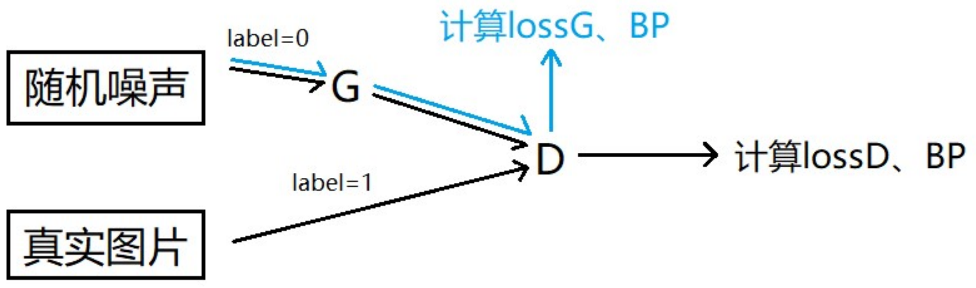
题目 1：构建并训练单生成器-判别器的基础 GAN 模型

架构设计：

模型架构：使用 DCGAN 进行网络构建，生成器 G：四层(反卷积 Conv+归一化 Batchnorm2d+激活函数 Relu)+一层(反卷积 Conv+激活函数 Tanh)。

训练流程：

如图：



对于 G：用随机噪声作为输入，G 生成输出后交由 D 判别，G 的目标是判别为真，朝这个目标进行训练与梯度下降，使用二元交叉熵作为损失函数。

对于 D：对 G 生成的假数据与真数据进行判别，D 的目标是分类正确，朝这个目标进行训练与梯度下降，使用二元交叉熵作为损失函数。

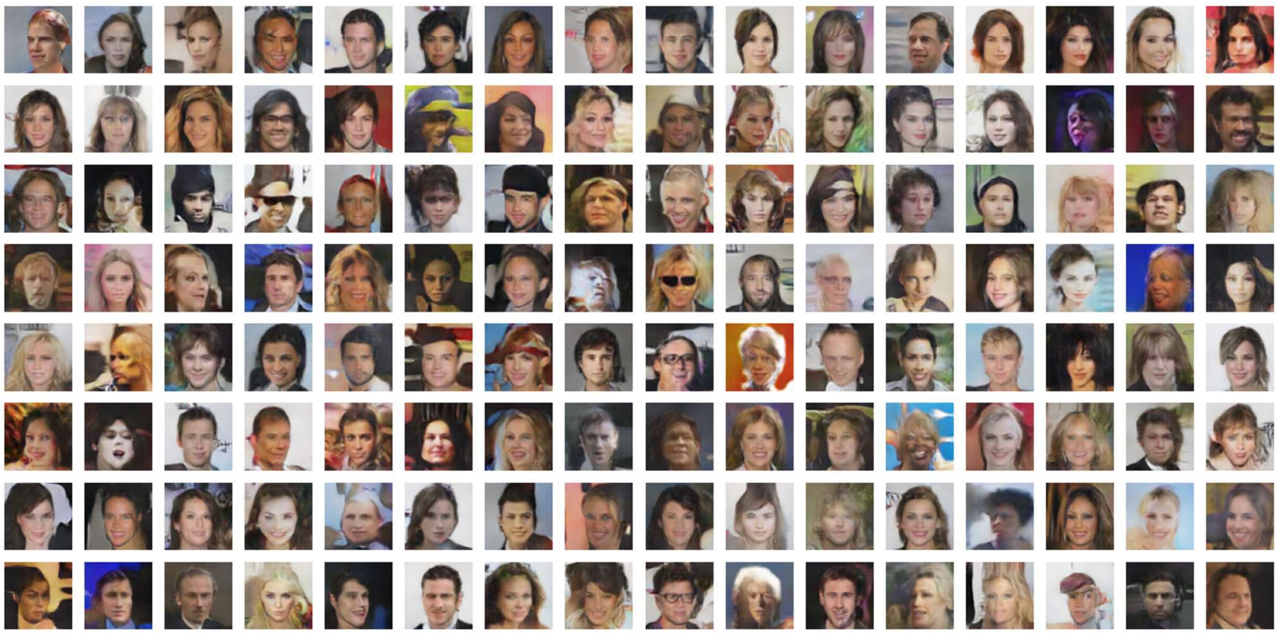
参数选择：

预处理：根据生产环境将图片放缩、裁剪为 64*64 像素大小，再将 RGB 通道(0,1)标准化为(-1,1)，方便后续处理，Batchsize 设为 128。

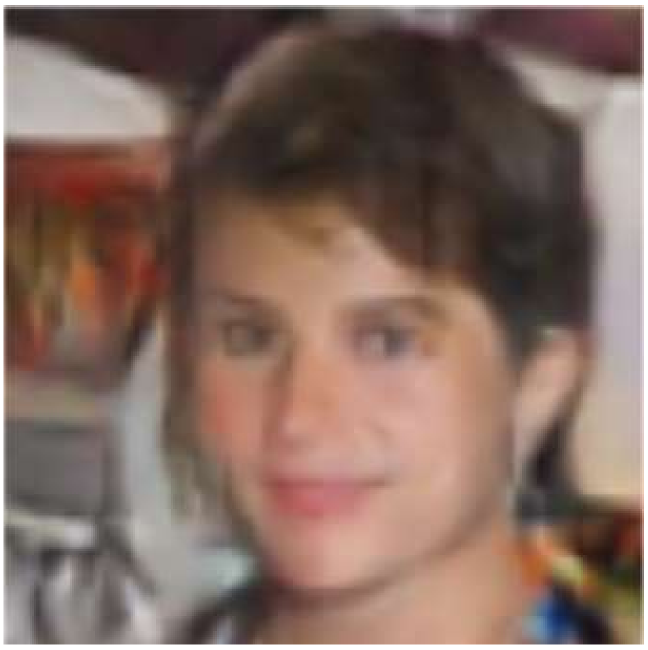
模型初始化：采用 DCGAN 论文推荐的权重初始化：均值为 0，标准差为 0.02。

