CyberGAN：基于 CycleGAN 的赛博朋克风格图像生成迁移

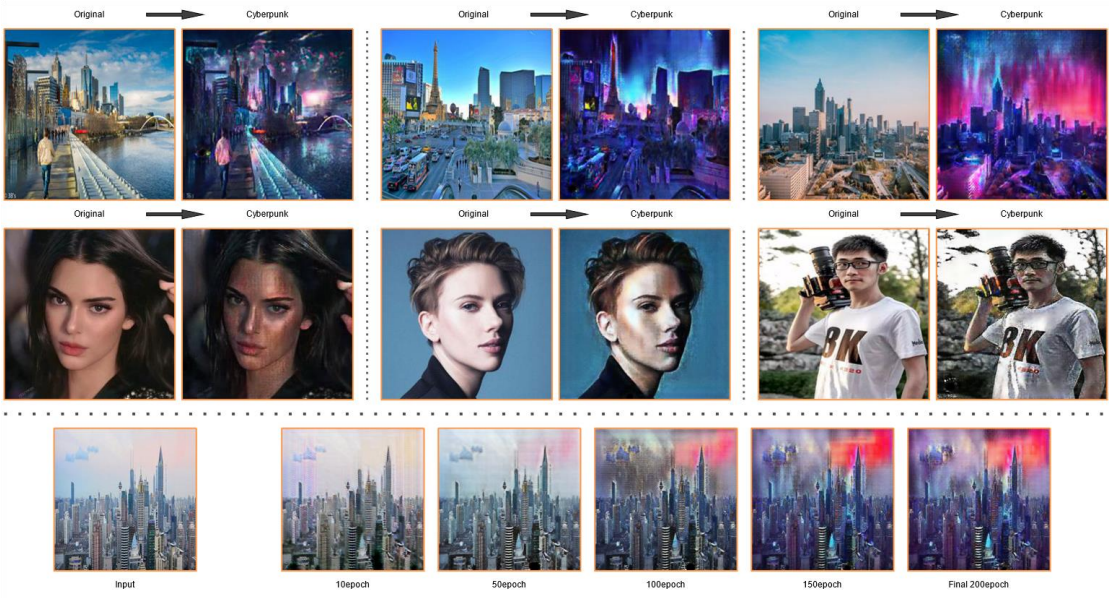

图1.　项目效果展示

1 摘要(Abstract)

本项目利用 CycleGAN[1]模型实现图像向赛博朋克风格的转换。赛博朋克风格以其独特的未来主义视觉特征和浓厚的文化氛围，在艺术创作和影视制作等领域备受青睐。本项目通过训练 CycleGAN 的生成器和判别器，成功实现了赛博朋克风格图像的初步生成。

在资源有限的条件下，我们完成了一个效果简明的演示版本，展示了 CycleGAN 在图像风格迁移中的基础应用。尽管当前效果尚有提升空间，但本项目为未来更深入的赛博朋克风格生成提供了有效的基线和方案。未来工作将

在本项目的基础上，进一步优化模型结构和训练策略，以实现更高质量的图像风格转换。

2 引言(Introduction)

赛博朋克风格源自 20 世纪 80 年代的科幻文化，以其黑暗、未来主义和高科技低生活的特点著称。这种风格广泛出现在文学、电影、电子游戏和艺术作品中，其独特的视觉效果和文化氛围吸引了大量的关注和喜爱。随着科技的发展，尤其是深度学习技术的进步，图像风格迁移成为了一个热门的研究领域，为实现不同风格之间的自动转换提供了可能性。

2.1 风格迁移

风格迁移（Style Transfer）是计算机视觉领域的一个重要任务，旨在将一种图像的风格应用到另一种图像的内容上，从而生成具有特定艺术风格的图像。风格迁移方法主要包括：

(1). 基于生成对抗网络（GANs）：如 Karras 等人提出的 StyleGAN[2]等通过无监督学习方式，通过生成器网络合成图像，判别器网络区分真实图像和生成图像，以帮助生成器学习到图像风格实现风格迁移。

图2.　StyleGAN 生成图片[2]

(2). 基于卷积神经网络（CNN）的方法：如 Gatys 等人提出的基于 CNN 的神经

风格迁移算法[3]，通过将风格图像的纹理特征和内容图像的结构特征结合，生成具有目标风格的图像。

图3.　输入内容图像和风格图像，输出风格迁移后的合成图像[8]

(3). 其他方法：除了上述两种常见的方法外，还有基于物理模型模拟表面，基于稀疏表示分析图像特点，基于注意力机制强调图像特征等的风格迁移方式。

风格迁移技术在艺术创作、图像增强、影视制作等领域有着广泛的应用。通过风格迁移，用户可以轻松地将普通照片转换为特定风格的艺术作品，如油画风格、卡通风格或梵高绘画风格等。随着深度学习和人工智能技术的发展，风格迁移技术也在不断进步，变得更加高效和灵活。

2.2 项目介绍

图像风格迁移的核心问题在于如何在保持图像内容的同时，应用目标风格的视觉特征。传统方法依赖成对的图像数据进行训练，而这种数据在实际应用中往往难以获取。在基于生成对抗网络的风格迁移技术中，CycleGAN（循环生成对抗网络）以其无需成对训练数据的优势脱颖而出。CycleGAN 通过引入循环一致性损失（Cycle Consistency Loss），使得模型能够在没有成对数据的情况下学习到从一种风格到另一种风格的映射关系，从而实现高质量的图像风格迁移。

本项目旨在探索 CycleGAN 在赛博朋克风格迁移中的应用。我们通过训练 CycleGAN 模型，将普通图像转换为具有赛博朋克风格的图像。通过这一项目，我们希望为未来进一步实现赛博朋克风格迁移提供一个有效的基准，并为进一步优化和扩展赛博朋克风格迁移技术提供参考。

3 原理介绍

3.1 生成对抗网络

生成对抗网络（GAN，Generative Adversarial Networks）是一种深度学习模型，由 Ian Goodfellow 等人[4]在 2014 年提出。GAN 基于非合作零和博弈对抗学习原理，通过生成器（Generator）和判别器（Discriminator）的相互对抗，生成器尽可能生成逼真样本，判别器则尽可能趋判别该样本是否为真实样本或生成的虚假样本，最后使得生成器能生成骗过判别器的图像。举例来说，生成模型可以被认为类似于造假者团队，试图生产虚假货币并在没有检测的情况下使用它，而判别模型则是类似于警察，试图检测假币。这场比赛中的比赛开始了两个团队都在改进他们的方法，直到伪造品与真品无法区分用品。

GAN 在结构上受博弈论中的二人零和博弈（即二人的利益之和为零，一方的所得正是另一方的所失）的启发，系统由一个生成模型（G）和一个判别模型（D）构成。G 捕捉真实数据样本的潜在分布，并生成新的数据样本；D 是一个二分类器，判别输入是真实数据还是生成的样本。生成器和判别器均可以采用深度神经网络。GAN 的优化过程是一个极小极大博弈(Minimaxgame)问题，优化目标是达到纳什均衡。

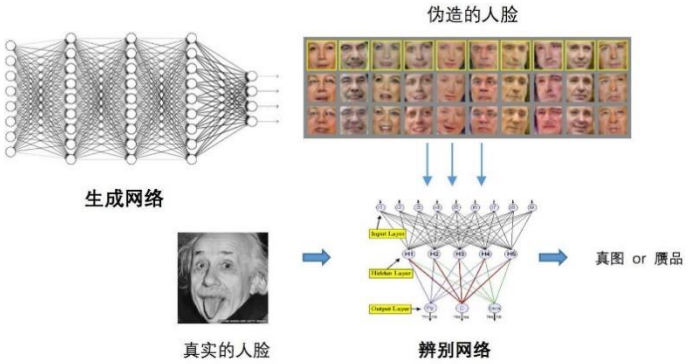

图4. GAN 原理示意图

3.1.1 生成器

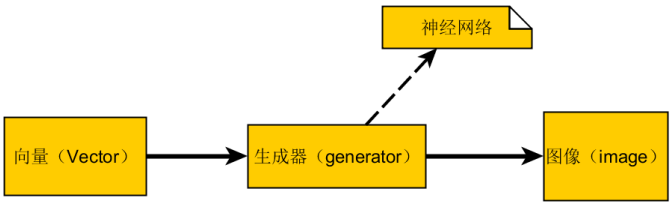

图5. 生成器功能示意图

生成器（Generator）通常是一个深度神经网络，其目的是从随机噪声 $z \sim p_z(z)$ 中生成数据 $G(z)$，使得这些生成的数据看起来像来自真实数据分布 $p_{data}(x)$。生成器的目标函数：

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right] \quad (1)$$

这个目标函数由两部分组成，第一部分是真实数据的对数似然，第二部分是生成数据的对数似然（取反）。生成器的目标是最小化这个目标函数，即生成尽可能让判别器判断为真实的数据。

3.1.2 判别器

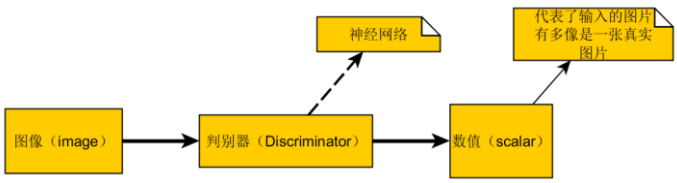

图6. 判别器功能示意图

判别器（Discriminator）也是一个深度神经网络，其任务是区分输入的数据

是来自生成器的假数据还是真实数据。判别器的目标是最大化正确分类真实数据和假数据的概率。判别器的目标函数：

$$\max_{D} V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}\left[\log\left(1 - D\left(G(z)\right)\right)\right] \quad (2)$$

