Домашка

tldr:

- Выбрать архитектуру из рассказанных NST, pix2pix, CycleGAN[1]
- Подберите к ней задачу, чтобы она вам нравилась
- Подберите еще одну задачу, которая уже решена (если не NST)
- Повторите решение, которое уже есть[2] (если не NST)
- Решите свою задачу

---

1. Расположены в порядке возрастания сложности и крутизны
2. Поверьте если вы сделаете этот пункт следующий будет в *разы* легче

Если вы выбрали Neural Style Transfer

Тут все довольно просто на первый и на второй взгляд. Поэтому недосотаточно просто нап ноутбук. Если вы хотите приличных баллов, то у вас есть две опции:

1. Вы разделяете картинку на две части и переносите на них разные стили.

   Нельзя просто взять и два раза применить обычную архитектуру сначала к одной чат

   От вас ожидается, что вы отдадите нейросети два картинки стиля и она внутри себя(с выходную картинку на две части и к одной части применит один стиль, а к другой - вт

2. Вы переносите *одновременно* два стиля на одну картинку контента.

   Нельзя просто взять и два раза применить обычную архитектуру сначала с одним сти

   От вас ожидается, что вы модифицируете модель(скорее лосс модели) для того, чтобы весами.

Если вы выбрали pix2pix

Здесь от вас ожидается, что вы реализуете свою архитектуру для pix2pix модели. Пожалуйс репозиториев. Этот факт очень легко обнаружить. Перед тем, как приступить проверьте, чт влезают на вашу видеокарту или на карту Google Colab. Если они не влезают, но вам все ра израсходовать все безплатные триалы облаков(Google, Amazon, .. etc) во вселенной.

Если вы выбрали CycleGAN

Здесь от вас ожидается, что вы реализуете свою архитектуру для CycleGAN модели. Пожал
репозиториев. Этот факт очень легко обнаружить. Перед тем, как приступить проверьте, чт
влезают на вашу видеокарту или на карту Google Colab. CycleGAN в этом смысле хуже, чем
влезают, но вам все равно очень хочется, то вы можете израсходовать все бесплатные три
вселенной

Remarks:

