# DCGAN人物头像生成

参考教程：[DCGAN实现面孔生成（Celeb-A数据集）]

## 1. 项目背景和目标

本项目使用PyTorch实现了一个基于**深度卷积生成对抗网络**（DCGAN, Deep Convolutional Generative Adversarial Network）的模型。DCGAN是一种用于生成图像的生成对抗网络（GAN）变体，它通过卷积神经网络（CNN）来生成和判别图像。目标是训练一个生成器（Generator）生成逼真的图像，并训练一个判别器（Discriminator）来区分生成的图像和真实图像。

## 2. 主要技术栈

- **PyTorch**：深度学习框架，提供了神经网络的构建、训练和优化功能。
- **卷积神经网络（CNN）**：通过卷积层来提取图像的特征，生成器和判别器均使用卷积操作。
- **生成对抗网络（GAN）**：包括生成器（Generator）和判别器（Discriminator）两个神经网络，生成器生成图像，判别器判断图像的真假。
- **Adam优化器**：用于训练生成器和判别器，Adam优化器通过自适应调整学习率来加速训练。

## 3. 模型架构

该实现包含两个主要部分：生成器和判别器。它们的结构如下：

### 生成器（Generator）：

- 输入为一个固定大小的噪声向量（100维），通过一系列反卷积（转置卷积）层将其逐步扩展成一个64x64的RGB图像。
- 使用 `Tanh` 激活函数确保生成图像的像素值在 `[-1, 1]` 的范围内。

```
# 生成器定义

class Generator(nn.Module):

    def __init__(self, ngpu):

        super(Generator, self).__init__()

        self.ngpu = ngpu
```

```
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 2, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )


    def forward(self, input):
        return self.main(input)
```

### 判别器（Discriminator）：

- 输入为一个64x64的RGB图像，网络通过多层卷积层逐步压缩图像，最终输出一个值，表示该图像是真实的（1）还是生成的（0）。
- 使用 `LeakyReLU` 作为激活函数，以及 `Sigmoid` 函数输出概率值。

```
# 判别器定义

class Discriminator(nn.Module):

    def __init__(self, ngpu):

        super(Discriminator, self).__init__()

        self.ngpu = ngpu

        self.main = nn.Sequential(

            # input is ``(nc) x 64 x 64``

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

            nn.LeakyReLU(0.2, inplace=True),

            # state size. ``(ndf) x 32 x 32``

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

            nn.BatchNorm2d(ndf * 2),

            nn.LeakyReLU(0.2, inplace=True),

            # state size. ``(ndf*2) x 16 x 16``

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

            nn.BatchNorm2d(ndf * 4),

            nn.LeakyReLU(0.2, inplace=True),

            # state size. ``(ndf*4) x 8 x 8``

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

            nn.BatchNorm2d(ndf * 8),

            nn.LeakyReLU(0.2, inplace=True),

            # state size. ``(ndf*8) x 4 x 4``
```

```
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),

            nn.Sigmoid()#映射到区间为(0, 1)

        )

    def forward(self, input):
        return self.main(input)
```

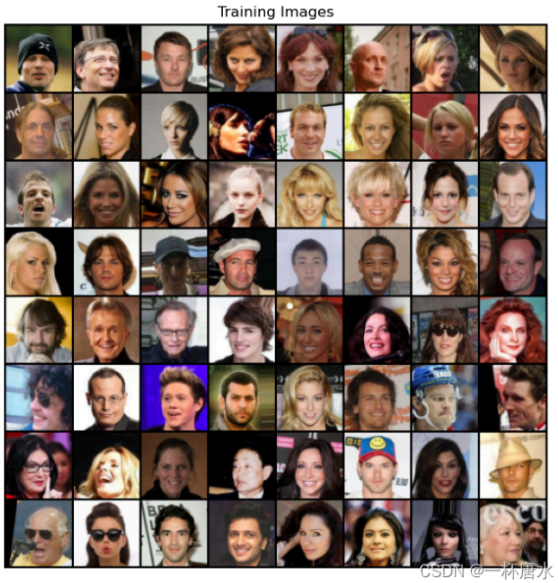
## 4. 数据处理与加载

- **缓存数据集**：为了加快训练速度，使用预处理后的图像数据，存储为PyTorch的 `.pt` 格式文件，并加载到内存中。数据集通过 `CachedDataset` 类加载，这样避免了每次训练时重复加载和处理图像数据。
- **数据加载器（DataLoader）**：使用 `DataLoader` 进行批处理、随机打乱和多进程数据加载。

```
# 加载数据 (启用多进程和pin_memory)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=4,
                                         pin_memory=True)
```