判别器的目标是最大化这个目标函数，即正确区分真实数据和生成数据。

3.1.3 模型训练

训练过程通常遵循以下步骤：

(1). 初始化生成器和判别器的网络参数。

(2). 进行迭代训练，每次迭代包括两个子步骤

a) 判别器的更新：在当前生成器参数固定的情况下，对判别器进行优化，以最大化其区分真假数据的能力。

b) 生成器的更新：在当前判别器参数固定的情况下，对生成器进行优化，以最小化其生成数据被判别器正确分类的概率。

(3). 重复上述迭代过程，直到生成的数据质量达到满意的水平，或者达到预设的迭代次数。

如图为使用 GAN 生成手写数字的 50 轮次效果，可以看到生成的手写数字是逐渐清晰明确的。

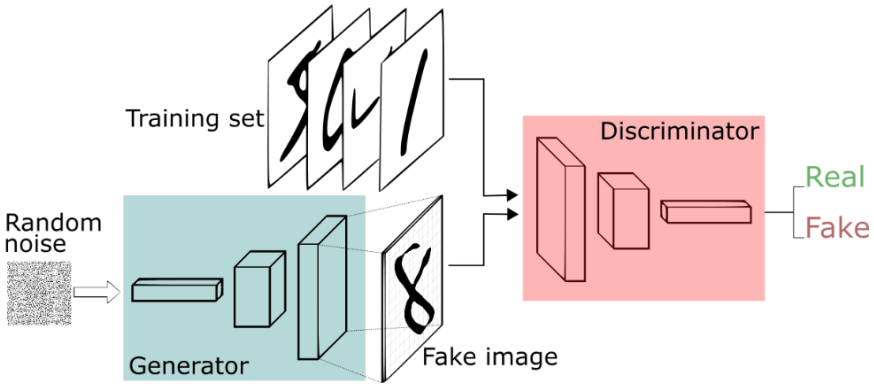

图7. GAN 实现手写数字生成模型示意图

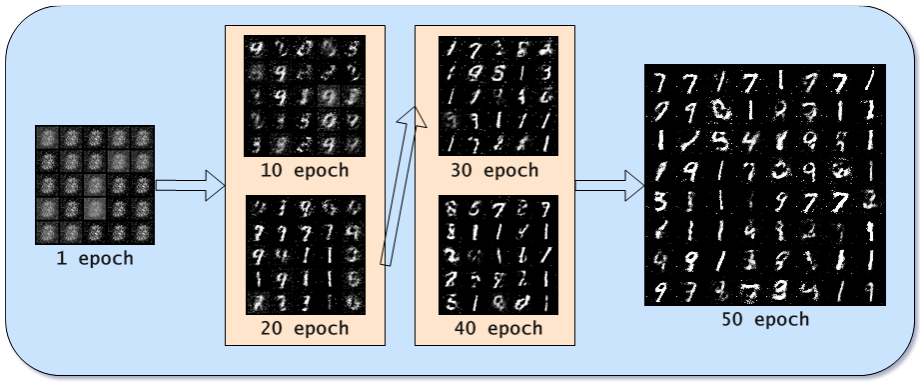

图8.　1-50 轮手写数字生成效果图

3.2 CycleGAN

CycleGAN（Cycle-Consistent Generative Adversarial Networks）是一种用于图像到图像风格迁移的无监督学习模型，由 Zhu 等人[1]在 2017 年提出。传统的图像到图像翻译方法通常需要成对的训练数据，但 CycleGAN 通过引入循环一致性损失，使得模型能够在无配对数据的情况下实现高质量的图像翻译。

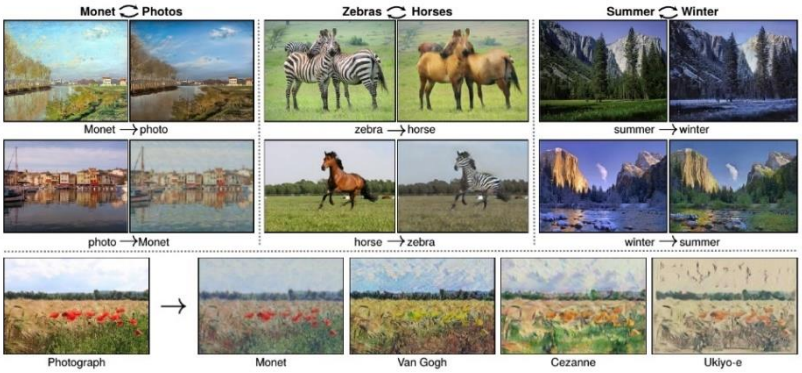

图9.　CycleGAN 实现风格迁移效果

3.2.1 CycleGAN 的基本结构

CycleGAN 包含两个生成器和两个判别器，分别用于两个域之间的映射。设两个域为 X 和 Y，生成器 $G: X \to Y$ 将域 X 中的图像转换为域 Y 中的图像，而生成器 $F: Y \to X$ 则将域 Y 中的图像转换为域 X 中的图像。两个判别器 D_X 和 D_Y 分别用于判别图像是否来自域 X 和 域 Y。

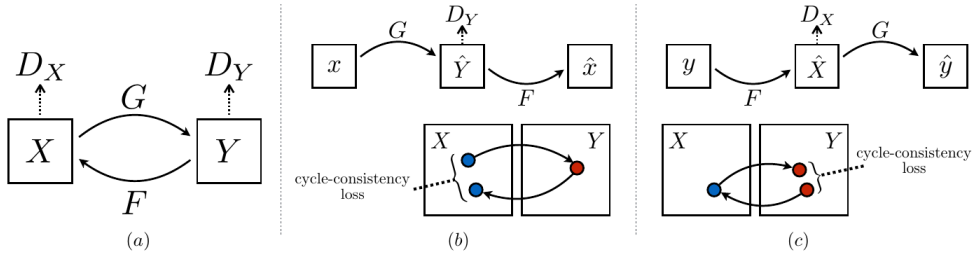

图10. CycleGAN 基本结构-循环[1]

3.2.2 对抗性损失

CycleGAN 使用生成对抗网络（GAN）的对抗性损失来训练生成器和判别器。对于生成器 G 和判别器 D_Y，对抗性损失定义如下：

$$\mathcal{L}_{\mathcal{GAN}}(G, D_Y, X, Y) = E_{y\sim p_{data}(y)}[\log D_Y(y)] + E_{x\sim p_{data}(x)}\left[\log\left(1 - D_Y\left(G(x)\right)\right)\right] \quad (3)$$

类似地，对于生成器 F 和判别器D_X，对抗性损失定义如下：

$$\mathcal{L}_{\mathcal{GAN}}(F, D_X, Y, X) = E_{x\sim p_{data}(x)}[\log D_X(x)] + E_{y\sim p_{data}(y)}\left[\log\left(1 - D_X\left(F(y)\right)\right)\right] \quad (4)$$

3.2.3 循环一致性损失

为了确保生成的图像能够保留原始图像的内容，CycleGAN 引入了循环一致性损失。具体来说，如果我们从域X映射到域Y，然后再映射回域X，我们希望得到的图像与原始图像尽可能相似。循环一致性损失定义如下：

$$\mathcal{L}_{cyc}(G, F) = E_{x\sim p_{data}(x)}\left[|F\left(G(x)\right) - x|_1\right] + E_{y\sim p_{data}(y)}\left[|G\left(F(y)\right) - y|_1\right] \quad (5)$$

这里，$|\cdot|_1$表示 $L1$ 范数，用于衡量图像之间的像素差异。

3.2.4 总损失函数

CycleGAN 的总损失函数是对抗性损失和循环一致性损失的加权和。总损失函数定义如下：

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{\mathcal{GAN}}(G, D_Y, X, Y) + \mathcal{L}_{\mathcal{GAN}}(F, D_X, Y, X) + \lambda\mathcal{L}_{cyc}(G, F) \quad (6)$$

其中，λ 是平衡对抗性损失和循环一致性损失的权重系数，通常通过实验

确定其最优值。

3.2.5 模型训练

CycleGAN 的训练过程包括以下步骤：

(1). 初始化参数：初始化生成器和判别器的参数。

(2). 对抗训练：交替优化生成器和判别器，使生成器生成的图像逐渐逼真，判别器能够区分真实图像和生成图像。

(3). 循环一致性训练：通过循环一致性损失，确保生成的图像能够恢复原始图像，保留内容的一致性。

(4). 迭代训练：重复上述步骤，直到模型收敛。

CycleGAN 的创新在于无需成对的训练数据，通过循环一致性损失解决了无配对图像翻译的问题，广泛应用于风格迁移、图像修复等领域。

4 项目实现

4.1 项目简介

本项目为实现基于 CycleGAN 的赛博朋克图像生成风格迁移，尝试训练一个 GAN 模型实现普通图像转换为具有赛博朋克风格的图像。通过项目分析，在考虑到缺少对应数据图像的情况下，我们确定了使用 CycleGAN 网络架构来实现本项目。并完成了数据集准备与分析，环境搭建，模型架构代码实现与模型训练，模型效果测试与评估的流程。由于计算资源有限和数据集质量未达到极高治疗和多样化大规模，本项目最终获得的风格迁移效果未达到预期效果，但是为后续提升效果提供了 baseline。