- Это задание нужно для того, чтобы вы наступили на все грабли, что есть. Узнали об и
Посмотрели на неработающие модели и поняли, что все тлен. Изгуглили весь интерне
Поверьте, оно того стиот. Не откладывайте это задание на ночь перед сдачей, так как
- У вас два союзника в этой борьбе:
    1. Оригинальная статья, те психи, что ее писала как то заставили свою модель раб
    спроводили свое детище, позволят вам написать свой вариант алгоритма.
    2. Гугл, он знает ответы на почти все ваши вопросы, но у него есть две ипостаси од
    занаете(русскоязычная), а есть еще одна, которая кусается, но знает больше(анг
    на ходу :)
- На самом деле у вас есть еще один союзник, это ментор проекта(или лектор или семи
пользоваться в ситуации, в которой вы не можете(занчит попытались и не вышло) на
- Сдавать это все нужно следующим образом. Код вы кидаете на github и отправляете
степик или еще куда-то)

```
!pip3 install pillow
!pip3 install torch
!pip3 install torchvision
!pip3 install tqdm
!pip3 install matplotlib
import numpy
!pip3 install opencv-python
```

```
            nn.ReflectionPad2d(3),
            nn.Conv2d(3, 64, 7),
            nn.LayerNorm(256),
            nn.ReLU(inplace=True),

            # Downsampling
            nn.Conv2d(64, 128, 3, stride=2, padding=1),
            nn.LayerNorm(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, 3, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 512, 3, stride=2, padding=1),
            nn.InstanceNorm2d(512),
            nn.ReLU(inplace=True),

            # Residual blocks
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),
            ResidualBlock(512),

            # Upsampling
            nn.Upsample(2),
            nn.Conv2d(512, 256, 3, stride=2, padding=1),
            nn.LayerNorm(1),
            nn.ReLU(inplace=True),
            nn.Upsample(2),
            nn.Conv2d(256, 128, 3, stride=2, padding=1),
            nn.LayerNorm(1),
            nn.ReLU(inplace=True),
            nn.Upsample(2),
            nn.Conv2d(128, 64, 3, stride=2, padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),

            # Output layer
            nn.InstanceNorm2d(3),
            nn.Conv2d(64, 3, 7),
            nn.Tanh()
        )

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


class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super(ResidualBlock, self).__init__()

        self.res = nn.Sequential(nn.ReflectionPad2d(1),
```

```
Requirement already satisfied: pillow in /usr/local/lib/python3.6/dist-packages (7.0.
```

```
!mkdir weights && cd weights && mkdir rain
```

```
Requirement already satisfied: torch==1.5.0 in /usr/local/lib/python3.6/dist-packages
```

```
# модель из следующего источника: https://github.com/Lornatang/CycleGAN-PyTorch
# использовался датасет cezanne2photo со следующего источника: https://people.eecs.berkele
"""
import torch
import torch.nn as nn
import torch.nn.functional as F


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, stride=2, padding=1),
            nn.PReLU(init = 0.2),

            nn.Conv2d(64, 128, 4, stride=2, padding=1),
            nn.LayerNorm(128),
            nn.PReLU(init = 0.2),

            nn.Conv2d(128, 256, 4, stride=2, padding=1),
            nn.LayerNorm(256),
            nn.PReLU(init = 0.2),

            nn.Conv2d(256, 512, 4, padding=1),
            nn.LayerNorm(512),
            nn.PReLU(init = 0.2),

            nn.Conv2d(512, 1024, 4, padding=1),
            nn.LayerNorm(1024),
            nn.PReLU(init = 0.2),

            nn.Conv2d(1024, 1, 4, padding=1),
        )

    def forward(self, x):
        x = self.main(x)
        x = F.avg_pool2d(x, x.size()[2:])
        x = torch.flatten(x, 1)
        return x


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # Initial convolution block
```

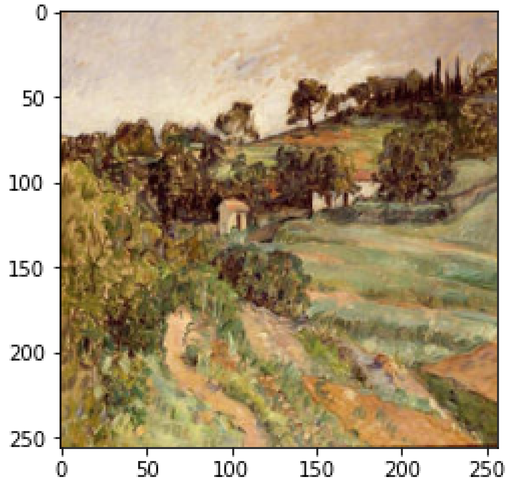
```
        self.res = nn.Sequential(nn.ReflectionPad2d(1),
                                 nn.Conv2d(in_channels, in_channels, 3),
                                 nn.LayerNorm(int(in_channels/16)),
                                 nn.ReLU(inplace=True),
                                 nn.ReflectionPad2d(1),
                                 nn.Conv2d(in_channels, in_channels, 3),
                                 nn.LayerNorm(int(in_channels/16)))

    def forward(self, x):
        return x + self.res(x)
```

Обученная нейросеть неплохо переводит картины(пейзажи) в фотографии, но при обработк
Пример ниже.

```
# оригинальное
import matplotlib.pyplot as plt
img=Image.open('/content/data/cezanne2photo/test/A/00220.jpg')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f71e8b0fdd8>
```