优化器参数均使用 GAN 模型常见初始值。

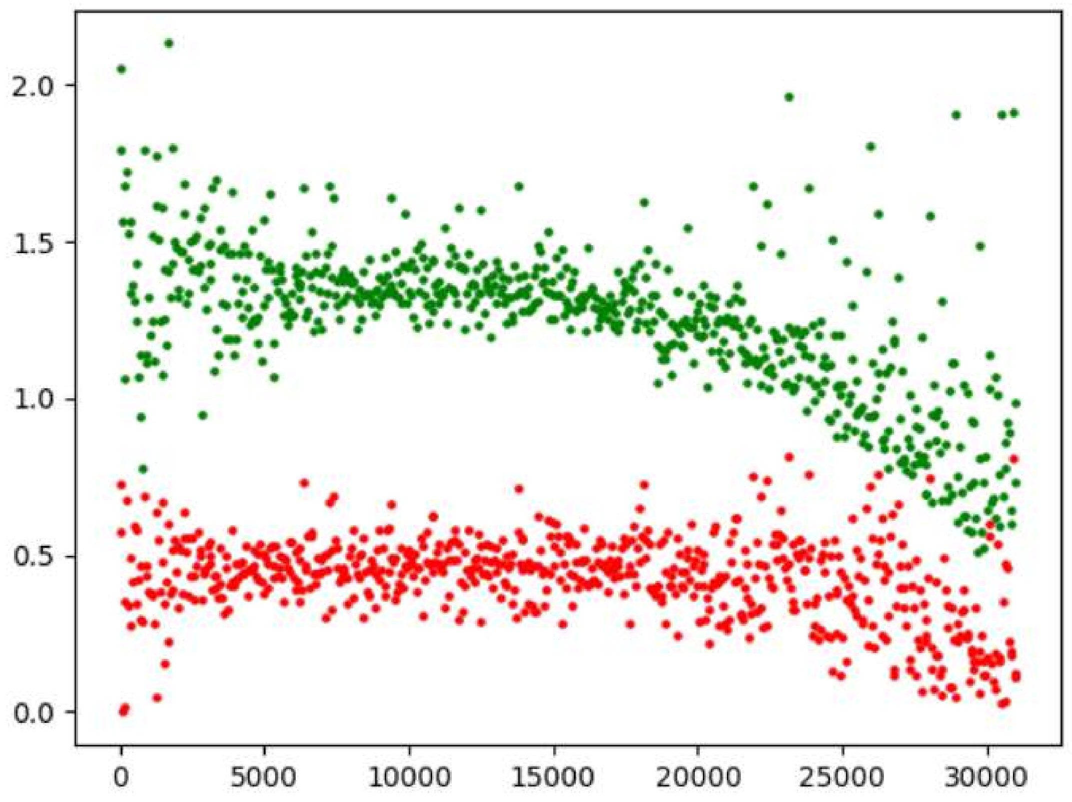
模型的生成效果:



包括生成样本的图片示例：



损失曲线：



完整代码：

```
import torch
import torchvision
import torch.nn as nn
from torchvision import transforms
import random
from tqdm import tqdm
import matplotlib.pyplot as plt
random.seed(722)
torch.manual_seed(722)

#预处理，加载数据集
dataset = torchvision.datasets.ImageFolder(root='./img',
transform=transforms.Compose([
    transforms.Resize(64),
    transforms.CenterCrop(64),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
]))
dataloader = torch.utils.data.DataLoader(dataset, batch_size=128, shuffle=True,
num_workers=4, drop_last = True)
```

```
noise_size = 100

#设置网络架构
class Gnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(in_channels=noise_size, out_channels=64 * 8,
kernel_size=4, stride=1, padding=0,
                               bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(),

            nn.ConvTranspose2d(in_channels=64 * 8, out_channels=64 * 4,
kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(),

            nn.ConvTranspose2d(in_channels=64 * 4, out_channels=64 * 2,
kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(),

            nn.ConvTranspose2d(in_channels=64 * 2, out_channels=64,
kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),

            nn.ConvTranspose2d(in_channels=64, out_channels=3, kernel_size=4,
stride=2, padding=1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)
class Dnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2,
padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64 * 2, kernel_size=4,
stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(in_channels=64 * 2, out_channels=64 * 4, kernel_size=4,
stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(in_channels=64 * 4, out_channels=64 * 8, kernel_size=4,
stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(in_channels=64 * 8, out_channels=1, kernel_size=4,
stride=1, padding=0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.net(x)
loss = nn.BCELoss()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
G_NET = Gnet().to(device)
D_NET = Dnet().to(device)
opt_G = torch.optim.Adam(G_NET.parameters(), lr=0.0002, betas=(0.5, 0.999))
opt_D = torch.optim.Adam(D_NET.parameters(), lr=0.0002, betas=(0.5, 0.999))
true_label = torch.ones(128).to(device)
false_label = torch.zeros(128).to(device)

#模型初始化函数
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        print("init:%s"%classname)
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        print("init:%s"%classname)
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
if __name__ == '__main__':
    G_NET.apply(weights_init)
    D_NET.apply(weights_init)
    epoch = 20

    for e in tqdm(range(epoch)):
        for i, (image, label) in enumerate(dataloader):
            #训练 G
            G_NET.zero_grad()
            noise = torch.randn(128, noise_size, 1, 1, device=device)
            fake = G_NET(noise)
            output = D_NET(fake).view(-1)
            loss_G = loss(output,true_label)
            loss_G.backward()
            loss_G_mean = output.mean().item()
            opt_G.step()
            #训练 D
            if i%2 == 0:
                D_NET.zero_grad()
                real_image = image.to(device)
                output_real = D_NET(real_image).view(-1)
                loss_D_real = loss(output_real, true_label)
                output_fake = D_NET(fake.detach()).view(-1)
                loss_D_fake = loss(output_fake, false_label)
                loss_D = loss_D_real + loss_D_fake
                loss_D_mean = loss_D.mean().item()
                loss_D.backward()
                opt_D.step()

            if i % 50 == 0:
                print("epoch:%i/[%i]
iter:%i/[1550]"%(e,epoch,i),'|',"loss_D:%f"%loss_D_mean,'|',"loss_G:%f"%loss_G_
mean)
                iter = e*1550+i
                plt.scatter(iter,loss_G_mean, c='r',s = 5)
                plt.scatter(iter,loss_D_mean, c='g',s = 5)



torch.save(obj=G_NET.state_dict(),f='weight/G_Net_parameter_%i_epoch.pth'%e)
        torch.save(obj=D_NET.state_dict(),
f='weight/D_Net_parameter_%i_epoch.pth' % e)

    plt.show()

#验证、查看统计数据部分
noise = torch.randn(128, noise_size, 1, 1, device=device)
```

```
img = G_NET(noise)
img = img.detach().cpu()
img = img.permute(0, 2, 3, 1).numpy()
img = (img + 1) / 2
fig, axes = plt.subplots(8, 16, figsize=(16, 8))
axes = axes.flatten()

for i in range(128):
    axes[i].imshow(img[i])
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```

题目 2：扩展为多生成器-多判别器结构

实现细节：

在使用架构进行实例时进行多个生成器-判别器的实例化，并放在同一个列表中，训练时保证相对于每个 batch 为并行训练。多个生成器-判别器组之间完全独立，互不影响。

相对于题目一的改动：

创建实例时：

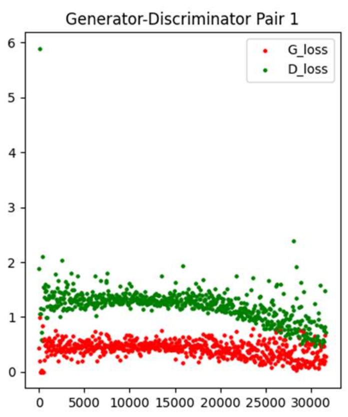
```
G_NETS = [Gnet().to(device) for _ in range(3)]
D_NETS = [Dnet().to(device) for _ in range(3)]
opt_G = [torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
for G in G_NETS]
opt_D = [torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
for D in D_NETS]
```

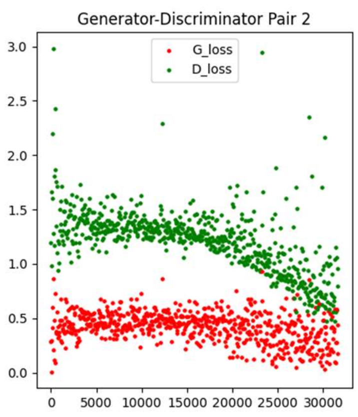
main 函数中：

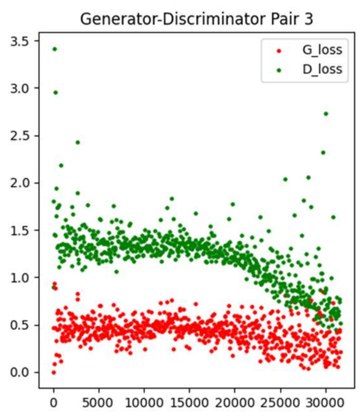
```
    for G in G_NETS:
        G.apply(weights_init)
    for D in D_NETS:
        D.apply(weights_init)
     for idx in range(3):
                # 生成器训练
                G_NETS[idx].zero_grad()
```

```
fake = G_NETS[idx](noise_batch[idx])
output = D_NETS[idx](fake).view(-1)
loss_G = loss(output, true_label)
loss_G.backward()
opt_G[idx].step()

# 判别器训练
if i % 2 == 0:
    D_NETS[idx].zero_grad()
    output_real = D_NETS[idx](real_image).view(-1)
    loss_D_real = loss(output_real, true_label)

    output_fake = D_NETS[idx](fake.detach()).view(-1)
    loss_D_fake = loss(output_fake, false_label)
    loss_D = loss_D_real + loss_D_fake
    loss_D.backward()
    opt_D[idx].step()

if i % 50 == 0:
    print(f"epoch: {e}/{epoch}, iter: {i}, G{idx+1}_loss:
{loss_G.item():.4f}, D{idx+1}_loss: {loss_D.item():.4f}")
```

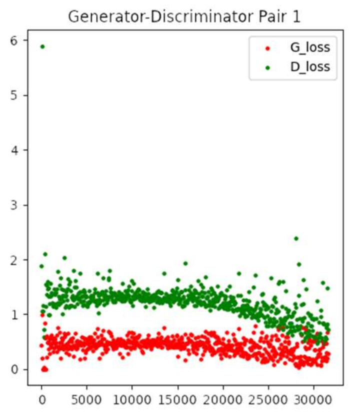
模型性能比较：

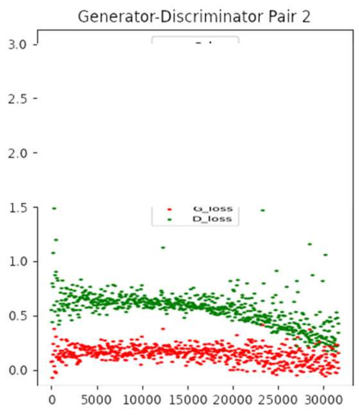


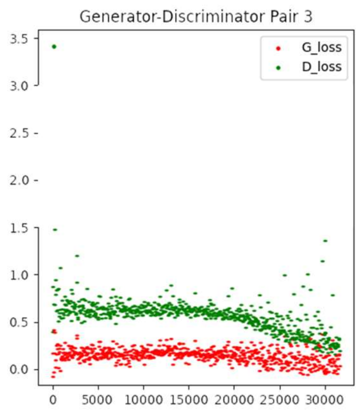




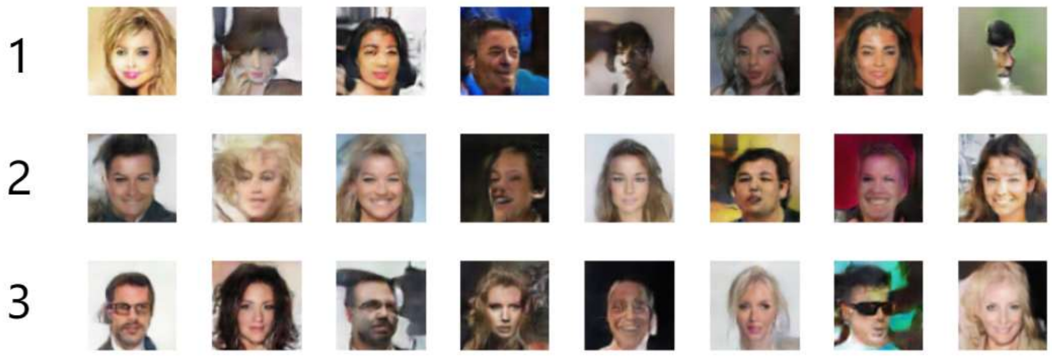
放缩到相同尺度：







模型结果比较：



题目 3：构建评估机制，筛选相对最优生成器

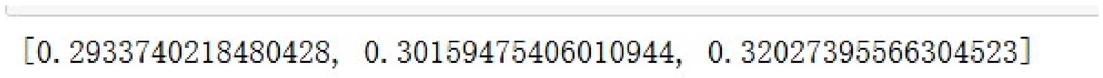
筛选机制：

使用判别器的输出值作为评估分数，为了防止 G 只生成满足同组对应 D 要求的图片，评估时采用三组 D 对应一组 G 的方式，如第一组的 G 的得分为：全三组 D 的打分的平均值。

实现代码&筛选函数：

