ANN实验报告4

> 张天祺 2021010719 计12

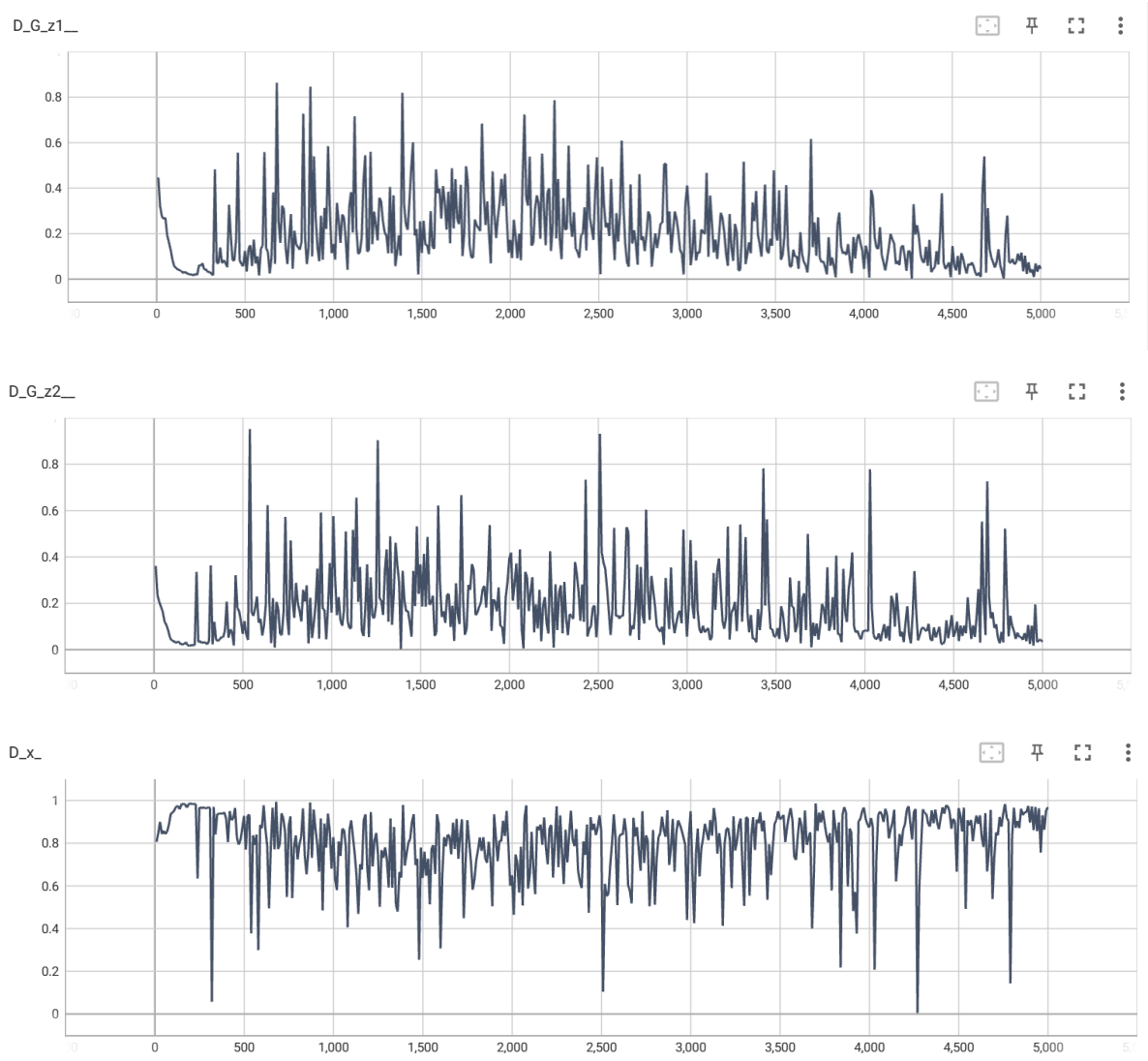
按照默认参数训练，调整`latent_dim`和`hidden_dim`

保持其他训练参数为默认设置，分别调整`latent_dim`和`hidden_dim`参数为16, 100。共做四次实验：

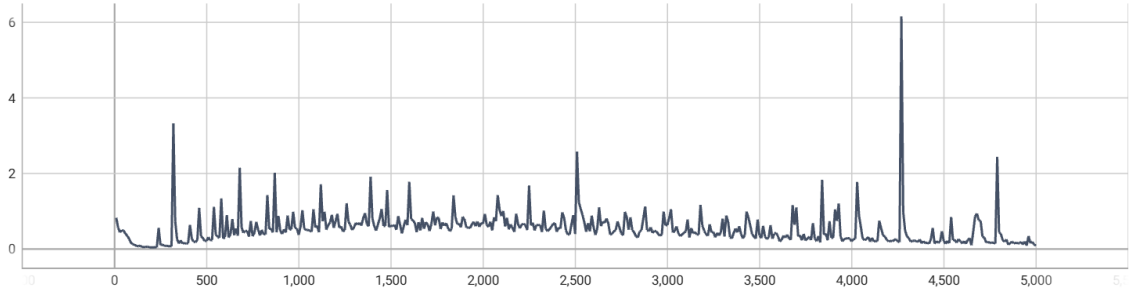
以下实验报告中模型的命名方式为`{latent_dim}_{hidden_dim}`