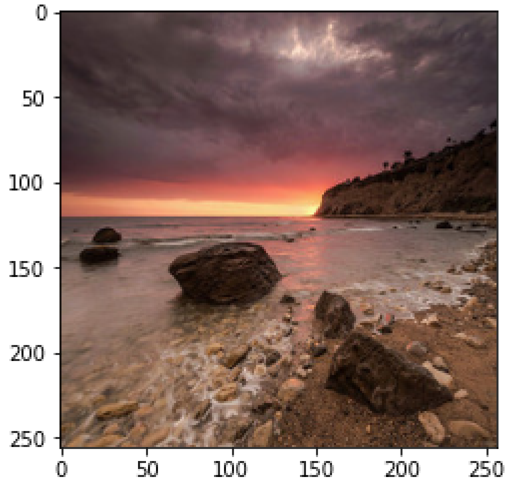

```
#результат
import matplotlib.pyplot as plt
img=Image.open('result.png')
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7f71e8b4f780>

При переводе фотографий в картины -- работает хуже. Пример ниже

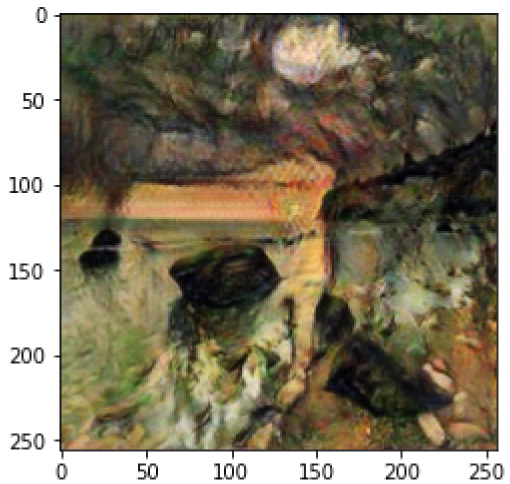
```
#оригинальное
import matplotlib.pyplot as plt
img=Image.open('/content/data/cezanne2photo/test/B/2014-08-04 23:37:50.jpg')
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7f71e8a60160>



```
#результат
import matplotlib.pyplot as plt
img=Image.open('result.png')
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7f71e8b32fd0>



**Своя архитектура CycleGAN **

```
# реализация своей архитектуры для CycleGAN

import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.main = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(3,  64, 3, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64,  128, 3, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 3, 1, bias=False),

        )


    def forward(self, x):
        x = self.main(x)
        x = F.avg_pool2d(x, x.size()[2:])
        x = torch.flatten(x, 1)
        return x


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # Initial convolution block

            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.InstanceNorm2d(256),
            nn.ReLU(inplace=True))

        self.main3 = nn.Sequential(
            nn.Conv2d(256, 128, kernel_size=3, padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 3, kernel_size=3, padding=1),
            nn.Tanh()

        )

    def forward(self, x):
        x=self.main(x)
```

```
        x=self.main3(x)
        return x
```

```
import glob
import os
import random
import time
from threading import Thread

import cv2
import numpy as np
from PIL import Image
from torch.utils.data import Dataset


class ImageDataset(Dataset):
    def __init__(self, root, transform=None, unaligned=False, mode="train"):
        self.transform = transform
        self.unaligned = unaligned

        self.files_A = sorted(glob.glob(os.path.join(root, f"{mode}/A") + "/*.*"))
        self.files_B = sorted(glob.glob(os.path.join(root, f"{mode}/B") + "/*.*"))

    def __getitem__(self, index):
        item_A = self.transform(Image.open(self.files_A[index % len(self.files_A)]))

        if self.unaligned:
            item_B = self.transform(Image.open(self.files_B[random.randint(0, len(self.fil
        else:
            item_B = self.transform(Image.open(self.files_B[index % len(self.files_B)]))

        return {"A": item_A, "B": item_B}

    def __len__(self):
        return max(len(self.files_A), len(self.files_B))


class DecayLR:
    def __init__(self, epochs, offset, decay_epochs):
        epoch_flag = epochs - decay_epochs
        assert (epoch_flag > 0), "Decay must start before the training session ends!"
        self.epochs = epochs
        self.offset = offset
        self.decay_epochs = decay_epochs

    def step(self, epoch):
        return 1.0 - max(0, epoch + self.offset - self.decay_epochs) / (
```