```
scores = []
noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _ in
range(3)]
for idx in range(len(G_NETS)):
    output = torch.zeros(128, device=device)
    for idx2 in range(len(G_NETS)):
        fake = G_NETS[idx](noise_batch[idx])
        output += D_NETS[idx2](fake).view(-1)
    score = output.mean().item()/3
    scores.append(score)
print(scores)
```

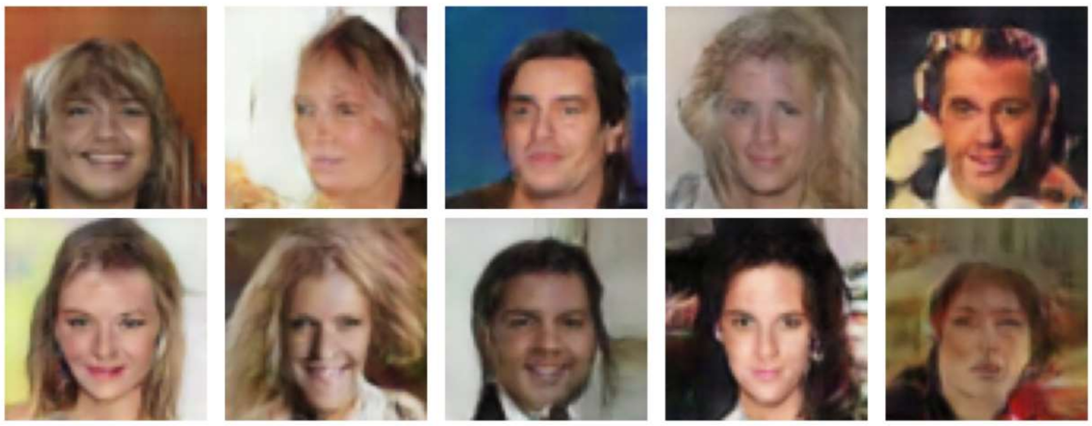
评分：



```
[0.2933740218480428, 0.30159475406010944, 0.32027395566304523]
```

因此第三个生成器的效果最好

相对最优生成器的生成结果：



题目 4、5：实现“赛马”中的对齐机制（循环）

原理：

利用题目 3 的评估方法，选出最优生成器，在每一个 epoch 结束后进行对齐操作。具体操作为，将最优的生成器与当前生成器进行加权结合，加权比例为动态计算：

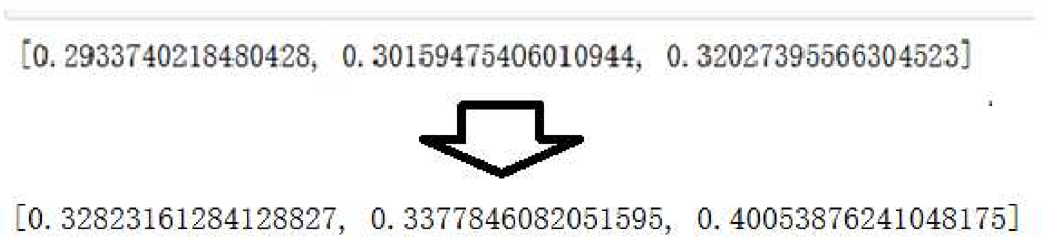
最优生成器的权重=(最优生成器得分-当前生成器得分)/ 最优生成器得分

这样，越差的生成器向最优生成器对齐的幅度越大，而最优生成器则几乎没有变化，实现了“对齐”的操作。

新增代码：

```
#对齐机制（循环）（p4p5）
    scores = []
    noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _ in
range(3)]
    for idx in range(len(G_NETS)):
        output = torch.zeros(128, device=device)
        for idx2 in range(len(G_NETS)):
            fake = G_NETS[idx](noise_batch[idx])
            output += D_NETS[idx2](fake).view(-1)
        score = output.mean().item()/3
        scores.append(score)
    best_i = scores.index(max(scores))
    print(scores)
    for i in range(3):
        weight_update_ratio = (scores[best_i]-scores[i])/scores[best_i]
        if i == best_i:
            continue  # 不调整最优生成器的权重
        for param_g, param_best_g in zip(G_NETS[i].parameters(),
G_NETS[best_i].parameters()):
            param_g.data = param_g.data * (1 - weight_update_ratio) +
param_best_g.data * weight_update_ratio
        print("set",i,"to",best_i)
#对齐机制（循环）（p4p5）
```

相对于题目三，评分分布发生的变化：



