实验六

填充代码

GAN

```
# ============================== 生成器（Generator） ==============================
class Generator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            #TODO
            nn.Linear(input_dim, hidden_dim),            # 使用线性层将随机噪声映射到第一个隐藏层
            nn.ReLU(),       # 使用 ReLU 作为激活函数，帮助模型学习非线性特征
            #TODO
            nn.Linear(hidden_dim, hidden_dim),            # 使用线性层将第一个隐藏层映射到第二个隐
藏层
            nn.ReLU(),       # 再次使用 ReLU 激活函数
            #TODO
            nn.Linear(hidden_dim, output_dim),            # 使用线性层将第二个隐藏层映射到输出层，
输出为图像的像素大小
            nn.Tanh()        # 使用 Tanh 将输出归一化到 [-1, 1]，适用于图像生成
        )

    def forward(self, x):
        #TODO                # 前向传播：将输入 x 通过模型进行计算，得到生成的图像
        return self.model(x)

# ============================== 判别器（Discriminator） ==============================
class Discriminator(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            #TODO   # 输入层到第一个隐藏层，使用线性层
            nn.Linear(input_dim, hidden_dim),
            #TODO   # 使用 LeakyReLU 激活函数，避免梯度消失问题，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1),
            #TODO   # 第一个隐藏层到第二个隐藏层，使用线性层
            nn.Linear(hidden_dim, hidden_dim),
            #TODO   # 再次使用 LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1),
            #TODO   # 第二个隐藏层到输出层，使用线性层
            nn.Linear(hidden_dim, input_dim),
            #TODO   # 使用 Sigmoid 激活函数，将输出范围限制在 [0, 1]
            nn.Sigmoid()
        )

    def forward(self, x):
        #TODO        # 前向传播：将输入 x 通过模型进行计算，得到判别结果
        return self.model(x)
```

```
# 创建生成器G和判别器D，并移动到 GPU（如果可用）
    #TODO   # 生成器G
    G = Generator(input_dim=input_dim, hidden_dim= hidden_dim, output_dim=
output_dim).to(device)
    #TOOD   # 判别器D
    D = Discriminator(input_dim= output_dim, hidden_dim= hidden_dim).to(device)

    # 定义针对生成器G的优化器optim_G和针对判别器D的优化器optim_D，要求使用Adam优化器，学习率设置为0.0002
    #TODO  # 生成器优化器optim_G
    optim_G = torch.optim.Adam(G.parameters(), lr=0.0002)
    #TODO  # 判别器优化器optim_D
    optim_D = torch.optim.Adam(D.parameters(), lr=0.0002)

    loss_func = nn.BCELoss()  # 使用二元交叉熵损失
```

```
# ============================== 训练判别器 ==============================
def train_discriminator(real_images, D, G, loss_func, optim_D, batch_size, input_dim,
device):
    '''训练判别器'''
    real_images = real_images.view(-1, 28 * 28).to(device)  # 获取真实图像并展平
    real_output = D(real_images)  # 判别器预测真实图像
    #TODO   # 计算真实样本的损失real_loss
    real_loss = loss_func(real_output, real_images)
    noise = torch.randn(real_images.shape[0], input_dim, device=device)  # 生成随机噪声
    fake_images = G(noise).detach()  # 生成假图像（detach 避免梯度传递给 G）
    fake_output = D(fake_images)  # 判别器预测假图像

    #TODO   # 计算假样本的损失fake_loss
    fake_loss = loss_func(fake_output, real_images)

    loss_D = real_loss + fake_loss  # 判别器总损失
    optim_D.zero_grad()  # 清空梯度
    loss_D.backward()  # 反向传播
    optim_D.step()  # 更新判别器参数

    return loss_D.item()  # 返回标量损失

# ============================== 训练生成器 ==============================
def train_generator(D, G, loss_func, optim_G, batch_size, input_dim, device):
    '''训练生成器'''
    noise = torch.randn(batch_size, input_dim, device=device)  # 生成随机噪声
    fake_images = G(noise)  # 生成假图像
    fake_output = D(fake_images)  # 判别器对假图像的判断
    #TODO # 计算生成器损失（希望生成的图像判别为真）
    loss_G = loss_func(fake_output, torch.ones_like(fake_images))
    optim_G.zero_grad()  # 清空梯度
    loss_G.backward()  # 反向传播
    optim_G.step()  # 更新生成器参数

    return loss_G.item()  # 返回标量损失
```