4.2 数据集

4.2.1 数据集获取

本项目需要准备两个风格的图像数据集：普通图像和赛博朋克风格图像

普通图像数据集：从 Kaggle 的 Unpaired Day and Night city view images dataset[5]获取 749 张城市图像数据，从 Google Image 爬虫获取 251 张人物图像数据。(爬虫代码涉及权限条约，不提供)

赛博朋克更改图像数据集：从 Kaggle 的 Cyber Verse Cyberpunk_Image Dataset[6]提取 1000 张图像数据。

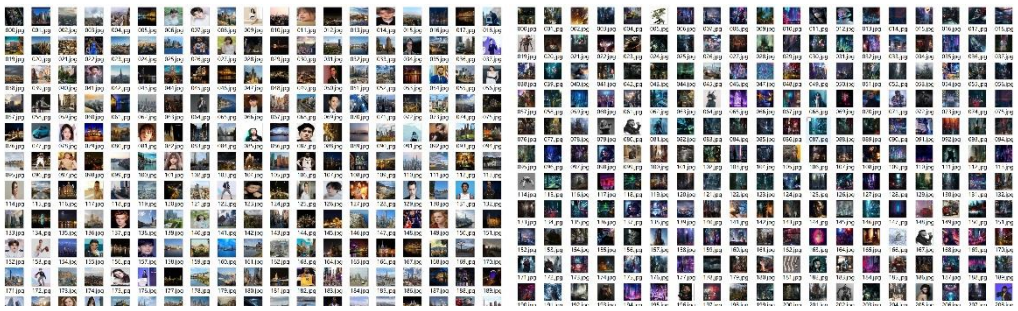

图11. 数据概览

4.2.2 数据预处理

所有图像在使用前进行统一的预处理，调整图像大小为 $size = 256 \times 256$, 颜色空间全部转换为 RGB。

```
1.import os
2.from PIL import Image
3.
4.# 设置图像数据路径和输出路径
5.dataset_path='./dataset'
6.output_path='./processed_dataset'
7.
8.# 如果输出目录不存在，则创建它
9.ifnotos.path.exists(output_path):
10.os.makedirs(output_path)
11.
12.# 定义预处理函数
13.defpreprocess_image(image_path,output_size=(256,256)):
```

```
14.withImage.open(image_path)asimg:
15.img=img.convert('RGB')转换为 RGB
16.img=img.resize(output_size,Image.ANTIALIAS)调整图像大小
17.returnimg
18.
19.# 遍历数据集中的所有图像文件
20.forfilenameinos.listdir(dataset_path):
21.iffilename.lower().endswith(('.png','.jpg','.jpeg','.bmp','.gif')):检查文件扩展名
22.# 完整的图像文件路径
23.image_path=os.path.join(dataset_path,filename)
24.
25.# 预处理图像
26.processed_img=preprocess_image(image_path)
27.
28.# 保存预处理后的图像到输出目录
29.output_image_path=os.path.join(output_path,filename)
30.processed_img.save(output_image_path)
31.
32.print(f'所有图像已预处理并保存到{output_path}')
33.
34.fromtorchvisionimporttransforms
35.
36.# 定义一个转换序列，包括转换为 Tensor 和归一化等操作
37.transform=transforms.Compose([
38.transforms.Resize((256,256)),
39.transforms.ToTensor(),
40.transforms.Normalize(mean=[0.485,0.456,0.406],std=[0.229,0.224,0.225]),
41.])
42.
43.# 然后应用这个转换到 PIL 图像上
44.tensor_image=transform(processed_img)
```

将数据集划分为普通图像 TrainA800 张图片，TestA200 张图像，赛博朋克图像 TrainB800 张图像，TestB200 张图像。

```
1.import os
2.import shutil
3.from sklearn.model_selection import train_test_split
4.
5.normal_path=os.path.join(dataset_path,'normal')
6.cyberpunk_path=os.path.join(dataset_path,'cyberpunk')
7.
```

```
8.# 设置训练集和测试集的图像数量
9.train_size=800
10.test_size=200
11.
12.# 为普通图像和赛博朋克图像创建训练和测试目录
13.trainA_path=os.path.join(dataset_path,'TrainA')
14.testA_path=os.path.join(dataset_path,'TestA')
15.trainB_path=os.path.join(dataset_path,'TrainB')
16.testB_path=os.path.join(dataset_path,'TestB')
17.
18.# 如果目录不存在，则创建它们
19.os.makedirs(trainA_path,exist_ok=True)
20.os.makedirs(testA_path,exist_ok=True)
21.os.makedirs(trainB_path,exist_ok=True)
22.os.makedirs(testB_path,exist_ok=True)
23.
24.# 获取所有图像文件并划分为训练集和测试集
25.defsplit_data(image_path,train_path,test_path,train_count,test_count):
26.# 获取所有图像文件
27.images=[fforfinos.listdir(image_path)ifos.path.isfile(os.path.join(image_path,f))]
28.# 划分数据集
29.train_images,test_images=train_test_split(images,test_size=test_count,random_state=42
)
30.
31. # 创建训练集和测试集的图像列表
32.train_images=train_images[:train_count]
33.test_images=test_images[:test_count]
34.
35.# 复制图像到相应的训练集和测试集目录
36.forimgintrain_images:
37.shutil.copy(os.path.join(image_path,img),os.path.join(train_path,img))
38.forimgintest_images:
39.shutil.copy(os.path.join(image_path,img),os.path.join(test_path,img))
40.
41. # 对普通图像和赛博朋克图像执行数据划分
42.split_data(normal_path,trainA_path,testA_path,train_size,test_size)
43.split_data(cyberpunk_path,trainB_path,testB_path,train_size,test_size)
44.
45.print("数据集划分完成。")
```

4.3 环境搭建

本项目在 conda 虚拟环境使用 Python 基于 PyTorch 框架实现模型。

如下为 requirements.txt，按照此环节配置 conda 虚拟环境。

```
python==3.10.14
torch==2.2.2+cuda11.8
torchvision==0.17.2
matplotlib.pyplot==3.8.4
pillow==10.3.0
tensorboard==2.16.2
# 其余库没有限制
```

4.4 模型架构

本项目的模型核心是残差块，用于构建生成器和判别器，以实现图像风格转换。生成器基于残差网络架构，通过卷积层、归一化和激活函数逐步提取和转换特征，最终生成赛博朋克风格的图像；判别器则采用 PatchGAN 结构，通过局部特征判别生成图像的真实性。

4.4.1 残差块

残差块（ResidualBlock）是生成器的基本构建单元，用于避免深层网络中的梯度消失问题。

```
# 定义残差块，用于避免深层网络中的梯度消失问题
class ResidualBlock(nn.Module):
    def __init__(self, in_features):
        super(ResidualBlock, self).__init__()

        # 定义残差块内的卷积层序列
        conv_block = [
            # 第一个卷积层，使用 3x3 的卷积核，1 的 stride 和 1 的 padding
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1, padding=1),
            # 实例归一化层，用于归一化卷积层的输出
            nn.InstanceNorm2d(in_features),
            # ReLU 激活函数，inplace=True 表示原地操作，减少内存消耗
            nn.ReLU(inplace=True),
            # 第二个卷积层，与第一个卷积层参数相同
            nn.Conv2d(in_features, in_features, kernel_size=3, stride=1, padding=1),
            # 另一个实例归一化层
            nn.InstanceNorm2d(in_features)
        ]

```

```
        # 将卷积层序列封装为一个 Sequential 模块
        self.conv_block = nn.Sequential(*conv_block)

    # 前向传播函数，实现残差块的计算
    def forward(self, x):
        # 残差连接：输入直接加上卷积层的输出
        return x + self.conv_block(x)
```

残差块包含两个卷积层和一个残差连接，通过将输入直接添加到卷积操作的输出，保持了原始特征信息，从而改善了深层网络的训练效果。

4.4.2 生成器

生成器（Generator）的作用是将普通图像转换为赛博朋克风格图像。生成器采用基于残差网络（ResNet）的架构，包含初始卷积层、下采样层、残差块和上采样层。

```
# 定义生成器网络
class Generator(nn.Module):
    def __init__(self, input_nc, output_nc, n_residual_blocks=9):
        super(Generator, self).__init__()

        # 初始卷积层，用于将输入图像的通道数从 input_nc 转换到 64
        model = [
            nn.Conv2d(input_nc, 64, kernel_size=7, stride=1, padding=3),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        ]

        # 下采样过程，逐步将图像尺寸减半，同时增加通道数
        in_features = 64
        out_features = in_features * 2
        for _ in range(2):
            model += [
                nn.Conv2d(in_features, out_features, kernel_size=3, stride=2,
padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True)
            ]
            in_features = out_features
            out_features *= 2

```

```
        # 添加残差块
        for _ in range(n_residual_blocks):
            model += [ResidualBlock(in_features)]

        # 上采样过程，逐步将图像尺寸加倍，同时减少通道数
        out_features = in_features // 2
        for _ in range(2):
            model += [
                nn.ConvTranspose2d(in_features, out_features, kernel_size=3, stride=2,
padding=1, output_padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True)
            ]
            in_features = out_features
            out_features //= 2

        # 输出层，将 64 通道的特征图转换为输出_nc 通道的图像，并使用 Tanh 激活函数
        model += [
            nn.Conv2d(64, output_nc, kernel_size=7, stride=1, padding=3),
            nn.Tanh()
        ]

        # 将模型各部分封装为一个 Sequential 模块
        self.model = nn.Sequential(*model)

    # 生成器的前向传播函数
    def forward(self, x):
        return self.model(x)
```

