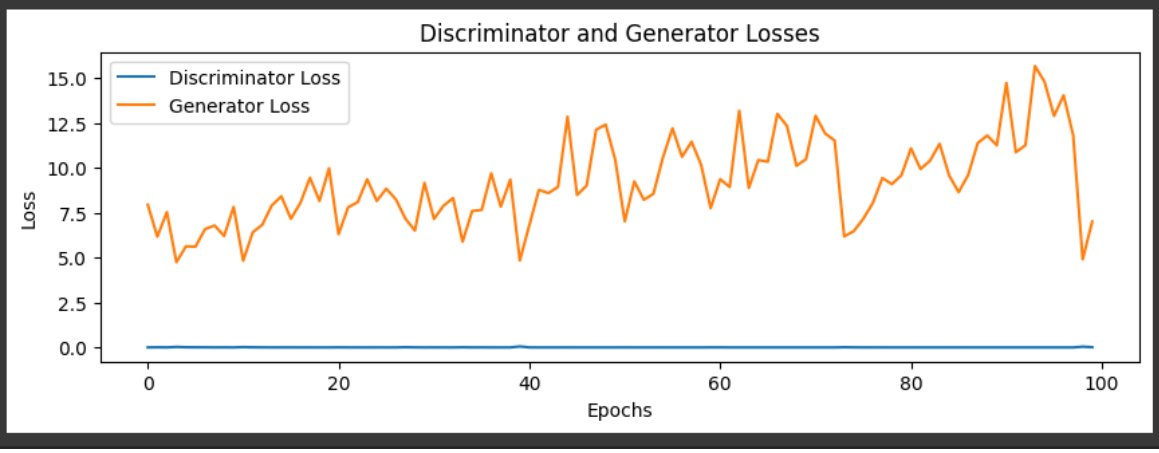
رسم نمودار تغییرات loss (در هر plot_frequency تا epoch) تشخیص دهنده و مولد

```python
import matplotlib.pyplot as plt

# Plot discriminator loss
plt.plot(range(n_epochs), d_losses, label='Discriminator Loss')

# Plot generator loss
plt.plot(range(n_epochs), g_losses, label='Generator Loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Discriminator and Generator Losses')
plt.legend()
plt.show()
```

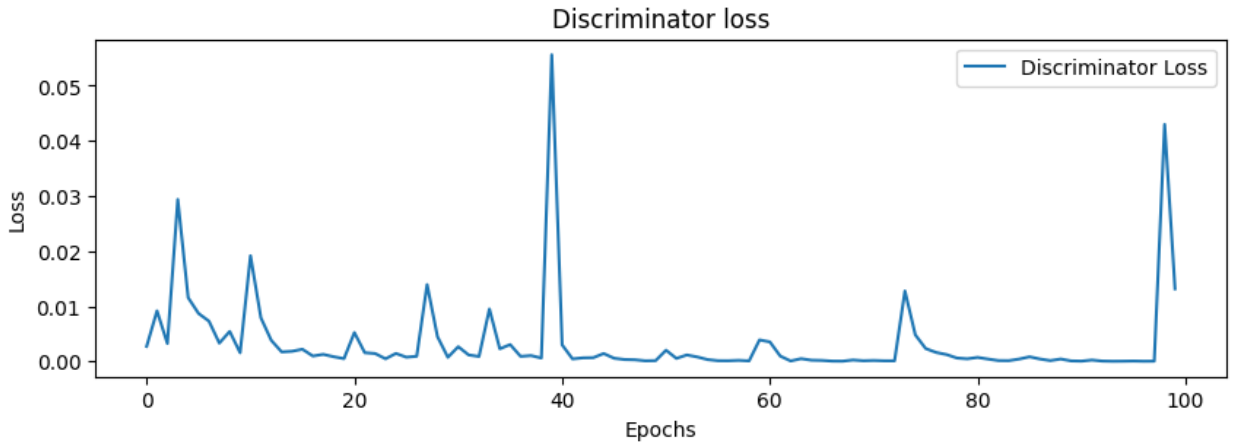


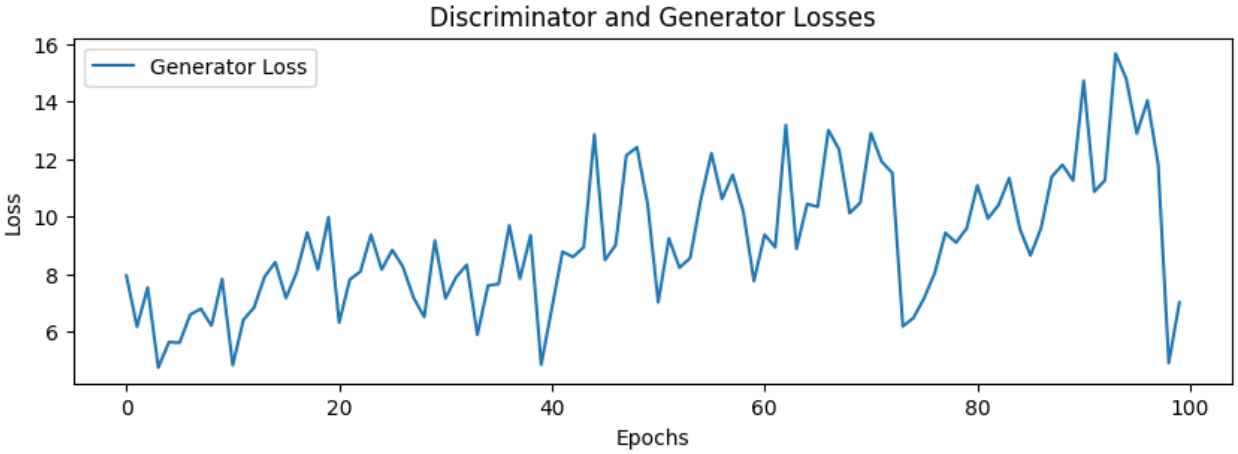
همانطور که دیده می‌شود با توجه به تغییرات کوچک loss تشخیص دهنده نسبت به مولد در نمودار دیده نمی‌شود لذا نمودار آن‌ها را به صورت جدا نیز رسم می‌کنیم.

```python
[28] import matplotlib.pyplot as plt

    # Plot discriminator loss
    plt.plot(range(n_epochs), d_losses, label='Discriminator Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Discriminator loss')
    plt.legend()
    plt.show()

    # Plot generator loss
    plt.plot(range(n_epochs), g_losses, label='Generator Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Discriminator and Generator Losses')
    plt.legend()
    plt.show()
```