```
                self.epochs - self.decay_epochs)
```

```
import random

import torch


class ReplayBuffer:
    def __init__(self, max_size=50):
        assert (max_size > 0), "Empty buffer or trying to create a black hole. Be careful.
        self.max_size = max_size
        self.data = []

    def push_and_pop(self, data):
        to_return = []
        for element in data.data:
            element = torch.unsqueeze(element, 0)
            if len(self.data) < self.max_size:
                self.data.append(element)
                to_return.append(element)
            else:
                if random.uniform(0, 1) > 0.5:
                    i = random.randint(0, self.max_size - 1)
                    to_return.append(self.data[i].clone())
                    self.data[i] = element
                else:
                    to_return.append(element)
        return torch.cat(to_return)


# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    elif classname.find("BatchNorm") != -1:
        torch.nn.init.normal_(m.weight, 1.0, 0.02)
        torch.nn.init.zeros_(m.bias)
```

```
import argparse
import random
import time

import torch.backends.cudnn as cudnn
import torch.utils.data.distributed
import torchvision.transforms as transforms
import torchvision.utils as vutils
from PIL import Image

#from cyclegan_pytorch import Generator
```

```
file = "/content/gdrive/My Drive/Colab Notebooks/GanR/data/rain/test/B/38.jpg"
model_name = "/content/gdrive/My Drive/Colab Notebooks/GanR/weights/rain149/netG_B2A_epoch_

image_size = 256

cudnn.benchmark = True
device = torch.device("cuda:0")

# create model
model = Generator().to(device)

# Load state dicts
model.load_state_dict(torch.load(model_name))

# Set model mode
model.eval()

# Load image
image = Image.open(file)
pre_process = transforms.Compose([transforms.Resize(image_size),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5
                                  ])
image = pre_process(image).unsqueeze(0)
image = image.to(device)

start = time.clock()
fake_image = model(image)
elapsed = (time.clock() - start)
print(f"cost {elapsed:.4f}s")
vutils.save_image(fake_image.detach(), "result.png", normalize=True)
```

```
cost 0.0264s
```

Преобразование дождливой погоды в ясную

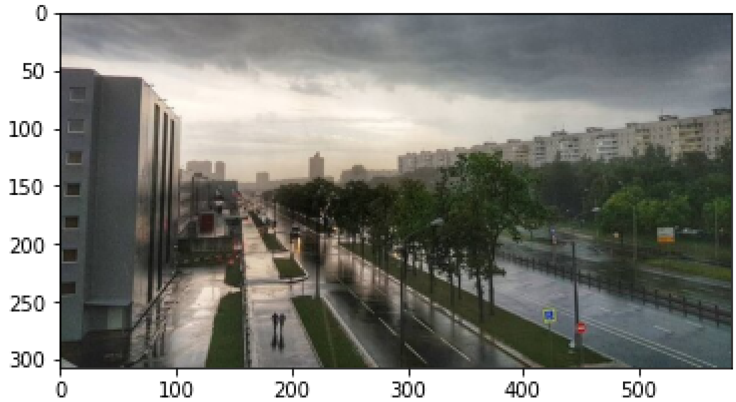
На примере изображений №199 хорошо видно, что линии дождя/града исчезают и освеще

```
# 199
from IPython.display import Image, display
display(Image('/content/gdrive/My Drive/Colab Notebooks/GanR/data/rain/test/B/199.jpg'))
```