生成器的具体结构为：

(1). 初始卷积层：用于特征提取。

(2). 下采样层：通过卷积操作减小特征图尺寸，提高特征抽象能力。

(3). 残差块：保持特征信息，防止梯度消失。

(4). 上采样层：通过反卷积操作恢复特征图尺寸。

(5). 输出层：生成目标风格的图像，使用 Tanh 激活函数将输出归一化到 [-1,1] 范围。

4.4.3 判别器

判别器（Discriminator）用于判断生成的图像是否真实。判别器采用 PatchGAN 架构，通过卷积层对输入图像的局部区域进行判别。

PatchGAN[7]是一种用于生成对抗网络（GAN）的判别器架构。与传统的 GAN 判别器不同，它不输出一个单一的真假评价值，而是输出一个 N×N 的矩阵，每个元素代表图像中相应小块（patch）的真实性评分。这种方法允许判别器专注于图像的局部区域，从而更好地捕捉细节和纹理，特别适合于图像风格迁移等任务，其中高分辨率和细节保持是关键。

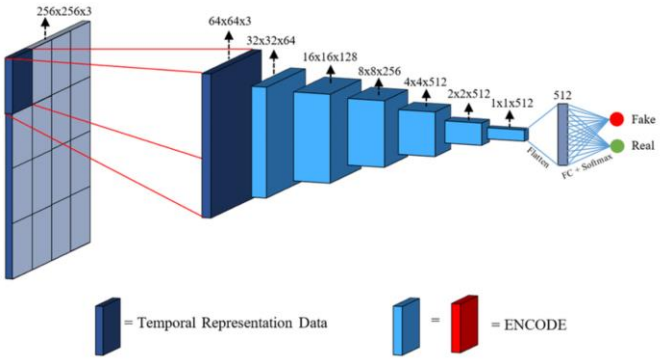

图12. 判别器的 PatchGAN 架构示意图

```
# 定义判别器网络，用于 CycleGAN
class Discriminator(nn.Module):
    def __init__(self, input_nc):
        super(Discriminator, self).__init__()

        # 判别器的卷积层序列
        model = [
            nn.Conv2d(input_nc, 64, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        ]

        model += [
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True)
        ]

```

```
        model += [
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True)
        ]

        model += [
            nn.Conv2d(256, 512, kernel_size=4, stride=1, padding=1),
            nn.InstanceNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True)
        ]

        # 最后一个卷积层，输出单个通道的预测结果
        model += [nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=1)]

        # 将模型各部分封装为一个 Sequential 模块
        self.model = nn.Sequential(*model)

    # 判别器的前向传播函数
    def forward(self, x):
        return self.model(x)
```

判别器的具体结构为：

(1). 卷积层和 LeakyReLU 激活函数：用于特征提取和判别，逐层增加特征图的深度，降低空间维度。

(2). 输出层：生成一个单通道输出，表示图像中各区域的真伪评分。

4.5 模型训练

本节将详细描述 CycleGAN 模型的训练过程，包括数据集加载与处理、模型定义、超参数设置、损失函数与优化器选择以及训练步骤。

4.5.1 加载图像数据集并处理

首先，我们需要加载和处理图像数据集。为此，定义一个自定义的数据集类 ImageDataset，用于加载图像并进行预处理操作。通过将图像调整为 256x256 大小、转为 Tensor 类型并标准化处理，可以提高模型的训练效率和泛

化能力。

```
# 自定义用于加载和处理图像的数据集类
class ImageDataset(Dataset):
    def __init__(self, root, transforms_=None, mode='train'):
        # 初始化 transforms 和文件名列表，以及 root 路径和模式（训练或测试）
        self.transform = transforms_
        self.files_A = sorted(os.listdir(os.path.join(root, f'{mode}A')))  # 获取训练或测试集中 A 的文件列表
        self.files_B = sorted(os.listdir(os.path.join(root, f'{mode}B')))  # 获取训练或测试集中 B 的文件列表
        self.root = root
        self.mode = mode

    def __getitem__(self, index):
        # 根据索引 index 获取数据集中的一个样本
        # 从 A 和 B 文件夹中读取相应索引的图像，应用 transforms_变换
        item_A = self.transform(Image.open(os.path.join(self.root, f'{self.mode}A/{self.files_A[index % len(self.files_A)]}')))
        item_B = self.transform(Image.open(os.path.join(self.root, f'{self.mode}B/{self.files_B[index % len(self.files_B)]}')))
        # 返回一个字典，包含 A 和 B 的图像
        return {'A': item_A, 'B': item_B}

    def __len__(self):
        # 返回数据集中样本的数量，取 A 和 B 文件列表的长度最大值
        return max(len(self.files_A), len(self.files_B))

# 数据预处理，包括调整图像大小，转换为张量，归一化
transform = transforms.Compose([
    transforms.Resize((256, 256)),  # 调整图像大小为 256x256
    transforms.ToTensor(),  # 将图像转换为 torch.FloatTensor
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 归一化，使图像像素值在[-1, 1]范围内
])
```

4.5.2 定义 CycleGAN 模型

接下来，定义 CycleGAN 的生成器和判别器。生成器负责将普通图像转换为赛博朋克风格图像，判别器负责判别生成的图像是否真实。

```
# 初始化生成器和判别器
# 假设 Generator 和 Discriminator 类已在其他地方定义
G_AB = Generator(input_nc=3, output_nc=3)
```

```
G_BA = Generator(input_nc=3, output_nc=3)
D_A = Discriminator(input_nc=3)
D_B = Discriminator(input_nc=3)
# 如果 GPU 可用，则使用 GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 将模型和损失函数移动到 GPU
G_AB = G_AB.to(device)
G_BA = G_BA.to(device)
D_A = D_A.to(device)
D_B = D_B.to(device)
criterion_GAN = criterion_GAN.to(device)
criterion_cycle = criterion_cycle.to(device)
criterion_identity = criterion_identity.to(device)
```

4.5.3 超参数设置

在训练过程中，设置合适的超参数对于模型的收敛和性能至关重要。我们选择以下超参数进行训练：

(1). 学习率：0.0002

(2). 批量大小：1

(3). 训练周期：200 个 epoch

```
# 训练参数设置
n_epochs = 200  # 训练周期数
batch_size = 1  # 每个批次的样本数
lr = 2e-4  # 学习率
start_decay_epoch = 100  # 开始衰减学习率的周期
```

4.5.4 损失函数与优化器

CycleGAN 的损失函数由生成对抗损失、循环一致性损失和身份损失组成：

(1). 生成对抗损失（GAN Loss）：用于保证生成的图像在目标域中的真实性。

(2). 循环一致性损失（Cycle Consistency Loss）：用于保证生成图像可以转换回原始图像。

(3). 身份损失（Identity Loss）：用于保持颜色和风格一致性。

优化器选择 Adam，并使用学习率调度器进行动态学习率调整。

```
# 损失函数定义
criterion_GAN = nn.MSELoss()  # GAN 损失函数
criterion_cycle = nn.L1Loss()  # 循环一致性损失函数
criterion_identity = nn.L1Loss()  # 身份损失函数

# 优化器定义
optimizer_G = optim.Adam(itertools.chain(G_AB.parameters(), G_BA.parameters()), lr=lr, betas=(0.5, 0.999))
optimizer_D_A = optim.Adam(D_A.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_D_B = optim.Adam(D_B.parameters(), lr=lr, betas=(0.5, 0.999))
# 学习率衰减策略
def lambda_rule(epoch):
    lr_l = 1.0 - max(0, epoch + 1 - start_decay_epoch) / float(n_epochs - start_decay_epoch + 1)
    return lr_l

# 创建学习率调度器
lr_scheduler_G = torch.optim.lr_scheduler.LambdaLR(optimizer_G, lr_lambda=lambda_rule)
lr_scheduler_D_A = torch.optim.lr_scheduler.LambdaLR(optimizer_D_A, lr_lambda=lambda_rule)
lr_scheduler_D_B = torch.optim.lr_scheduler.LambdaLR(optimizer_D_B, lr_lambda=lambda_rule)
```

4.5.5 训练

本项目模型在 AutoDL 平台如下配置进行训练，训练时长约 8.4 小时

GPU	NVIDIA Tesla V100-32GB(32GB) * 1
CPU	10 vCPU Intel Xeon Processor (Skylake, IBRS)