模型训练曲线如下：

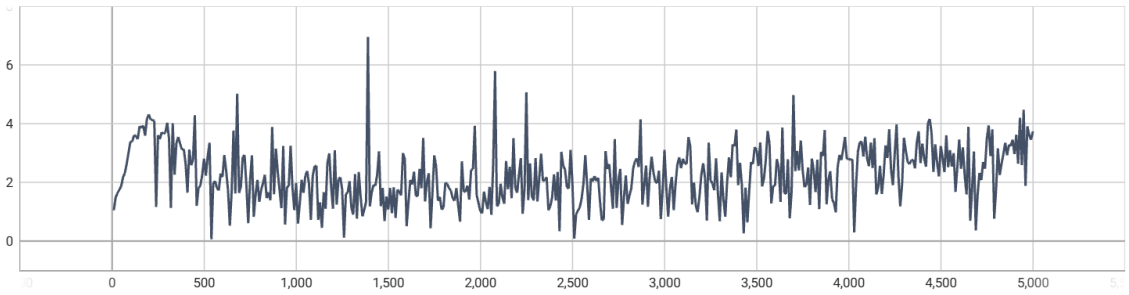
16_16



discriminator_loss

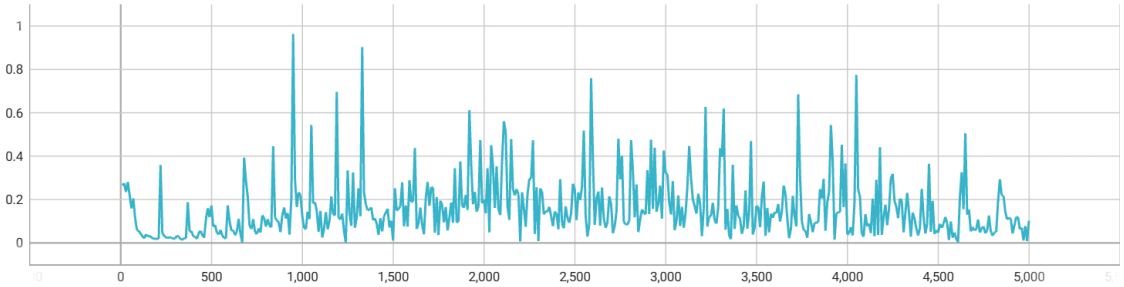


generator_loss

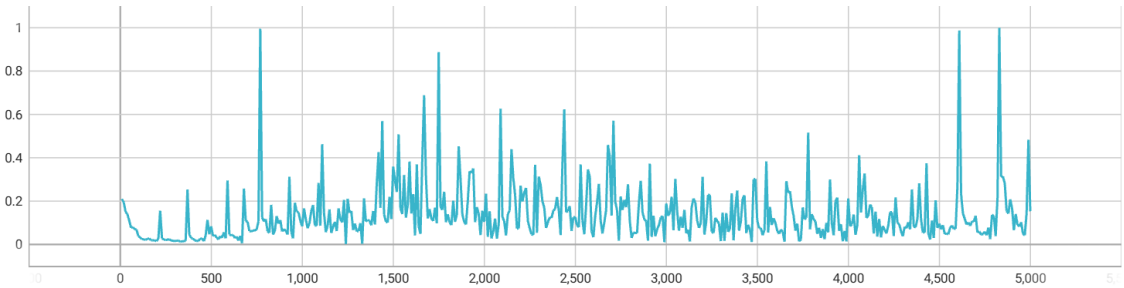


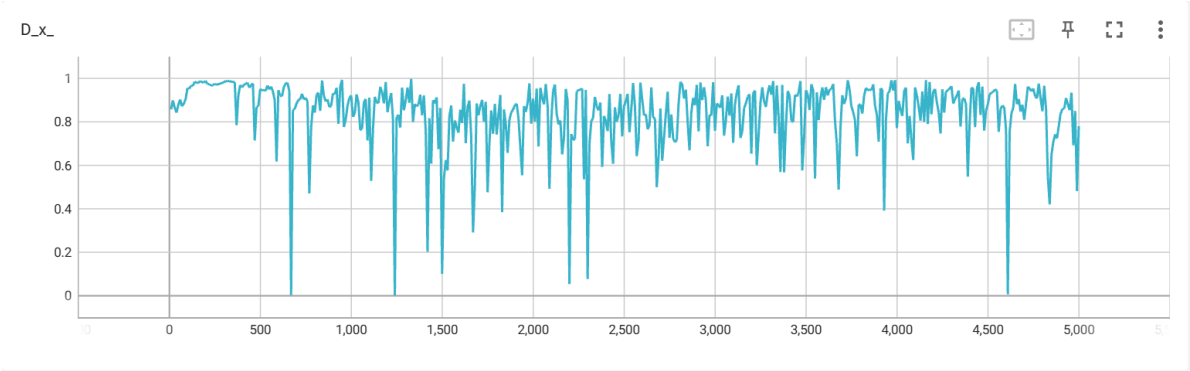
100_16

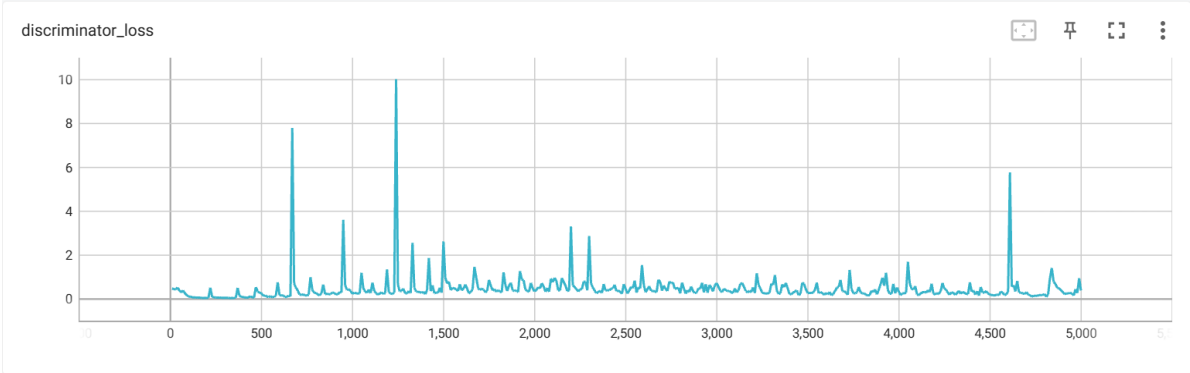
D_G_z1__

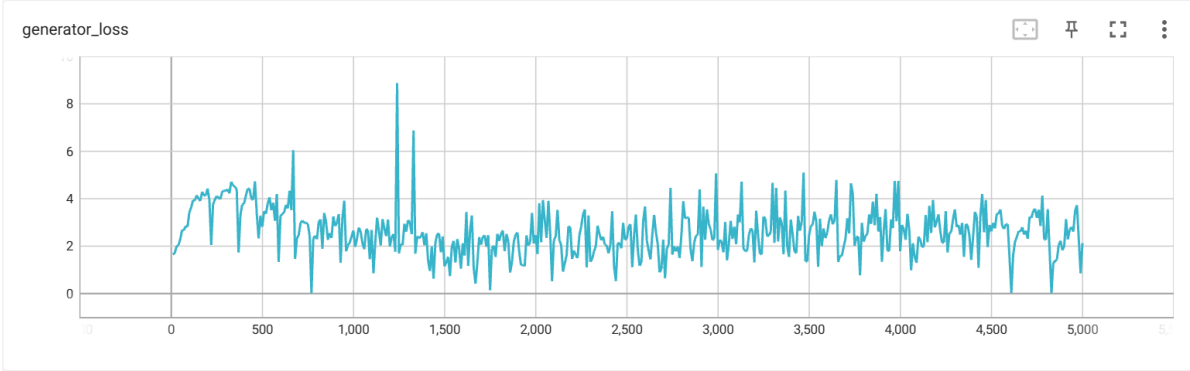


D_G_z2__

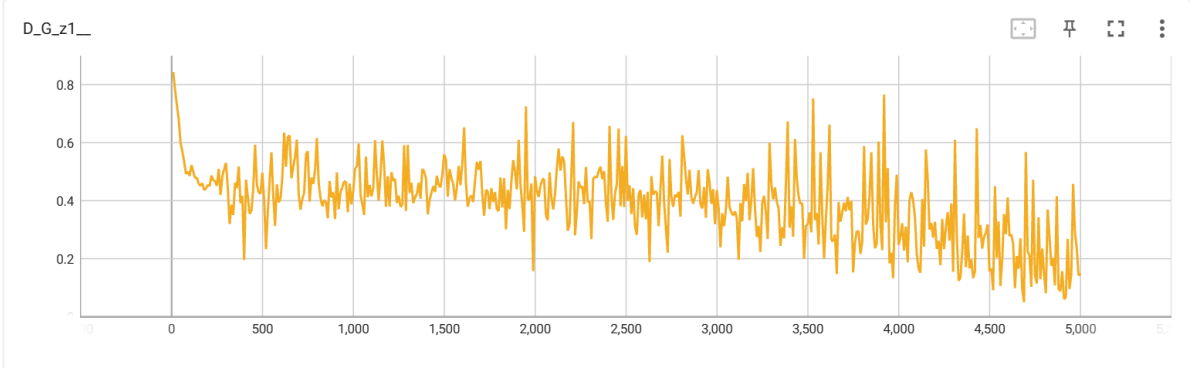



D_x_
1
0.8
0.6
0.4
0.2
0
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000