```
[0.2933740218480428, 0.30159475406010944, 0.32027395566304523]
```

```
[0.32823161284128827, 0.3377846082051595, 0.40053876241048175]
```

题目 6：引入随机扰动机制以增强多样性

随机扰动的选择：

在每一个 epoch 结束后，对三组 G-D 进行评估，按一定的权重添加标准差为 0.01 的高斯噪声。由于噪声会对模型产生干扰，为了增强模型的多样性，只需要向非最优的 G-D 添加噪声，且分数越低噪声越大。

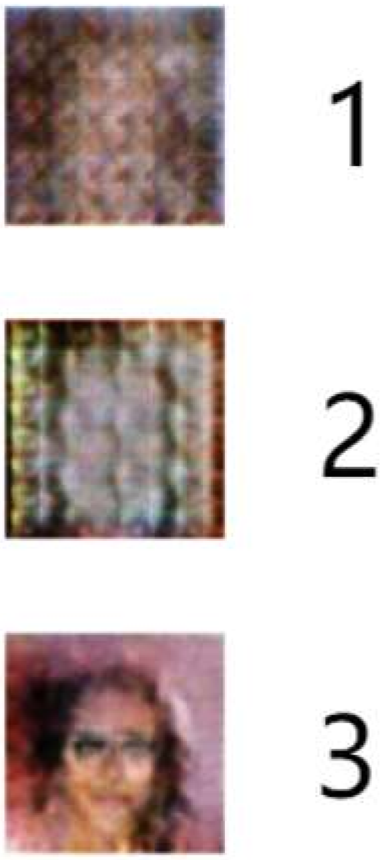
代码：

```
# 定义随机扰动函数
def add_random_noise(param, noise_std=0.01):
    noise = torch.randn_like(param) * noise_std
    param.data += noise
```

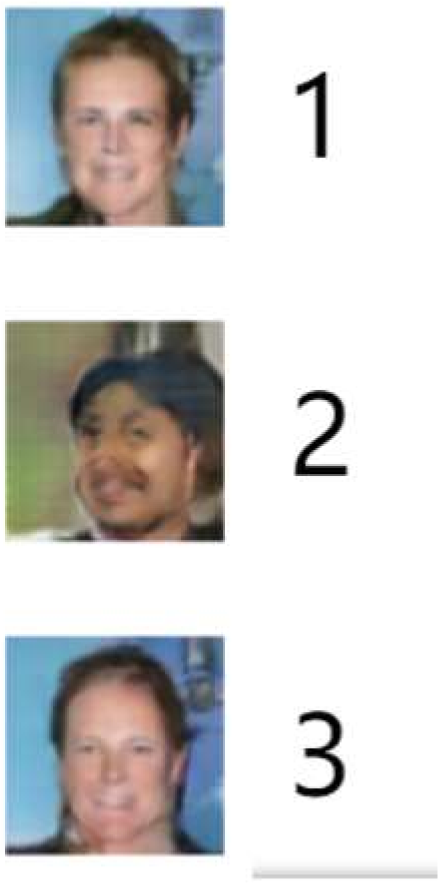
```
# 每个 epoch 添加随机扰动
noise_std = 0.01 * (1 - weight_update_ratio)  # 噪声的标准差与更新比例相关
add_random_noise(param_g, noise_std)  # 向当前生成器的权重添加随机噪声
```

效果：

刚刚添加噪声时:



重新迭代后:



可以看出早期 1、2 组为非最优组，加入噪声后明显受到干扰。经过训练后 2 组出现了明显的不同，产生了多样化。

题目 7：实现动态学习率调整

为了让非最优 G-D 组尽快追赶最优组，我们希望他的学习率更大些，而最优组偏小些。

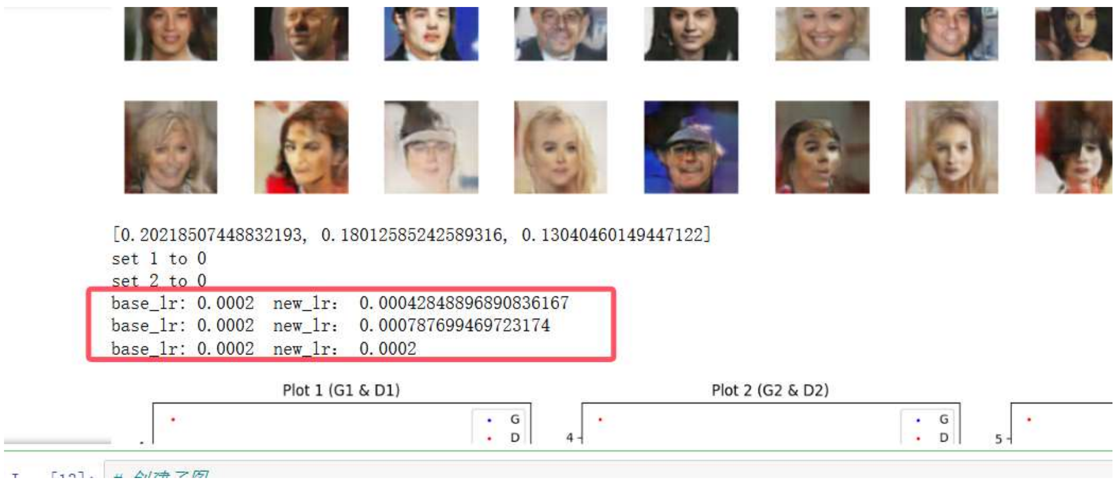
学习率调整函数设计：

```
#动态学习率调整（p7）
def adjust_learning_rate(optimizer, base_lr, current_loss, best_loss,
lr_factor=0.1):
    # 计算学习率调整系数
    loss_diff = current_loss - best_loss
    lr_multiplier = 1 + (loss_diff / best_loss)
    new_lr = base_lr * max(lr_multiplier, lr_factor)  # 确保最小学习率为
lr_factor * base_lr
    print("base_lr:",base_lr," new_lr: ", new_lr)
    # 更新优化器的学习率
    for param_group in optimizer.param_groups:
        param_group['lr'] = new_lr
```

训练过程中添加下面的代码：

```
adjust_learning_rate(opt_G[idx], base_lr=0.0002, current_loss=loss_G_mean[idx],
best_loss=min(loss_G_mean))
```

训练过程中学习率的变化：