DCGAN

```
# ============================== 生成器（Generator） ==============================
class Generator(nn.Module):
    def __init__(self, input_dim):
        super(Generator, self).__init__()

        # 1. 输入层：将 100 维随机噪声投影到 32x32（1024 维）
        #TODO   # 线性变换fc1，将输入噪声扩展到 1024 维
        self.fc1 = nn.Linear(input_dim, 32*32)
        self.br1 = nn.Sequential(
            #TODO   # 批归一化，加速训练并稳定收敛
            nn.BatchNorm1d(32*32),
            #TODO   # ReLU 激活函数，引入非线性
            nn.ReLU()
        )

        # 2. 第二层：将 1024 维数据映射到 128 * 7 * 7 维特征
        #TODO   # 线性变换fc2，将数据变换为适合卷积层的维数大小
        self.fc2 = nn.Linear(32*32, 128*7*7)
        self.br2 = nn.Sequential(
            #TODO   # 批归一化
            nn.BatchNorm1d(128*7*7),
            #TODO   # ReLU 激活函数
            nn.ReLU()
        )

        # 3. 反卷积层 1：上采样，输出 64 通道的 14x14 特征图
        self.conv1 = nn.Sequential(
            #TODO   # 反卷积：将 7x7 放大到 14x14，kernel size设置为4，stride设置为2，padding设置
为1
            nn.ConvTranspose2d(in_channels=128 ,out_channels=64,kernel_size=4, stride=2,
padding=1),
            #TODO   # 归一化，稳定训练
            nn.BatchNorm2d(64),
            #TODO   # ReLU 激活函数
            nn.ReLU()
        )

        # 4. 反卷积层 2：输出 1 通道的 28x28 图像
        self.conv2 = nn.Sequential(
            #TODO   # 反卷积：将 14x14 放大到 28x28，将 7x7 放大到 14x14，kernel size设置为4，
stride设置为2，padding设置为1
            nn.ConvTranspose2d(in_channels=64 ,out_channels=1 ,kernel_size=4, stride=2,
padding=1),
            #TODO   # Sigmoid 激活函数，将输出归一化到 [0,1]
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.br1(self.fc1(x))  # 通过全连接层，进行 BatchNorm 和 ReLU 激活
        x = self.br2(self.fc2(x))  # 继续通过全连接层，进行 BatchNorm 和 ReLU 激活
```

```
        x = x.reshape(-1, 128, 7, 7)  # 变形为适合卷积输入的形状 (batch, 128, 7, 7)
        x = self.conv1(x)  # 反卷积：上采样到 14x14
        output = self.conv2(x)  # 反卷积：上采样到 28x28
        return output  # 返回生成的图像

# ============================== 判别器（Discriminator） ==============================
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        # 1. 第一层：输入 1 通道的 28x28 图像，输出 32 通道的特征图，然后通过MaxPool2d降采样
        self.conv1 = nn.Sequential(
            #TODO  # 5x5 卷积核，步长为1
            nn.Conv2d(in_channels=1, out_channels=32 ,kernel_size=5, stride=1),
            #TODO   # LeakyReLU，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )
        self.pl1 = nn.MaxPool2d(2, stride=2)

        # 2. 第二层：输入 32 通道，输出 64 通道特征，然后通过MaxPool2d降采样
        self.conv2 = nn.Sequential(
            #TODO   # 5x5 卷积核，步长为1
            nn.Conv2d(in_channels=32, out_channels=64 ,kernel_size=5, stride=1),