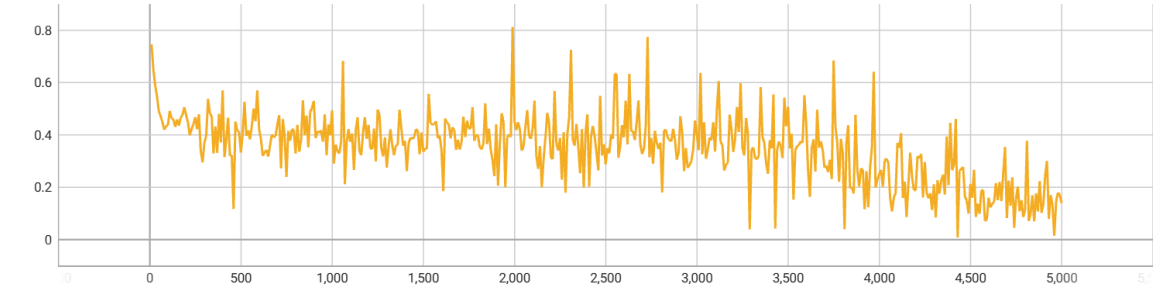

discriminator_loss
10
8
6
4
2
0
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


generator_loss
8
6
4
2
0
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000

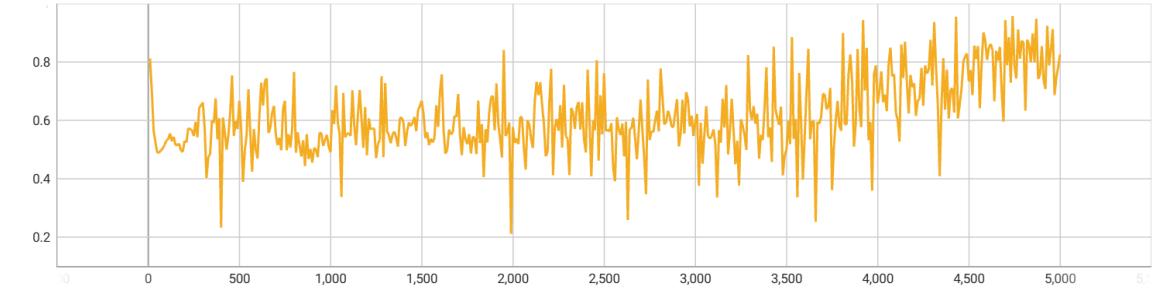
16_100


D_G_z1__
0.8
0.6
0.4
0.2
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000