```
# 199
from IPython.display import Image, display
display(Image('result.png'))
```

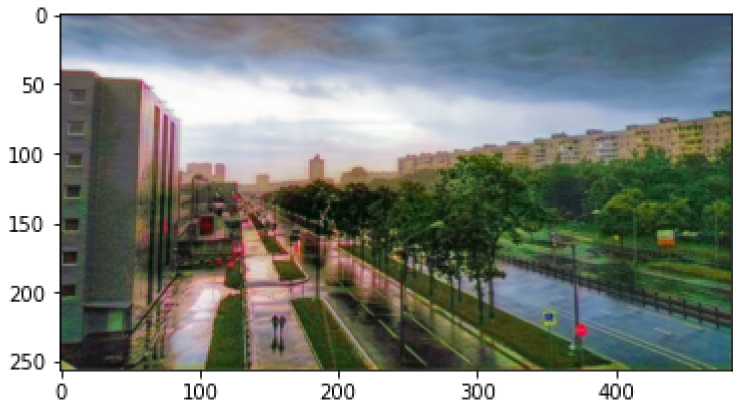
```
#22 оригинальное
import matplotlib.pyplot as plt
img=Image.open('/content/gdrive/My Drive/Colab Notebooks/GanR/data/rain/test/B/22.jpg')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa16d279668>
```




```
#22 результат
img=Image.open('result.png')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa16d253d68>
```



Теперь проверим работу в обратной направлении: дорисовывание дождя

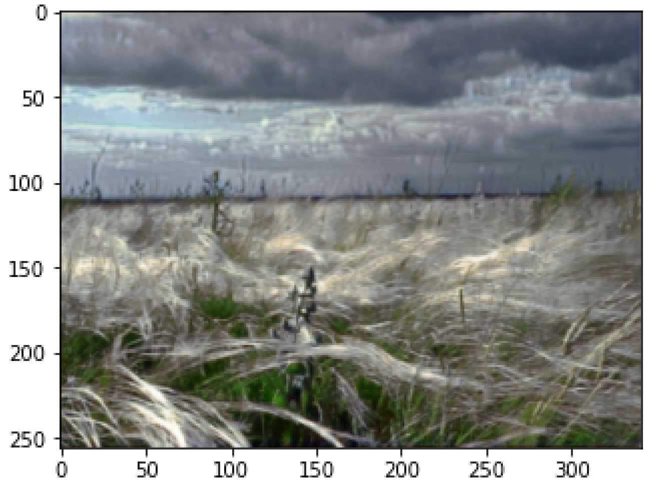
На втором изображении №129 видны появления полос осадков

```
#129 оригинальное
import matplotlib.pyplot as plt
img=Image.open('/content/gdrive/My Drive/Colab Notebooks/GanR/data/rain/test/A/129.jpg')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa18002e898>
```

```
#129 результат (дорисовка дождя)
img=Image.open('result.png')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa180141fd0>
```



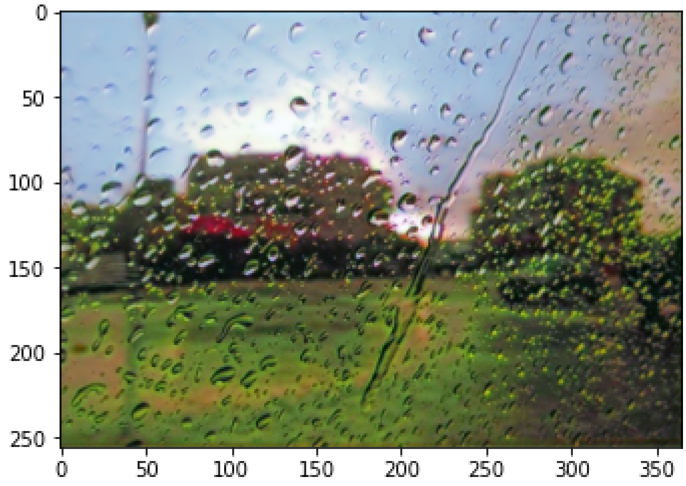
Данную архитектуру нейронной сети хорошо применять для раскрашивания изображений. приведены изображения под №13.