```
[0.20218507448832193, 0.18012585242589316, 0.13040460149447122]
set 1 to 0
set 2 to 0
base_lr: 0.0002  new_lr:  0.00042848896890836167
base_lr: 0.0002  new_lr:  0.0007876994697231 74
base_lr: 0.0002  new_lr:  0.0002
```

在每个 epoch 中进行一次学习率的改变

题目 8：构建多层对抗机制，提升训练效果

设计：

添加 G-D 的中间一层进行对抗，因此需要 G-D 中间层的输出，因此需要对 DCGAN 架构进行改动：

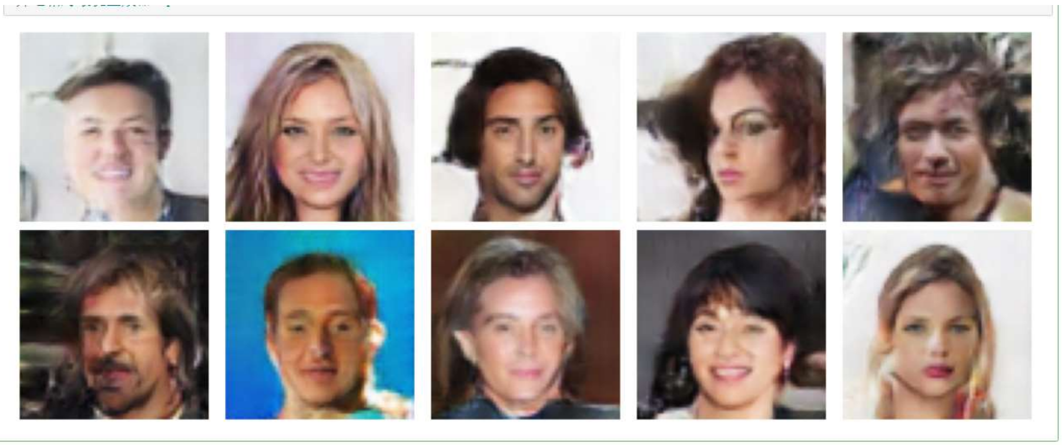
```
class Gnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(noise_size, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(),
            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(),

            nn.ConvTranspose2d(64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(),
        )

        self.final_layers = nn.Sequential(
            nn.ConvTranspose2d(64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.net(x)
        mid_output = x.clone()
        x = self.final_layers(x)
        return x, mid_output

class Dnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.final_layers = nn.Sequential(
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.net(x)
        mid_output = x.clone()
        x = self.final_layers(x)
        return x, mid_output
```

有了中间层输出，我们规定中间层的损失函数为 L1，并且加上原本的损失函数（末尾层）共同组成新的损失函数：

```
lossmid = nn.L1Loss()

real_out, dis_mid = D_NETS[idx](real_image)
fake_out, fake_dis_mid = D_NETS[idx](fake)

loss_Gend = loss(fake_out.view(-1), true_label)
loss_Gmid = lossmid(fake_dis_mid.view(-1), dis_mid.view(-1)) #中间层损失(L1)
loss_G = loss_Gend + loss_Gmid
```

后续对 loss_G 进行梯度下降即可。

结果：



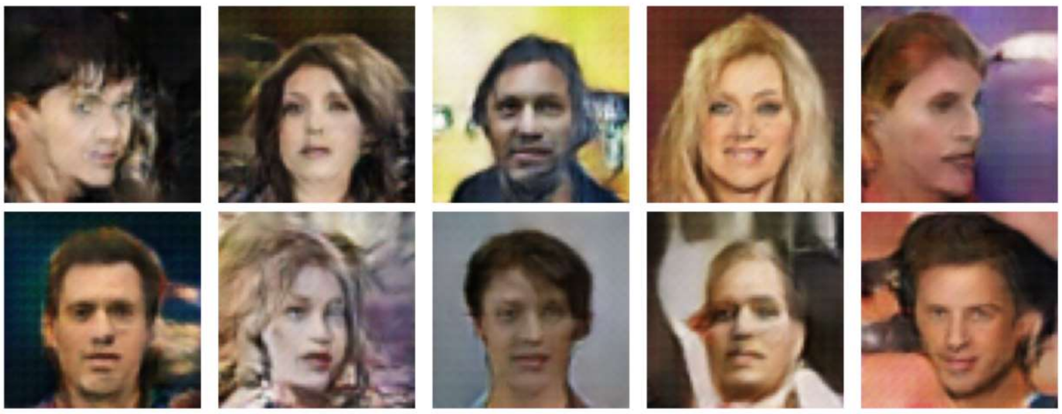
题目 9、10：引入权重惩罚机制， 实现完整训练

在最后几个 epoch 按照三个 G-D 的得分进行加权，合并成一个新的生成器：

```
G_sum = Gnet().to(device)
for i in range(3):
    for param1, param2 in zip(G_sum.parameters(), G_NETS[i].parameters()):
            param1.data = param1.data + param2.data * (scores[i]/sum(scores))
```

合并后完成剩下的 epoch。

结果：



到此，所有的机制已经全部应用。由于时间有限，无法对所有机制进行调整至合适的参数、轮数、方式。因此最终效果并不为最优效果，更多侧重于技术测试。

下面是包含全部机制的代码：

```
import os
img_folder = 'weight'
if not os.path.exists(img_folder):
    os.makedirs(img_folder)
import torch
import torchvision
import torch.nn as nn
from torchvision import transforms
import random
from tqdm import tqdm
```

```
import matplotlib.pyplot as plt
from PIL import Image
random.seed(722)
torch.manual_seed(722)
dataset = torchvision.datasets.ImageFolder(root='./img',
transform=transforms.Compose([
    transforms.Resize(64),
    transforms.CenterCrop(64),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
]))
dataloader = torch.utils.data.DataLoader(dataset, batch_size=128, shuffle=True,
num_workers=4, drop_last = True)
noise_size = 100
class Gnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(noise_size, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(),