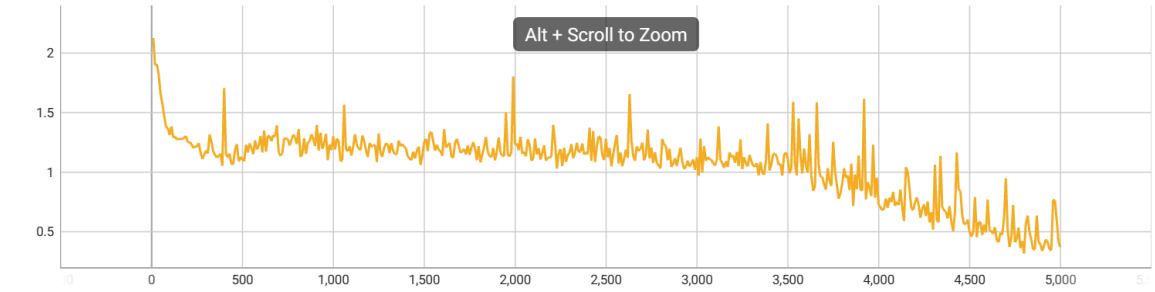
D_G_z2__



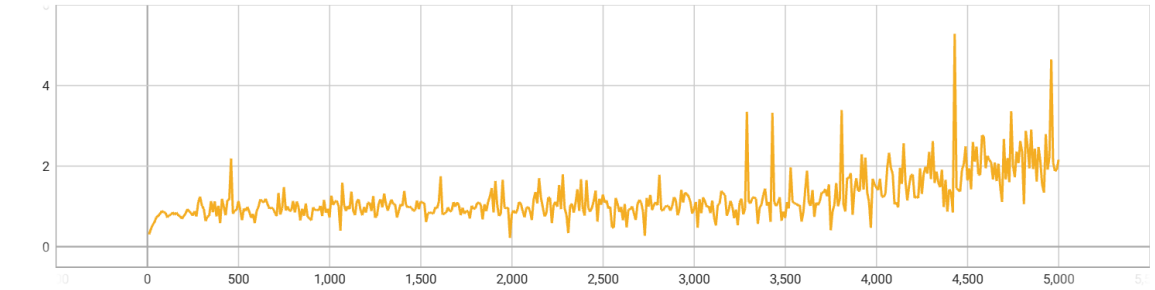
D_x_



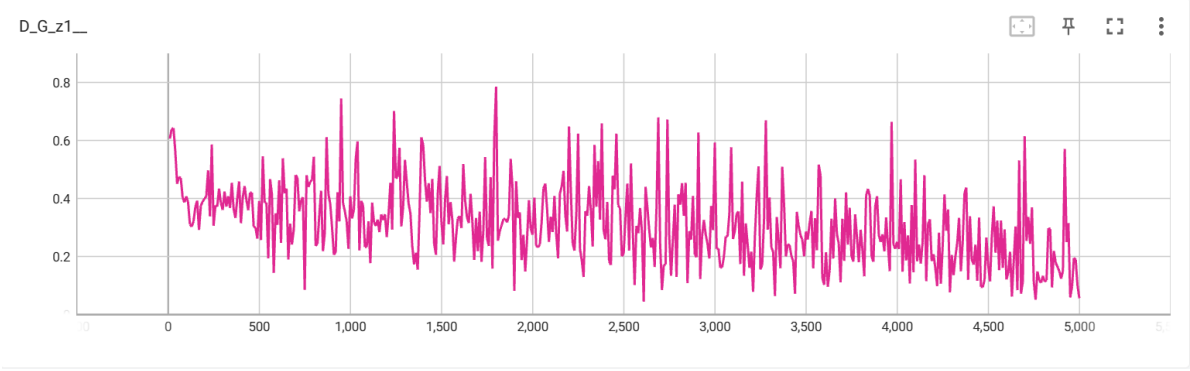
discriminator_loss

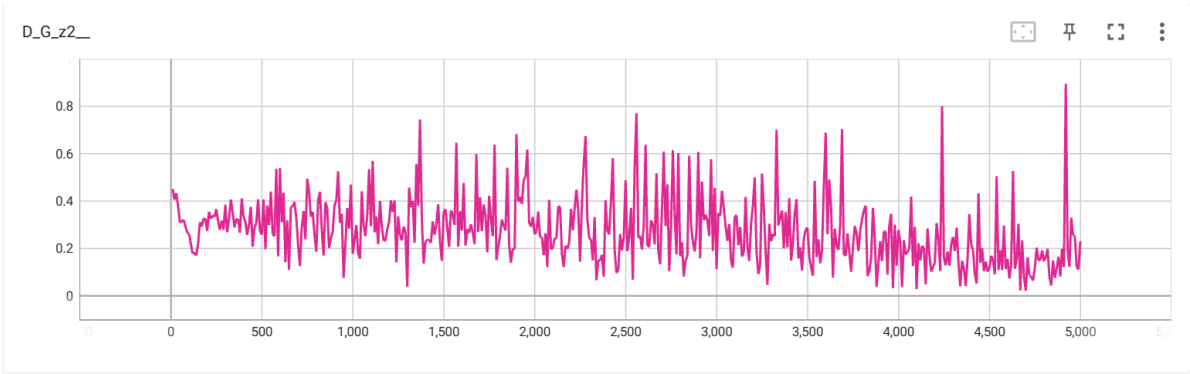


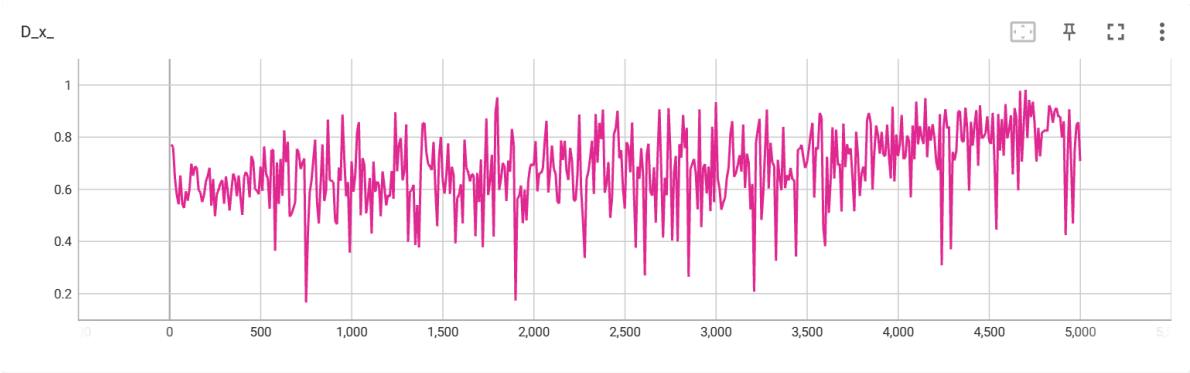
generator_loss

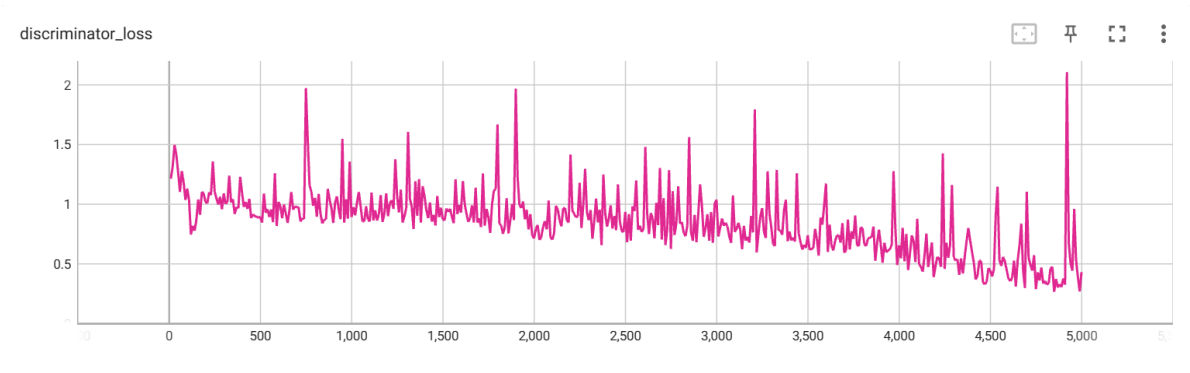


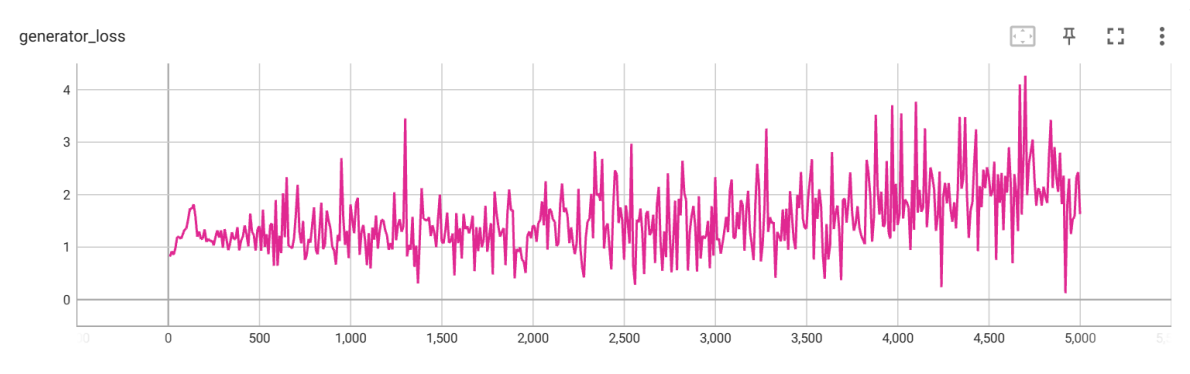
100_100

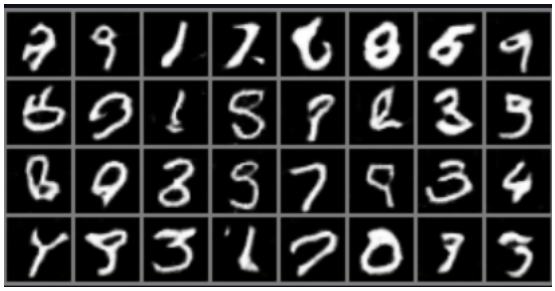

D_G_z1__
0.8
0.6
0.4
0.2
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


D_G_z2__
0.8
0.6
0.4
0.2
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


D_x_
1
0.8
0.6
0.4
0.2
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


discriminator_loss
2
1.5
1
0.5
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000


generator_loss
4
3
2
1
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000



FID结果

model_config	FID score
16_16	101.892
100_16	101.226
16_100	36.538
100_100	**33.350**