```
#13 оригинальное
import matplotlib.pyplot as plt
img=Image.open('/content/gdrive/My Drive/Colab Notebooks/GanR/data/rain/test/B/13.jpg')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa16d194710>
```

```
#13 результат
img=Image.open('result.png')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa16d1b28d0>
```




```
# Обучение
import torch.backends.cudnn as cudnn
import torch.utils.data
import torchvision.transforms as transforms
import torchvision.utils as vutils
from PIL import Image
from tqdm import tqdm
import argparse
import itertools
import os
import random



adataroot = "/content/gdrive/My Drive/Colab Notebooks/GanR/data"
adataset = "rain"
aepochs = 150
adecay_epochs = 100
abatch_size = 6
alr = 0.0002
ap = 100
anetG_A2B = ""
anetG_B2A = ""
anetD_A = ""
anetD_B = ""
aimage_size = 256
aoutf = "./outputs"
aprint_freq = 100

try:
    os.makedirs(aoutf)
except OSError:
    pass
```

```
try:
    os.makedirs("weights")
except OSError:
    pass


amanualSeed = random.randint(1, 10000)
print("Random Seed: ", amanualSeed)
random.seed(amanualSeed)
torch.manual_seed(amanualSeed)

cudnn.benchmark = True


# Dataset
dataset = ImageDataset(root=os.path.join(adataroot, adataset),
                       transform=transforms.Compose([
                           transforms.Resize(int(aimage_size * 1.12), Image.BICUBIC),
                           transforms.RandomCrop(aimage_size),
                           transforms.RandomHorizontalFlip(),
                           transforms.ToTensor(),
                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),
                       unaligned=True)

dataloader = torch.utils.data.DataLoader(dataset, batch_size=abatch_size, shuffle=True, pi

try:
    os.makedirs(os.path.join(aoutf, adataset, "A"))
    os.makedirs(os.path.join(aoutf, adataset, "B"))
except OSError:
    pass

try:
    os.makedirs(os.path.join("weights", adataset))
except OSError:
    pass

device = torch.device("cuda:0")

# create model
netG_A2B = Generator().to(device)
netG_B2A = Generator().to(device)
netD_A = Discriminator().to(device)
netD_B = Discriminator().to(device)

netG_A2B.apply(weights_init)
netG_B2A.apply(weights_init)
netD_A.apply(weights_init)
netD_B.apply(weights_init)


# define loss function (adversarial_loss) and optimizer
cycle_loss = torch.nn.L1Loss().to(device)
identity_loss = torch.nn.L1Loss().to(device)
```

```
adversarial_loss = torch.nn.MSELoss().to(device)

# Optimizers
optimizer_G = torch.optim.Adam(itertools.chain(netG_A2B.parameters(), netG_B2A.parameters(
                               lr=alr, betas=(0.5, 0.999))
optimizer_D_A = torch.optim.Adam(netD_A.parameters(), lr=alr, betas=(0.5, 0.999))
optimizer_D_B = torch.optim.Adam(netD_B.parameters(), lr=alr, betas=(0.5, 0.999))

lr_lambda = DecayLR(aepochs, 0, adecay_epochs).step
lr_scheduler_G = torch.optim.lr_scheduler.LambdaLR(optimizer_G, lr_lambda=lr_lambda)
lr_scheduler_D_A = torch.optim.lr_scheduler.LambdaLR(optimizer_D_A, lr_lambda=lr_lambda)
lr_scheduler_D_B = torch.optim.lr_scheduler.LambdaLR(optimizer_D_B, lr_lambda=lr_lambda)

g_losses = []
d_losses = []

identity_losses = []
gan_losses = []
cycle_losses = []

fake_A_buffer = ReplayBuffer()
fake_B_buffer = ReplayBuffer()

for epoch in range(0, aepochs):
    progress_bar = tqdm(enumerate(dataloader), total=len(dataloader))

    for i, data in progress_bar:

        # get batch size data
        real_image_A = data["A"].to(device)
        real_image_B = data["B"].to(device)
        batch_size = real_image_A.size(0)

        # real data label is 1, fake data label is 0.
        real_label = torch.full((batch_size, 1), 1, device=device, dtype=torch.float32)
        fake_label = torch.full((batch_size, 1), 0, device=device, dtype=torch.float32)

        ##############################################
        # (1) Update G network: Generators A2B and B2A
        ##############################################

        # Set G_A and G_B's gradients to zero
        optimizer_G.zero_grad()


        # Identity loss
        # G_B2A(A) should equal A if real A is fed
        identity_image_A = netG_B2A(real_image_A)
        loss_identity_A = identity_loss(identity_image_A, real_image_A) * 5.0
        # G_A2B(B) should equal B if real B is fed
        identity_image_B = netG_A2B(real_image_B)
        loss_identity_B = identity_loss(identity_image_B, real_image_B) * 5.0

        # GAN loss
        # GAN loss D_A(G_A(A))
        fake_image_A = netG_B2A(real_image_B)
```

```
        fake_image_A = netG_B2A(real_image_B)
        fake_output_A = netD_A(fake_image_A)
        loss_GAN_B2A = adversarial_loss(fake_output_A, real_label)
        # GAN loss D_B(G_B(B))
        fake_image_B = netG_A2B(real_image_A)
        fake_output_B = netD_B(fake_image_B)
        loss_GAN_A2B = adversarial_loss(fake_output_B, real_label)

        # Cycle loss
        recovered_image_A = netG_B2A(fake_image_B)
        loss_cycle_ABA = cycle_loss(recovered_image_A, real_image_A) * 10.0

        recovered_image_B = netG_A2B(fake_image_A)
        loss_cycle_BAB = cycle_loss(recovered_image_B, real_image_B) * 10.0

        # Combined loss and calculate gradients
        errG = loss_identity_A + loss_identity_B + loss_GAN_A2B + loss_GAN_B2A + loss_cycl

        # Calculate gradients for G_A and G_B
        errG.backward()
        # Update G_A and G_B's weights
        optimizer_G.step()

        ##############################################
        # (2) Update D network: Discriminator A
        ##############################################

        # Set D_A gradients to zero
        optimizer_D_A.zero_grad()

        # Real A image loss
        real_output_A = netD_A(real_image_A)
        errD_real_A = adversarial_loss(real_output_A, real_label)

        # Fake A image loss
        fake_image_A = fake_A_buffer.push_and_pop(fake_image_A)
        fake_output_A = netD_A(fake_image_A.detach())
        errD_fake_A = adversarial_loss(fake_output_A, fake_label)

        # Combined loss and calculate gradients
        errD_A = (errD_real_A + errD_fake_A) / 2

        # Calculate gradients for D_A
        errD_A.backward()
        # Update D_A weights
        optimizer_D_A.step()

        ##############################################
        # (3) Update D network: Discriminator B
        ##############################################

        # Set D_B gradients to zero
        optimizer_D_B.zero_grad()

        # Real B image loss
        real_output_B = netD_B(real_image_B)
```

```
        real_output_B = netD_B(real_image_B)
        errD_real_B = adversarial_loss(real_output_B, real_label)

        # Fake B image loss
        fake_image_B = fake_B_buffer.push_and_pop(fake_image_B)
        fake_output_B = netD_B(fake_image_B.detach())
        errD_fake_B = adversarial_loss(fake_output_B, fake_label)