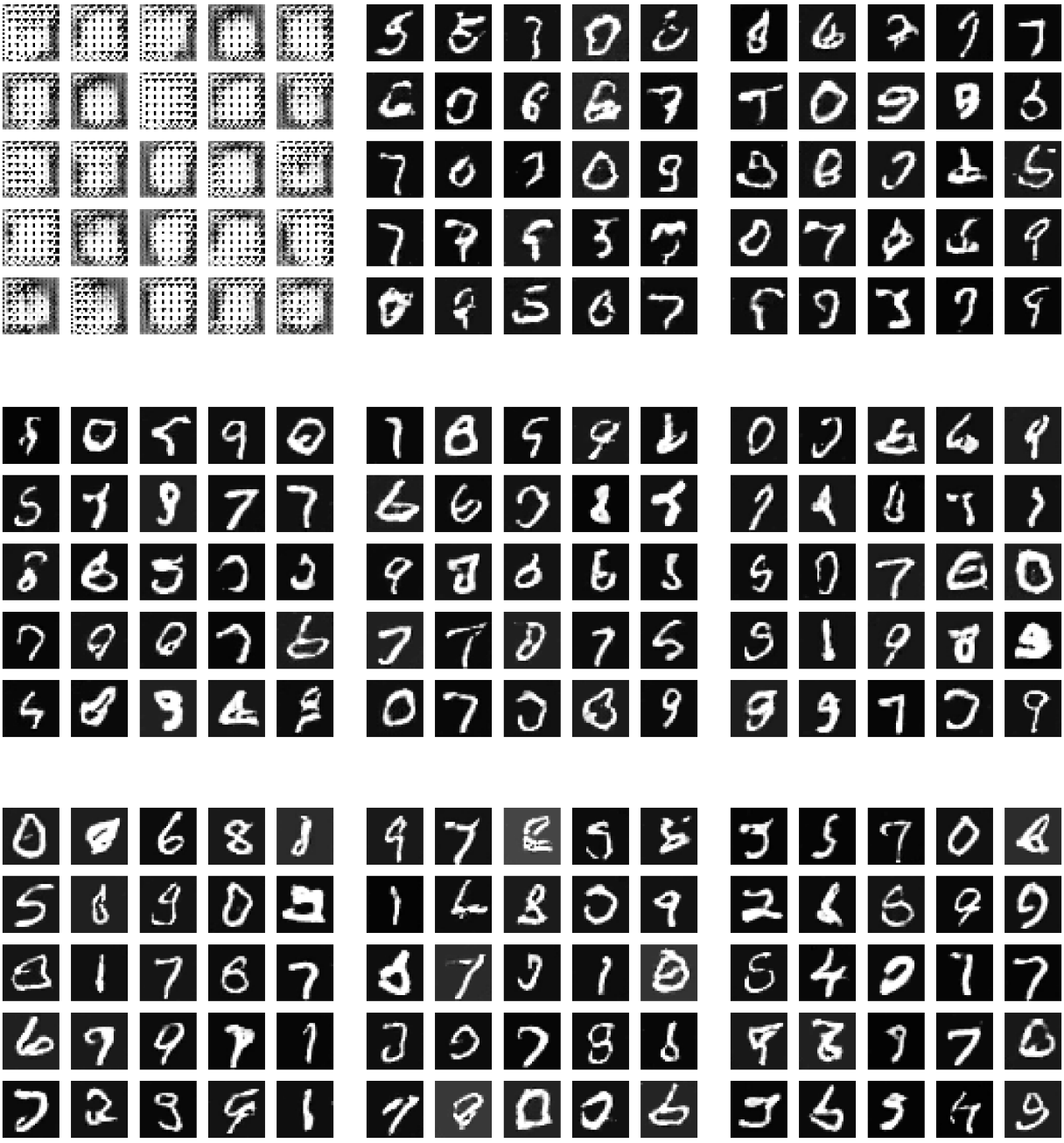


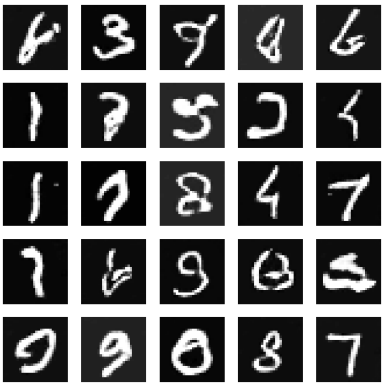
دلیل تغییرات و نوسانات lossها:

همانطور که در نمودارها و نیز در اعداد loss ذکر شده مشاهده می‌شود، loss تشخیص دهنده و مولد نسبت به یکدیگر (در اکثر اوقات) رابطه‌ی عکس دارند به این صورت که هنگامی loss تشخیص دهنده افزایش می‌یابد، loss مولد کاهش می‌یابد و برعکس؛ که این پدیده عادی می‌باشد زیرا با کم شدن loss یک طرف و در نتیجه قوی‌تر شدن آن، loss طرف دیگر کاهش می‌یابد. به همین دلیل loss ها نسبت به یکدیگر نوسان پیدا می‌کنند.

می دانیم تثبیت loss ها و نزدیک صفر شدن loss تشخیص دهنده می‌تواند معیار خوبی بر اساس loss ها و همینطور تولید خروجی واقعی و متنوع شبکه برای تعیین **همگرایی** باشد. لذا می‌توانیم با توجه به اعداد loss و نمودار آن‌ها و وضوح تصاویر ایجاد شده بگوییم که همگرایی رخ داده است.

