实验 7：生成对抗网络

2024 年 6 月 5 日

1 实验简介

本实验介绍如何利用 PyTorch 搭建生成对抗网络，并基于 MNIST 手写数字数据集搭建一个手写数字生成器。通过本实验，你将掌握以下知识和技能：

- 生成对抗网络的基本原理
- 利用 PyTorch 搭建并训练简单的生成对抗网络
- 组织数据并训练生成对抗网络

2 实验环境准备

首先，你需要安装完成本实验所需的必要组件。

2.1 安装 Miniconda

从以下链接下载 Miniconda 安装包并完成安装[Link]，安装过程中需要选中 `Register Anaconda as my default Python 3.10`。若计算机上已经安装好 Miniconda 或 Anaconda，请跳过此步骤。

2.2 安装必要的 Package

在 `Anaconda Prompt` 中，通过 `pip` 包管理器安装 `Jupyter Notebook`、`PyTorch`、`torchvision`、`matplotlib` 包。

```
pip install notebook -i https://pypi.tuna.tsinghua.edu.cn/simple/
pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

```
pip install torchvision -i https://pypi.tuna.tsinghua.edu.cn/simple/
pip install matplotlib -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

当然，可以直接通过以下命令进行全局修改，而不必每次都指定-i 参数：

```
pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

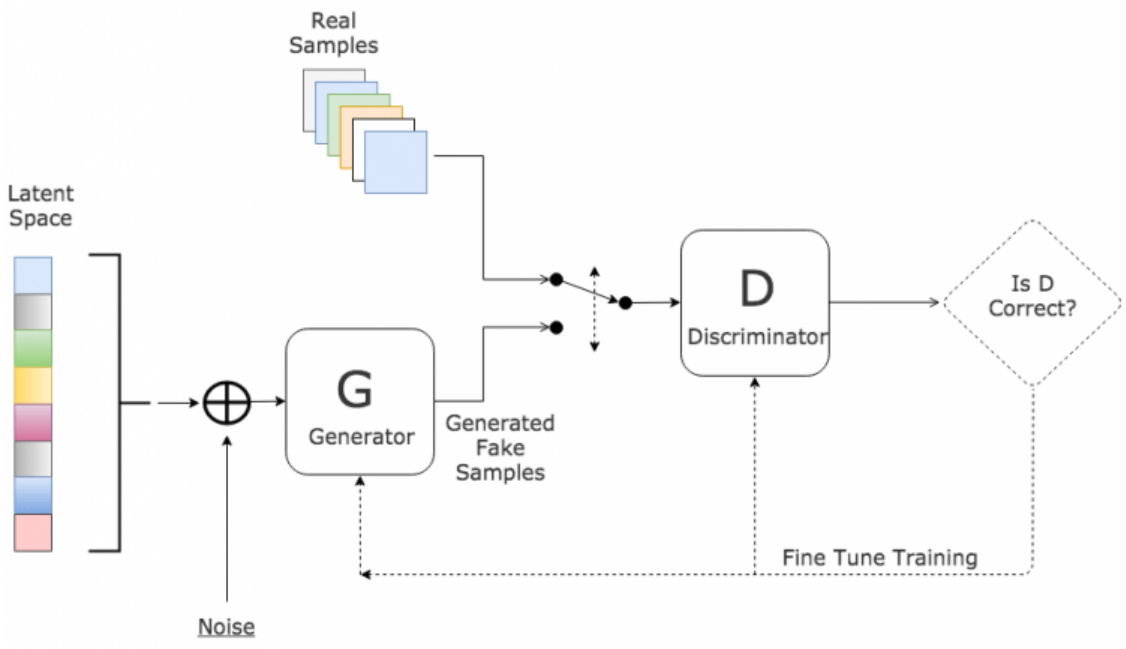
你可以通过执行以下 Python 代码，检查上述包是否已经安装完成。

[27]:
```python
import pkgutil