            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(),

            nn.ConvTranspose2d(64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(),
        )

        self.final_layers = nn.Sequential(
            nn.ConvTranspose2d(64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.net(x)
        mid_output = x.clone()
        x = self.final_layers(x)
        return x, mid_output
```

```
class Dnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.final_layers = nn.Sequential(
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.net(x)
        mid_output = x.clone()
        x = self.final_layers(x)
        return x, mid_output
loss = nn.BCELoss()
lossmid = nn.L1Loss()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
G_NETS = [Gnet().to(device) for _ in range(3)]
D_NETS = [Dnet().to(device) for _ in range(3)]
opt_G = [torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999)) for G
in G_NETS]
opt_D = [torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999)) for D
in D_NETS]
true_label = torch.ones(128).to(device)
false_label = torch.zeros(128).to(device)

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        print("init:%s"%classname)
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        print("init:%s"%classname)
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# new
# def weights_init(m):
#     if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):  # 只对有权重的
层应用初始化
#         print(f"Initializing {m.__class__.__name__}")
#         nn.init.normal_(m.weight.data, 0.0, 0.02)
#     elif isinstance(m, nn.BatchNorm2d):
#         print(f"Initializing {m.__class__.__name__}")
#         nn.init.normal_(m.weight.data, 1.0, 0.02)
#         nn.init.constant_(m.bias.data, 0)

# 添加随机扰动（p6）
def add_random_noise(param, noise_std=0.01):
    noise = torch.randn_like(param) * noise_std
    param.data += noise
# 添加随机扰动（p6）
#动态学习率调整（p7）
def adjust_learning_rate(optimizer, base_lr, current_loss, best_loss,
lr_factor=0.1):
    # 计算学习率调整系数
    loss_diff = current_loss - best_loss
    lr_multiplier = 1 + (loss_diff / best_loss)
    new_lr = base_lr * max(lr_multiplier, lr_factor)  # 确保最小学习率为
lr_factor * base_lr
    print("base_lr:",base_lr," new_lr: ", new_lr)
    # 更新优化器的学习率
    for param_group in optimizer.param_groups:
        param_group['lr'] = new_lr
#动态学习率调整（p7）
for G in G_NETS:
    G.apply(weights_init)
for D in D_NETS:
    D.apply(weights_init)

G1=[]
D1=[]
```

```
G2=[]
D2=[]
G3=[]
D3=[]
groups = [(G1, D1), (G2, D2), (G3, D3)]
epoch = 20

for e in tqdm(range(epoch)):
    loss_G_mean = [None]*3
    loss_D_mean = [None]*3
    for i, (image, _) in enumerate(dataloader):
        real_image = image.to(device)
        noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _
in range(3)]

        for idx in range(3):
            # 生成器训练
            G_NETS[idx].zero_grad()
            fake, gen_mid = G_NETS[idx](noise_batch[idx])
            real_out, dis_mid = D_NETS[idx](real_image)
            fake_out, fake_dis_mid = D_NETS[idx](fake)
            loss_Gend = loss(fake_out.view(-1), true_label)
            loss_Gmid = lossmid(fake_dis_mid.view(-1), dis_mid.view(-1))
            loss_G = loss_Gend + loss_Gmid
            loss_G.backward()
            loss_G_mean[idx] = fake_out.mean().item()
            opt_G[idx].step()

            # 判别器训练
            if i % 2 == 0:
                D_NETS[idx].zero_grad()
                output_real = D_NETS[idx](real_image)[0].view(-1)
                loss_D_real = loss(output_real, true_label)
                output_fake = D_NETS[idx](fake.detach())[0].view(-1)
                loss_D_fake = loss(output_fake, false_label)
                loss_D = loss_D_real + loss_D_fake
                loss_D_mean[idx] = loss_D.mean().item()
                loss_D.backward()
                opt_D[idx].step()

            if i % 50 == 0:
                iter = e * len(dataloader) + i
                new_spotG = (iter, loss_G_mean[idx])
                new_spotD = (iter, loss_D_mean[idx])
                groups[idx][0].append(new_spotG)
                groups[idx][1].append(new_spotD)
                print(f”epoch: {e}/{epoch}, iter: {i}, G{idx+1}_loss:
{loss_G.item():.4f}, D{idx+1}_loss: {loss_D.item():.4f}”)
                if idx == 2:
                    noisetest = torch.randn(128, noise_size, 1, 1,
device=device)  # 产生正态分布的噪声
                    for idx2 in range(3):
                        img = G_NETS[idx2](noisetest)[0]
                        img = img.detach().cpu()
                        img = img.permute(0, 2, 3, 1).numpy()
                        img = (img + 1) / 2
                        fig, axes = plt.subplots(1, 8, figsize=(8, 1))
                        axes = axes.flatten()

                        for j in range(8):
                            axes[j].imshow(img[j])
                            axes[j].axis(’off’)  # 关闭坐标轴

                        plt.tight_layout()  # 自动调整子图间距
                        plt.show()

#对齐机制（循环）（p4p5）
    scores = []
    noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _ in
range(3)]
    for idx in range(len(G_NETS)):
        output = torch.zeros(128, device=device)
        for idx2 in range(len(G_NETS)):
            fake = G_NETS[idx](noise_batch[idx])[0]
            output += D_NETS[idx2](fake)[0].view(-1)
        score = output.mean().item()/3
        scores.append(score)
    best_i = scores.index(max(scores))
    print(scores)
    for i in range(3):
        weight_update_ratio = (scores[best_i]-scores[i])/scores[best_i]
        if i == best_i:
            continue  # 不调整最优生成器的权重
        for param_g, param_best_g in zip(G_NETS[i].parameters(),
G_NETS[best_i].parameters()):
            param_g.data = param_g.data * (1 - weight_update_ratio) +
param_best_g.data * weight_update_ratio
            # 添加随机扰动（p6）
            noise_std = 0.01 * (1 - weight_update_ratio)  # 噪声的标准差与更新
比例相关
            add_random_noise(param_g, noise_std)  # 向当前生成器的权重添加随机
噪声
            # 添加随机扰动（p6）
        print("set",i,"to",best_i)
#对齐机制（循环）（p4p5）

    for idx in range(3):
        #动态学习率调整（p7）
        adjust_learning_rate(opt_G[idx], base_lr=0.0002,
current_loss=loss_G_mean[idx], best_loss=min(loss_G_mean))
        #动态学习率调整（p7）
        torch.save(G_NETS[idx].state_dict(),
f'weight/G_Net_{idx+1}_epoch_{e}.pth')
        torch.save(D_NETS[idx].state_dict(),
f'weight/D_Net_{idx+1}_epoch_{e}.pth')

