

**人工智能导论-实验报告**

**题目：生成对抗网络实验**

院 系 软件学院

专业班级

姓 名

学 号

指导教师

2024年 12月 26 日

**目录**

**[1引言 2](#_Toc21601)**

**1.1问题描述 3**

**1.2 目标 3**

**[2 数据集介绍 3](#_Toc4496)**

**2.1 数据来源 4**

**2.2 数据预处理 4**

**[3 模型架构 2](#_Toc2876)**

**3.1 生成器以及原理函数 4**

**3.2 判别器以及原理函数 5**

**3.3 损失函数/优化器 6**

**[4 数据可视化分析 7](#_Toc16576)**

**4.1 真实图像 7**

**4.2 生成器输出图像 7**

**5 实验结果与分析 7**

**6 讨论与总结 10**

**6.1 结果分析 10**

**6.2 遇到的问题以及解决方法 11**

**6.3 代码模型的优化 11**

**7 参考文献 11**

**8代码 11**

**9 附录 16**

**9.1 实验一运行结果图 16**

**9.2 实验二运行结果图 16**

**9.3 实验三运行结果图 17**

## 1 引言

### 1.1 问题描述

在深度学习中，生成对抗网络（Generative Adversarial Network, GAN）是近年来非常有影响力的一种无监督学习框架。它通过让两个网络（生成器和判别器）在相互博弈的过程中不断提升能力，从而实现对数据分布的逼真模拟。

* **生成器（Generator）**：从随机噪声中生成新的样本，力图“骗过”判别器；
* **判别器（Discriminator）**：判定输入样本是真实数据还是由生成器伪造的，以提升区分真假的能力。

本实验选择了**MNIST 手写数字**数据集作为实验对象，使用 **PyTorch** 框架构建 GAN（或其变体 **DCGAN**），并尝试通过对抗训练来生成较为逼真的数字图像。

### 1.2 目标

1. **掌握 GAN 的基本原理和对抗思想**，理解生成器与判别器之间的动态博弈；
2. **实现一个完整的 GAN / DCGAN 训练流程**，包括数据预处理、模型定义、损失函数与优化器设定、训练循环、可视化等；
3. **分析并评价 GAN 的训练效果**，包括生成器输出样本质量、训练损失曲线、模式崩溃等常见现象；
4. **探讨改进与扩展**，如在不同网络结构、超参数、或变体算法（WGAN、LSGAN 等）下的效果变化。

## 2 数据集介绍

### 2.1 数据来源

* 使用 torchvision.datasets.MNIST 自动下载 MNIST 数据集。该数据集包含 60000 张训练图片与 10000 张测试图片，每张图片是 28×28 的灰度手写数字（0~9）。
* MNIST 可从以下链接手动获取：http://yann.lecun.com/exdb/mnist/

### 2.2 数据预处理

**1.尺寸变换**

·在最基础的全连接 GAN 中，我们常将 28×28的图像直接 flatten 到 784 维向量。

·在 **DCGAN** 经典结构中，我们往往将图像调整为 32×32或 64×64，以对齐卷积/转置卷积层的步幅 (stride) 和填充 (padding)。本实验选择将 MNIST 调整为 32×323大小进行对抗训练。若保留原始 28×28，则需要手动设计适配的卷积核参数。

**2.归一化 (Normalization)**

为了匹配生成器最终 **Tanh** 输出的 [−1,+1]区间，使用：

xnorm​=(x−0.5)/0.5​⇒xnorm​∈[−1,1].

这一步通常使用 transforms.Normalize([0.5],[0.5])。

**3.DataLoader 批量加载**

设置合适的 batch\_size（如 128/256），随机打乱顺序 (shuffle=True)，以提高训练的稳定性和泛化能力。

## 3 模型架构

本实验使用的模型为 **DCGAN**，即在 GAN 框架的基础上采用卷积/转置卷积结构，对图像生成和判别更加高效。

### 3.1 生成器以及原理函数

#### 3.1.1 结构设计

1. 输入：大小为 (batch,latent\_dim,1,1) 的随机噪声 zzz，latent\_dim 一般取 64 或 100。
2. 转置卷积层 (ConvTranspose2d)：一步步上采样，在通道数和空间尺寸上逐层匹配到输出图像（例如 1×32×32）。
3. 每一层用 **BatchNorm2d** 和 **ReLU** 激活，防止梯度消失或爆炸；最后一层输出通道为 1（MNIST灰度图），并通过 **Tanh** 激活。

