第二次作业　使用 GAN 进行 CelebA 数据集图像填充生成

生成式人工智能

吴天阳 *4124136039* 人工智能 *B2480*

1　应用实践

使用 GAN 模型在 CelebA 数据集上进行图像填充实验，并进行改进，以下是各模块代码.

1.1　模型定义

基础版的 GAN 模型生成器采用编码-解码结构，将输入图像通过一系列卷积层降维后再利用转置卷积层还原回原始尺寸，实现图像的生成或重建. 判别器则通过多层卷积和全连接层对输入图像进行特征提取与分类，输出图像为真实或生成的概率. 代码如下

```python
import torch
import torch.nn as nn
from torch.nn.utils import spectral_norm

class Generator(nn.Module):
    def __init__(self):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1),  # 64x64 -> 32x32
            nn.ReLU(),
            nn.Conv2d(64, 128, 4, 2, 1),  # 32x32 -> 16x16
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 256, 4, 2, 1),  # -> 8x8
            nn.BatchNorm2d(256),
            nn.ReLU()
        )

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, 4, 2, 1),  # 8x8 -> 16x16
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, 4, 2, 1),   # 16x16 -> 32x32
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, 4, 2, 1),     # 32x32 -> 64x64
            nn.Tanh()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

class Discriminator(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Flatten(),
            nn.Linear(128 * 16 * 16, 1),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.main(x)

if __name__ == '__main__':
    # Test the models
    G = Generator()
    D = Discriminator()

    x = torch.randn(8, 3, 64, 64)  # Batch of 8 images
    fake_images = G(x)
    print("Generator output shape:", fake_images.shape)

    D_output = D(fake_images)
    print("Discriminator output shape:", D_output.shape)  # Should be (8,
    ↪  1) for batch size of 8
```

改进版 GAN 模型在生成器中引入了 UNet 结构，通过跳跃连接保留了编码器的中间特征，有效提升了生成图像的细节与清晰度. 同时，判别器采用 PatchGAN 架构，并引入谱归一化，增强了训练稳定性和对局部纹理的判别能力，使得模型在保持全局一致性的同时更擅长处理图像局部细节，整体效果优于基础版本. 代码如下

```
import torch
import torch.nn as nn
from torch.nn.utils import spectral_norm

# 使用 UNet 结构的生成器，支持 skip-connection，提升细节保留
class UNetGenerator(nn.Module):
    def __init__(self):
        super().__init__()
        # 编码器
        self.enc1 = nn.Sequential(nn.Conv2d(3, 64, 4, 2, 1), nn.ReLU())
        ↪  # 64x64 -> 32x32
        self.enc2 = nn.Sequential(nn.Conv2d(64, 128, 4, 2, 1),
        ↪  nn.BatchNorm2d(128), nn.ReLU())  # 32->16
        self.enc3 = nn.Sequential(nn.Conv2d(128, 256, 4, 2, 1),
        ↪  nn.BatchNorm2d(256), nn.ReLU()) # 16->8

        # 解码器
        self.dec1 = nn.Sequential(nn.ConvTranspose2d(256, 128, 4, 2, 1),
        ↪  nn.BatchNorm2d(128), nn.ReLU()) # 8->16
```

```
        self.dec2 = nn.Sequential(nn.ConvTranspose2d(256, 64, 4, 2, 1),
        ↪  nn.BatchNorm2d(64), nn.ReLU())   # 16->32
        self.dec3 = nn.Sequential(nn.ConvTranspose2d(128, 3, 4, 2, 1),
        ↪  nn.Tanh())                       # 32->64

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(e1)
        e3 = self.enc3(e2)

        d1 = self.dec1(e3)
        d2 = self.dec2(torch.cat([d1, e2], dim=1))
        out = self.dec3(torch.cat([d2, e1], dim=1))
        return out

class PatchDiscriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            spectral_norm(nn.Conv2d(3, 64, 4, 2, 1)),
            nn.LeakyReLU(0.2),
            spectral_norm(nn.Conv2d(64, 128, 4, 2, 1)),
            nn.LeakyReLU(0.2),
            spectral_norm(nn.Conv2d(128, 256, 4, 2, 1)),
            nn.LeakyReLU(0.2),
            spectral_norm(nn.Conv2d(256, 1, 4, 1, 0)),  # PatchGAN
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.main(x)

# Alias for compatibility
Generator = UNetGenerator
Discriminator = PatchDiscriminator
```