- تصاویر ایجاد شده بعد از آموزش در هر ۱۰ epoch (plot_frequency) (به ترتیب از سمت چپ به راست) با **Learning rate** های تشخیص دهنده و مولد یکسان و برابر با **0.001**:





میزان loss تشخیص دهنده و مولد در epochهای برابر با [۰و ۱۰۰] با learning rate تشخیص دهنده و مولد یکسان و برابر با **0.001**:

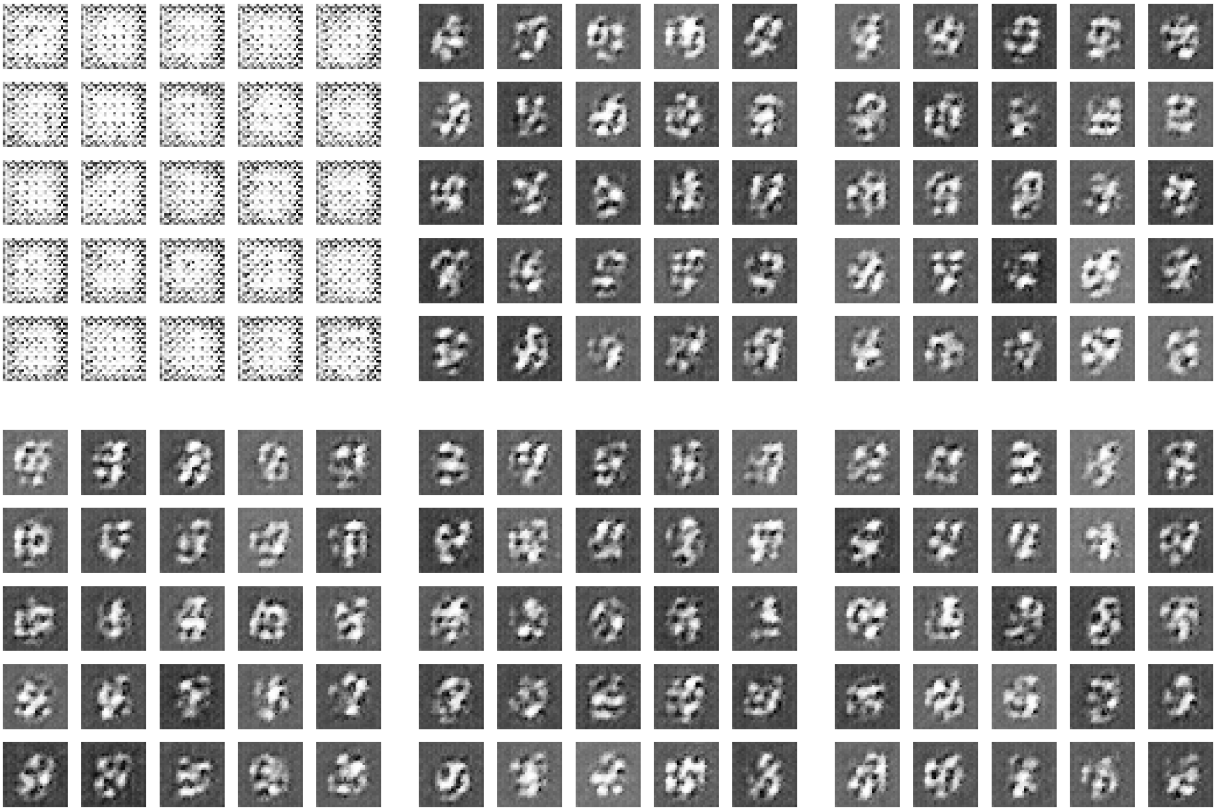
epoch: 1	discriminator last batch loss: 0.0005962662398815155	generator last batch loss: 9.784010887145996
epoch: 2	discriminator last batch loss: 0.008103284053504467	generator last batch loss: 7.220818996429443
epoch: 3	discriminator last batch loss: 0.00454923277720809	generator last batch loss: 7.22516393661499
epoch: 4	discriminator last batch loss: 0.021790826693177223	generator last batch loss: 4.214580059051514
epoch: 5	discriminator last batch loss: 0.031185060739517212	generator last batch loss: 5.012573719024658
epoch: 6	discriminator last batch loss: 0.14527946710586548	generator last batch loss: 2.305745840072632
epoch: 7	discriminator last batch loss: 0.02882670797407627	generator last batch loss: 5.040332794189453
epoch: 8	discriminator last batch loss: 0.07101012021303177	generator last batch loss: 5.164517402648926
epoch: 9	discriminator last batch loss: 0.052737489342689514	generator last batch loss: 5.025341033935547
.	.	.
.	.	.
.	.	.
epoch: 90	discriminator last batch loss: 0.035690173506736755	generator last batch loss: 7.571715831756592
epoch: 91	discriminator last batch loss: 0.05138183385133743	generator last batch loss: 4.915477275848389
epoch: 92	discriminator last batch loss: 0.24315106868743896	generator last batch loss: 8.246171951293945
epoch: 93	discriminator last batch loss: 0.17873889207839966	generator last batch loss: 4.784704208374023
epoch: 94	discriminator last batch loss: 0.22424335777759552	generator last batch loss: 4.954181671142578
epoch: 95	discriminator last batch loss: 0.03408059850335121	generator last batch loss: 5.355101585388184
epoch: 96	discriminator last batch loss: 0.08086640387773514	generator last batch loss: 7.032771110534668
epoch: 97	discriminator last batch loss: 0.07166945934295654	generator last batch loss: 5.621907711029053