# 检查 PyTorch 是否已安装
if pkgutil.find_loader('torch'):
    print('PyTorch is available.')

# 检查 torchvision 是否已安装
if pkgutil.find_loader('torchvision'):
    print('torchvision is available.')

# 检查 matplotlib 是否已安装
if pkgutil.find_loader('matplotlib'):
    print('matplotlib is available.')
```

```
PyTorch is available.
torchvision is available.
matplotlib is available.
```

3 生成对抗网络基本原理

生成对抗网络（Generative Adversarial Networks, GAN）是一类能够学习训练数据分布，并基于该分布生成新的数据的深度学习模型。模型由一个“生成器”（Generator）和一个“判别器”（Discriminator）组成，其中生成器的任务是基于数据分布生成“伪造”数据，而判别器的任务是对数据进行鉴别，即判断数据是真实样本还是“伪造”出的样本。在训练过程中，生成器不断产生更加逼真的“伪造”数据以欺骗判别器，判别器则不断提高鉴别真伪的能力，以更好地分类真实样本和伪造样本。当生成器能够生成近乎完美的“伪造”样本，使得判别器认为该样本是真实样本和“伪造样本”的置信度相等时，上述博弈达到均衡状态。



假设 x 代表图像数据，$D(x)$ 是判别器网络，其输出为 x 来自真实训练样本（而非生成器伪造的数据）的概率。显然，当 x 来自训练数据时，$D(x)$ 的输出结果较大；而当 x 是生成器伪造的数据时，$D(x)$ 的输出值较小。同时，假设 z 为从标准正态分布中采样的潜空间向量（Latent space vector），$G(z)$ 表示生成器，它将潜向量 z 映射到数据空间。G 的目标是估计训练数据的分布 (p_{data})，以便基于该分布生成伪造样本（p_g）。

进一步地，$D(G(z))$ 表示生成器 G 的输出是真实图像的概率。D 和 G 形成 Minimax 博弈，D 试图最大化其正确分类真假图像的概率（$logD(x)$），而 G 试图最小化 D 预测其输出为假的概率（$log(1-D(x))$）。生成对抗网络的损失函数为：

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z\sim p_z(z)}[log(1-D(G(z)))]$$

生成对抗网络首先由 Ian Goodfellow 于 2014 年提出（Generative Adversarial Nets）。

4 数据加载和查看

在本实验中，我们使用 MNIST 手写数字数据集，训练一个可以生成逼真的手写数字图片的生成对抗网络模型。

首先，加载必要的包。

[28]:
```
import os
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
```

定义超参数，并判断是否存在可用的 GPU 设备。

[29]:
```
# 定义优化器及模型训练超参数
LR = .0002
BETA1 = .5
EPOCHS = 1
BATCH_SIZE = 128

# 判断是否存在可用的 GPU 设备
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

下载 MNIST 数据，并对数据进行变换（transform）。当执行 `torchvision.datasets.MNIST` 方法时，将判断指定目录下是否已存在完整的 MNIST 数据集。若不存在，将自动下载 MNIST 数据集。

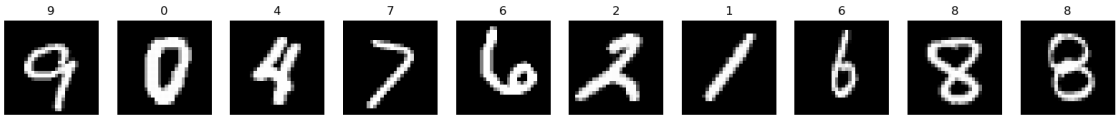
[30]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.,), (1.,))]
)

try:
    trainset = torchvision.datasets.MNIST('./data', transform=transform)
except RuntimeError:
    trainset = torchvision.datasets.MNIST('./data', download=True,␣
 ↪transform=transform)
```

数据准备完成后，可查看数据集的部分样本。

[31]:
```
def plot_digits(dataset):
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=10,␣
 ↪shuffle=True)
```