class Generator(nn.Module):

def \_\_init\_\_(self, nz=100, ngf=64):

super().\_\_init\_\_()

self.net = nn.Sequential(

nn.ConvTranspose2d(nz, ngf\*4, kernel\_size=4, stride=1, padding=0, bias=False),

nn.BatchNorm2d(ngf\*4),

nn.ReLU(True),

# ...依次上采样...

nn.ConvTranspose2d(ngf, 1, kernel\_size=4, stride=2, padding=1, bias=False),

nn.Tanh()

)

def forward(self, x):

return self.net(x)

#### 3.1.2 工作原理

* 生成器在训练中不断更新参数，学习如何将噪声映射到和真实数字近似分布的图像空间。
* 最终目的是“欺骗”判别器，让判别器相信这是“真图”。

### 3.2 判别器以及原理函数

#### 3.2.1 结构设计

1. 输入：形状 (batch,1,32,32)的图像（来自真实数据或生成器输出）。
2. 通过多层卷积 **down-sampling**，每层可用 stride=2、padding=1，减少空间分辨率，但增大通道数。
3. 在中间层使用 **BatchNorm2d** 和 **LeakyReLU(0.2)** 激活以提升稳定性；最后输出 (N, 1, 1, 1) → (N,1)。
4. Sigmoid() 将结果限制在 [0,1]判别真假。

class Discriminator(nn.Module):

def \_\_init\_\_(self, ndf=64):

super().\_\_init\_\_()

self.net = nn.Sequential(

nn.Conv2d(1, ndf, 4, 2, 1, bias=False),

nn.LeakyReLU(0.2, inplace=True),

# ...依次下采样...

nn.Conv2d(ndf\*8, 1, 4, 2, 1, bias=False),

nn.Sigmoid()

)

def forward(self, x):

return self.net(x)

#### 3.2.2 工作原理

* 判别器训练的目标：对真实图输出 1（真），对假图输出 0（假）。
* 与生成器对抗：判别器越准确，生成器就需“更逼真”地生成图像。

### 3.3 损失函数/优化器

#### 3.3.1 损失函数

**·二元交叉熵 (BCE)**： ℓ(x,y)=−[ylog(x)+(1−y)log(1−x)].

·对于判别器 (D)



·对于生成器 (G):LG​=−Ez∼pz​​[logD(G(z))].

直观而言，就是希望判别器对 G(z) 输出越接近 1 越好。

#### 3.3.2 优化器

* **Adam** 常用在 DCGAN 中：lr = 2e-4, betas=(0.5, 0.999).
* 训练采用 **交替更新**：先更新 D（仅对 D 的参数求梯度），再更新 G（仅对 G 的参数求梯度）。

## 4 数据可视化分析

### 4.1 真实图像

* 可从 DataLoader 取出一批 MNIST 原图或放大/缩放后的图像可视化，如下

images, \_ = next(iter(train\_loader))

save\_image(images, 'real\_samples.png')

观察图像结构，验证预处理（如 Resize, Normalize）是否有效。

### 4.2 生成器输出图像

* 每个 epoch 结束时，用一组**固定噪声**（fixed\_noise）输入生成器，得到一批生成样本并保存。
* 可以对比各个 epoch 的生成结果演变：初期完全像噪点，逐渐会出现数字轮廓，后期越来越逼真。

示例可视化方法：

fake\_images = G(fixed\_noise)

save\_image(denorm(fake\_images), f'fake\_samples\_epoch\_{epoch+1}.png', nrow=8)

## 5 实验结果与分析

1. **训练日志与损失曲线**

·在每个 batch 或若干 batch 之后记录并打印 d\_loss（判别器损失）、g\_loss（生成器损失），以及判别器打分 D(x) / D(G(z))。