            #TODO  # LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )
        self.pl2 = nn.MaxPool2d(2, stride=2)

        # 3. 全连接层 1：将 64x4x4 维特征图转换成 1024 维向量
        self.fc1 = nn.Sequential(
            #TODO   # 线性变换，将 64x4x4 映射到 1024 维
            nn.Linear(64*4*4, 1024),
            #TODO   # LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )

        # 4. 全连接层 2：最终输出真假概率
        self.fc2 = nn.Sequential(
            #TODO   # 线性变换，将 1024 维数据映射到 1 维
            nn.Linear(1024,1),
            #TODO   # Sigmoid 归一化到 [0,1] 作为概率输出
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.pl1(self.conv1(x))  # 第一层卷积，降维
        x = self.pl2(self.conv2(x))  # 第二层卷积，降维
        x = x.view(x.shape[0], -1)  # 展平成向量
        x = self.fc1(x)  # 通过全连接层
        output = self.fc2(x)  # 通过最后一层全连接层，输出真假概率
        return output  # 返回判别结果
```

```
    # 创建生成器和判别器，并移动到 GPU（如果可用）
    # TODO
    G = Generator(input_dim).to(device)
    # TODO
    D = Discriminator().to(device)

    # 定义优化器，优化器要求同任务一
    # TODO
    optim_G = torch.optim.Adam(G.parameters(), lr=0.0002)
    # TODO
    optim_D = torch.optim.Adam(D.parameters(), lr=0.0002)
    loss_func = nn.BCELoss()
```

```
 ============================== 训练判别器 ==============================
def train_discriminator(real_images, D, G, loss_func, optim_D, batch_size, input_dim,
device):
    '''训练判别器'''
    real_images = real_images.to(device)
    real_output = D(real_images) # 判别器预测真实图像
    real_target = real_images.mean(dim = [2,3])
    #TODO   # 计算真实样本的损失real_loss
    real_loss = loss_func(real_output, real_target)
    noise = torch.randn(real_images.shape[0], input_dim, device=device)  # 生成随机噪声
    fake_images = G(noise).detach()  # 生成假图像（detach 避免梯度传递给 G）
    fake_output = D(fake_images)  # 判别器预测假图像

    #TODO   # 计算假样本的损失fake_loss
    fake_loss = loss_func(fake_output, real_target)

    loss_D = real_loss + fake_loss  # 判别器总损失
    optim_D.zero_grad()  # 清空梯度
    loss_D.backward()  # 反向传播
    optim_D.step()  # 更新判别器参数

    return loss_D.item()  # 返回标量损失


# ============================== 训练生成器 ==============================
def train_generator(D, G, loss_func, optim_G, batch_size, input_dim, device):
    '''训练生成器'''
    noise = torch.randn(batch_size, input_dim, device=device).to(device)  # 生成随机噪声
    fake_images = G(noise)  # 生成假图像
    fake_output = D(fake_images)  # 判别器对假图像的判断
    #TODO # 计算生成器损失（希望生成的图像判别为真）
    loss_G = loss_func(fake_output, torch.ones(batch_size,1).to(device))
    optim_G.zero_grad()  # 清空梯度
    loss_G.backward()  # 反向传播
    optim_G.step()  # 更新生成器参数

    return loss_G.item()  # 返回标量损失
```