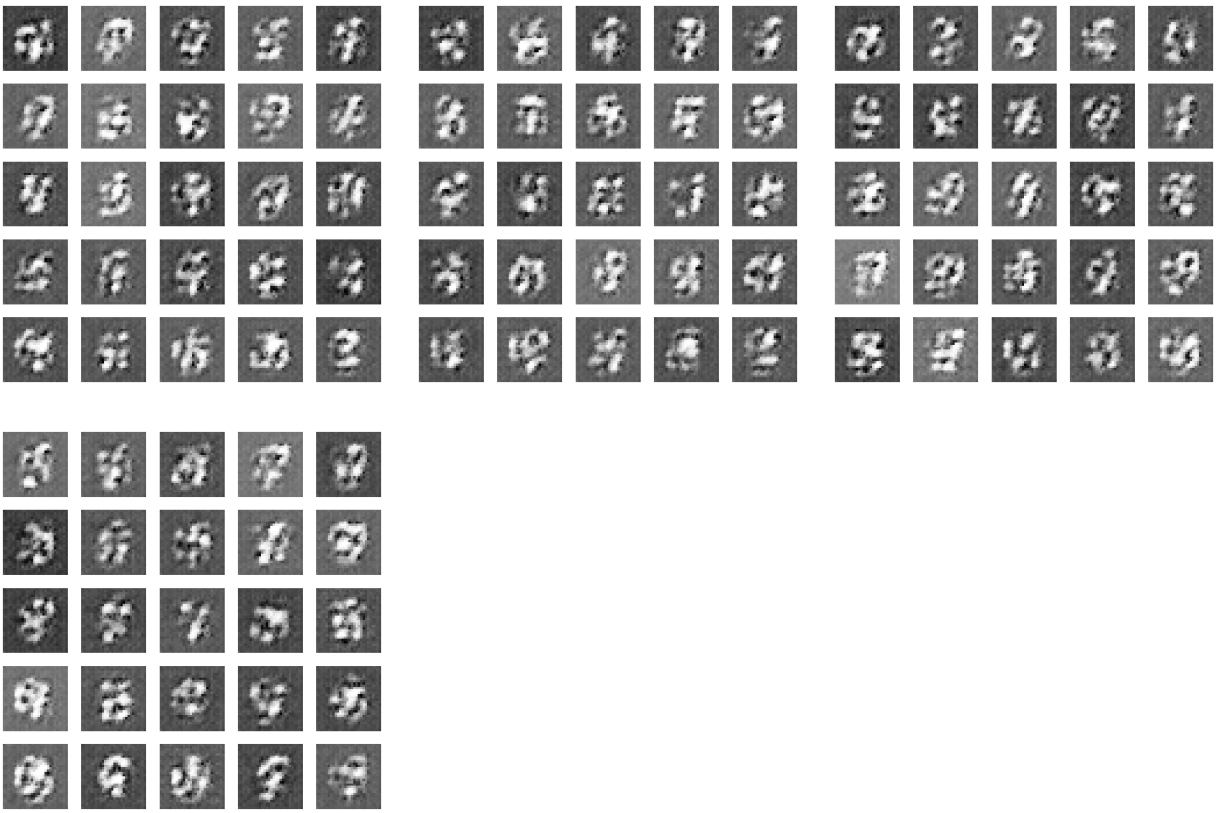
epoch: 98	discriminator last batch loss: 0.26175975799560547	generator last batch loss: 3.911574125289917
epoch: 99	discriminator last batch loss: 0.15268434584140778	generator last batch loss: 2.111323356628418
epoch: 100	discriminator last batch loss: 0.033834438771009445	generator last batch loss: 6.6198577880859375

با توجه به نکات ذکر شده در مورد همگرایی، می‌توانیم با توجه به اعداد loss و نمودار آن‌ها و وضوح تصاویر ایجاد شده بگوییم که همگرایی رخ داده است.

همچنین با افزایش learning rate سریع‌تر به وضوح تصاویر و تثبیت loss رسیدیم.

- تصاویر ایجاد شده بعد از آموزش در هر ۱۰ epoch (plot_frequency) (به ترتیب از سمت چپ به راست) با **Learning rate** تشخیص دهنده‌ی برابر با **0.001** و مولد یکسان و برابر با **0.0002**:





میزان loss تشخیص دهنده و مولد در epochهای برابر با [۰و ۱۰۰] با **Learning rate** تشخیص دهنده‌ی برابر با **0.001** و مولد یکسان و برابر با **0.0002**:

epoch: 1 discriminator last batch loss: 0.00015663070371374488 generator last batch loss: 10.217318534851074

epoch: 2 discriminator last batch loss: 0.0009905232582241297 generator last batch loss: 8.805638313293457

epoch: 3 discriminator last batch loss: 0.00028909792308695614 generator last batch loss: 9.354217529296875

epoch: 4 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 5 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 6 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 7 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 8 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 9 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 10 discriminator last batch loss: 100.0 generator last batch loss: 0.0

. . .

. . .

. . .

epoch: 91 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 92 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 93 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 94 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 95 discriminator last batch loss: 100.0 generator last batch loss: 0.0

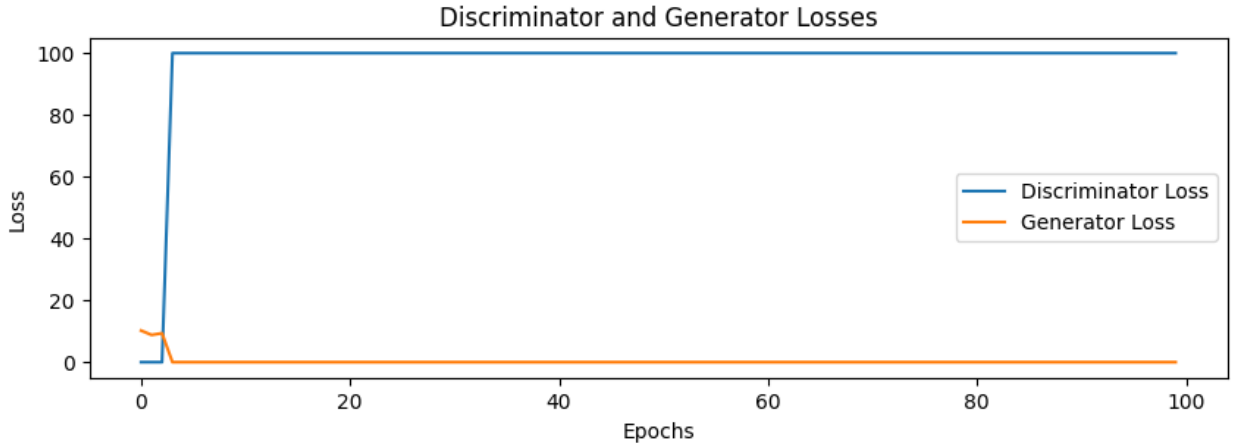
epoch: 96 discriminator last batch loss: 100.0 generator last batch loss: 0.0