值得一提的是，实际每次训练得到图片的FID波动较大。

hidden_dim和latent_dim的影响

从上面的图像和FID指标来看，`hidden_dim`增大对于提高训练的效果是有直接帮助的。由于增大`hidden_dim`可以说直接增大了模型的表现能力，提升了generator的能力，也因此会提高discriminator的判别能力，因此获得最低的FID的结果是意料之中的。

而增大`latent_dim`对于训练效果的影响并不显著。由于增大`hidden_dim`的结果是采样空间变大，更大的`hidden_dim`相当于更加随机的噪声，对于generator的生成能力并没有提升。

从两种图像的对比来看：

提高`hidden_dim`会让generator_loss降低，discriminator_loss升高，也是generator生成图片质量升高，能符合训练集数据分布，使得discriminator更难做出判别的表现。

而提高`latent_dim`并没有对generator_loss产生明显的影响。这也就意味着引入更大的随机噪声对于generator并不一定有好的效果。

纳什均衡

达到纳什均衡的条件是生成空间分布与训练空间分布相同，也就是`_netD(fake_imgs) = _netD(real_imgs) = 0.5`，而最终的结果是generator输出的图像完全能够以假乱真，而discriminator的判别结果完全随机输出。但是观察`D_x`和`D_G_z1`的结果并没有收敛到0.5，观察`discriminator_loss`和`generator_loss`也都在大幅度波动，因此可以很明显看出这个GAN的训练并没有达到纳什均衡。

对于使GAN的训练达到纳什均衡，可以有添加梯度惩罚、调整learning rate、使用batch normalization等方法来帮助实现。而本次实验中基本没有使用优化方法，随意没有达到纳什均衡是正常的事情。

线性插值

选择的最佳GAN：**100_100**

使用如下代码进行线性插值：

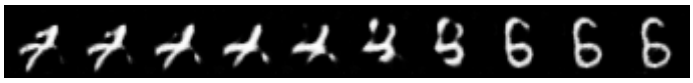
```
def interpolate_latent_space(generator, latent_size, num_steps=10,
device='cuda'):
    # 生成两个随机潜在向量
    z1 = torch.randn(1, latent_size, 1, 1, device=device)
    z2 = torch.randn(1, latent_size, 1, 1, device=device)

    # 在两个潜在向量之间进行线性插值
    alpha_values = torch.linspace(0, 1, num_steps, device=device)
    interpolated_z = z1 * (1 - alpha_values.view(-1, 1, 1, 1)) + z2 *
alpha_values.view(-1, 1, 1, 1)

    # 将生成器置于评估模式，并生成插值图像
    generator.eval()
    with torch.no_grad():
        interpolated_images = generator(interpolated_z)

    # 保存插值图像
    save_image(interpolated_images, 'results/interpolation.png', nrow=num_steps,
normalize=True)
```

```
generator = GAN.Generator(1, 100, 100).to('cuda')
checkpoint = torch.load('results/latent-100_hidden-100_batch-64_num-train-steps-5000/4999/generator.bin', map_location=torch.device('cuda'))
generator.load_state_dict(checkpoint)
generator.eval()
latent_size = 100
interpolate_latent_space(generator, latent_size)
```

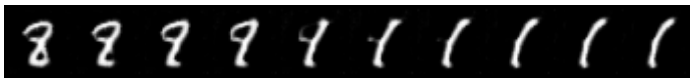
得到的结果：

> 0到9

> 4到6



> 8到1



> 4到1

> 9到3

可以看出，模型生成图片的过渡还算平滑，可以看出从一个数字到另外一个数字的逐渐变化，线性插值的方法总体来看表现不错。

对于模型的生成质量而言，由于原始训练得出的模型生成的质量本就一般，因此在插值中生成图片的质量可以接受。

Mode Collapse

模式崩溃（mode collapse）是GAN训练中存在的一个问题，指的是generator倾向于生成相似或者相同的样本，而不是覆盖数据分布的多个模式。在模式崩溃的情况下，生成器可能会丧失对数据分布中多个模式的表达能力，导致生成的样本缺乏多样性。

随机生成50个样本：

每个样本分别是数字：9、0、9、6、9、6、7、2、8、8、4、0、7、9、6、8、1、6、3、4、0、3、8、3、7、0、6、6、9、4、3、3、8、4、9、9、6、6、8、1、0、4、9、8、4、1、1、1、7、5

统计：

	0	1	2	3	4	5	6	7	8	9
出现次数	5	5	1	5	6	1	8	4	7	8

出现6、8、9的概率较高而出现2、5的概率较低，因此可以判断出现了一定的模式崩溃。

对此，我认为是generator学习到生成更多的圆圈、折线等，同时6、8、9本身图像的分布就比较接近，都是上下结构而且都存在圆圈或者半圆，因此generator将生成空间拟合到比较靠近6、8、9的数据分布的子集部分，因此生成这些数字的概率较高。反之，生成2，5等与6、8、9分布不相似的数字的概率就会更低。同时，对于5这种较难生成的数字（需要两笔写成），generator在生成后往往会被discriminator判断为false，因此可能会导致generator学到了躲避生成这些数字来对抗discriminator，因此导致最终这些数字的生成概率很低。

Ablation

MLP based GAN

修改模型中部分代码以使用mlp based GAN，具体修改如下：

```
class Generator(nn.Module):
    def __init__(self, num_channels, latent_dim, hidden_dim):
        super(Generator, self).__init__()
        self.num_channels = num_channels
        self.hidden_dim = hidden_dim
        self.latent_dim = latent_dim

        # TODO START
        self.decoder = nn.Sequential(
            # nn.ConvTranspose2d(in_channels=latent_dim,
out_channels=hidden_dim*4, kernel_size=4, stride=1, padding=0),
            # nn.BatchNorm2d(num_features=hidden_dim*4),
            # nn.ReLU(),
            # nn.ConvTranspose2d(in_channels=hidden_dim*4,
out_channels=hidden_dim*2, kernel_size=4, stride=2, padding=1),
            # nn.BatchNorm2d(num_features=hidden_dim*2),
            # nn.ReLU(),
            # nn.ConvTranspose2d(in_channels=hidden_dim*2,
out_channels=hidden_dim, kernel_size=4, stride=2, padding=1),
            # nn.BatchNorm2d(num_features=hidden_dim),
            # nn.ReLU(),
            # nn.ConvTranspose2d(in_channels=hidden_dim,
out_channels=num_channels, kernel_size=4, stride=2, padding=1),
            # nn.Tanh()

            # mlp implementation
            nn.Linear(latent_dim, 4*hidden_dim),
            nn.BatchNorm1d(4*hidden_dim),
            nn.ReLU(),
```

```
            nn.Linear(4*hidden_dim, 2*hidden_dim),
            nn.BatchNorm1d(2*hidden_dim),
            nn.ReLU(),
            nn.Linear(2*hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_channels * 32 * 32),
            nn.Tanh(),
        )
        # TODO END

    def forward(self, z):
        '''
        *   Arguments:
            *   z (torch.FloatTensor): [batch_size, latent_dim, 1, 1]
        '''
        z = z.to(next(self.parameters()).device)
        # mlp implementation
        z = z.view(-1, self.latent_dim)
        return self.decoder(z)
```