WGAN

```
# ============================== 生成器（Generator） ==============================
class Generator(nn.Module):
    def __init__(self, input_dim):
        super(Generator, self).__init__()

        # 1. 输入层：将 100 维随机噪声从input_dim投影到 32x32（1024 维）
        #TODO    # 线性变换fc1，将输入噪声扩展到 1024 维
        self.fc1 = nn.Linear(100,32*32)

        self.br1 = nn.Sequential(
            #TODO    # 批归一化，加速训练并稳定收敛
            nn.BatchNorm1d(32*32),
            #TODO    # ReLU 激活函数，引入非线性
            nn.ReLU()
        )

        # 2. 第二层：将 1024 维数据映射到 128 * 7 * 7 维
        #TODO    # 线性变换，将数据变换为适合卷积层的维数大小
        self.fc2 = nn.Linear(1024,128*7*7)
        self.br2 = nn.Sequential(
            #TODO    # 批归一化
            nn.BatchNorm1d(128*7*7),
            #TODO    # ReLU 激活函数
            nn.ReLU()
        )

        # 3. 反卷积层 1：上采样，输出 64 通道的 14x14 特征图
        self.conv1 = nn.Sequential(
            #TODO    # 反卷积：将 7x7 放大到 14x14，kernel size设置为4，stride设置为2，padding设置
为1
            nn.ConvTranspose2d(in_channels=128 ,out_channels=64,kernel_size=4, stride=2,
padding=1),
            #TODO    # 归一化，稳定训练
            nn.BatchNorm2d(64),
            #TODO    # ReLU 激活函数
            nn.ReLU()
        )

        # 4. 反卷积层 2：输出 1 通道的 28x28 图像
        self.conv2 = nn.Sequential(
            #TODO    # 反卷积：将 14x14 放大到 28x28，将 7x7 放大到 14x14，kernel size设置为4，
stride设置为2，padding设置为1
            nn.ConvTranspose2d(in_channels=64 ,out_channels=1 ,kernel_size=4, stride=2,
padding=1),
            #TODO     # WGAN 需要使用 Tanh 激活函数，将输出范围限制在 [-1, 1]
            nn.Tanh()
        )

    def forward(self, x):
        x = self.br1(self.fc1(x))  # 通过全连接层，进行 BatchNorm 和 ReLU 激活
```

```
        x = self.br2(self.fc2(x))  # 继续通过全连接层，进行 BatchNorm 和 ReLU 激活
        x = x.reshape(-1, 128, 7, 7)  # 变形为适合卷积输入的形状 (batch, 128, 7, 7)
        x = self.conv1(x)  # 反卷积：上采样到 14x14
        output = self.conv2(x)  # 反卷积：上采样到 28x28
        return output  # 返回生成的图像

# ============================== 判别器（Discriminator） ==============================
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        # 1. 第一层：输入 1 通道的 28x28 图像，输出 32 通道的特征图，然后通过MaxPool2d降采样
        self.conv1 = nn.Sequential(
             #TODO  # 5x5 卷积核，步长为1
            nn.Conv2d(in_channels=1, out_channels=32 ,kernel_size=5, stride=1),
            #TODO   # LeakyReLU，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )
        self.pl1 = nn.MaxPool2d(2, stride=2)

        # 2. 第二层：输入 32 通道，输出 64 通道特征，然后通过MaxPool2d降采样
        self.conv2 = nn.Sequential(
            #TODO   # 5x5 卷积核，步长为1
            nn.Conv2d(in_channels=32, out_channels=64 ,kernel_size=5, stride=1),
            #TODO  # LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )
        self.pl2 = nn.MaxPool2d(2, stride=2)

        # 3. 全连接层 1：将 64x4x4 维特征图转换成 1024 维向量
        self.fc1 = nn.Sequential(
           #TODO   # 线性变换，将 64x4x4 映射到 1024 维
            nn.Linear(64*4*4, 1024),
            #TODO   # LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(negative_slope=0.1)
        )

        # 4. 全连接层 2：最终输出
        #TODO # 输出一个标量作为判别结果
        self.fc2 = nn.Linear(1024,1)

    def forward(self, x):
        x = self.pl1(self.conv1(x))  # 第一层卷积，降维
        x = self.pl2(self.conv2(x))  # 第二层卷积，降维
        x = x.view(x.shape[0], -1)  # 展平成向量
        x = self.fc1(x)  # 通过全连接层
        output = self.fc2(x)  # 通过最后一层全连接层，输出标量
        return output  # 返回判别结果
```

```
  # 创建生成器G和判别器D，并移动到 GPU（如果可用）
    ## TODO
    G = Generator(input_dim).to(device)
```