```
    trainiter = iter(trainloader)
    batch = trainiter.__next__()

    fig = plt.figure(figsize=(20,200))
    for j in range(10):
        plt.subplot(1,10,j+1)
        plt.imshow(batch[0][j][0,:], cmap='gray')
        plt.title(batch[1][j].item())
        plt.axis('off')
```

[32]:
```
plot_digits(trainset)
```



5 模型定义

5.1 定义生成器和判别器

首先，定义生成器和判别器。生成器和判别器是各自独立的神经网络，其中：

- 生成器的输出为 28×28 的矩阵（即“伪造”的手写数字图片）；
- 判别器的输出是图片是否为真的概率。

这里为简便起见，生成器和判别器均使用全连接神经网络。

[33]:
```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.lin1 = nn.Linear(100, 256)
        self.lin2 = nn.Linear(256, 512)
        self.lin3 = nn.Linear(512, 1024)
        self.lin4 = nn.Linear(1024, 784)
        self.leaky_relu = nn.LeakyReLU(.2)
```

```
    def forward(self, input):
        x = self.leaky_relu(self.lin1(input))
        x = self.leaky_relu(self.lin2(x))
        x = self.leaky_relu(self.lin3(x))
        x = self.leaky_relu(self.lin4(x))
        return torch.tanh(x).view(-1, 1, 28, 28)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.lin1 = nn.Linear(784, 1024)
        self.lin2 = nn.Linear(1024, 512)
        self.lin3 = nn.Linear(512, 256)
        self.lin4 = nn.Linear(256, 1)
        self.leaky_relu = nn.LeakyReLU(.2)
        self.dropout = nn.Dropout(.3)

    def forward(self, input):
        x = input.view(-1, 784)
        x = self.leaky_relu(self.lin1(x))
        x = self.dropout(x)
        x = self.leaky_relu(self.lin2(x))
        x = self.dropout(x)
        x = self.leaky_relu(self.lin3(x))
        x = self.dropout(x)
        x = self.leaky_relu(self.lin4(x))
        return torch.sigmoid(x)
```

5.2 定义生成对抗网络模型

接下来，定义一个 GAN 类，用于定义生成对抗网络的模型结构和训练过程。

[34]:
```
class GAN():
    def __init__(self, generator, discriminator, loss, optimizerG, optimizerD,␣
 ↪batch_size=BATCH_SIZE):
```

```
        self.generator = generator
        self.discriminator = discriminator
        self.loss = loss
        self.optimG = optimizerG
        self.optimD = optimizerD
        self.batch_size = batch_size

    def generate_fake(self, batch):
        return self.generator(batch)

    def plot_ten_samples(self):
        noise = torch.randn(10, 100, device=device)
        fake = self.generate_fake(noise).cpu()
        fake = fake.detach().numpy()
        fig = plt.figure(figsize=(20, 200))
        for j in range(10):
            plt.subplot(1, 10, j + 1)
            plt.imshow(fake[j][0], cmap='gray')
            plt.axis('off')
        plt.show()

    def train(self, trainset, epochs=EPOCHS, verbose=True):
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=self.
↪batch_size, shuffle=True)
        errG_history = []
        errD_history = []
        try:
            for epoch in range(epochs):
                if epoch % 10 == 0:
                    self.plot_ten_samples()
                epoch_gerr = []
                epoch_derr = []
                for j, data in enumerate(trainloader, 0):

                    # 训练判别器
                    self.optimD.zero_grad()
```

```
                # 使用真实样本进行训练
                real = data[0].to(device)
                batch_size = real.size()[0]
                real_labels = torch.ones(batch_size, 1, device=device) * .9
                out = self.discriminator(real)
                errD_real = self.loss(out, real_labels)
                errD_real.backward()

                # 使用伪造样本进行训练
                noise = torch.randn(batch_size, 100, device=device)
                fake = self.generate_fake(noise)
                fake_labels = torch.ones(batch_size, 1, device=device) * .1
                out = self.discriminator(fake)
                errD_fake = self.loss(out, fake_labels)
                errD_fake.backward()
                self.optimD.step()
                epoch_derr += [(errD_real.item() + errD_fake.item()) * .5]

                # 训练生成器
                self.optimG.zero_grad()
                noise = torch.randn(batch_size, 100, device=device)
                fake = self.generate_fake(noise)
                fake_labels = torch.ones(batch_size, 1, device=device)
                out = self.discriminator(fake)
                errG = self.loss(out, fake_labels)
                errG.backward()
                self.optimG.step()
                epoch_gerr += [errG.item()]

            errD_history += [torch.tensor(epoch_derr).mean()]
            errG_history += [torch.tensor(epoch_gerr).mean()]

            print(f'EPOCH {epoch + 1:2}/{epochs}')
            print(f"disc_loss: {errD_history[-1]:.4f} - gen_loss:␣
↪{errG_history[-1]:.4f}")
```