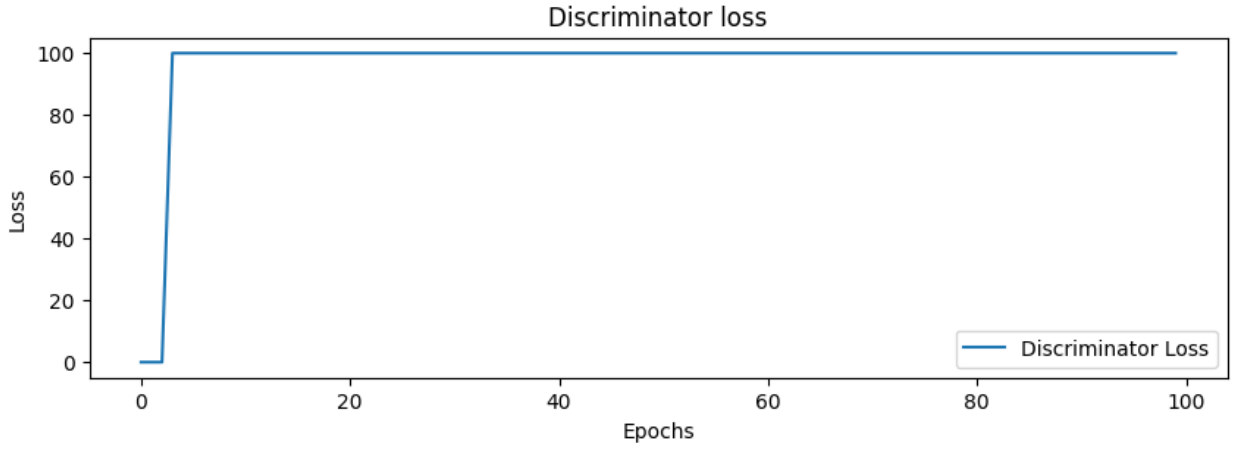
epoch: 97 discriminator last batch loss: 100.0 generator last batch loss: 0.0

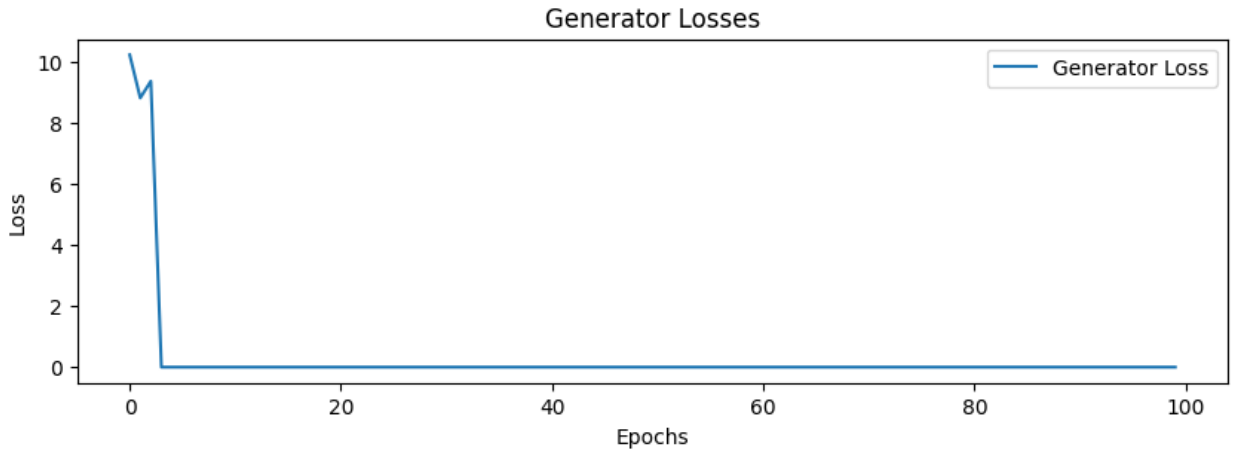
epoch: 98 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 99 discriminator last batch loss: 100.0 generator last batch loss: 0.0