训练过程中，交替更新生成器和判别器。首先更新生成器以欺骗判别器，然后更新判别器以区分真实图像和生成图像。整个训练过程包括以下步骤：

(1). 加载输入图像：从数据集中加载一批普通图像和赛博朋克风格图像。

(2). 生成图像：使用生成器将普通图像转换为赛博朋克风格图像，反之亦

然。

(3). 计算损失：计算生成对抗损失、循环一致性损失和身份损失。

(4). 更新生成器：最小化生成对抗损失和循环一致性损失。

(5). 更新判别器：最大化判别真实图像和生成图像的能力。

(6). 调整学习率：使用调度器动态调整学习率。

```
# 模型保存目录和日志目录设置
model_save_dir = 'desired_model_directory'
log_dir = 'desired_log_directory'
# 确保目录存在
os.makedirs(model_save_dir, exist_ok=True)
os.makedirs(log_dir, exist_ok=True)
# 初始化 TensorBoard 的 SummaryWriter
writer = SummaryWriter(log_dir)

# 创建 DataLoader 实例，用于加载数据
dataloader = DataLoader(ImageDataset(root='./data', transforms_=transform),
batch_size=batch_size, shuffle=True)

# 训练循环
for epoch in range(n_epochs):
    # 在每个 epoch 中遍历数据集
    for i, batch in enumerate(dataloader):
        # 将图像移动到 GPU
        real_A = batch['A'].to(device)
        real_B = batch['B'].to(device)

        # 清零梯度
        optimizer_G.zero_grad()

        # 计算 Identity Loss（可选）
        # 生成器尝试复制输入图像
        same_B = G_AB(real_B)
        loss_identity_B = criterion_identity(same_B, real_B) * 5.0
        same_A = G_BA(real_A)
        loss_identity_A = criterion_identity(same_A, real_A) * 5.0

        # 计算 GAN Loss
        # 生成器生成假图像，判别器尝试区分真假
        fake_B = G_AB(real_A)
        pred_fake = D_B(fake_B)
```

```
        loss_GAN_AB = criterion_GAN(pred_fake, torch.ones_like(pred_fake))
        fake_A = G_BA(real_B)
        pred_fake = D_A(fake_A)
        loss_GAN_BA = criterion_GAN(pred_fake, torch.ones_like(pred_fake))

        # 计算 Cycle Loss
        # 确保两个生成器转换后的图像尽可能接近原始图像
        recovered_A = G_BA(fake_B)
        loss_cycle_A = criterion_cycle(recovered_A, real_A) * 10.0
        recovered_B = G_AB(fake_A)
        loss_cycle_B = criterion_cycle(recovered_B, real_B) * 10.0

        # 计算总的生成器损失
        loss_G = loss_identity_A + loss_identity_B + loss_GAN_AB + loss_GAN_BA +
loss_cycle_A + loss_cycle_B
        loss_G.backward()  # 反向传播
        optimizer_G.step()  # 更新生成器参数

        # 接下来更新判别器 D_A
        optimizer_D_A.zero_grad()
        # 判别器在真实图像上的损失
        pred_real = D_A(real_A)
        loss_D_real = criterion_GAN(pred_real, torch.ones_like(pred_real))
        # 判别器在生成图像上的损失
        pred_fake = D_A(fake_A.detach())
        loss_D_fake = criterion_GAN(pred_fake, torch.zeros_like(pred_fake))
        loss_D_A = (loss_D_real + loss_D_fake) * 0.5
        loss_D_A.backward()
        optimizer_D_A.step()

        # 同样的方式更新判别器 D_B
        optimizer_D_B.zero_grad()
        pred_real = D_B(real_B)
        loss_D_real = criterion_GAN(pred_real, torch.ones_like(pred_real))
        pred_fake = D_B(fake_B.detach())
        loss_D_fake = criterion_GAN(pred_fake, torch.zeros_like(pred_fake))
        loss_D_B = (loss_D_real + loss_D_fake) * 0.5
        loss_D_B.backward()
        optimizer_D_B.step()

        # 记录训练和验证损失到 TensorBoard
        if i % 10 == 0:
            writer.add_scalar('Loss/G', loss_G.item(), epoch * len(dataloader) + i)
```

```
            writer.add_scalar('Loss/D_A', loss_D_A.item(), epoch * len(dataloader) + i)
            writer.add_scalar('Loss/D_B', loss_D_B.item(), epoch * len(dataloader) + i)

        # 每 100 批次打印一次训练状态
        if i % 100 == 0:
            print(f'Epoch [{epoch+1}/{n_epochs}] Batch [{i}/{len(dataloader)}] '
                  f'Loss G: {loss_G.item():.4f}, Loss D_A: {loss_D_A.item():.4f}, Loss D_B: {loss_D_B.item():.4f}')

    # 更新学习率
    lr_scheduler_G.step()
    lr_scheduler_D_A.step()
    lr_scheduler_D_B.step()

    # 保存模型参数
    if epoch % 10 == 0 or epoch == n_epochs - 1:
        torch.save(G_AB.state_dict(), f'{model_save_dir}/G_AB_epoch_{epoch}.pth')
        torch.save(G_BA.state_dict(), f'{model_save_dir}/G_BA_epoch_{epoch}.pth')
        torch.save(D_A.state_dict(), f'{model_save_dir}/D_A_epoch_{epoch}.pth')
        torch.save(D_B.state_dict(), f'{model_save_dir}/D_B_epoch_{epoch}.pth')

    # 可视化生成的图像
    with torch.no_grad():
        fake_B = G_AB(real_A)
        fake_A = G_BA(real_B)
        # 使用 matplotlib 展示和保存图像
        plt.figure(figsize=(10, 5))
        plt.subplot(2, 2, 1)
        plt.title('Real A')
        plt.imshow((real_A[0].cpu().numpy().transpose(1, 2, 0) * 0.5 + 0.5))
        plt.subplot(2, 2, 2)
        plt.title('Fake B')
        plt.imshow((fake_B[0].cpu().numpy().transpose(1, 2, 0) * 0.5 + 0.5))
        plt.subplot(2, 2, 3)
        plt.title('Real B')
        plt.imshow((real_B[0].cpu().numpy().transpose(1, 2, 0) * 0.5 + 0.5))
        plt.subplot(2, 2, 4)
        plt.title('Fake A')
        plt.imshow((fake_A[0].cpu().numpy().transpose(1, 2, 0) * 0.5 + 0.5))
        plt.show()

        # 将图像添加到 TensorBoard
```

```
118.         writer.add_images('Real A', real_A * 0.5 + 0.5, epoch)
119.         writer.add_images('Fake B', fake_B * 0.5 + 0.5, epoch)
120.         writer.add_images('Real B', real_B * 0.5 + 0.5, epoch)
121.         writer.add_images('Fake A', fake_A * 0.5 + 0.5, epoch)
122.
123. # 训练结束后保存最终模型参数
124. torch.save(G_AB.state_dict(), f'{model_save_dir}/G_AB_final.pth')
125. torch.save(G_BA.state_dict(), f'{model_save_dir}/G_BA_final.pth')
126. torch.save(D_A.state_dict(), f'{model_save_dir}/D_A_final.pth')
127. torch.save(D_B.state_dict(), f'{model_save_dir}/D_B_final.pth')
128.
129. # 关闭 TensorBoard SummaryWriter
130. writer.close()
```

上述代码通过记录训练损失、保存模型参数和可视化生成结果，确保模型在训练过程中的稳定性和可调试性。

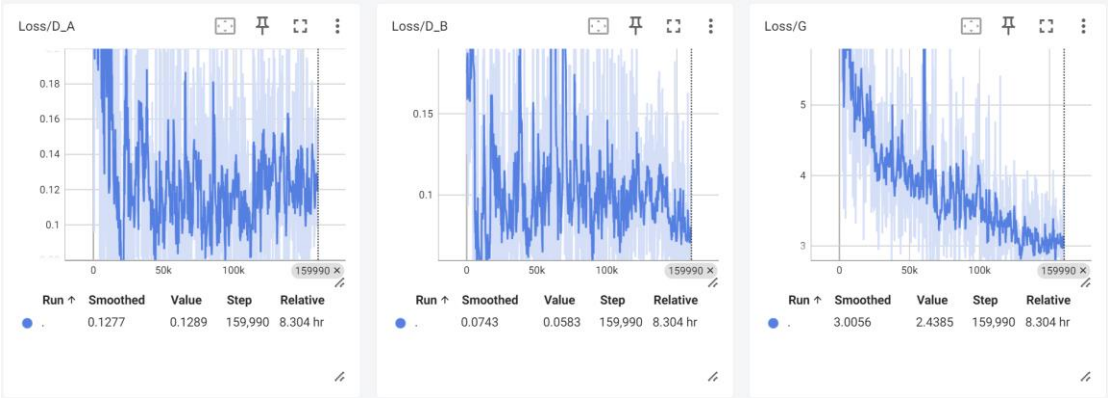

图13. 模型训练损失

4.5.6 注意

GAN 训练模型是消耗计算资源较大的，在设置 input_size=(3, 256, 256)模型 G_AB 进行测试可以发现总估计 Size 为 791.15MB。由于 CycleGAN 循环的特性，消耗的显存会大幅增加，如果对 batch_size 进行加倍的话，显存消耗也会显著加倍，在设置 batch_size=1 的情况下消耗显存 14186MiB/32768Mib，设置 batch_size=2 即翻倍导致 CUDA out of memory。由于算力资源和资金有限，本项目对数据集大小、数据集多样性和完备性，训练规模等都做了较大减法，也导致了最后训练出来的模型效果较差，仅能实现部分边缘特征和色彩的迁移。