```
        except KeyboardInterrupt:
            return errG_history, errD_history

        return errG_history, errD_history
```

实例化生成器和判别器。

[35]:
```
dnet = Discriminator().to(device)
gnet = Generator().to(device)
```

指定模型的损失函数、优化器，并实例化生成对抗网络模型。

[36]:
```
loss = nn.BCELoss()
optimizerD = torch.optim.Adam(dnet.parameters(), lr=LR, betas=(BETA1, 0.999))
optimizerG = torch.optim.Adam(gnet.parameters(), lr=LR, betas=(BETA1, 0.999))

gan = GAN(gnet, dnet, loss, optimizerG, optimizerD)
```

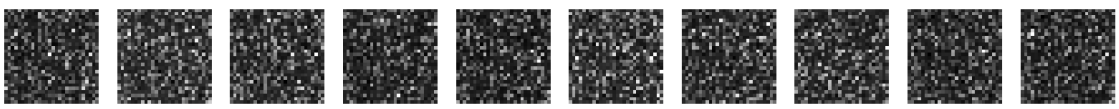
6 模型训练

调用 gan 对象的 train 方法，以完成模型训练。

训练刚刚开始时，产生的样本是完全随机的噪声；随着训练 epoch 数的增多，生成的手写数字图像越来越接近真实样本。

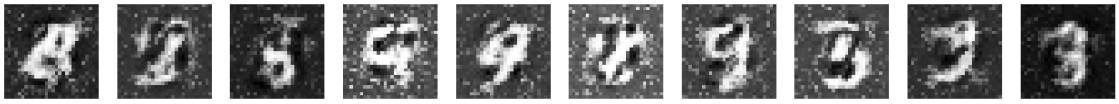
此处可按需设置 epochs，一般而言，至少需要训练 20 个 epoch 才能生成较为真实的图像。当使用 CPU 进行训练时，完成 1 个 epoch 的训练大约耗时 30 秒。

[37]:
```
errorG_history, errorD_history = gan.train(trainset, epochs=100)
```



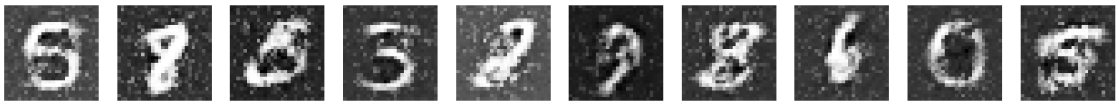
```
EPOCH  1/100
disc_loss: 0.5517 - gen_loss: 1.0981
```

```
EPOCH  2/100
disc_loss: 0.4877 - gen_loss: 2.0706
EPOCH  3/100
disc_loss: 0.4757 - gen_loss: 2.1770
EPOCH  4/100
disc_loss: 0.4729 - gen_loss: 2.1203
EPOCH  5/100
disc_loss: 0.4723 - gen_loss: 2.0579
EPOCH  6/100
disc_loss: 0.4785 - gen_loss: 1.9058
EPOCH  7/100
disc_loss: 0.4751 - gen_loss: 1.8901
EPOCH  8/100
disc_loss: 0.4715 - gen_loss: 1.8532
EPOCH  9/100
disc_loss: 0.4755 - gen_loss: 1.8098
EPOCH 10/100
disc_loss: 0.4743 - gen_loss: 1.7893
```

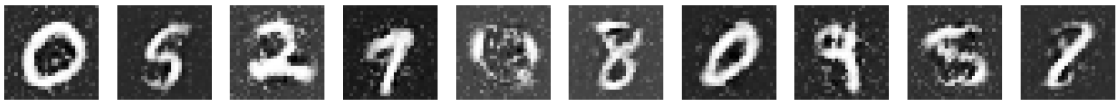