epoch: 100 discriminator last batch loss: 100.0 generator last batch loss: 0.0

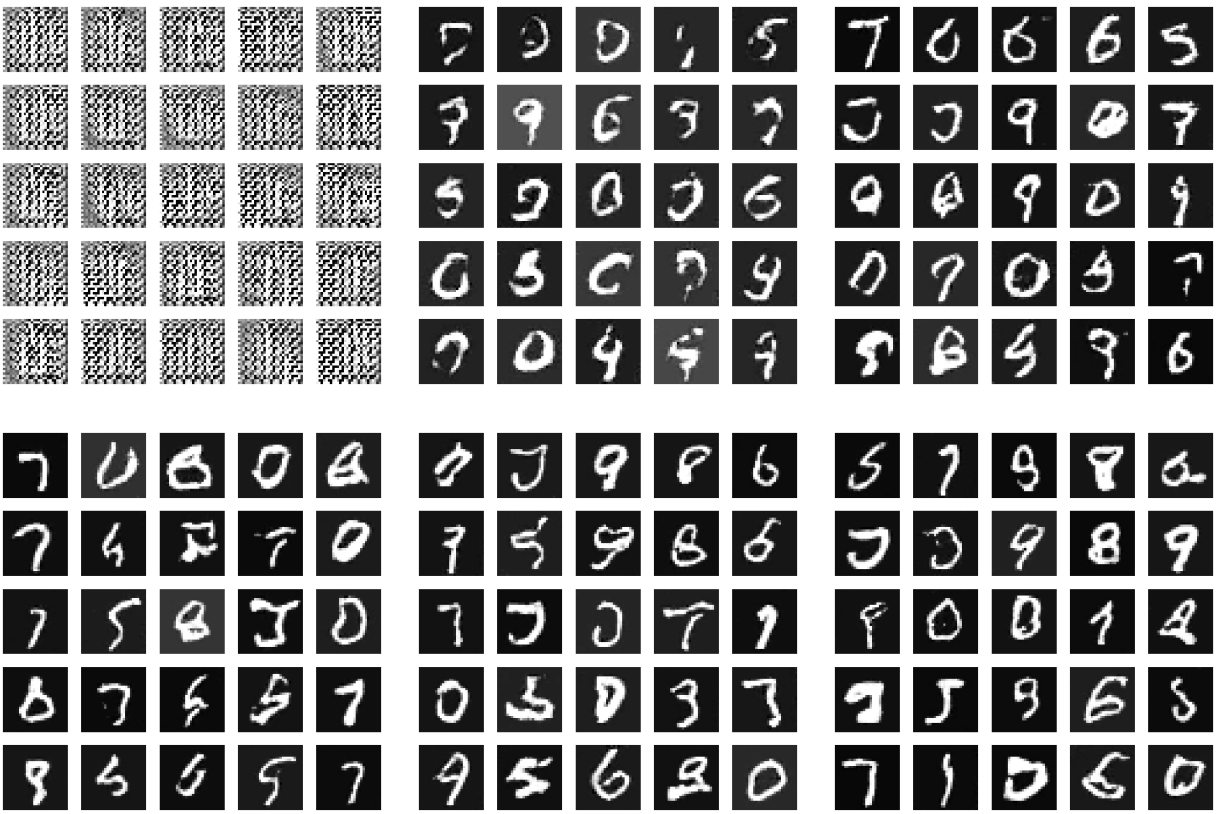


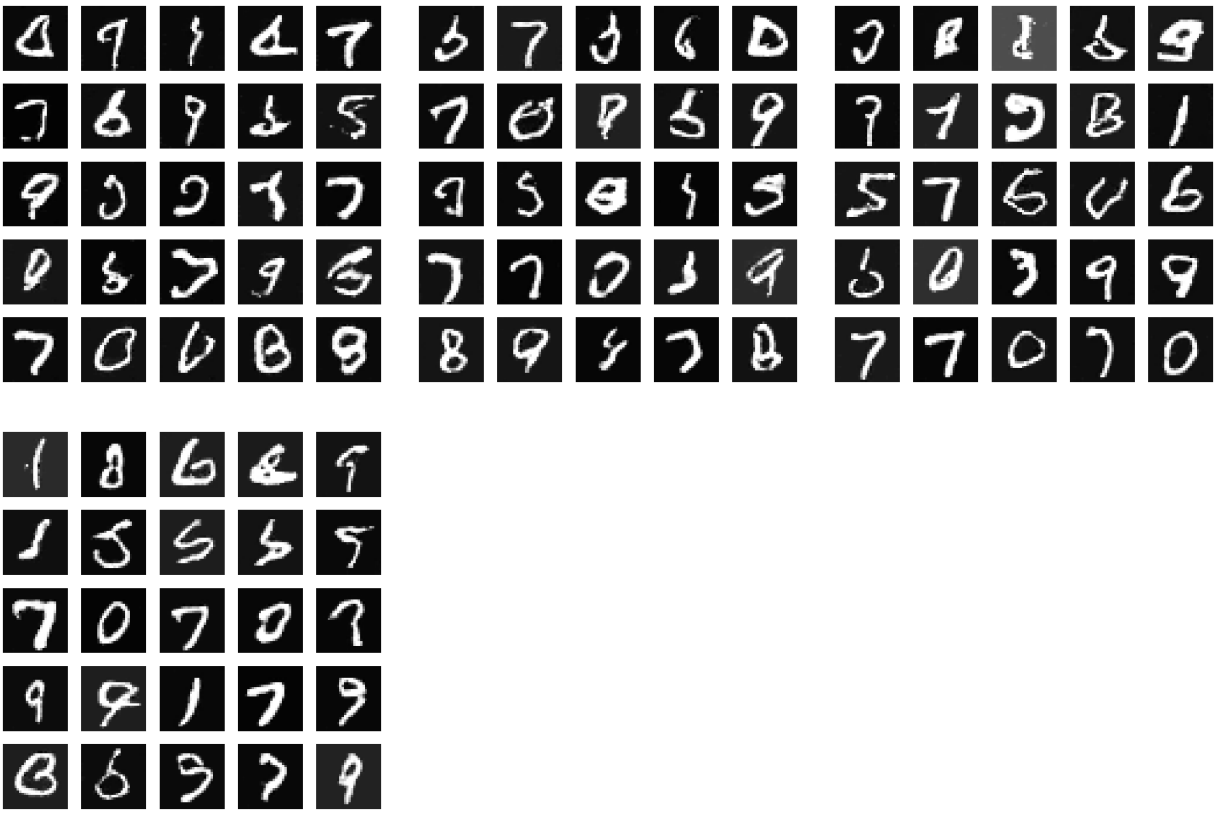




همانطور که مشاهده می‌شود با توجه به اینکه نرخ یادگیری تشخیص دهنده بزرگتر از مولد بود (حدودا ۱۰ برابر) منجر به این شد که تشخیص دهنده سریع‌تر از مولد قوی شده و مولد نتواند با توجه به بازخوردهای تشخیص دهنده آموزش ببیند.