    # 创建子图
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    groups = [(G1, D1), (G2, D2), (G3, D3)]

    # 绘制每个子图
    for i, (G, D) in enumerate(groups):
        # 提取 G 和 D 的坐标
        G_x = [point[0] for point in G]
        G_y = [point[1] for point in G]
        D_x = [point[0] for point in D]
        D_y = [point[1] for point in D]

        # 绘制 G 和 D
        axes[i].scatter(G_x, G_y, color='blue',s=3 , label='G')
        axes[i].scatter(D_x, D_y, color='red',s=3 , label='D')
        axes[i].set_title(f'Plot {i+1} (G{i+1} & D{i+1})')
        axes[i].set_xlabel('X')
        axes[i].set_ylabel('Y')
        axes[i].legend()
    # 显示图形
    plt.tight_layout()
    plt.show()


plt.show()
```

```
for ax in axes:
    ax.set_xlim(0, epoch * len(dataloader))  # 横坐标统一范围
    ax.set_ylim(0, 6)  # 纵坐标统一范围
plt.show()
noise = torch.randn(128, noise_size, 1, 1, device=device)  # 产生正态分布的噪声

for i in range(3):
    img = G_NETS[i](noise)[0]
    img = img.detach().cpu()
    img = img.permute(0, 2, 3, 1).numpy()

    img = (img + 1) / 2

    fig, axes = plt.subplots(1, 8, figsize=(8, 1))
    axes = axes.flatten()

    for j in range(8):
        axes[j].imshow(img[j])
        axes[j].axis('off')  # 关闭坐标轴

    plt.tight_layout()  # 自动调整子图间距
    plt.show()
#筛选相对最优生成器（p3）
scores = []
noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _ in
range(3)]
for idx in range(len(G_NETS)):
    output = torch.zeros(128, device=device)
    for idx2 in range(len(G_NETS)):
        fake = G_NETS[idx](noise_batch[idx])[0]
        output += D_NETS[idx2](fake)[0].view(-1)
    score = output.mean().item()/3
    scores.append(score)
print(scores)
noise = torch.randn(128, noise_size, 1, 1, device=device)
i = 1
img = G_NETS[i](noise)[0]
img = img.detach().cpu()
img = img.permute(0, 2, 3, 1).numpy()

img = (img + 1) / 2

fig, axes = plt.subplots(2, 5, figsize=(10, 4))
axes = axes.flatten()
```

```
for j in range(10):
    axes[j].imshow(img[j])
    axes[j].axis('off')
plt.tight_layout()
plt.show()
#筛选相对最优生成器（p3）

#合并后继续训练（p9）
G_sum = Gnet().to(device)
for i in range(3):
    for param1, param2 in zip(G_sum.parameters(), G_NETS[i].parameters()):
            param1.data = param1.data + param2.data * (scores[i]/sum(scores))

epoch = 2

for e in tqdm(range(epoch)):

    for i, (image, _) in enumerate(dataloader):
        real_image = image.to(device)
        noise_batch = [torch.randn(128, noise_size, 1, 1, device=device) for _
in range(3)]

        for idx in range(3):
            # 生成器训练
            G_sum.zero_grad()
            fake, gen_mid = G_sum(noise_batch[idx])
            real_out, dis_mid = D_NETS[idx](real_image)
            fake_out, fake_dis_mid = D_NETS[idx](fake)
            loss_Gend = loss(fake_out.view(-1), true_label)
            loss_Gmid = lossmid(fake_dis_mid.view(-1), dis_mid.view(-1))
            loss_G = loss_Gend + loss_Gmid
            loss_G.backward()
            opt_G[idx].step()

            # 判别器训练
            if i % 2 == 0:
                D_NETS[idx].zero_grad()
                output_real = D_NETS[idx](real_image)[0].view(-1)
                loss_D_real = loss(output_real, true_label)
                output_fake = D_NETS[idx](fake.detach())[0].view(-1)
                loss_D_fake = loss(output_fake, false_label)
                loss_D = loss_D_real + loss_D_fake
                loss_D.backward()
                opt_D[idx].step()

            if i % 50 == 0:
                iter = e * len(dataloader) + i
                print(f"epoch: {e}/{epoch}, iter: {i}, G{idx+1}_loss:
{loss_G.item():.4f}, D{idx+1}_loss: {loss_D.item():.4f}")
                if idx == 2:
                    noisetest = torch.randn(128, noise_size, 1, 1,
device=device)  # 产生正态分布的噪声
                    img = G_sum(noisetest)[0]
                    img = img.detach().cpu()
                    img = img.permute(0, 2, 3, 1).numpy()
                    img = (img + 1) / 2
                    fig, axes = plt.subplots(1, 8, figsize=(8, 1))
                    axes = axes.flatten()

                    for j in range(8):
                        axes[j].imshow(img[j])
                        axes[j].axis('off')  # 关闭坐标轴

                    plt.tight_layout()  # 自动调整子图间距
                    plt.show()


    for idx in range(3):
        torch.save(G_sum.state_dict(), f'weight/G_sum.pth')


noise = torch.randn(128, noise_size, 1, 1, device=device)
img = G_sum(noise)[0]
img = img.detach().cpu()
img = img.permute(0, 2, 3, 1).numpy()

img = (img + 1) / 2

fig, axes = plt.subplots(2, 5, figsize=(10, 4))
axes = axes.flatten()

for j in range(10):
    axes[j].imshow(img[j])
    axes[j].axis('off')

plt.tight_layout()
plt.show()
```

```
#合并后继续训练（p9）
```