·期望看到 D(G(z)) 从接近 0 慢慢升高，而 D(x) 保持相对较高值（判别器对真实图依旧信任度高）。

1. **生成图像质量**

·通过查看保存的 fake\_samples\_epoch\_XX.png，如果能够明显看到清晰的手写数字形状，则说明生成器已学到合理的分布；若出现混淆或模式崩溃 (mode collapse)，则需要调整超参。

1. **训练稳定性**

·DCGAN 在 MNIST 上一般较稳定。如果判别器太强 / 太弱，都可能导致不稳定现象：生成器学不到有用信息或判别器失去鉴别力。

在本实验中，若训练 **50~200 epoch**，通常能在 MNIST 数据上生成较逼真的数字图像。随着 epoch 的增加，生成样本从起初的随机噪声逐渐变得更有结构。

epoch增加结果：

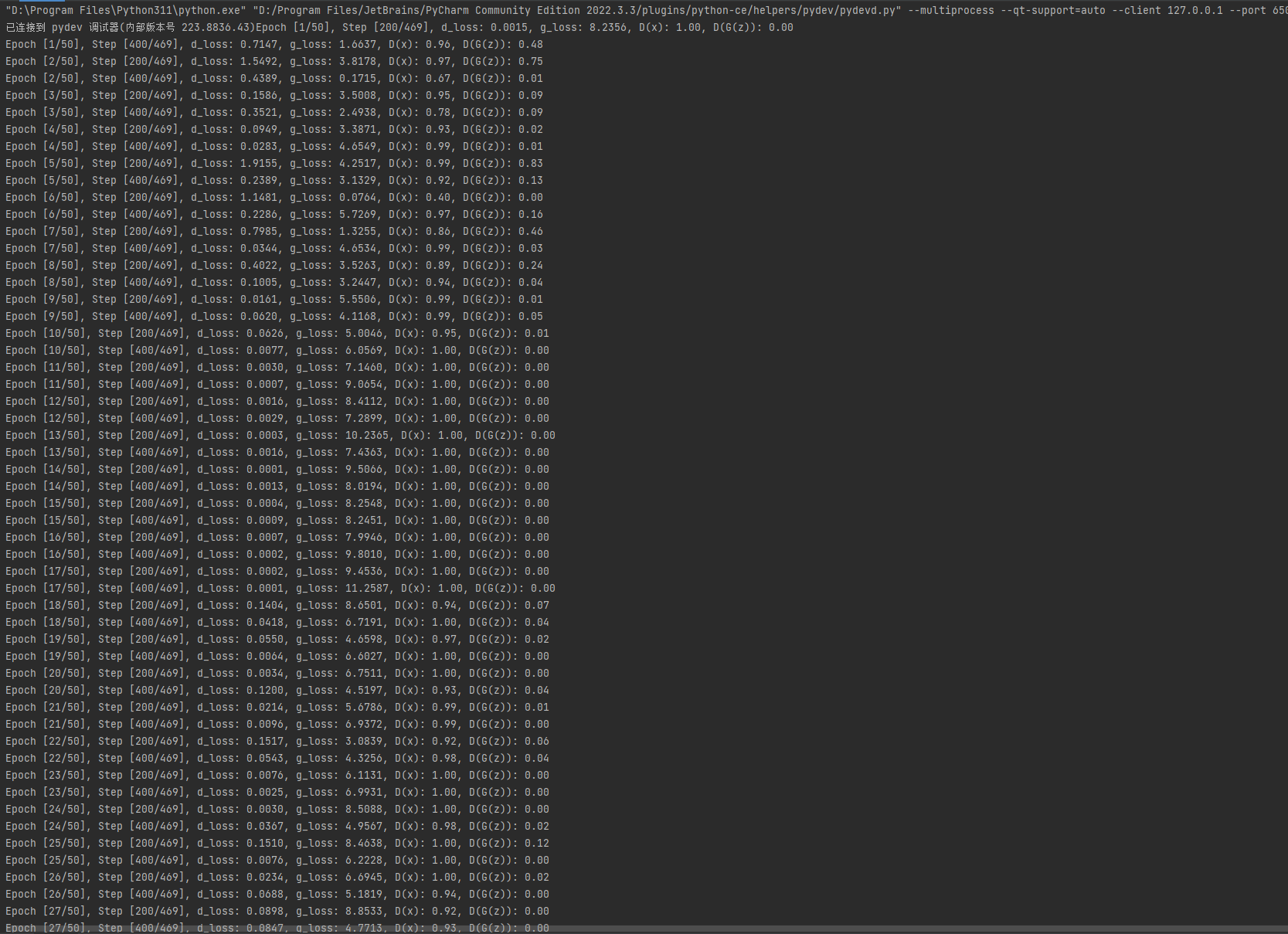


图5-1 生成结果

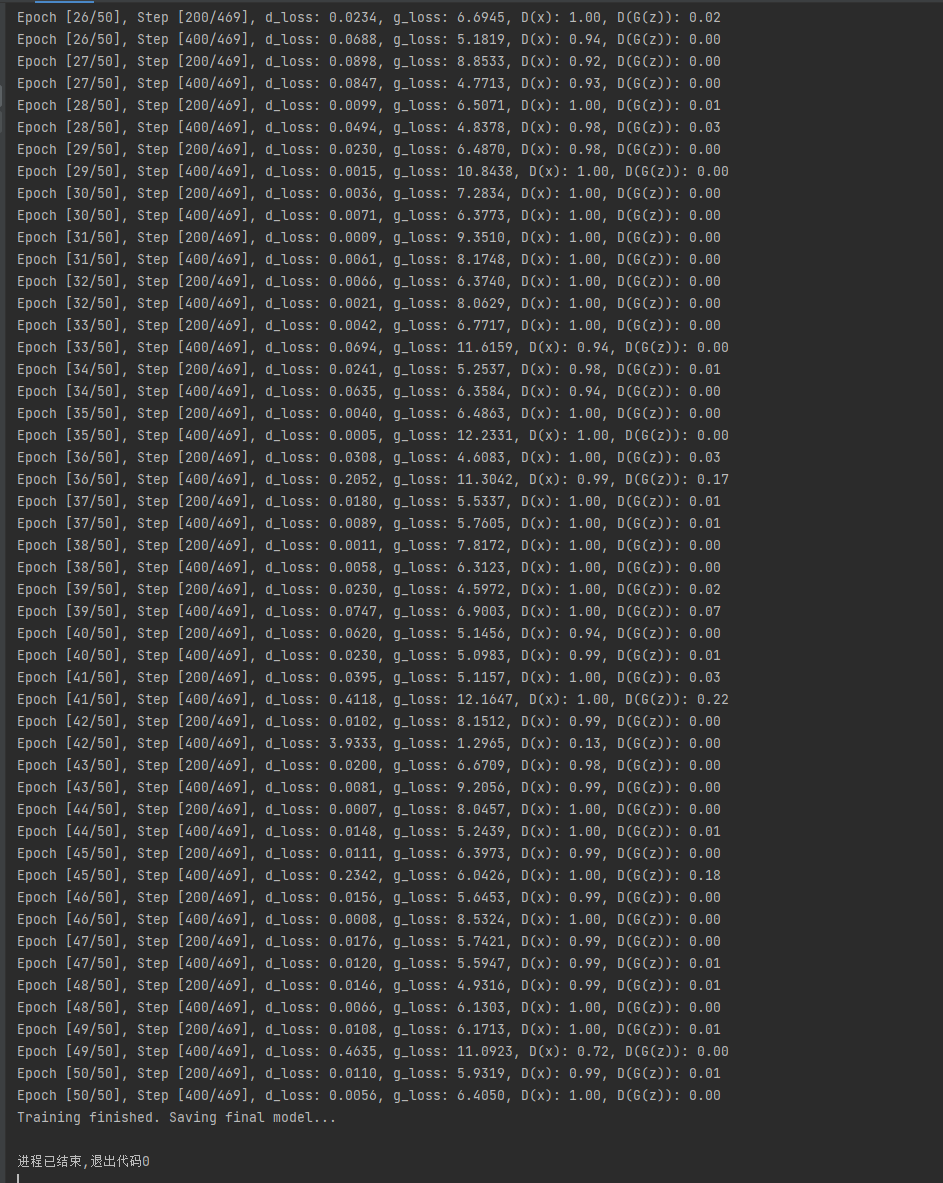


图5-2 epoch增加生成结果

生成图像结果：

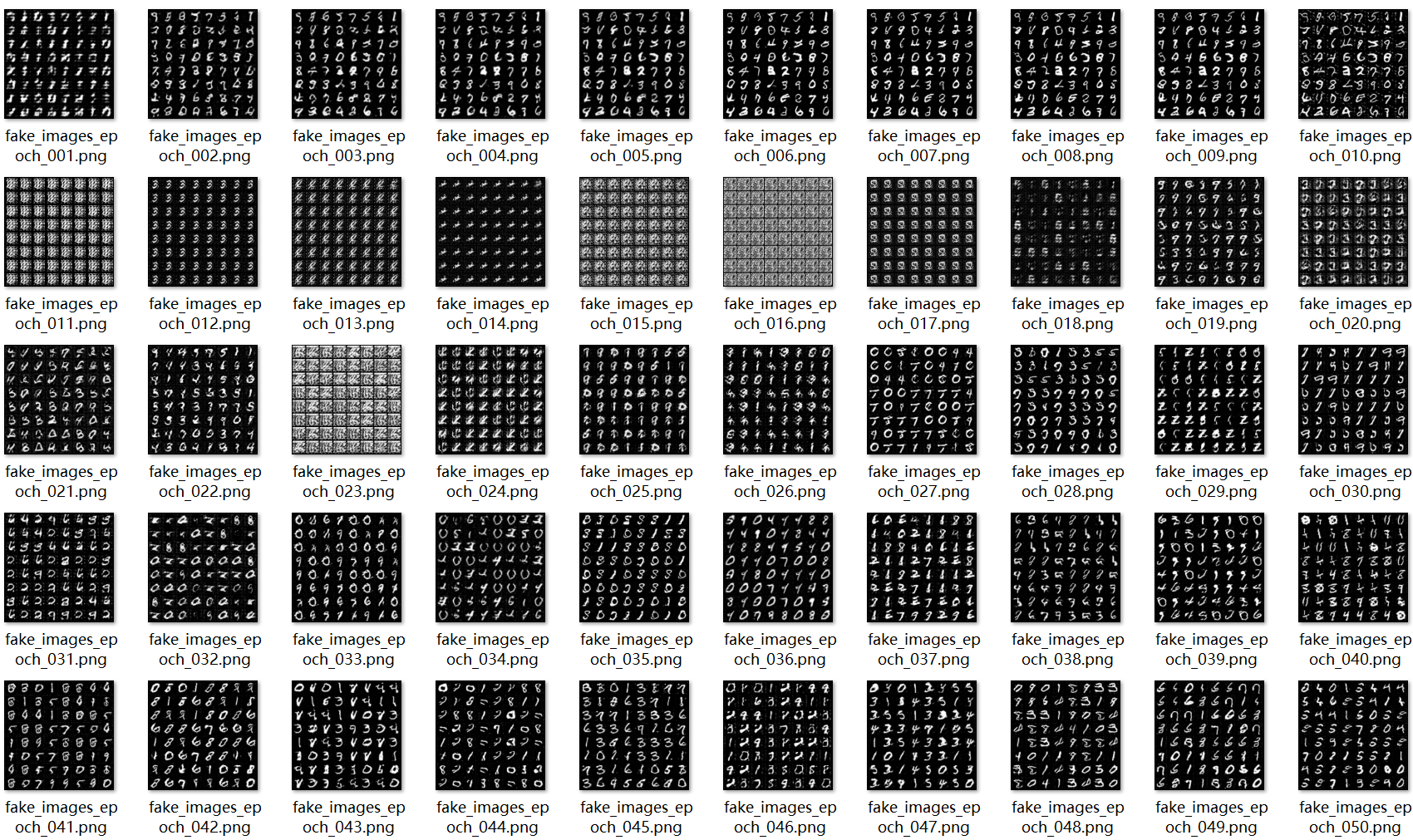


图5-3 生成图像结果

注：所有详细生成结果图片见9附录

## 6 讨论与总结

### 6.1 结果分析

* 本次实验表明，基于 **DCGAN** 的网络结构在手写数字生成任务上具有良好的收敛性。
* 在可视化图像中，数字轮廓较清晰，出现错误或模糊的概率相对较低（相对于全连接式 GAN 而言）。
* GAN 模型训练过程表现出“互博弈”特性：判别器变得越强，生成器就越需要逼真地伪造图像，二者相辅相成。

### 6.2 遇到的问题及解决方法

**1.卷积核 / 输入尺寸不匹配**

出现错误：RuntimeError: Kernel size can't be greater than...