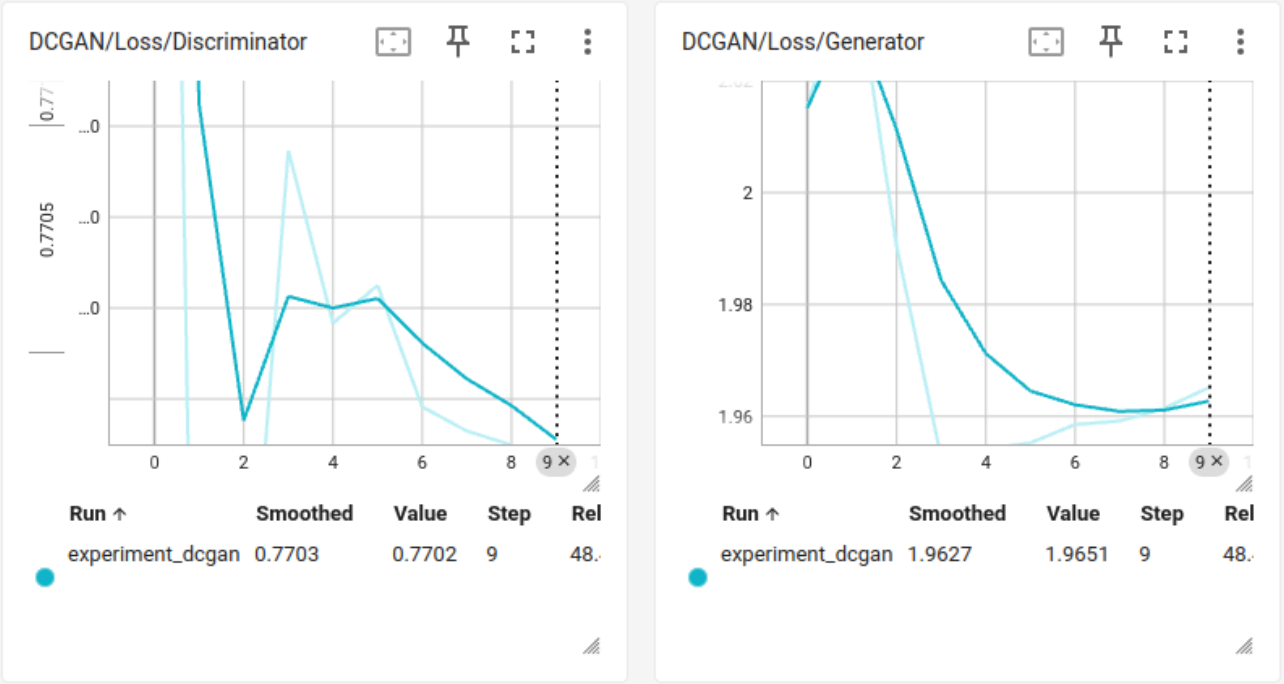
```
    # TODO
    D = Discriminator().to(device)

    # 定义优化器optim_G和optim_D：使用RMSprop，学习率设置为0.00005
    # TODO
    optim_G = torch.optim.RMSprop(G.parameters(), lr = 0.00005)
    # TODO
    optim_D = torch.optim.RMSprop(D.parameters(), lr = 0.00005)
    # 初始化 TensorBoard
    writer = SummaryWriter(log_dir='./logs/experiment_wgan')

    # 开始训练
    for epoch in range(num_epoch):
        total_loss_D, total_loss_G = 0, 0
        for i, (real_images, _) in enumerate(train_loader):
            # 判别器训练5次，然后训练生成器1次。提示：for循环，记得修改total_loss_D和total_loss_G的值
            # TODO  # 判别器训练 5 次
            for __ in range(5):
                loss_D = train_discriminator(real_images, D, G, optim_D, clip_value,
batch_size, input_dim, device)
                total_loss_D += loss_D
            # TODO  # 生成器训练 1 次
            loss_G = train_generator(D, G, optim_G, batch_size, input_dim, device)
            total_loss_G += loss_G

            # 每 100 步打印一次损失
            if (i + 1) % 100 == 0 or (i + 1) == len(train_loader):
                print(f'Epoch {epoch:02d} | Step {i + 1:04d} / {len(train_loader)} | Loss_D
{total_loss_D / (i + 1):.4f} | Loss_G {total_loss_G / (i + 1):.4f}')
```