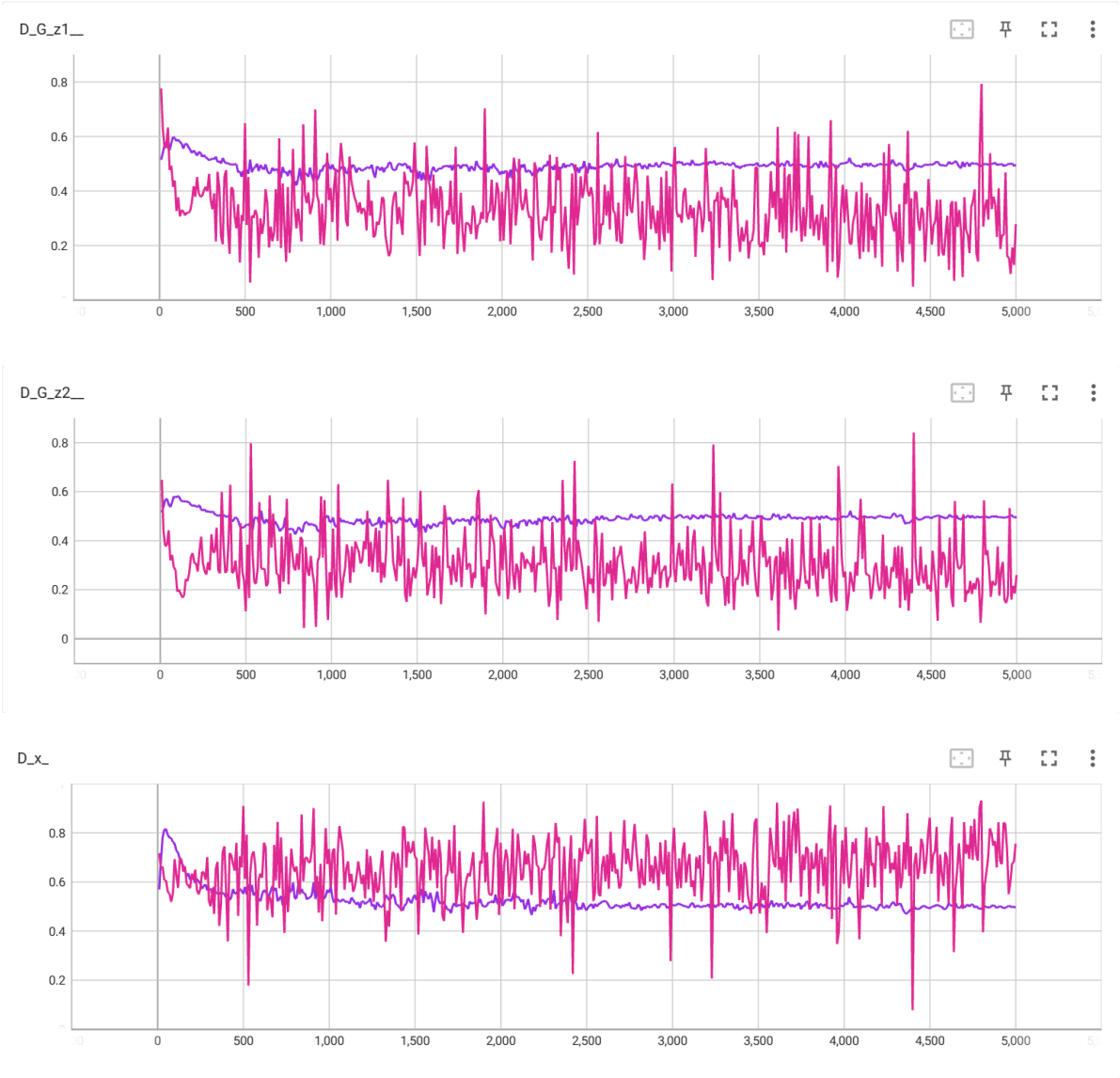
```
class Discriminator(nn.Module):
    def __init__(self, num_channels, hidden_dim):
        super(Discriminator, self).__init__()
        self.num_channels = num_channels
        self.hidden_dim = hidden_dim
        self.clf = nn.Sequential(
            # # input is (num_channels) x 32 x 32
            # nn.Conv2d(num_channels, hidden_dim, 4, 2, 1, bias=False),
            # nn.LeakyReLU(0.2, inplace=True),
            # # state size. (hidden_dim) x 16 x 16
            # nn.Conv2d(hidden_dim, hidden_dim * 2, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(hidden_dim * 2),
            # nn.LeakyReLU(0.2, inplace=True),
            # # state size. (hidden_dim*2) x 8 x 8
            # nn.Conv2d(hidden_dim * 2, hidden_dim * 4, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(hidden_dim * 4),
            # nn.LeakyReLU(0.2, inplace=True),
            # # state size. (hidden_dim*4) x 4 x 4
            # nn.Conv2d(hidden_dim * 4, 1, 4, 1, 0, bias=False),
            # nn.Sigmoid()

            # mlp implementation
            nn.Linear(num_channels * 32 * 32, hidden_dim),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, 2 * hidden_dim),
            nn.Dropout(0.2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(2 * hidden_dim, 4 * hidden_dim),
            nn.Dropout(0.2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(4 * hidden_dim, 1),
            nn.Sigmoid(),
        )
```

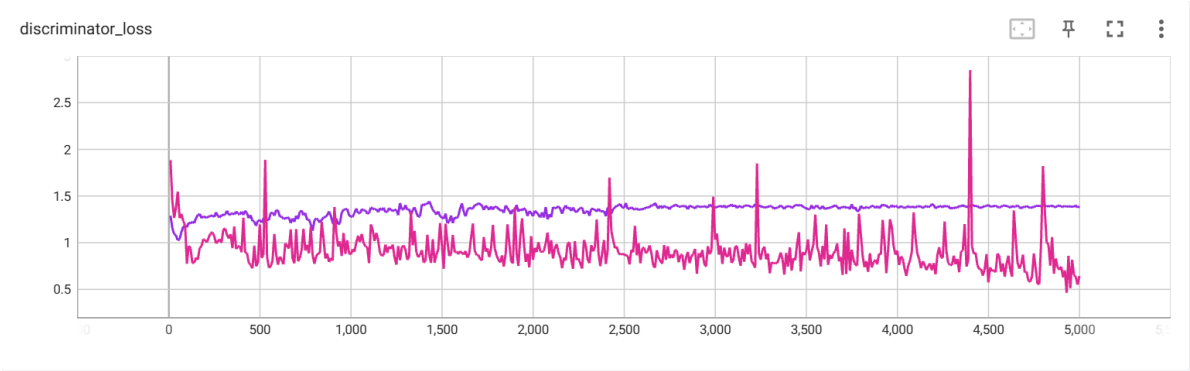
```
    def forward(self, x):
        # # mlp implementation
        x = x.view(-1, self.num_channels * 32 * 32)
        return self.clf(x).view(-1, 1).squeeze(1)
```