解决：确保生成器最后输出与判别器输入形状一致；若使用 28×28，需仔细调节每层 kernel\_size, stride, padding；或将图片 resize 到 32×32，并采用经典 DCGAN 参数。

**2.模式崩溃 (mode collapse)**

问题：生成器有时会反复输出相似的图像，多样性下降。

解决：降低判别器学习率、引入 **label smoothing**、尝试 **WGAN** / **WGAN-GP** 等。

**3.训练时间较长**

问题：在 CPU 环境下时间较长；若有 GPU 可加速。

减少 batch\_size 或减少网络层数能缩短单 epoch 时间，但可能影响结果质量。

### 6.3 代码模型的优化

1. **使用更深的网络**：

增加生成器 / 判别器层数和通道数，在更复杂数据集（如 CIFAR-10）上也能取得更好效果。

1. **改进优化策略**：

试用 **RMSProp**、**SGD**、或 **AdamW**，观察训练速度与稳定性。

可在判别器使用 **学习率衰减**，在后期 fine-tune。

1. **GAN 变体**：

**WGAN / WGAN-GP** 解决普通 GAN 在训练稳定性上的问题，引入 Wasserstein 距离衡量真实和生成分布之间的差异。

**LSGAN**、**EBGAN** 等也能在一定程度上缓解训练不稳定和模式崩溃。

## 7 参考文献 / 学习资源

1. **Ian Goodfellow et al.** Generative Adversarial Networks. Communications of the ACM, 2014.
2. **Radford, Metz, Chintala.** Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. ICLR, 2016. (DCGAN 经典论文)
3. PyTorch 官方文档: [https://pytorch.org](https://pytorch.org" \t "_new)
4. MNIST: <http://yann.lecun.com/exdb/mnist/>

**8代码**

import os

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

from torchvision import transforms

from torchvision.utils import save\_image

# ============ 1. 环境与超参数 ============

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

# 数据及网络相关超参数

batch\_size = 128 # 批大小

lr = 2e-4 # 学习率

num\_epochs = 50 # 训练轮数

latent\_dim = 100 # 生成器输入噪声向量维度

ngf = 64 # 生成器中 feature map 基数

ndf = 64 # 判别器中 feature map 基数

# 生成图像的输出文件夹

sample\_dir = 'samples\_dcgan\_32x32'

os.makedirs(sample\_dir, exist\_ok=True)

# ============ 2. 数据准备：MNIST → 32x32 ============

transform = transforms.Compose([

transforms.Resize(32), # 将 28×28 缩放到 32×32

transforms.ToTensor(),

transforms.Normalize([0.5], [0.5]) # 归一化到 [-1, +1]

])

train\_dataset = torchvision.datasets.MNIST(

root='./data',

train=True,

transform=transform,

download=True

)

train\_loader = torch.utils.data.DataLoader(

train\_dataset,

batch\_size=batch\_size,

shuffle=True

)

# ============ 3. 定义生成器 (G) ============

# 目标：输入 (N, latent\_dim, 1, 1) -> 输出 (N, 1, 32, 32)

class Generator(nn.Module):

def \_\_init\_\_(self, nz=100, ngf=64):

super().\_\_init\_\_()

self.net = nn.Sequential(

# (nz, 1, 1) -> (ngf\*4, 4, 4)

nn.ConvTranspose2d(nz, ngf \* 4, kernel\_size=4, stride=1, padding=0, bias=False),

nn.BatchNorm2d(ngf \* 4),

nn.ReLU(True),

# (ngf\*4, 4, 4) -> (ngf\*2, 8, 8)

nn.ConvTranspose2d(ngf \* 4, ngf \* 2, kernel\_size=4, stride=2, padding=1, bias=False),

nn.BatchNorm2d(ngf \* 2),

nn.ReLU(True),

# (ngf\*2, 8, 8) -> (ngf, 16, 16)

nn.ConvTranspose2d(ngf \* 2, ngf, kernel\_size=4, stride=2, padding=1, bias=False),

nn.BatchNorm2d(ngf),

nn.ReLU(True),

# (ngf, 16, 16) -> (1, 32, 32)

nn.ConvTranspose2d(ngf, 1, kernel\_size=4, stride=2, padding=1, bias=False),

nn.Tanh()