## 5. 训练过程

训练图像：




训练过程中，生成器和判别器交替优化：

1. **判别器优化**：

- 判别器首先接收真实图像，并计算其与标签（真实标签1）之间的损失。
- 然后，生成器生成假图像，判别器计算这些假图像的损失，并更新其参数。

2. **生成器优化**：
   - 生成器生成图像并通过判别器进行评估，目标是最小化判别器对生成图像的误判概率，即希望生成器生成的图像被判别器判定为真实图像。
3. **标签平滑**：为了提高判别器的鲁棒性，使用标签平滑技术，将真实图像的标签设置为0.9，而非严格的1。

## 6. 优化与学习率调整

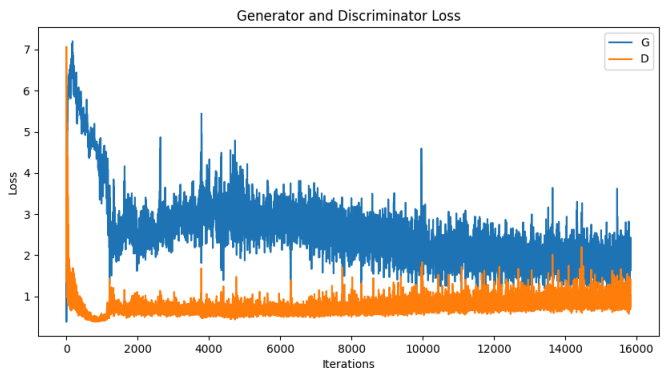
- **生成器与判别器的学习率不同**：判别器的学习率设定为生成器的1/4（`lr_D = lr / 4`），以平衡两个网络的训练速度，避免判别器过于强大，导致生成器难以学习。
- **Adam优化器**：使用 `Adam` 优化器进行训练，设置了不同的超参数，`beta1` 被调整为0.7。

## 7. 训练中的监控与可视化

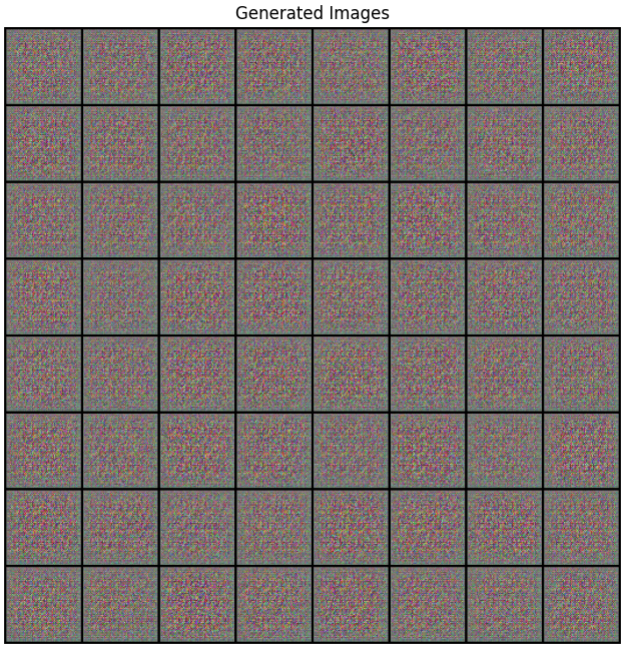
- **损失曲线**：记录并绘制生成器和判别器的损失曲线，以监控训练过程。
  最终loss值如下：

```
[19/20][100/792]        Loss_D: 1.0467  Loss_G: 1.6168  D(x): 0.4627    D(G(z)): 0.1649 / 0.2136
[19/20][150/792]        Loss_D: 0.7415  Loss_G: 1.7547  D(x): 0.7851    D(G(z)): 0.2654 / 0.1775
[19/20][200/792]        Loss_D: 0.9443  Loss_G: 1.8691  D(x): 0.8047    D(G(z)): 0.3917 / 0.1520
[19/20][250/792]        Loss_D: 0.7905  Loss_G: 1.7826  D(x): 0.6308    D(G(z)): 0.1824 / 0.1748
[19/20][300/792]        Loss_D: 0.7331  Loss_G: 2.0058  D(x): 0.6482    D(G(z)): 0.1598 / 0.1408
[19/20][350/792]        Loss_D: 0.7366  Loss_G: 2.3520  D(x): 0.7573    D(G(z)): 0.2501 / 0.0939
[19/20][400/792]        Loss_D: 0.8936  Loss_G: 1.9911  D(x): 0.6687    D(G(z)): 0.2837 / 0.1432
[19/20][450/792]        Loss_D: 0.9599  Loss_G: 1.3507  D(x): 0.5010    D(G(z)): 0.1515 / 0.2806
[19/20][500/792]        Loss_D: 0.7876  Loss_G: 1.8303  D(x): 0.5758    D(G(z)): 0.1316 / 0.1685
[19/20][550/792]        Loss_D: 0.7920  Loss_G: 1.9100  D(x): 0.7561    D(G(z)): 0.2788 / 0.1522
[19/20][600/792]        Loss_D: 0.9185  Loss_G: 1.8143  D(x): 0.4931    D(G(z)): 0.1163 / 0.1719
[19/20][650/792]        Loss_D: 0.9343  Loss_G: 1.9414  D(x): 0.5267    D(G(z)): 0.1783 / 0.1511
[19/20][700/792]        Loss_D: 0.6200  Loss_G: 2.5906  D(x): 0.6800    D(G(z)): 0.0934 / 0.0765
[19/20][750/792]        Loss_D: 0.8153  Loss_G: 2.0360  D(x): 0.7694    D(G(z)): 0.3047 / 0.1381
```

