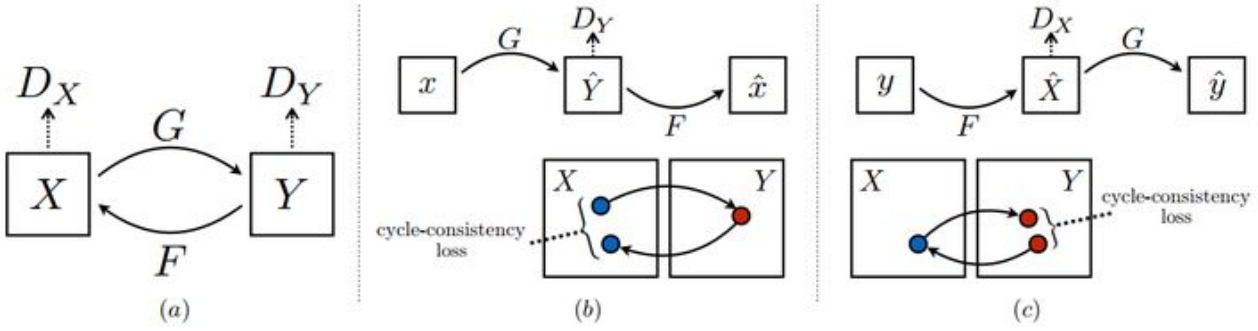
cycleGAN

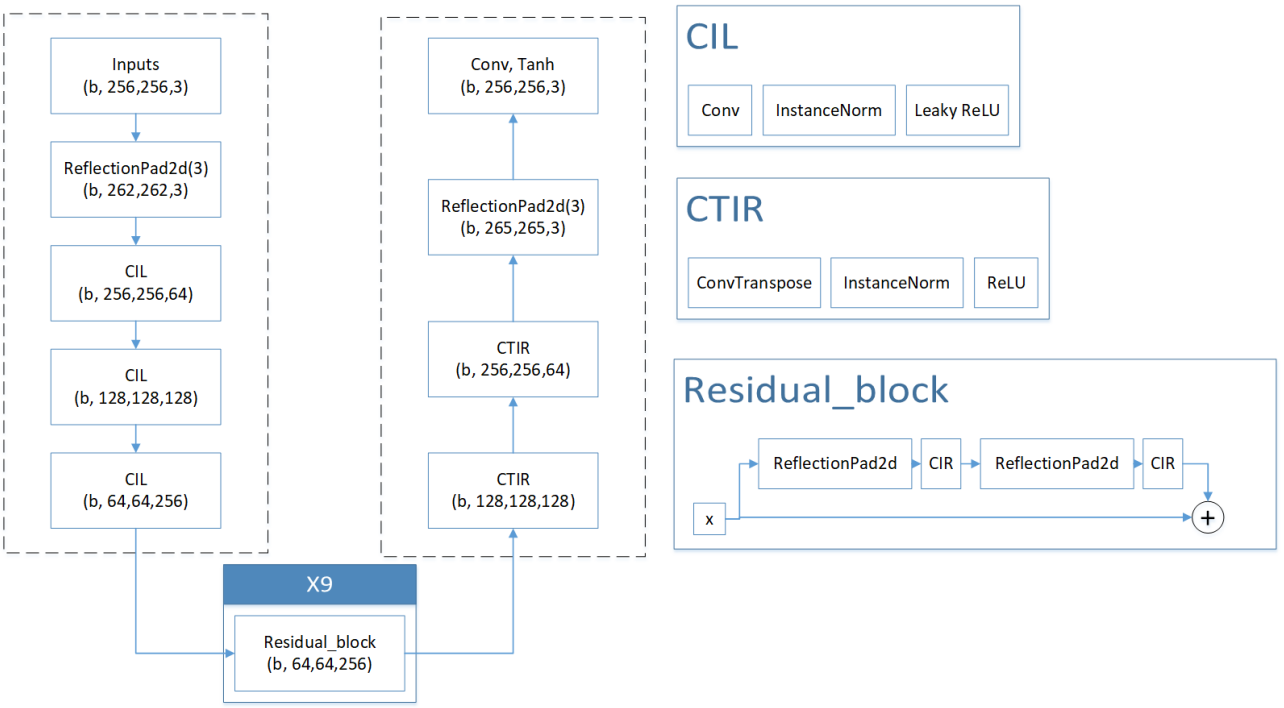
前言

cycleGAN是一种实现图像风格转换功能的GAN网络。早在它出现之前就存在着Pix2Pix实现图像风格转换，但pip2pip具有很大的局限性，主要是针对两种风格图像要对应出现，而现实中很难找到一些风格不同相同图像，也能难去拍摄获得，于是CycleGan就实现这个功能，在两种类型图像之间进行转换，而不需要对应关系。
想要做到这点，有两个比较重要的点，第一个就是双判别器。如下图a所示，两个分布X,Y，生成器G，F分别是X到Y和Y到X的映射，两个判别器Dx,Dy可以对转换后的图片进行判别。第二个点就是cycle-consistency loss，用数据集中其他的图来检验生成器，这是防止G和F过拟合，比如想把一个小狗照片转化成梵高风格，如果没有cycle-consistency loss，生成器可能会生成一张梵高真实画作来骗过Dx，而无视输入的小狗。

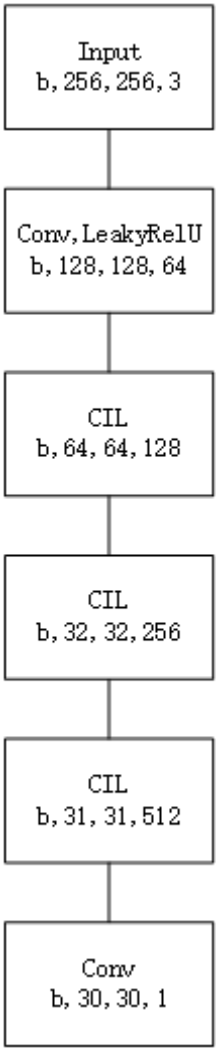


生成器和判别器