- تصاویر ایجاد شده بعد از آموزش در هر ۱۰ epoch (plot_frequency) (به ترتیب از سمت چپ به راست) با **Learning rate** تشخیص دهنده‌ی برابر با **0.0002** و مولد یکسان و برابر با **0.001**:





میزان loss تشخیص دهنده و مولد در epochهای برابر با [۰و ۱۰۰] با Learning rate تشخیص دهنده‌ی برابر با 0.0002 و مولد یکسان و برابر با 0.001:

epoch: 1 discriminator last batch loss: 0.004210371058434248 generator last batch loss: 13.798469543457031

epoch: 2 discriminator last batch loss: 0.08711402118206024 generator last batch loss: 5.0494384765625

epoch: 3 discriminator last batch loss: 0.020572438836097717 generator last batch loss: 4.847954750061035

epoch: 4 discriminator last batch loss: 0.016974596306681633 generator last batch loss: 6.314212322235107

epoch: 5 discriminator last batch loss: 0.04944019019603729 generator last batch loss: 4.624159812927246

epoch: 6 discriminator last batch loss: 0.028470326215028763 generator last batch loss: 5.271955490112305

epoch: 7 discriminator last batch loss: 0.014987034723162651 generator last batch loss: 5.45535945892334

epoch: 8 discriminator last batch loss: 0.010741319507360458 generator last batch loss: 6.404854774475098

epoch: 9 discriminator last batch loss: 0.013831131160259247 generator last batch loss: 5.4231390953063965

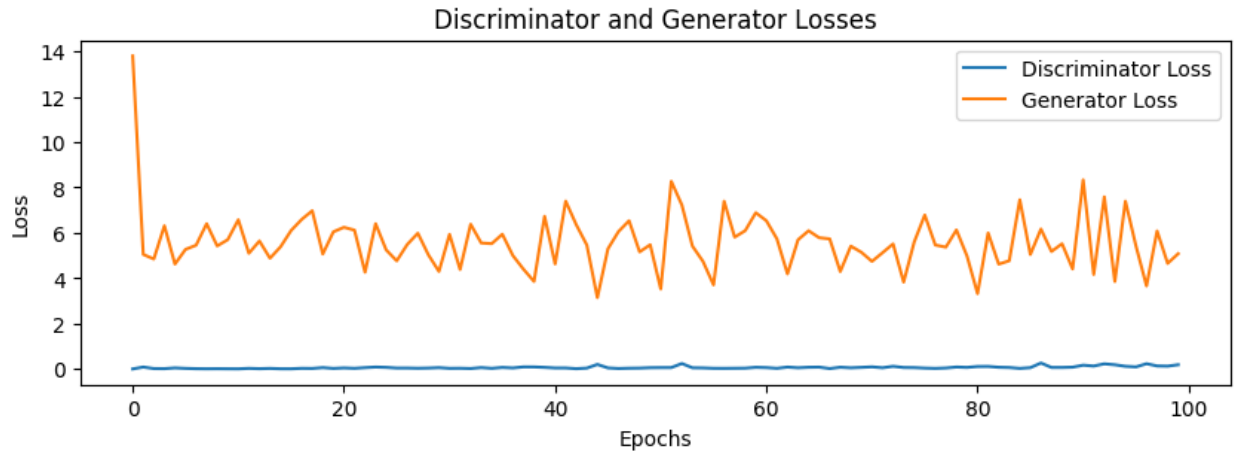
epoch: 10 discriminator last batch loss: 0.010894648730754852 generator last batch loss: 5.700761795043945

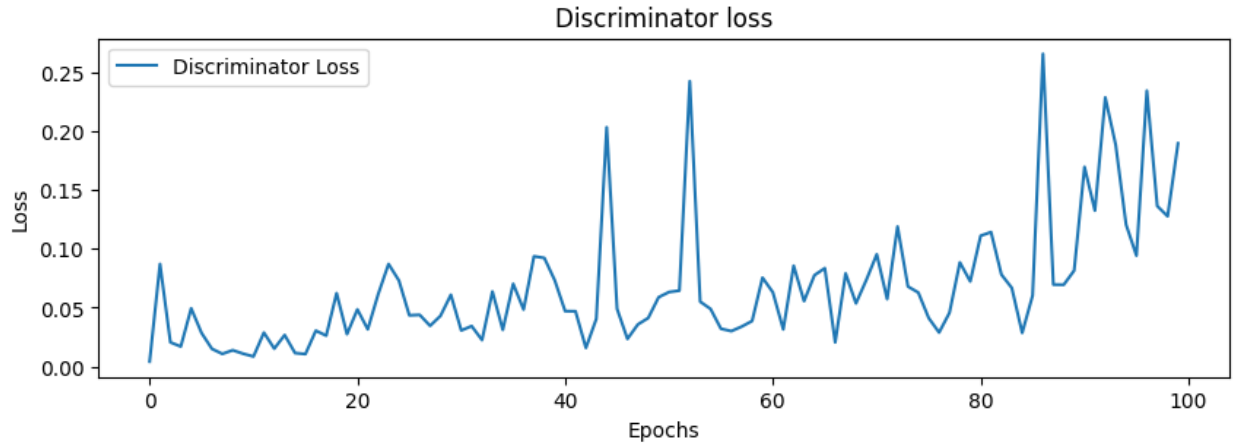
. . .

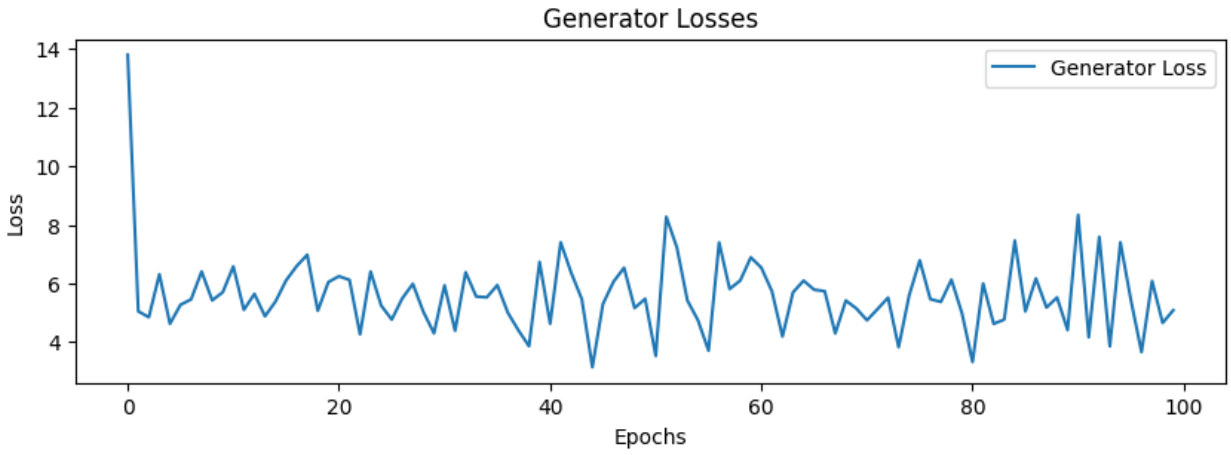
. . .