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 256, 256]           9,472
    InstanceNorm2d-2         [-1, 64, 256, 256]               0
              ReLU-3         [-1, 64, 256, 256]               0
            Conv2d-4        [-1, 128, 128, 128]          73,856
    InstanceNorm2d-5        [-1, 128, 128, 128]               0
              ReLU-6        [-1, 128, 128, 128]               0
            Conv2d-7          [-1, 256, 64, 64]         295,168
    InstanceNorm2d-8          [-1, 256, 64, 64]               0
              ReLU-9          [-1, 256, 64, 64]               0
           Conv2d-10          [-1, 256, 64, 64]         590,080
   InstanceNorm2d-11          [-1, 256, 64, 64]               0
             ReLU-12          [-1, 256, 64, 64]               0
           Conv2d-13          [-1, 256, 64, 64]         590,080
   InstanceNorm2d-14          [-1, 256, 64, 64]               0
    ResidualBlock-15          [-1, 256, 64, 64]               0
           Conv2d-16          [-1, 256, 64, 64]         590,080
   InstanceNorm2d-17          [-1, 256, 64, 64]               0
             ReLU-18          [-1, 256, 64, 64]               0
           Conv2d-19          [-1, 256, 64, 64]         590,080
   InstanceNorm2d-20          [-1, 256, 64, 64]               0
    ResidualBlock-21          [-1, 256, 64, 64]               0
           Conv2d-22          [-1, 256, 64, 64]         590,080
...
Forward/backward pass size (MB): 747.00
Params size (MB): 43.40
Estimated Total Size (MB): 791.15
----------------------------------------------------------------
```

图14. CycleGAN 模型架构和资源消耗

```
Every 1.0s: nvidia-smi

Tue Jun  4 08:42:20 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100S-PCI...  On   | 00000000:00:08.0 Off |                  Off |
| N/A   59C    P0   217W / 250W |  14186MiB / 32768MiB |     95%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