生成器的结构主要分为编码器结构，其中，解码器采用了CIL（卷积，IN正则化、Leaky ReLU激活函数），另外使用了图像增强(ReflectionPad2d，这是将图像沿着边缘上下左右进行对称，增大图像的分辨率)的方式。在两个结构链接区域使用Residual block模块，默认是9个重复模块，将数据进行恢复增强。编码器部分使用反卷积、IN标准化、ReLU激活函数去恢复图像的大小，最后通过ReflectionPad2d将图像分辨率提升，通过卷积恢复到图像原大小，有效的解决对象边缘信息。



patchGAN判别器的重点是输出结构为NxN的矩阵进行比较，这种产生了全局比较的概念，考虑了全局感受野信息的差别。判别器结构很简单，直接一个链式结构，进行变换，最后展平成(b，1)结构，这是第二个github链接的判别器



模型损失

生成器和判别器的loss函数和GAN是一样的，判别器D尽力检测出生成器G产生的假图片，生成器尽力生成图片骗过判别器

对抗loss

两部分组成：

$L_{GAN}(G, D_Y, X, Y) = E_{y \sim P_{data}(y)}[logD_Y(y)] + E_{y \sim P_{data}(y)}[log(1 - D_Y(G(x)))]$

$L_{GAN}(G, D_Y, X, Y) = E_{y \sim P_{data}(y)}[[D_Y(y)]^2] + E_{y \sim P_{data}(y)}[D_Y(G(x)) - 1_{n*n}]^2$