1.2 训练设置

在训练中，优化模型采用 Two Time-Scale Update Rule，将判别器的学习率设为生成器的一半，有助于平衡两者训练速度，避免判别器过快收敛导致生成器训练困难. 利用预训练 VGG 网络提取的高层特征来衡量生成图像与真实图像的感知差异，增强生成图像的结构和语义一致性，提升视觉质量. 对生成器使用 0.9 的平滑标签代替 1，提高判别器对真实图像的容错性，从而缓解过拟合并提升稳定性.

```
batch_size = 64
lr = 2e-4
num_epochs = 50
image_size = 64

# 准备数据集
transform = transforms.Compose([
```

```
    transforms.CenterCrop(128),
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

random_mask_applier = RandomMaskApplier(mask_dir=const.mask_dir, mask_size=(64, 64))
train_dataset = CelebaCustomDataset(
    img_dir=const.celeba_dir,
    transform=transform,
    mask_fn=random_mask_applier.apply_mask,
    train=True
)
eval_dataset = CelebaCustomDataset(
    img_dir=const.celeba_dir,
    transform=transform,
    mask_fn=random_mask_applier.apply_mask,
    train=False
)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
eval_dataloader = DataLoader(eval_dataset, batch_size=batch_size, shuffle=False)

# 模型、损失、优化器
G = Generator().to(device)
D = Discriminator().to(device)
criterion = torch.nn.BCELoss()
l1_loss = torch.nn.L1Loss()
opt_G = torch.optim.Adam(G.parameters(), lr=lr, betas=(0.5, 0.999))  # TTUR G
opt_D = torch.optim.Adam(D.parameters(), lr=lr * 0.5, betas=(0.5, 0.999))  # TTUR D



from torchvision.models import vgg16

# 加载 VGG 感知损失网络
vgg = vgg16(pretrained=True).features[:16].eval().to(device)
for param in vgg.parameters():
    param.requires_grad = False

def perceptual_loss(fake, real):
    return torch.nn.functional.l1_loss(vgg(fake), vgg(real))

from torchvision.models import vgg16

# 感知损失网络
vgg = vgg16(pretrained=True).features[:16].eval().to(device)
```

```
for param in vgg.parameters():
    param.requires_grad = False

def perceptual_loss(fake, real):
    return torch.nn.functional.l1_loss(vgg(fake), vgg(real))
```

损失函数计算如下：

```
masked, real = masked.to(device), real.to(device)
fake = G(masked)

# 判别器
D_real = D(real).view(-1)
D_fake = D(fake.detach()).view(-1)
loss_D = criterion(D_real, torch.ones_like(D_real)) + \
            criterion(D_fake, torch.zeros_like(D_fake))

opt_D.zero_grad()
loss_D.backward()
opt_D.step()

# 生成器
D_fake = D(fake).view(-1)
vgg_loss = perceptual_loss(fake, real)

# 加上和真实图片的 l1 损失
loss_G = criterion(D_fake, torch.ones_like(D_fake) * 0.9) + 10 *
↪  l1_loss(fake, real) + 0.1 * vgg_loss

opt_G.zero_grad()
loss_G.backward()
opt_G.step()
```