```
EPOCH 11/100
disc_loss: 0.4875 - gen_loss: 1.6892
EPOCH 12/100
disc_loss: 0.5120 - gen_loss: 1.5627
EPOCH 13/100
disc_loss: 0.5224 - gen_loss: 1.5165
EPOCH 14/100
disc_loss: 0.5265 - gen_loss: 1.4544
EPOCH 15/100
disc_loss: 0.5250 - gen_loss: 1.4586
EPOCH 16/100
disc_loss: 0.5258 - gen_loss: 1.4514
```

```
EPOCH 17/100
disc_loss: 0.5270 - gen_loss: 1.4362
EPOCH 18/100
disc_loss: 0.5252 - gen_loss: 1.4385
EPOCH 19/100
disc_loss: 0.5265 - gen_loss: 1.4199
EPOCH 20/100
disc_loss: 0.5251 - gen_loss: 1.4362
```



```
EPOCH 21/100
disc_loss: 0.5233 - gen_loss: 1.4389
EPOCH 22/100
disc_loss: 0.5203 - gen_loss: 1.4534
EPOCH 23/100
disc_loss: 0.5180 - gen_loss: 1.4532
EPOCH 24/100
disc_loss: 0.5135 - gen_loss: 1.4643
EPOCH 25/100
disc_loss: 0.5103 - gen_loss: 1.4921
EPOCH 26/100
disc_loss: 0.5113 - gen_loss: 1.4847
EPOCH 27/100
disc_loss: 0.5081 - gen_loss: 1.4955
EPOCH 28/100
disc_loss: 0.5049 - gen_loss: 1.5035
EPOCH 29/100
disc_loss: 0.5015 - gen_loss: 1.5190
EPOCH 30/100
disc_loss: 0.4985 - gen_loss: 1.5269
```



```
EPOCH 31/100
disc_loss: 0.4972 - gen_loss: 1.5390
EPOCH 32/100
disc_loss: 0.4919 - gen_loss: 1.5512
EPOCH 33/100
disc_loss: 0.4913 - gen_loss: 1.5595
EPOCH 34/100
disc_loss: 0.4883 - gen_loss: 1.5695
EPOCH 35/100
disc_loss: 0.4849 - gen_loss: 1.5899
EPOCH 36/100
disc_loss: 0.4811 - gen_loss: 1.6059
EPOCH 37/100
disc_loss: 0.4810 - gen_loss: 1.6041
EPOCH 38/100
disc_loss: 0.4803 - gen_loss: 1.6138
EPOCH 39/100
disc_loss: 0.4767 - gen_loss: 1.6338
EPOCH 40/100
disc_loss: 0.4736 - gen_loss: 1.6434
```

```
EPOCH 41/100
disc_loss: 0.4729 - gen_loss: 1.6542
EPOCH 42/100
disc_loss: 0.4716 - gen_loss: 1.6612
EPOCH 43/100
```

```
disc_loss: 0.4704 - gen_loss: 1.6657
EPOCH 44/100
disc_loss: 0.4679 - gen_loss: 1.6867
EPOCH 45/100
disc_loss: 0.4655 - gen_loss: 1.6851
EPOCH 46/100
disc_loss: 0.4655 - gen_loss: 1.6963
EPOCH 47/100
disc_loss: 0.4622 - gen_loss: 1.6986
EPOCH 48/100
disc_loss: 0.4635 - gen_loss: 1.7061
EPOCH 49/100
disc_loss: 0.4622 - gen_loss: 1.7106
EPOCH 50/100
disc_loss: 0.4612 - gen_loss: 1.7129
```

```
EPOCH 51/100
disc_loss: 0.4610 - gen_loss: 1.7161
EPOCH 52/100
disc_loss: 0.4591 - gen_loss: 1.7198
EPOCH 53/100
disc_loss: 0.4597 - gen_loss: 1.7294
EPOCH 54/100
disc_loss: 0.4583 - gen_loss: 1.7222
EPOCH 55/100
disc_loss: 0.4568 - gen_loss: 1.7303
EPOCH 56/100
disc_loss: 0.4558 - gen_loss: 1.7422
EPOCH 57/100
disc_loss: 0.4569 - gen_loss: 1.7379
EPOCH 58/100
```

```
disc_loss: 0.4561 - gen_loss: 1.7335
EPOCH 59/100
disc_loss: 0.4556 - gen_loss: 1.7378
EPOCH 60/100
disc_loss: 0.4555 - gen_loss: 1.7336
```