理论上，对抗训练可以学习映射出G和F，它们分别作为目标域Y和X产生相同的分布。然而，具有足够大的容量，网络可以将相同的输入图像集合映射到目标域中的任何图像的随机排列。因此，单独的对抗性loss不能保证可以映射单个输入。需要另外一个loss保证G和F不仅能满足各自的判别器，还能应用于其他图片。cycle consistency loss的作用就是制止这种情况的发生。它用梵高其他的画作测试FG，用另外真实照片测试GF，看看能否变回到原来的样子，这样保证了GX在整个X,Y分布区间的普适性。

$L_{cyc}(G, F) = E_{x \sim p_{data}(x)}[||F(G(x)) - x||_1] + E_{y \sim p_{data}(x)}[||F(G(y)) - y||_1]$

整体：

$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F)$

作者训练对抗损失时没有使用BCELoss（二分类交叉熵损失函数），而使用的MSE（均方根误差），作者解释MSE的训练效果更好

论文中没有写identity损失，但是源码中是体现了Identity损失的，这个损失主要是训练网络的识别能力，表示为AtoB(real B)AtoB(real_B)AtoB(real B)，就是将真实的B输入到A生成B的判别器中，查看判别器的识别损失，希望越小越好,说明生成器网络真正的理解了B的结构。

$$L_{GAN} = \frac{1}{2} * E_{x - p_{data}(x)}[D_Y(G(X)) - 1]^2 + \frac{1}{2} * E_{y - p_{data}(y)}[D_X(F(Y)) - 1]^2$$

$$L_{cycle} = E_{x - p_{data}(x)}[||F(G(X)) - X||_1] + E_{y - p_{data}(y)}[||G(F(Y)) - Y||_1]$$

$$L_{identity} = E_{x - p_{data}(x)}[||F(X) - X||_1] + E_{y - p_{data}(y)}[||G(Y) - Y||_1]$$

$$L = L_{GAN} + L_{cycle} + L_{identity}$$

训练：

环境：

Linux
python3
CPU or NVIDIA GPU + CUDA CuDNN

数据集：

使用官网提供的数据集：

```
# 下载.sh文件
bash ./datasets/download_cyclegan_dataset.sh maps
```

.sh文件里详细介绍了数据集的url地址，如：https://people.eecs.berkeley.edu/~teasung_park/CycleGAN/datasets/$FILE.zip

使用自己的数据集：

如果使用自己的数据集，格式需要与官网提供的一样。
需要在datasets文件夹中创建新的文件夹，比如马转斑马命名为'horse2zebra'，'horse2zebra'这个文件夹下有四个文件夹，分别为
'trainA':马的照片，训练集
'trainB':斑马的照片，训练集
'testA':马的照片，测试集
'testB':斑马的照片，测试集

训练模型：

```
#!./scripts/train_pix2pix.sh
python train.py --dataroot ./datasets/facades --name
facades_pix2pix --model pix2pix --direction BtoA
```

预训练模型

当然，你也可以使用官方提供的预训练模型,这样就不用自己训练了。

下载预训练模型

```
bash ./scripts/download_cyclegan_model.sh horse2zebra
```

测试模型：

如果是自己训练的模型：

```
#!./scripts/test_cyclegan.sh
python test.py --dataroot ./datasets/maps --name maps_cyclegan --
model cycle_gan
```

如果是预训练模型：

```
python test.py --dataroot datasets/horse2zebra/testA --name
horse2zebra_pretrained --model test --no_dropout
```

test.py的参数配置：
--dataroot datasets/horse2zebra/testA为测试数据的路径
--name maps_cyclegan/horse2zebra_pretrained为测试使用的模型，前者为自己训练的模型，后者为预训练模型
--model test模式为test模式
结果储存在results文件里