)

def forward(self, x):

return self.net(x)

# ============ 4. 定义判别器 (D) ============

# 目标：输入 (N, 1, 32, 32) -> 输出 (N, 1)

# 卷积层逐步下采样到 1×1

class Discriminator(nn.Module):

def \_\_init\_\_(self, ndf=64):

super().\_\_init\_\_()

self.net = nn.Sequential(

# (1, 32, 32) -> (ndf, 16, 16)

nn.Conv2d(1, ndf, kernel\_size=4, stride=2, padding=1, bias=False),

nn.LeakyReLU(0.2, inplace=True),

# (ndf, 16, 16) -> (ndf\*2, 8, 8)

nn.Conv2d(ndf, ndf \* 2, kernel\_size=4, stride=2, padding=1, bias=False),

nn.BatchNorm2d(ndf \* 2),

nn.LeakyReLU(0.2, inplace=True),

# (ndf\*2, 8, 8) -> (ndf\*4, 4, 4)

nn.Conv2d(ndf \* 2, ndf \* 4, kernel\_size=4, stride=2, padding=1, bias=False),

nn.BatchNorm2d(ndf \* 4),

nn.LeakyReLU(0.2, inplace=True),

# (ndf\*4, 4, 4) -> (ndf\*8, 2, 2)

nn.Conv2d(ndf \* 4, ndf \* 8, kernel\_size=4, stride=2, padding=1, bias=False),

nn.BatchNorm2d(ndf \* 8),

nn.LeakyReLU(0.2, inplace=True),

# (ndf\*8, 2, 2) -> (1, 1, 1)

nn.Conv2d(ndf \* 8, 1, kernel\_size=4, stride=2, padding=1, bias=False),

nn.Sigmoid()

)

def forward(self, x):

return self.net(x)

# ============ 5. 实例化网络 ============

G = Generator(nz=latent\_dim, ngf=ngf).to(device)

D = Discriminator(ndf=ndf).to(device)

# ============ 6. 损失函数 & 优化器 ============

criterion = nn.BCELoss()

g\_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.5, 0.999))

d\_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(0.5, 0.999))

# ============ 7. 训练循环 ============

def denorm(x):

""" 将 [-1,1] 的生成结果映射回 [0,1] 用于可视化 """

out = (x + 1) / 2

return out.clamp(0, 1)

fixed\_noise = torch.randn(64, latent\_dim, 1, 1, device=device) # 固定噪声，观测训练进度

step = 0

for epoch in range(num\_epochs):

for i, (images, \_) in enumerate(train\_loader):

bs = images.size(0)

real\_labels = torch.ones(bs, 1, device=device)

fake\_labels = torch.zeros(bs, 1, device=device)

# ---------------------

# (1) 训练判别器

# ---------------------

images = images.to(device)

# 判别器对真实图的结果

outputs = D(images).view(-1, 1)

d\_loss\_real = criterion(outputs, real\_labels)

real\_score = outputs.mean().item()

# 判别器对假图的结果

z = torch.randn(bs, latent\_dim, 1, 1, device=device)

fake\_images = G(z)

outputs = D(fake\_images.detach()).view(-1, 1)

d\_loss\_fake = criterion(outputs, fake\_labels)

fake\_score = outputs.mean().item()

d\_loss = d\_loss\_real + d\_loss\_fake

d\_optimizer.zero\_grad()

d\_loss.backward()

d\_optimizer.step()