图15. 项目训练资源消耗情况

4.6 模型效果

4.6.1 模型生成图片风格迁移效果

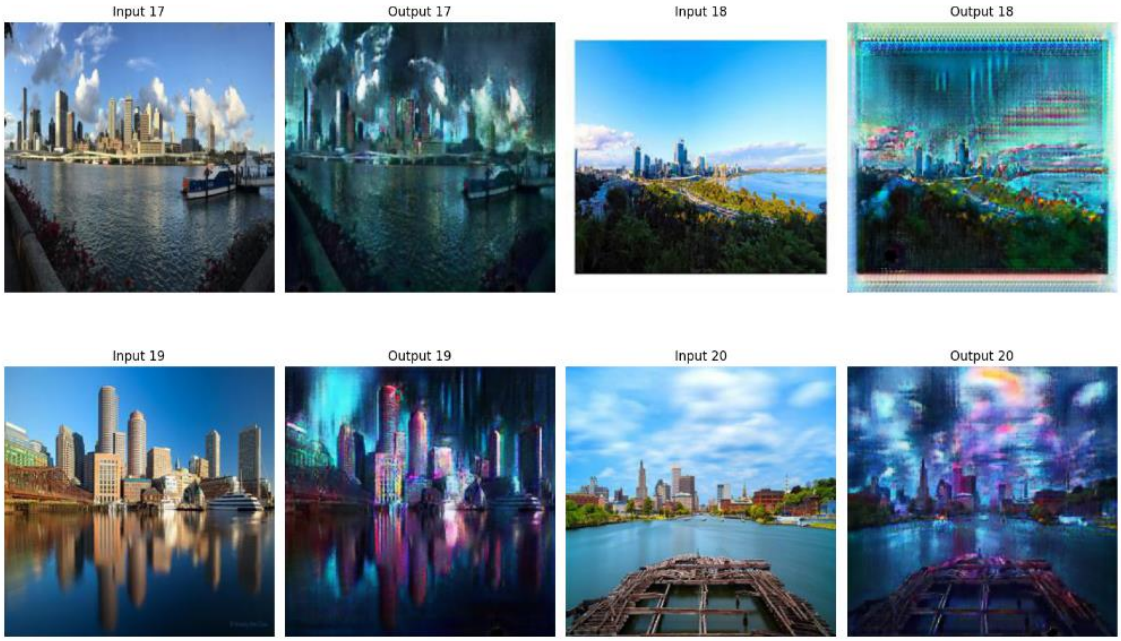

图16. 模型风格迁移效果图（1）

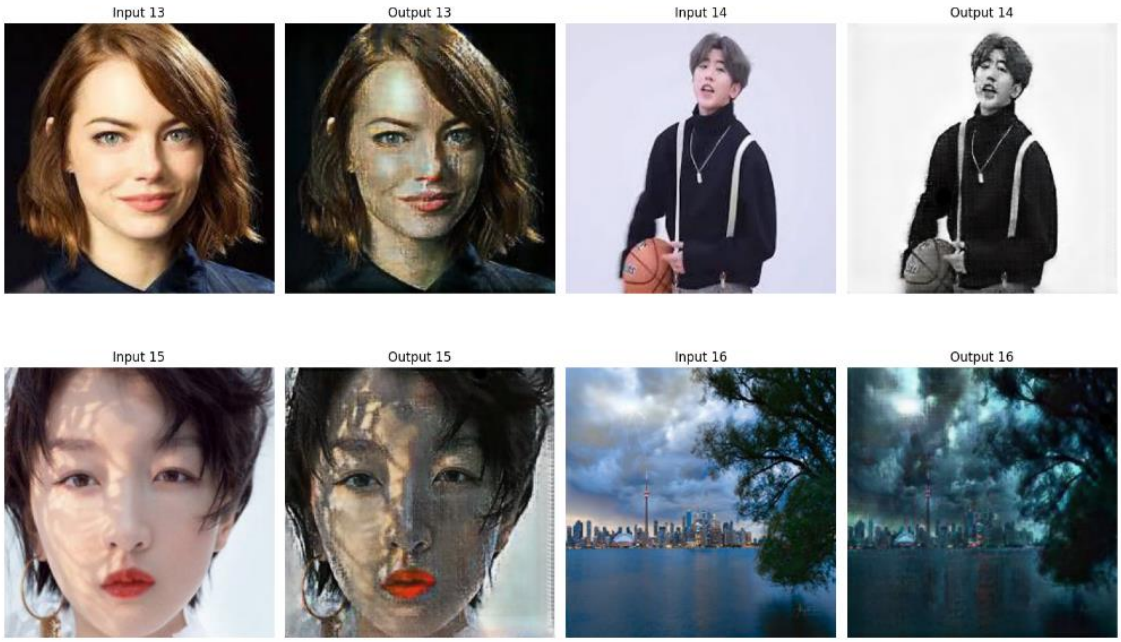

图17. 模型风格迁移效果图（2）

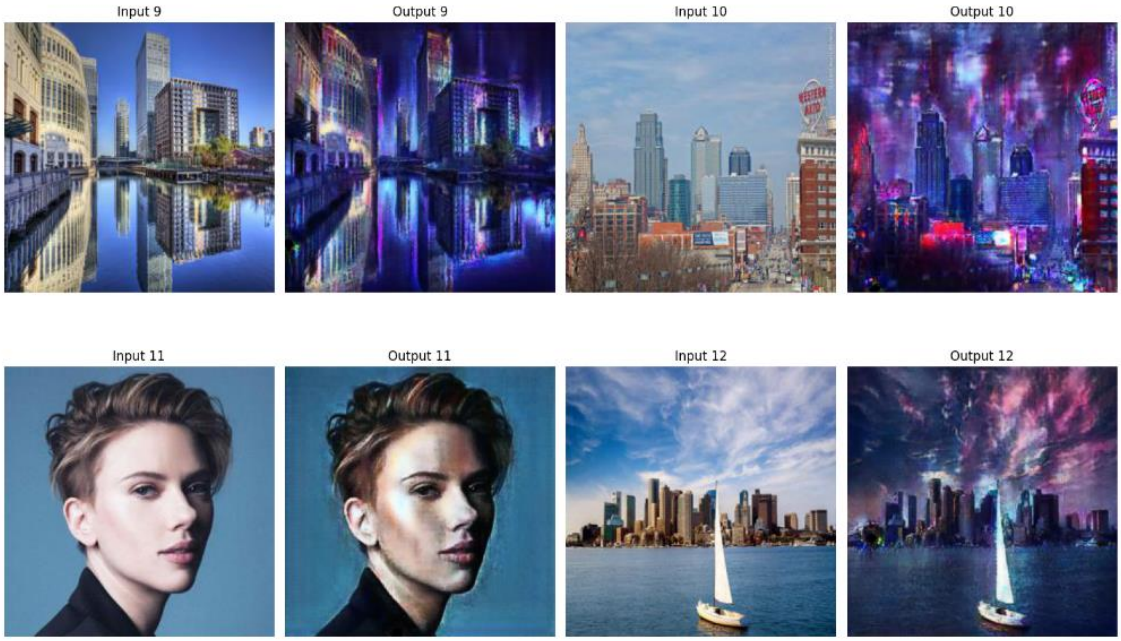

图18. 模型风格迁移效果图（3）

图19. 模型风格迁移效果图（4）

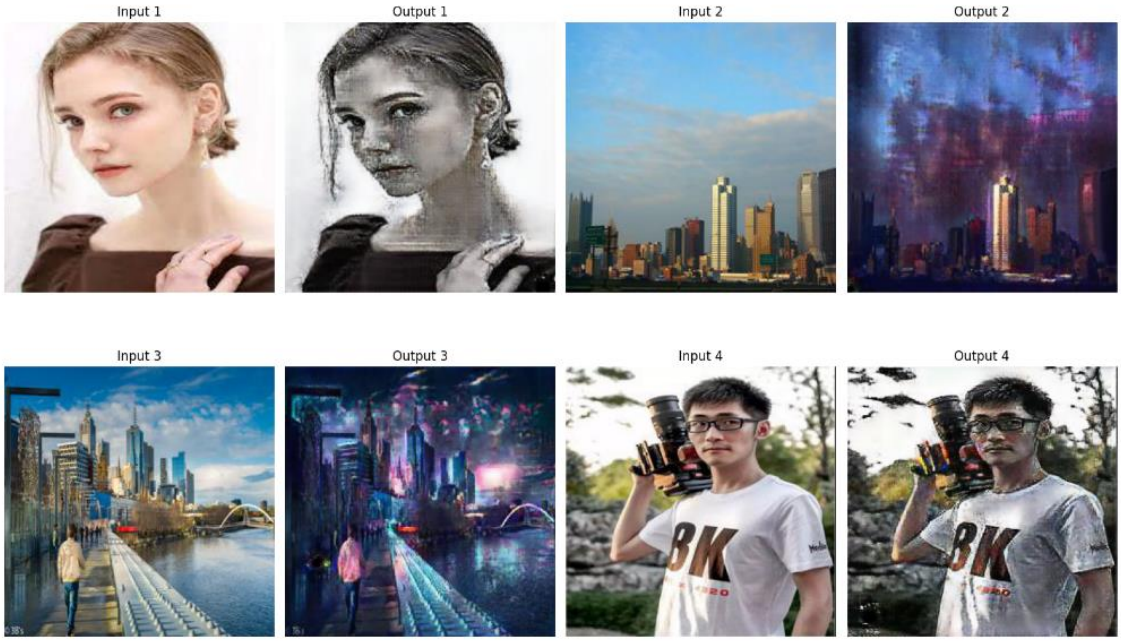

图20. 模型风格迁移效果图（5）

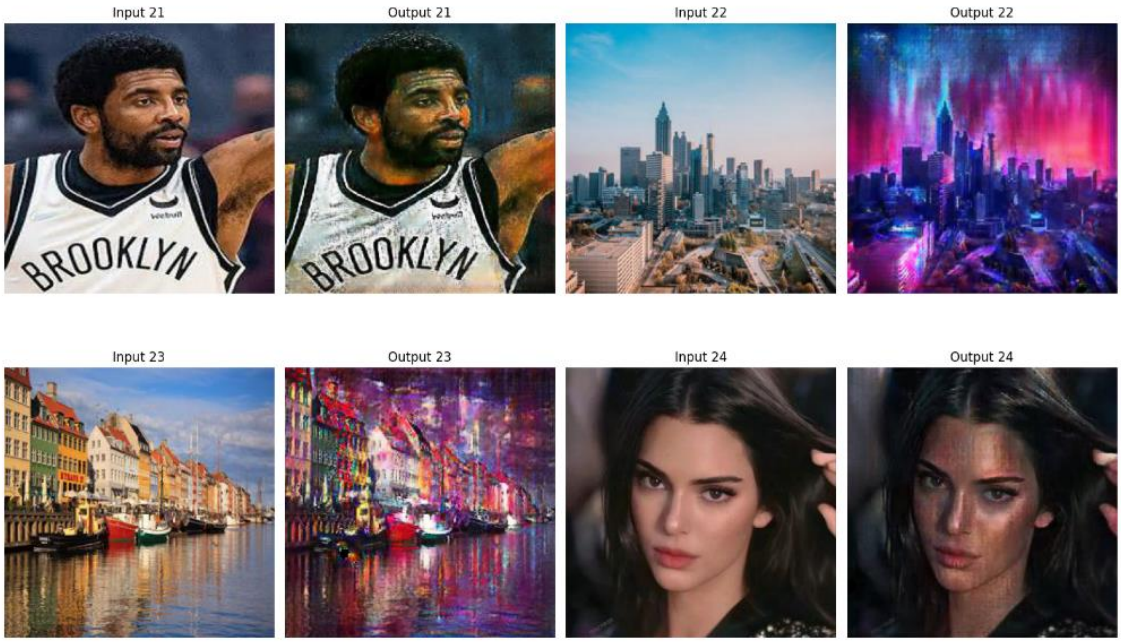

图21. 模型风格迁移效果图（6）

4.6.2 每十轮生成图片风格迁移效果

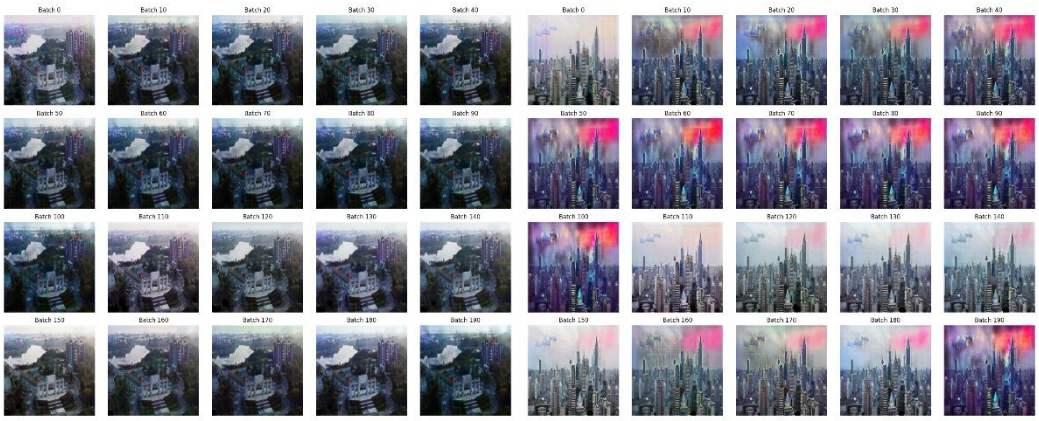

图22. 每十轮生成图片风格迁移效果图（1）

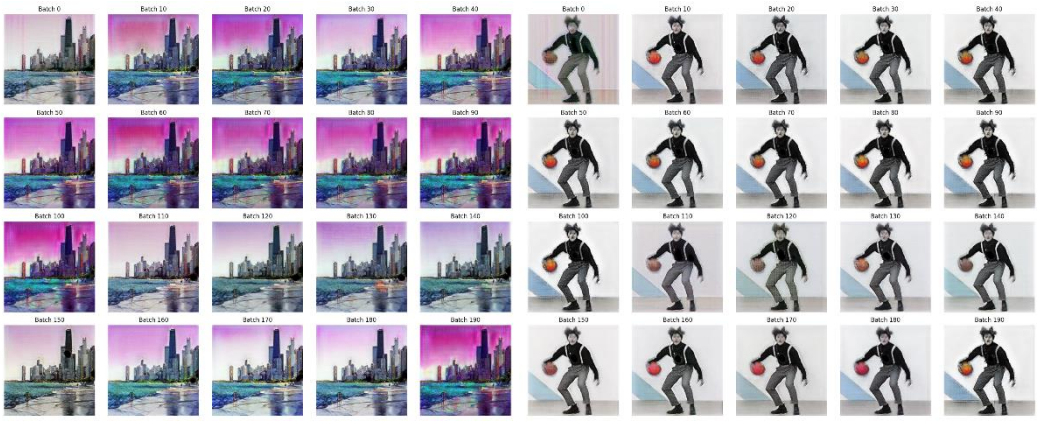

图23. 图 21 每十轮生成图片风格迁移效果图（2）

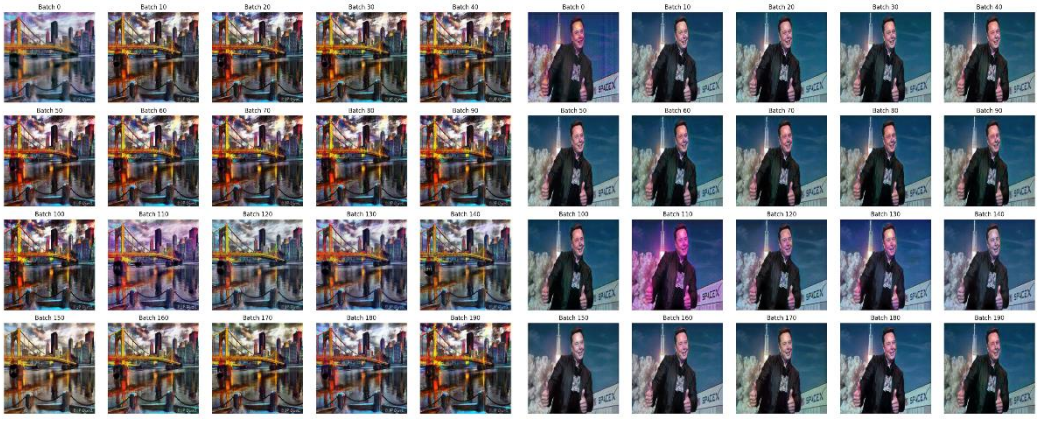

图24. 每十轮生成图片风格迁移效果图（3）

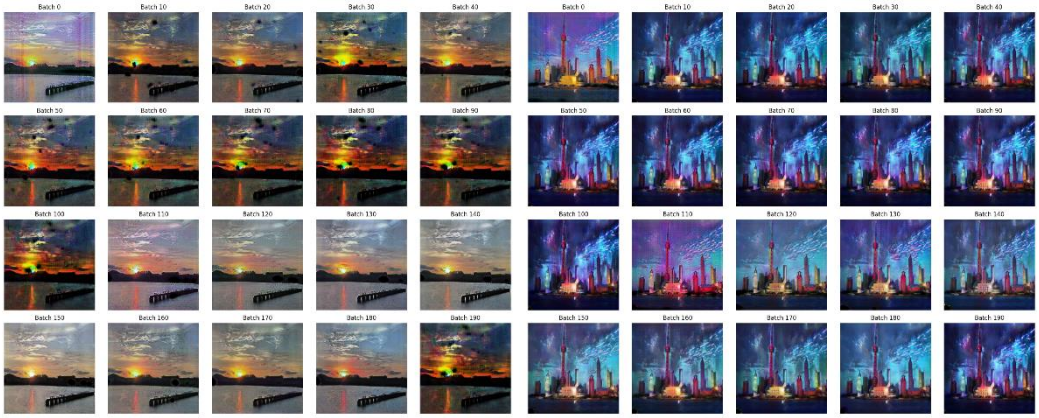

图25. 每十轮生成图片风格迁移效果图（4）

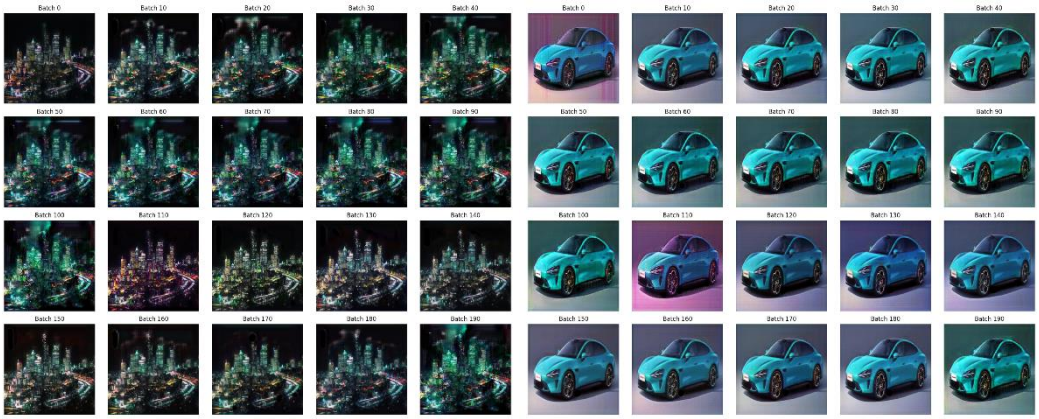

图26. 每十轮生成图片风格迁移效果图（5）

4.7 评估和测试

在生成对抗网络（GAN）的风格迁移任务中，评估模型效果的标准化定量指标相对缺乏，因此通常需要综合运用多种评价手段。评估过程不仅依赖于直观的视觉感知，还涉及定量分析，以确保结果的客观性和准确性。

尽管个人视觉感知是评估风格迁移效果的重要手段，但为了更精确地衡量模型性能，我们采用了几种定量指标。首先，我们使用了峰值信噪比（PSNR）和结构相似性指数（SSIM）来评估生成图像的质量。这些指标能够反映图像的清晰度和结构相似性，但它们并不专门针对风格迁移任务设计。

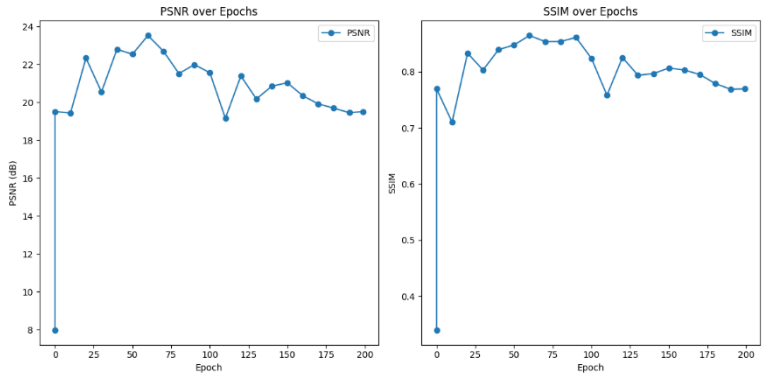

图27. 模型 PSNR 与 SSIM 随训练变化图

为了更深入地评估风格迁移的效果，我们还采用了基于 VGG 网络的感知损失。感知损失通过比较生成图像和原始图像在 VGG 特征空间的差异，能够更好地捕捉到风格迁移的细微变化。理论上，若风格迁移成功，生成图像与原始图像在视觉上应有显著差异，因此 PSNR 和 SSIM 的值应该较低，而 VGG 感知损失的值应该较高。

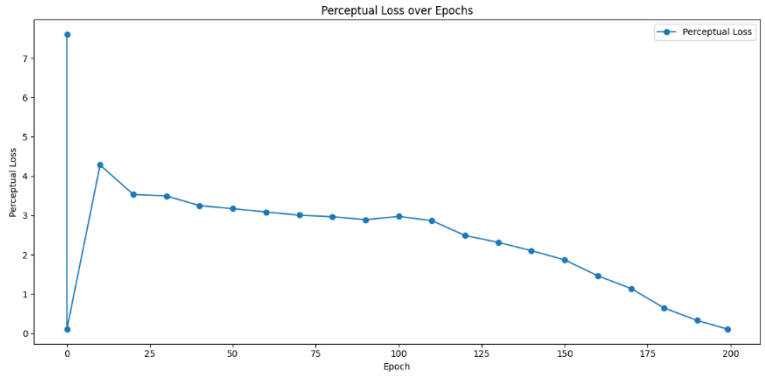

图28. VGG 感知损失随训练变化图

在实验中我们观察到，尽管模型在某些样本上表现出了预期的风格迁移效果，整体上并未完全达到我们设想中的赛博朋克风格。这一现象提示我们，可能需要进一步调整模型架构或训练过程，以改进风格迁移的效果。

5 讨论与改进

本项目利用 CycleGAN 实现了赛博朋克风格的图像生成迁移。在有限的算力资源和实践条件下，我们对数据集进行了简化，模型架构也未深入优化，导

致模型训练结果未能达到预期的赛博朋克风格迁移效果。此外，由于赛博朋克风格的多样性和复杂性，对图像进行尺寸压缩处理损失了许多特征。项目在部分图像上取得了一定的风格迁移效果，某些图像明显展示了赛博朋克风格中的霓虹灯色彩和暗黑格调等元素。未来可以基于本项目在更大的数据集上进行训练，并进一步优化模型架构，以实现更好的风格迁移效果。

尽管在有限的资源和数据集条件下取得了初步成果，但仍存在一些挑战和改进空间。以下是一些可能的改进方向：

(1). 数据集的扩展与优化：当前的数据集规模和多样性有限，限制了模型的泛化能力。未来的工作可以增加更多赛博朋克风格的图像样本，以提高模型的多样性和迁移效果。