```
EPOCH 61/100
disc_loss: 0.4542 - gen_loss: 1.7480
EPOCH 62/100
disc_loss: 0.4498 - gen_loss: 1.7662
EPOCH 63/100
disc_loss: 0.4544 - gen_loss: 1.7510
EPOCH 64/100
disc_loss: 0.4508 - gen_loss: 1.7507
EPOCH 65/100
disc_loss: 0.4541 - gen_loss: 1.7417
EPOCH 66/100
disc_loss: 0.4560 - gen_loss: 1.7510
EPOCH 67/100
disc_loss: 0.4501 - gen_loss: 1.7615
EPOCH 68/100
disc_loss: 0.4527 - gen_loss: 1.7543
EPOCH 69/100
disc_loss: 0.4514 - gen_loss: 1.7669
EPOCH 70/100
disc_loss: 0.4494 - gen_loss: 1.7628
```

```
EPOCH 71/100
disc_loss: 0.4495 - gen_loss: 1.7707
EPOCH 72/100
disc_loss: 0.4508 - gen_loss: 1.7606
EPOCH 73/100
disc_loss: 0.4486 - gen_loss: 1.7698
EPOCH 74/100
disc_loss: 0.4502 - gen_loss: 1.7590
EPOCH 75/100
disc_loss: 0.4487 - gen_loss: 1.7630
EPOCH 76/100
disc_loss: 0.4512 - gen_loss: 1.7606
EPOCH 77/100
disc_loss: 0.4502 - gen_loss: 1.7597
EPOCH 78/100
disc_loss: 0.4450 - gen_loss: 1.7847
EPOCH 79/100
disc_loss: 0.4496 - gen_loss: 1.7748
EPOCH 80/100
disc_loss: 0.4489 - gen_loss: 1.7632
```

```
EPOCH 81/100
disc_loss: 0.4475 - gen_loss: 1.7671
EPOCH 82/100
disc_loss: 0.4492 - gen_loss: 1.7743
EPOCH 83/100
disc_loss: 0.4481 - gen_loss: 1.7697
EPOCH 84/100
disc_loss: 0.4481 - gen_loss: 1.7775
EPOCH 85/100
disc_loss: 0.4494 - gen_loss: 1.7646
```

```
EPOCH 86/100
disc_loss: 0.4499 - gen_loss: 1.7620
EPOCH 87/100
disc_loss: 0.4478 - gen_loss: 1.7765
EPOCH 88/100
disc_loss: 0.4484 - gen_loss: 1.7678
EPOCH 89/100
disc_loss: 0.4477 - gen_loss: 1.7739
EPOCH 90/100
disc_loss: 0.4516 - gen_loss: 1.7460
```

```
EPOCH 91/100
disc_loss: 0.4485 - gen_loss: 1.7557
EPOCH 92/100
disc_loss: 0.4475 - gen_loss: 1.7770
EPOCH 93/100
disc_loss: 0.4476 - gen_loss: 1.7746
EPOCH 94/100
disc_loss: 0.4485 - gen_loss: 1.7652
EPOCH 95/100
disc_loss: 0.4442 - gen_loss: 1.7831
EPOCH 96/100
disc_loss: 0.4518 - gen_loss: 1.7594
EPOCH 97/100
disc_loss: 0.4476 - gen_loss: 1.7636
EPOCH 98/100
disc_loss: 0.4478 - gen_loss: 1.7865
EPOCH 99/100
disc_loss: 0.4480 - gen_loss: 1.7688
EPOCH 100/100
disc_loss: 0.4475 - gen_loss: 1.7666
```

模型训练结束后，可通过调用以下方法，随机生成 10 张“伪造”图片，以检验模型的生成效果。

[38]:
```
gan.plot_ten_samples()
```

通过执行以下代码块，可查看训练过程中生成器、判别器损失函数值的变化情况。

[40]:
```
plt.figure(figsize=(10,5))
plt.plot(errorG_history)
plt.plot(errorD_history)
plt.title('Loss graph')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Generator','Discriminator'])
```

[40]:
```
<matplotlib.legend.Legend at 0x1a0d600b250>
```