# ---------------------

# (2) 训练生成器

# ---------------------

# 让生成器骗过判别器 => 输出应为 real\_labels

outputs = D(fake\_images).view(-1, 1)

g\_loss = criterion(outputs, real\_labels)

g\_optimizer.zero\_grad()

g\_loss.backward()

g\_optimizer.step()

step += 1

if (i+1) % 200 == 0:

print(f"Epoch [{epoch+1}/{num\_epochs}], Step [{i+1}/{len(train\_loader)}], "

f"d\_loss: {d\_loss.item():.4f}, g\_loss: {g\_loss.item():.4f}, "

f"D(x): {real\_score:.2f}, D(G(z)): {fake\_score:.2f}")

# 每个 epoch 结束后，用固定噪声生成一批图像并保存

G.eval()

with torch.no\_grad():

fake = G(fixed\_noise).cpu()

G.train()

fake = denorm(fake)

save\_image(fake, os.path.join(sample\_dir, f'fake\_images\_epoch\_{epoch+1:03d}.png'), nrow=8)

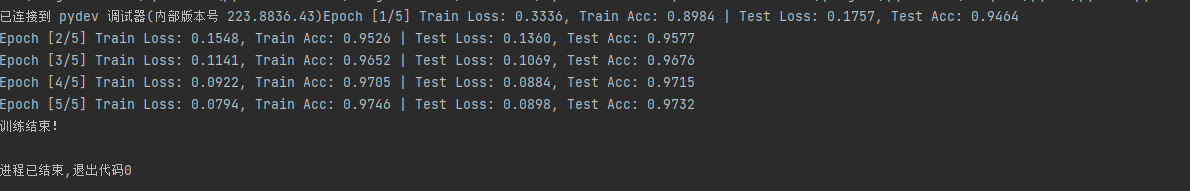
print("Training finished. Saving final model...")

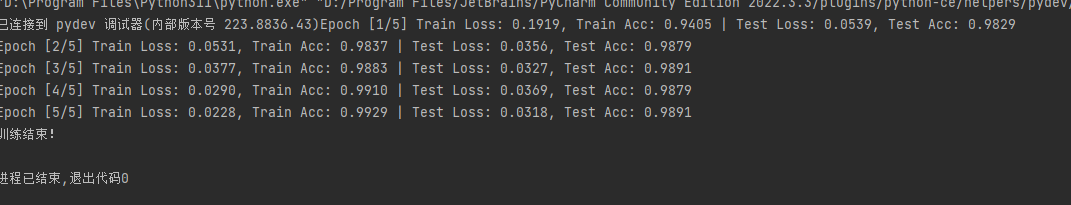
torch.save(G.state\_dict(), "generator\_dcgan\_32x32.pth")

torch.save(D.state\_dict(), "discriminator\_dcgan\_32x32.pth")

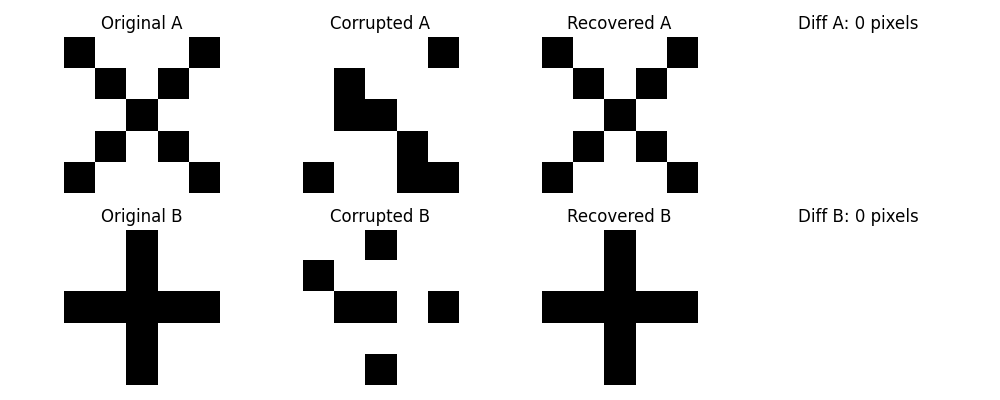
**9 附录**

9.1 实验1运行结果图





9.2 实验2 运行结果图



9.3 实验3运行结果图