代码分析：

1数据读取与预处理操作

```
opt = TrainOptions().parse()
dataset = create_dataset(opt) #create_dataset定义如下
dataset_size = len(dataset)
```

```
def create_dataset(opt):
    data_loader = CustomDatasetDataLoader(opt)
    dataset = data_loader.load_data()
    return dataset
```

unaligned_dataset.py

```
class UnalignedDataset(BaseDataset):

    def __init__(self, opt):

        BaseDataset.__init__(self, opt)
        self.dir_A = os.path.join(opt.dataroot, opt.phase + 'A')  #
指定trainA路径
        self.dir_B = os.path.join(opt.dataroot, opt.phase + 'B')  #
指定trainB路径

        self.A_paths = sorted(make_dataset(self.dir_A,
opt.max_dataset_size))
        self.B_paths = sorted(make_dataset(self.dir_B,
opt.max_dataset_size))
        self.A_size = len(self.A_paths)  # 得到trainA的数据数量
        self.B_size = len(self.B_paths)  # 得到trainB的数据数量
        btoA = self.opt.direction == 'BtoA'
        input_nc = self.opt.output_nc if btoA else
self.opt.input_nc       # 指定input channel
        output_nc = self.opt.input_nc if btoA else
self.opt.output_nc      # 指定output channel，一般为三通道，因为默认情况下
为彩色图
        self.transform_A = get_transform(self.opt, grayscale=
(input_nc == 1)) # get_transform()定义在base_dataset.py,下面有展示
        self.transform_B = get_transform(self.opt, grayscale=
(output_nc == 1))

    def __getitem__(self, index):

        A_path = self.A_paths[index % self.A_size]
        if self.opt.serial_batches:
            index_B = index % self.B_size
        else:
            index_B = random.randint(0, self.B_size - 1)
```

```
        B_path = self.B_paths[index_B]
        A_img = Image.open(A_path).convert('RGB')
        B_img = Image.open(B_path).convert('RGB')
        A = self.transform_A(A_img)
        B = self.transform_B(B_img)

        return {'A': A, 'B': B, 'A_paths': A_path, 'B_paths':
B_path}

    def __len__(self):
        return max(self.A_size, self.B_size)
```

base_dataset.py

```
def get_transform(opt, params=None, grayscale=False,
method=transforms.InterpolationMode.BICUBIC, convert=True):
    transform_list = []
    if grayscale:
        transform_list.append(transforms.Grayscale(1))
    if 'resize' in opt.preprocess:
        #首先判断需不需要做resize操作，如需要则resize为286*286
        osize = [opt.load_size, opt.load_size]
        transform_list.append(transforms.Resize(osize, method))
    elif 'scale_width' in opt.preprocess:
        transform_list.append(transforms.Lambda(lambda img:
__scale_width(img, opt.load_size, opt.crop_size, method)))

    if 'crop' in opt.preprocess:
        #然后再crop操作为256*256
        if params is None:

 transform_list.append(transforms.RandomCrop(opt.crop_size))
        else:
            transform_list.append(transforms.Lambda(lambda img:
__crop(img, params['crop_pos'], opt.crop_size)))

    if opt.preprocess == 'none':
        transform_list.append(transforms.Lambda(lambda img:
__make_power_2(img, base=4, method=method)))

    if not opt.no_flip:
```

```
        # 数据增强，是否进行反转操作，有50%的概率进行反转操作
        if params is None:

 transform_list.append(transforms.RandomHorizontalFlip())
        elif params['flip']:
            transform_list.append(transforms.Lambda(lambda img:
__flip(img, params['flip'])))

    if convert:
        # 进行归一化操作
        #首先转换为tensor格式
        transform_list += [transforms.ToTensor()]
        # 0.5是为了让我们的取值范围变为-1到1
        if grayscale:
            transform_list += [transforms.Normalize((0.5,),
(0.5,))]
        else:
            transform_list += [transforms.Normalize((0.5, 0.5,
0.5), (0.5, 0.5, 0.5))]
    return transforms.Compose(transform_list)
```