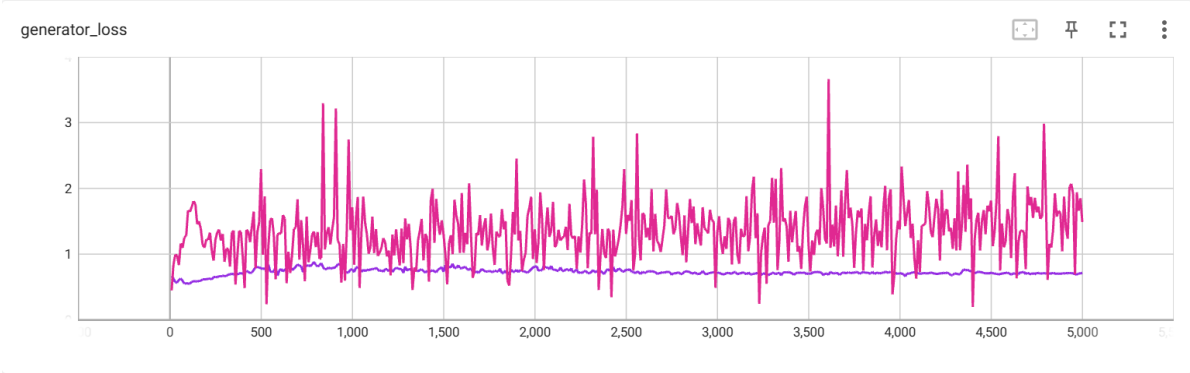
```
if (i + 1) % saving_steps == 0:
    dirname = self._netD.save(self._ckpt_dir, i)
    dirname = self._netG.save(self._ckpt_dir, i)
    self._netG.eval()
    # imgs = make_grid(self._netG(fixed_noise)) * 0.5 + 0.5
    # mlp implementation
    imgs = make_grid(self._netG(fixed_noise)).view(-1, 1, 32, 32) * 0.5 + 0.5
    # self._tb_writer.add_image('samples', imgs, global_step=i)
    save_image(imgs, os.path.join(dirname, "samples.png"))
```

修改后仍然使用**100_100**的参数进行训练，得到的训练曲线如下，将其与同样参数的卷积层训练曲线进行对比：

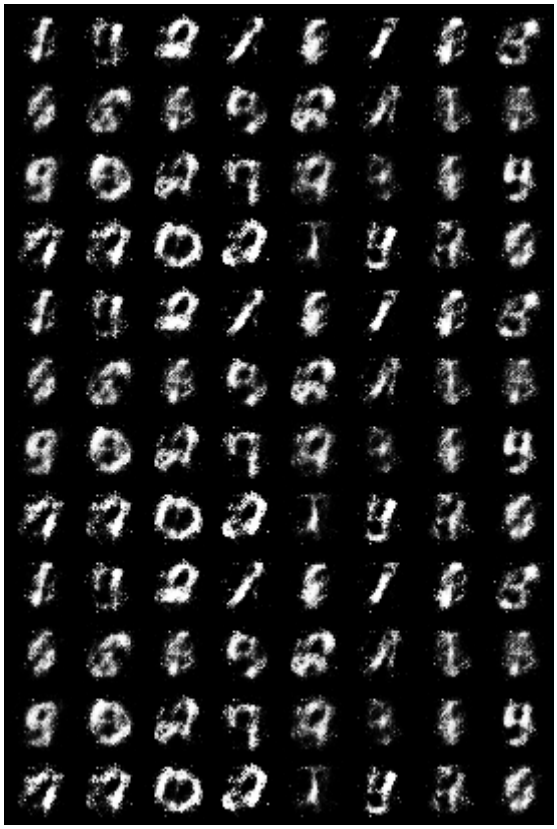
图标：紫色为使用mlp实现的GAN，粉色为作为对比的卷积实现的GAN







得到的图片结果如下：



从训练曲线可以看出，mlp的训练曲线的波动远小于CNN，loss曲线波动的范围很小，说明模型基本没有能够学习到训练集数据的分布。

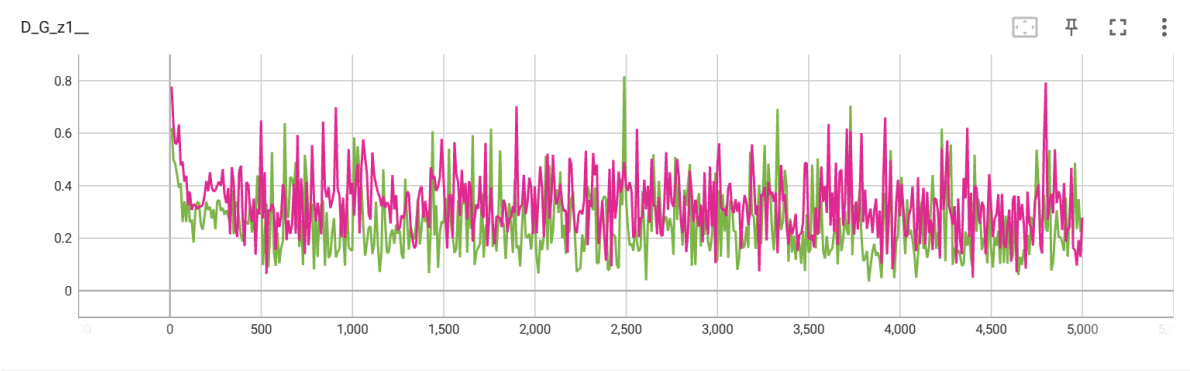
从最终生成的图片可以很明显地看出，生成图片的质量远远不如使用CNN的效果。图片生成的效果模糊、破碎，同时还分布着很多噪点。