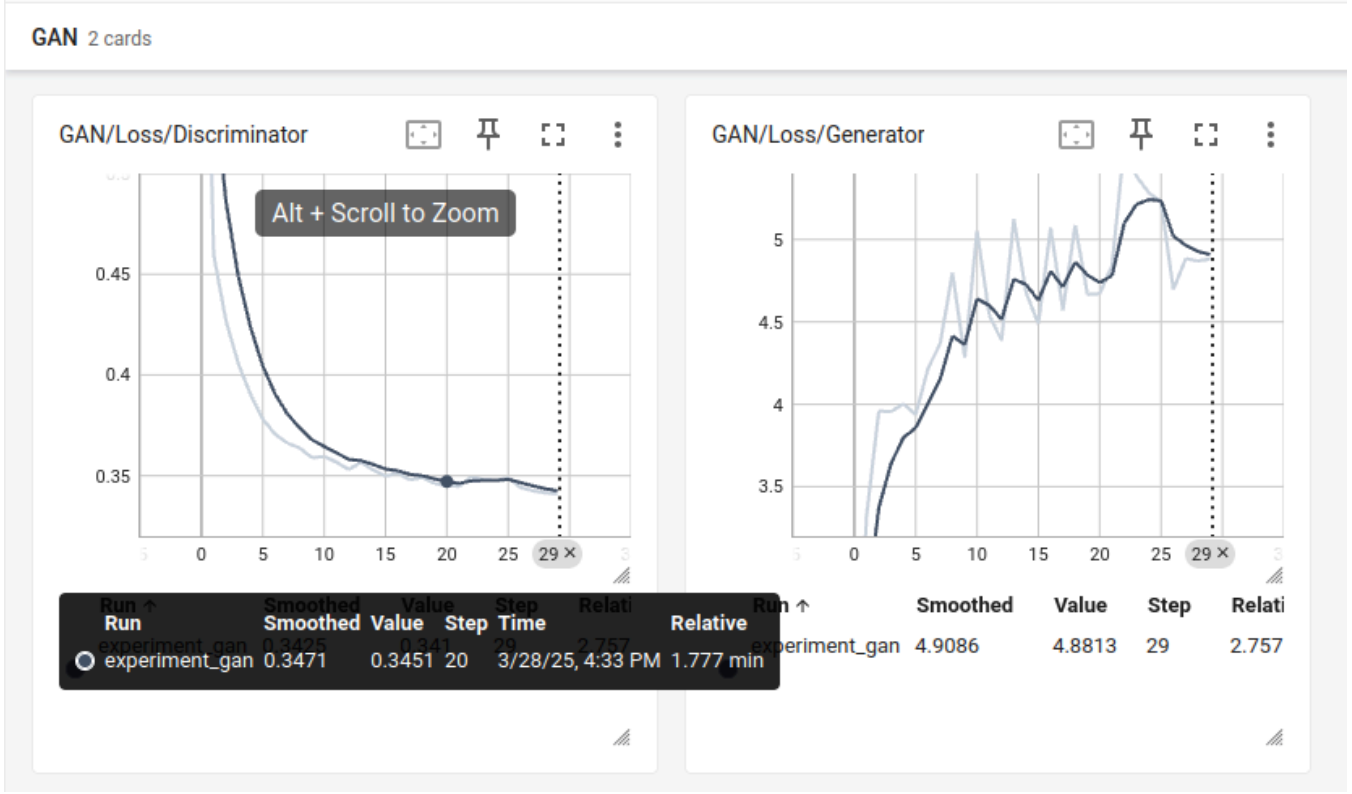
```
# ============================== 训练判别器 ==============================
def train_discriminator(real_images, D, G, optim_D, clip_value, batch_size, input_dim,
device):
    '''训练判别器'''

    real_images = real_images.to(device)
    real_output = D(real_images)

    noise = torch.randn(batch_size, input_dim, device=device)
    fake_images = G(noise).detach()
    fake_output = D(fake_images)

    #TODO  # 计算判别器的损失函数
    loss_D = -(torch.mean(real_output) - torch.mean(fake_output))
    optim_D.zero_grad()
    loss_D.backward()
    optim_D.step()

    # 对判别器参数进行裁剪
    for p in D.parameters():
        p.data.clamp_(-clip_value, clip_value)
```

```
    return loss_D.item()

# ============================== 训练生成器 ==============================
def train_generator(D, G, optim_G, batch_size, input_dim, device):
    '''训练生成器'''

    noise = torch.randn(batch_size, input_dim, device=device)
    fake_images = G(noise)
    fake_output = D(fake_images)

    #TODO  # 计算生成器的损失函数
    loss_G = -torch.mean(fake_output)
    optim_G.zero_grad()
    loss_G.backward()
    optim_G.step()

    return loss_G.item()
```