(2). 模型架构的调整：当前的模型架构可能需要进一步优化。例如，可以尝试引入更多的卷积层或调整残差块的数量，以增强模型的特征提取和风格迁移能力。

(3). 训练过程的优化：训练过程中的超参数（如学习率、批量大小、训练周期等）对模型性能有显著影响。通过更细致的调整和使用高级的优化算法，可以进一步提高模型的训练效率和效果。

(4). 损失函数的改进：当前的损失函数可能不足以精确捕捉赛博朋克风格的特点。可以考虑设计或引入新的损失函数，以更准确地指导模型学习赛博朋克风格的特征。

(5). 多风格迁移的支持：目前模型专注于赛博朋克风格的迁移。未来的工作可以探索支持多种风格的迁移，甚至实现用户自定义风格的迁移。

6 应用前景

赛博朋克风格图像生成技术具有广泛的应用前景，以下是一些潜在的应用

领域：

(1). 艺术创作：艺术家和设计师可以利用这一技术快速生成具有赛博朋克风格的艺术作品，激发新的创意和表达方式。

(2). 影视制作：在电影和电视剧的制作过程中，可以应用这一技术创建具有未来主义视觉效果的场景和背景，降低制作成本，提高制作效率。

(3). 游戏开发：在电子游戏领域，风格迁移技术可以用于快速生成游戏世界的视觉元素，为玩家提供沉浸式的赛博朋克体验。

(4). 广告与市场营销：企业可以利用这一技术为产品或品牌创造独特的视觉形象，吸引消费者注意力，提升品牌形象。

(5). 社交媒体：用户可以在社交媒体上使用风格迁移应用，将自己的照片转换为赛博朋克风格，增加个人表达的趣味性和独特性。

7 结论

本项目成功实现了利用 CycleGAN 模型进行赛博朋克风格图像生成迁移的目标，并取得了初步的迁移效果。我们的 CycleGAN 能够有效捕捉并再现赛博朋克风格中的关键视觉元素，如霓虹灯的鲜艳色彩、暗黑基调等。尽管当前的模型在图像质量和风格一致性方面还有待提高，但本项目为后续进一步提升效果奠定了坚实的基础。通过进一步优化模型结构、调整训练参数，扩大训练数据集的多样性和数据量，可以显著提升生成图像的质量和多样性。未来可以进一步完善改进本项目模型，或者应用于其他场景。

8 附录

[1]. Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2020). Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv:1703.10593* [cs.CV].

[2]. Karras, T., Laine, S., & Aila, T. (2019). A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv:1812.04948* [cs.NE].

[3]. Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2414-2423).
[4]. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A.,& Bengio, Y. (2014). Generative Adversarial Networks. *arXiv:1406.2661* [stat.ML].
[5]. https://www.kaggle.com/datasets/heonh0/daynight-cityview
[6]. https://www.kaggle.com/datasets/cyanex1702/cyberversecyberpunk-imagesdataset
[7]. Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2018). Image-to-Image Translation with Conditional Adversarial Networks. *arXiv:1611.07004* [cs.CV].
[8]. https://zh-v2.d2l.ai/
[9]. https://kimi.moonshot.cn/
[10]. https://chatgpt.com/