2生成网络和判别网络模块构造

init.py

```
model = create_model(opt)
```

cycle_gan_model.py

```
def __init__(self, opt):
        BaseModel.__init__(self, opt)

        self.loss_names = ['D_A', 'G_A', 'cycle_A', 'idt_A', 'D_B',
'G_B', 'cycle_B', 'idt_B']
        # D_A和D_B为两个D网络
        # G_A和G_B为两个G网络
        # cycleA和cycleB保证循环一致性，输入经过GAB和GBA之后要跟原始一样
        # idt_A和idt_B的意思为正常输入B到GBA应该输出的就是B，论文中没提到
        visual_names_A = ['real_A', 'fake_B', 'rec_A']
        visual_names_B = ['real_B', 'fake_A', 'rec_B']
```

```
        if self.isTrain and self.opt.lambda_identity > 0.0:
            visual_names_A.append('idt_B')
            visual_names_B.append('idt_A')

        self.visual_names = visual_names_A + visual_names_B
        if self.isTrain:
            self.model_names = ['G_A', 'G_B', 'D_A', 'D_B']
        else:
            self.model_names = ['G_A', 'G_B']
        # 生成器，具体定义在networks.py，下面有展示
        self.netG_A = networks.define_G(opt.input_nc,
opt.output_nc, opt.ngf, opt.netG, opt.norm,
                                        not opt.no_dropout,
opt.init_type, opt.init_gain, self.gpu_ids)
        self.netG_B = networks.define_G(opt.output_nc,
opt.input_nc, opt.ngf, opt.netG, opt.norm,
                                        not opt.no_dropout,
opt.init_type, opt.init_gain, self.gpu_ids)
        # 判别器
        if self.isTrain:
            self.netD_A = networks.define_D(opt.output_nc, opt.ndf,
opt.netD,
                                            opt.n_layers_D,
opt.norm, opt.init_type, opt.init_gain, self.gpu_ids)
            self.netD_B = networks.define_D(opt.input_nc, opt.ndf,
opt.netD,
                                            opt.n_layers_D,
opt.norm, opt.init_type, opt.init_gain, self.gpu_ids)

        if self.isTrain:
            if opt.lambda_identity > 0.0:
                assert(opt.input_nc == opt.output_nc)
            self.fake_A_pool = ImagePool(opt.pool_size)
            self.fake_B_pool = ImagePool(opt.pool_size)
            self.criterionGAN =
networks.GANLoss(opt.gan_mode).to(self.device)  # GAN损失函数
            self.criterionCycle = torch.nn.L1Loss()
            self.criterionIdt = torch.nn.L1Loss()
            self.optimizer_G =
torch.optim.Adam(itertools.chain(self.netG_A.parameters(),
self.netG_B.parameters()), lr=opt.lr, betas=(opt.beta1, 0.999))
```

```
            self.optimizer_D =
torch.optim.Adam(itertools.chain(self.netD_A.parameters(),
self.netD_B.parameters()), lr=opt.lr, betas=(opt.beta1, 0.999))
            self.optimizers.append(self.optimizer_G)
            self.optimizers.append(self.optimizer_D)
```

networks.py

```
def define_G(input_nc, output_nc, ngf, netG, norm='batch',
use_dropout=False, init_type='normal', init_gain=0.02, gpu_ids=[]):
    net = None
    norm_layer = get_norm_layer(norm_type=norm)

    if netG == 'resnet_9blocks':
        # 先进行残差网络结构，具体定义在下面展示
        net = ResnetGenerator(input_nc, output_nc, ngf,
norm_layer=norm_layer, use_dropout=use_dropout, n_blocks=9)
    elif netG == 'resnet_6blocks':
        net = ResnetGenerator(input_nc, output_nc, ngf,
norm_layer=norm_layer, use_dropout=use_dropout, n_blocks=6)
    elif netG == 'unet_128':
        net = UnetGenerator(input_nc, output_nc, 7, ngf,
norm_layer=norm_layer, use_dropout=use_dropout)
    elif netG == 'unet_256':
        net = UnetGenerator(input_nc, output_nc, 8, ngf,
norm_layer=norm_layer, use_dropout=use_dropout)
    else:
        raise NotImplementedError('Generator model name [%s] is not
recognized' % netG)
    return init_net(net, init_type, init_gain, gpu_ids)
```