实验结果

DCGAN 2 cards





思考题

思考题1: 为什么GAN的训练被描述为一个对抗过程？这种对抗机制如何促进生成器的改进？

生成器的目标是生成逼真的数据以欺骗判别器，而判别器的目标是准确区分真实数据与生成数据。

训练中，生成器通过最小化判别器正确识别的概率（即最大化判别器的误判率），而判别器则通过最大化自身判断的准确率。这种竞争迫使生成器不断优化其生成能力。

思考题2: ReLU和LeakyReLU各有什么特征？为什么在生成器中使用ReLU而在判别器中使用LeakyReLU?

- **生成器**：通常需要生成高维、复杂的结构化数据。ReLU的稀疏激活特性有助于生成器聚焦关键特征，同时其计算高效性适合生成任务。此外，生成器的输出层常结合`tanh`或`sigmoid`函数约束值域，中间层使用ReLU可平衡表达能力和效率。
- **判别器**：需要处理真实与生成数据的细微差异。若使用ReLU，负梯度消失可能导致判别器过早失效，削弱对生成器的指导信号。而LeakyReLU保留负区间的微弱梯度，确保判别器在训练中持续更新，提供更稳定的对抗反馈，从而提升整体训练鲁棒性。

思考题1: DCGAN与传统GAN的主要区别是什么？为什么DCGAN更适合图像生成任务?

传统GAN使用全连接层,DCGAN采用全卷积网络,在生成器和判别器中广泛使用批归一化

卷积操作天然适合图像的局部相关性，通过多层卷积可提取从边缘到纹理的高层语义特征。转置卷积生成图像时能逐步恢复细节，判别器通过卷积压缩冗余信息，提升对抗训练的效率和生成分辨率

思考题2: DCGAN的生成器和判别器分别使用了哪些关键的网络结构？这些结构如何影响生成效果?

生成器的关键结构:

转置卷积层,批归一化,ReLU激活函数.

判别器的关键结构:

卷积层,LeakyReLU激活函数,全局平均池化

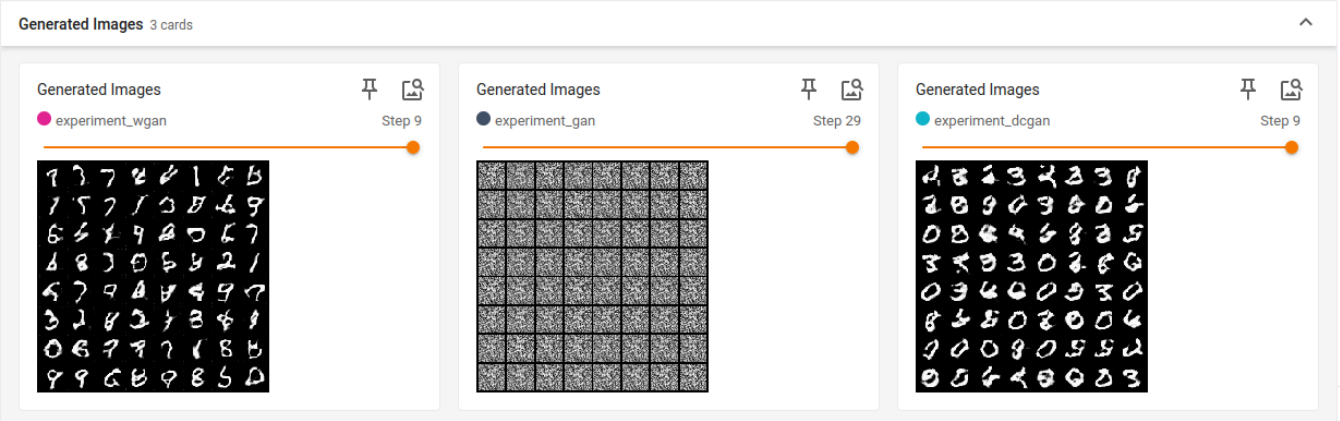
思考题3: DCGAN中为什么使用批归一化（Batch Normalization）？它对训练过程有什么影响?