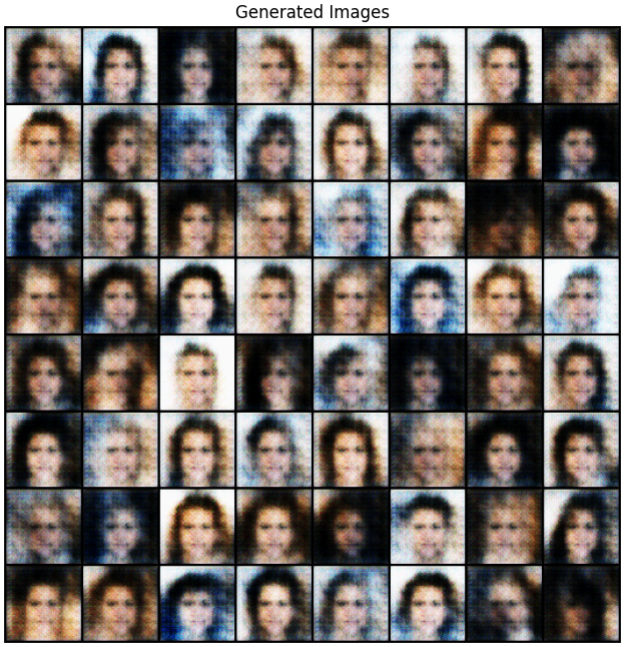
loss曲线：



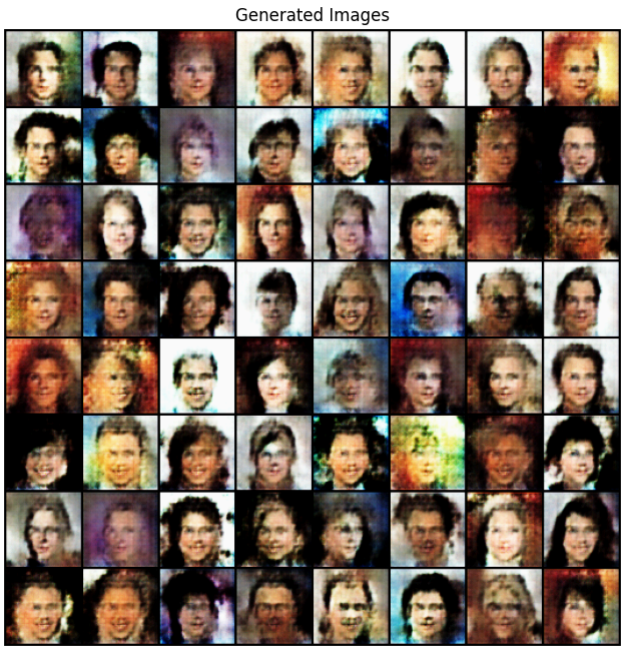

- **生成图像可视化**：每500次迭代生成并保存一张图像，使用 `matplotlib` 进行可视化并保存为图像文件。训练过程中产生的图片保存在train_result目录下。
  迭代第0次：




迭代第2000次：




迭代第5000次：




迭代第13000次：

## 8. 模型保存与加载

- **模型保存**：每训练一个epoch后，保存生成器的模型参数（`model_DCGAN_epoch_{epoch}.pth`）。训练完成后，保存最终模型（`model_DCGAN_final.pth`）。
- **缓存数据加载**：使用缓存的数据加速训练，并避免每次都重新加载和预处理数据。

## 9. 代码结构与运行

- 数据处理与加载：通过 `CachedDataset` 类加载缓存数据集，并使用 `DataLoader` 进行批次处理。
- 训练循环：每个epoch中更新判别器和生成器，计算损失并优化参数。
- 可视化：每500次迭代生成并保存一张图像，以观察生成图像的质量变化。

## 10. 测试效果

- 直接运行，即可生成n张人脸图片，保存在generate_images文件目录下。

示例：