networks.py

```
class ResnetGenerator(nn.Module):

    def __init__(self, input_nc, output_nc, ngf=64,
norm_layer=nn.BatchNorm2d, use_dropout=False, n_blocks=6,
padding_type='reflect'):
        assert(n_blocks >= 0)
        super(ResnetGenerator, self).__init__()
        if type(norm_layer) == functools.partial:
```

```
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d

        model = [nn.ReflectionPad2d(3),
                 nn.Conv2d(input_nc, ngf, kernel_size=7, padding=0,
bias=use_bias),
                 norm_layer(ngf),
                 nn.ReLU(True)]

        n_downsampling = 2
        for i in range(n_downsampling):
            mult = 2 ** i
            model += [nn.Conv2d(ngf * mult, ngf * mult * 2,
kernel_size=3, stride=2, padding=1, bias=use_bias),
                      norm_layer(ngf * mult * 2),
                      nn.ReLU(True)]

        mult = 2 ** n_downsampling
        for i in range(n_blocks):

            model += [ResnetBlock(ngf * mult,
padding_type=padding_type, norm_layer=norm_layer,
use_dropout=use_dropout, use_bias=use_bias)]

        for i in range(n_downsampling):
            mult = 2 ** (n_downsampling - i)
            model += [nn.ConvTranspose2d(ngf * mult, int(ngf * mult
/ 2),
                                         kernel_size=3, stride=2,
                                         padding=1,
output_padding=1,
                                         bias=use_bias),
                      norm_layer(int(ngf * mult / 2)),
                      nn.ReLU(True)]
        model += [nn.ReflectionPad2d(3)]
        model += [nn.Conv2d(ngf, output_nc, kernel_size=7,
padding=0)]
        model += [nn.Tanh()]

        self.model = nn.Sequential(*model)
```

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

3.identity loss计算方法

cycle_gan_model.py

```
lambda_idt = self.opt.lambda_identity
        lambda_A = self.opt.lambda_A
        lambda_B = self.opt.lambda_B
        # Identity loss
        if lambda_idt > 0:
            # 将B输入A2B网络中应该生成尽可能与B相同的图片
            self.idt_A = self.netG_A(self.real_B)
            self.loss_idt_A = self.criterionIdt(self.idt_A,
self.real_B) * lambda_B * lambda_idt
            # 将A输入B2A网络中应该生成尽可能与A相同的图片
            self.idt_B = self.netG_B(self.real_A)
            self.loss_idt_B = self.criterionIdt(self.idt_B,
self.real_A) * lambda_A * lambda_idt
        else:
            self.loss_idt_A = 0
            self.loss_idt_B = 0
```

4.生成与判别损失函数指定

cycle_gan_model.py

```
# GAN loss D_A(G_A(A)) 将A输入A2B网络中生成假的B，蒙骗判别器
self.loss_G_A = self.criterionGAN(self.netD_A(self.fake_B), True)
# GAN loss D_B(G_B(B)) 将B输入B2A网络生成假的A，蒙骗判别器
self.loss_G_B = self.criterionGAN(self.netD_B(self.fake_A), True)
# Forward cycle loss || G_B(G_A(A)) - A|| 计算输入和还原回A之后的差异度有多大
self.loss_cycle_A = self.criterionCycle(self.rec_A, self.real_A) * lambda_A
# Backward cycle loss || G_A(G_B(B)) - B|| 计算输入和还原回B之后的差异度有多大
self.loss_cycle_B = self.criterionCycle(self.rec_B, self.real_B) * lambda_B
# 将三种损失加载一起就可以了
self.loss_G = self.loss_G_A + self.loss_G_B + self.loss_cycle_A + self.loss_cycle_B + self.loss_idt_A + self.loss_idt_B
self.loss_G.backward()
```