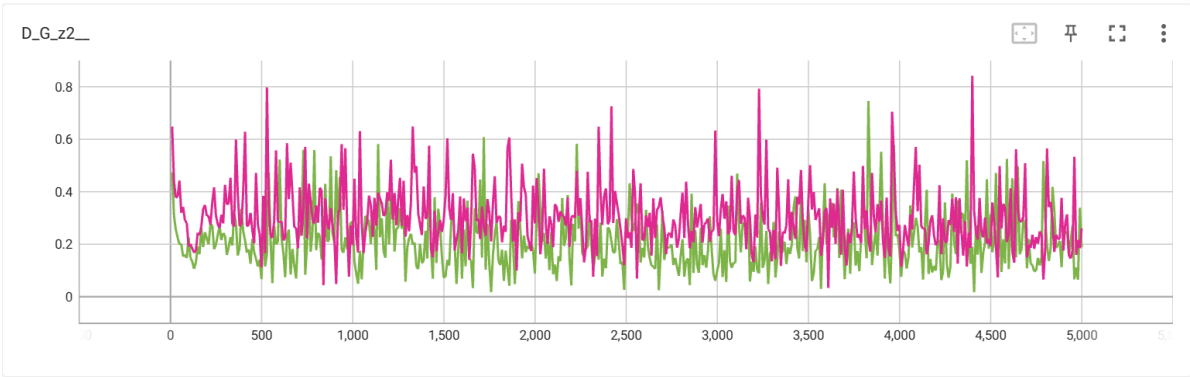
mlp训练的效果较差很大程度上是视觉任务的原因，CNN模型有着视觉的先验经验，能够更好地处理图像相邻像素点之间的相互关系。而mlp虽然作为全连接层尽管有着更高的计算复杂度和更多的参数，但是由于将输入的高维图像数据直接展开到一维，因此导致很多相邻像素点之间的相互关系丢失，即使拥有更多的参数，也很难拟合好图像信息。

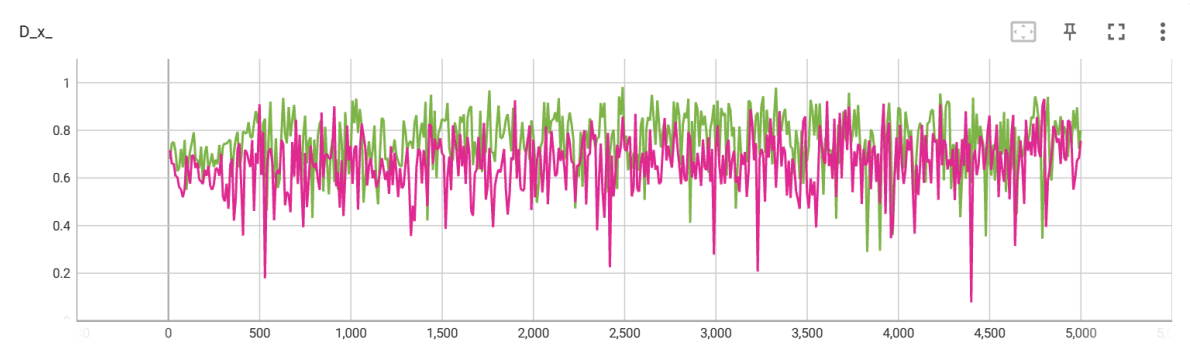
Replace the `LeakyReLU` implementation with `ReLU`

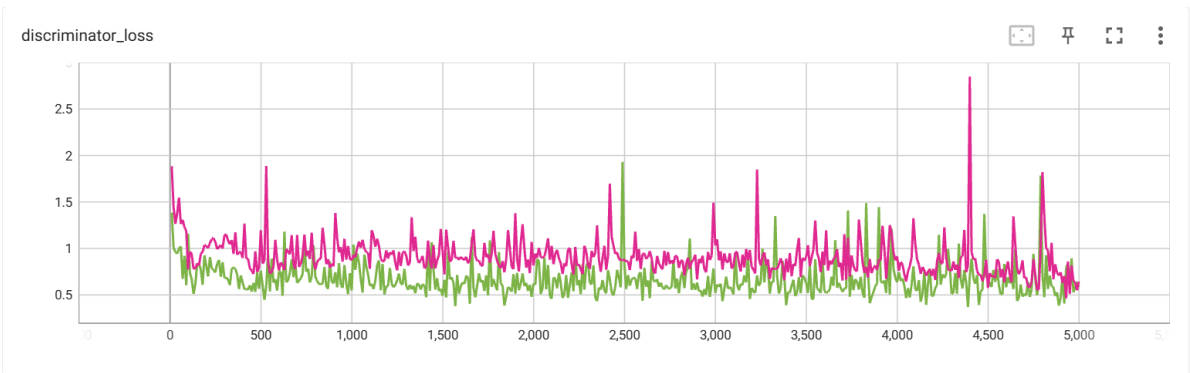
将discriminator中的`LeakyReLU`的实现改为`ReLU`，仍然选择**100_100**的模型参数。

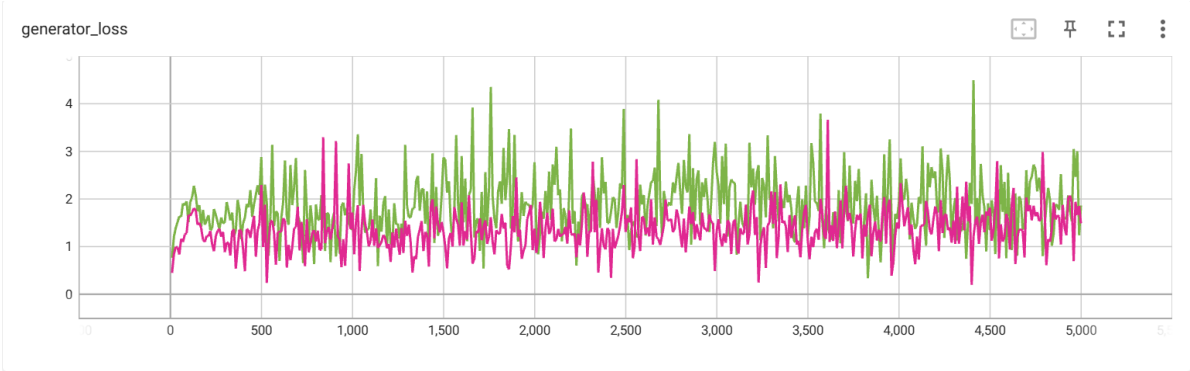
训练曲线如下，同样与**100_100**参数的原始模型进行对比，进行ablation的曲线为绿色：



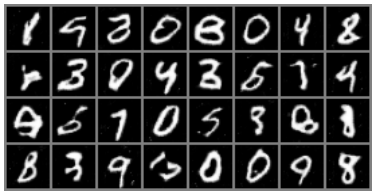








最终生成图片的FID为54.830



对比来看，将`LeakyReLU`改为`ReLU`整体来说会使图片生成的质量稍微下降，而且从训练曲线上看，discriminator loss要低于使用`LeakyReLU`的情况，而generator loss要高。说明generator生成的图片更难使discriminator做出错误判断。

`LeakyReLU`和`ReLU`的区别在于对于负数输入的处理，`ReLU`会直接将负数输入截断为0，而`LeakyReLU`会在负数输入上引入一个小的斜率，以使得负数输入时不再是完全的零。`LeakyReLU`的引入可以一定程度上缓解梯度消失问题，使网络在负数输入上仍然有梯度。

在本次实验中，选择`LeakyReLU`相对更好，可以做到有效避免梯度消失和神经元死亡的问题，能在一定程度上提高discriminator的表现，也就能够通过对抗训练提高generator生成图片的表现。