. . .

epoch: 91	discriminator last batch loss: 0.16961129009723663	generator last batch loss: 8.337634086608887
epoch: 92	discriminator last batch loss: 0.13247667253017426	generator last batch loss: 4.166770935058594
epoch: 93	discriminator last batch loss: 0.22864747047424316	generator last batch loss: 7.5879974365234375
epoch: 94	discriminator last batch loss: 0.18839097023010254	generator last batch loss: 3.8576173782348633
epoch: 95	discriminator last batch loss: 0.12063232064247131	generator last batch loss: 7.400165557861328
epoch: 96	discriminator last batch loss: 0.09419306367635727	generator last batch loss: 5.422227382659912
epoch: 97	discriminator last batch loss: 0.2344161719083786	generator last batch loss: 3.6592350006103516
epoch: 98	discriminator last batch loss: 0.136379212141037	generator last batch loss: 6.08396053314209
epoch: 99	discriminator last batch loss: 0.12766578793525696	generator last batch loss: 4.655241012573242
epoch: 100	discriminator last batch loss: 0.18963414430618286	generator last batch loss: 5.08441162109375



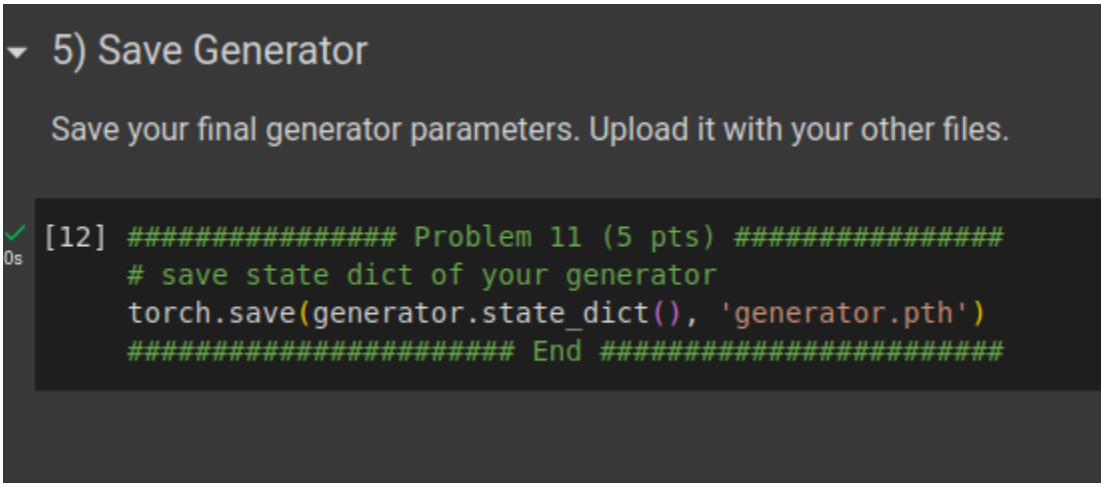


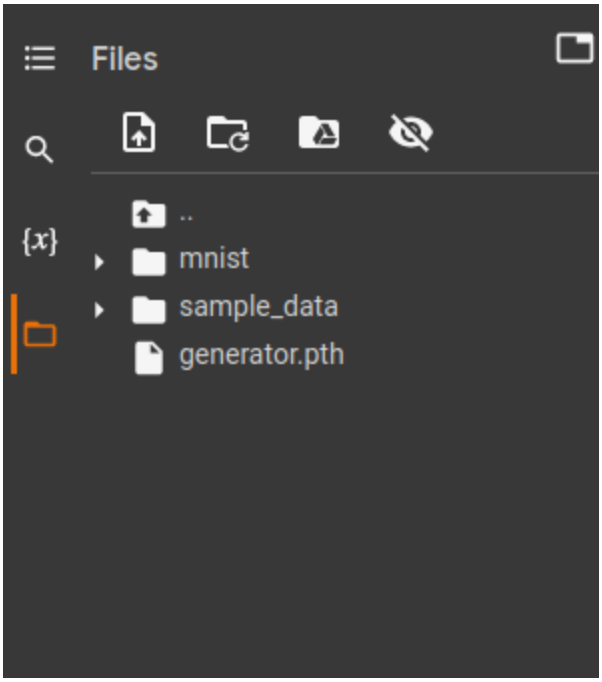


همانطور که مشاهده می‌شود با توجه به اینکه نرخ یادگیری مولد بزرگتر از تشخیص دهنده بود (حدودا ۱۰ برابر) منجر به این شد که loss تشخیص دهنده نسبت به حالت های قبل با learning rate برابر افزایش پیدا کند؛ اما با این وجود loss ها حدودا در یک محدوده قرار دارند و تصاویر تقریبا واضح هستند؛ لذا می‌توان گفت همگرایی نسبی‌ای رخ داده است.

لذا تغییر نرخ یادگیری باعث تغییر خروجی شبکه، میزان همگرایی، تثبیت loss ها و سرعت تثبیت آن‌ها می‌شود.

در این کد، از تابع torch.save() برای ذخیره دیکشنری حالت مولد استفاده شده است. دیکشنری حالت شامل تمام پارامترهای قابل یادگیری مدل مولد است. فایل generator.pth نام فایل ذخیره شده است. (فایل pth یک فرمت فایل است که در PyTorch برای ذخیره و ذخیره پارامترهای مدل استفاده می شود. مخفف "PyTorch Model Checkpoint" یا "PyTorch Model Parameters" است.)




Files
..
mnist
sample_data
generator.pth