1.3 训练结果

分别训练两个模型各 50 个 epochs，进行图像填充测试，测试结果如下

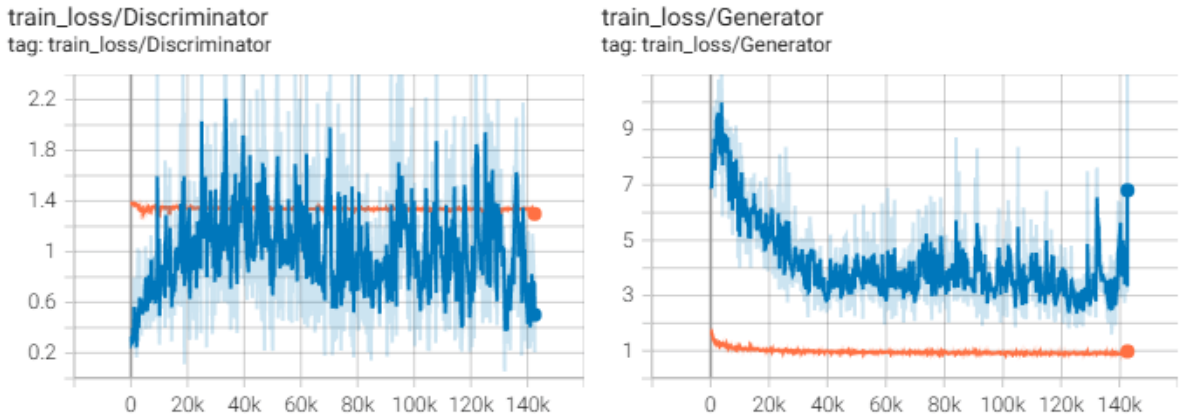

图 1: 训练集曲线图（蓝色为普通版，橙色为改进版）

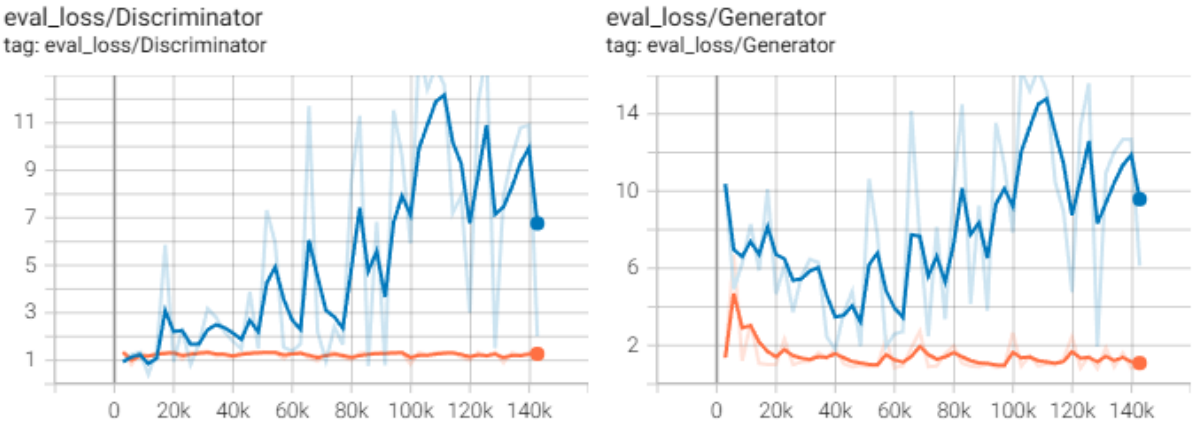

图 2: 验证集曲线图（蓝色为普通版，橙色为改进版）

1.4 图像填充效果

(a) 普通版生成效果　　(b) 改进版生成效果

图 3: 图像填充效果对比（左为普通版，右为改进版）

可以发现改进版的训练更加稳定，修复后的图像在细节、颜色和结构上更接近真实图像，在遮挡边界的处理更自然，过渡流畅，而基础版会出现模糊、颜色偏差或畸变.

2 理论分析

从理论上看，两者的差异主要源于损失函数、优化算法和训练稳定性的改进. 基础版模型只使用对抗损失，容易忽略图像的结构与细节，导致还原模糊. 优化版本引入感知损失、重建损失等多种辅助项，使模型在保持真实性的同时更好地恢复语义和细节信息. 在优化算法上，基础版本采用标准 Adam 优化器，若参数选择不当，训练容易不稳定，而优化版本则引入 RAdam、TTUR 等策略，使生成器与判别器更新更协调. 在训练稳定性方面，优化版本结合谱归一化、梯度惩罚等技术，有效缓解模型发散和震荡问题，使最终生成图像更清晰、自然. 综合来看，损失设计的合理性、优化器调度机制和稳定性策略的协同是优化后的 GAN 获得更优填充结果的关键.