1. **稳定梯度传播**：GAN的生成器和判别器存在训练不均衡问题，BN通过归一化每层的输入分布，缓解内部协变量偏移
2. **防止模式崩溃**：BN对每批样本的统计量进行归一化，强制生成器关注多样化的特征分布，减少生成样本的同质化。
3. **加速收敛**：归一化后的数据分布更接近标准正态分布，允许使用更大的学习率，缩短训练时间。

思考题1: WGAN与原始GAN的主要区别是什么？为什么WGAN能够改善GAN的训练稳定性?

1. **损失函数设计**:
    - **原始GAN**：基于JS散度当生成分布与真实分布无重叠时，梯度消失，导致训练停滞。
    - **WGAN**：改用Wasserstein距离，衡量两分布之间的“推土成本”。即使分布无重叠，仍能提供有意义的梯度。
2. **判别器（Critic）的输出**:
    - **原始GAN**：判别器输出概率，目标是二分类（真/假）。
    - **WGAN**：判别器输出标量分数，无需Sigmoid，目标是直接估计Wasserstein距离。
3.

思考题2: 对于每个GAN模型（GAN， DCGAN， WGAN），在报告中展示TensorBoard中记录的损失函数变化曲线图和不同epoch时输出的图像（分别添加在epoch总数的0%、25%、50%、75%、100%处输出的图像）；直观分析损失函数的变化趋势和生成图像的变化趋势。



- **GAN**：
    - 判别器损失快速下降至接近0，生成器损失上升，反映模式崩溃。
- **DCGAN**：
    - D_loss与G_loss交替波动，但整体趋势逐渐收敛，反映稳定的对抗过程。
- **WGAN**：
    - Critic损失逐渐下降，生成器损失平稳降低，反映Wasserstein距离的单调优化。

  图像:
    - **GAN**：部分样本质量高，但多样性差。
    - **DCGAN**：细节丰富，接近真实分布，但部分区域过平滑。
    - **WGAN**：图像清晰且多样性最佳，边缘更自然。

思考题3: 尝试调整超参数提升生成图片的质量。从生成的图片上直观来看，GAN，DCGAN和WGAN的效果有什么差别？你认为是什么导致了这种差别?

关键超参数调整策略：

1. **学习率**：
    - 降低学习率可提升WGAN稳定性；GAN需更谨慎调整以防梯度爆炸。
2. **批归一化**：
    - 在DCGAN中增加BN层数可减少生成噪声，但过量会导致过平滑。

生成效果对比：

- **GAN**：
    - 生成图像模糊，易出现模式重复（如相似纹理）。
    - 原因：JS散度梯度消失导致生成器无法精细调整。
- **DCGAN**：
    - 细节更清晰（如人脸五官），但部分区域存在伪影（如不规则斑点）。
    - 原因：卷积结构增强局部特征提取，但对抗不平衡仍引入噪声。
- **WGAN**：

- 图像边缘平滑，多样性最佳（如不同角度、光照的物体）。
- 原因：Wasserstein距离提供连续优化目标，Critic的Lipschitz约束抑制过拟合。

差异根源：

1. **损失函数性质**：
   - GAN的JS散度对分布重叠敏感，而Wasserstein距离无此限制，使WGAN更鲁棒。
2. **网络结构**：
   - DCGAN的卷积层优于GAN的全连接层，但未解决梯度消失本质问题。
3. **训练动态**：
   - WGAN的Critic通过权重约束维持稳定的梯度流，而GAN的判别器易“压倒”生成器。

实验感想与收获

通过本次对GAN、DCGAN和WGAN的对比实验，我对生成对抗网络的原理、优化策略以及实际应用有了更深刻的理解,从损失函数的设计到激活函数的选择，每一个环节都直接影响最终性能。TensorBoard的图像可视化输出给了最终性能一个直观的展现