        # Combined loss and calculate gradients
        errD_B = (errD_real_B + errD_fake_B) / 2

        # Calculate gradients for D_B
        errD_B.backward()
        # Update D_B weights
        optimizer_D_B.step()


        progress_bar.set_description(
            f"[{epoch}/{aepochs - 1}][{i}/{len(dataloader) - 1}] "
            f"Loss_D: {(errD_A + errD_B).item():.4f} "
            f"Loss_G: {errG.item():.4f} "
            f"Loss_G_identity: {(loss_identity_A + loss_identity_B).item():.4f} "
            f"loss_G_GAN: {(loss_GAN_A2B + loss_GAN_B2A).item():.4f} "
            f"loss_G_cycle: {(loss_cycle_ABA + loss_cycle_BAB).item():.4f}")


        if i % aprint_freq == 0:
            vutils.save_image(real_image_A,
                              f"{aoutf}/{adataset}/A/real_samples.png",
                              normalize=True)
            vutils.save_image(real_image_B,
                              f"{aoutf}/{adataset}/B/real_samples.png",
                              normalize=True)

            fake_image_A = 0.5 * (netG_B2A(real_image_B).data + 1.0)
            fake_image_B = 0.5 * (netG_A2B(real_image_A).data + 1.0)

            vutils.save_image(fake_image_A.detach(),
                              f"{aoutf}/{adataset}/A/fake_samples_epoch_{epoch}.png",
                              normalize=True)
            vutils.save_image(fake_image_B.detach(),
                              f"{aoutf}/{adataset}/B/fake_samples_epoch_{epoch}.png",
                              normalize=True)


    # do check pointing
    torch.save(netG_A2B.state_dict(), f"weights/{adataset}/netG_A2B_epoch_{epoch}.pth")
    torch.save(netG_B2A.state_dict(), f"weights/{adataset}/netG_B2A_epoch_{epoch}.pth")
    torch.save(netD_A.state_dict(), f"weights/{adataset}/netD_A_epoch_{epoch}.pth")
    torch.save(netD_B.state_dict(), f"weights/{adataset}/netD_B_epoch_{epoch}.pth")

    #test_images(model_name=f"weights/{adataset}/netG_A2B_epoch_{epoch}.pth")

    model_save_nameG_A2B = f"weights/{adataset}/netG_A2B_epoch_{epoch}.pth"
    model_save_nameG_B2A = f"weights/{adataset}/netG_B2A_epoch_{epoch}.pth"
    model_save_nameD_A = f"weights/{adataset}/netD_A_epoch_{epoch}.pth"
```

```
    model_save_nameD_A = f"weights/{adataset}/netD_A_epoch_{epoch}.pth"
    model_save_nameD_B = f"weights/{adataset}/netD_B_epoch_{epoch}.pth"

    path = F"/content/gdrive/My Drive/Colab Notebooks/GanR/{model_save_nameG_A2B}"
    torch.save(netG_A2B.state_dict(), path)
    path = F"/content/gdrive/My Drive/Colab Notebooks/GanR/{model_save_nameG_B2A}"
    torch.save(netG_B2A.state_dict(), path)
    path = F"/content/gdrive/My Drive/Colab Notebooks/GanR/{model_save_nameD_A}"
    torch.save(netD_A.state_dict(), path)
    path = F"/content/gdrive/My Drive/Colab Notebooks/GanR/{model_save_nameD_B}"
    torch.save(netD_B.state_dict(), path)



    # Update learning rates
    lr_scheduler_G.step()
    lr_scheduler_D_A.step()
    lr_scheduler_D_B.step()

# save last check pointing
torch.save(netG_A2B.state_dict(), f"weights/{adataset}/netG_A2B.pth")
torch.save(netG_B2A.state_dict(), f"weights/{adataset}/netG_B2A.pth")
torch.save(netD_A.state_dict(), f"weights/{adataset}/netD_A.pth")
torch.save(netD_B.state_dict(), f"weights/{adataset}/netD_B.pth")
```

```
Random Seed:  2995
  0%|          | 0/44 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-packages/torch/nn
  return F.mse_loss(input, target, reduction=self.reduction)
[0/149][42/43] Loss_D: 0.4876 Loss_G: 6.6877 Loss_G_identity: 1.9877 loss_G_GAN: 0.52
  return F.mse_loss(input, target, reduction=self.reduction)
[0/149][43/43] Loss_D: 0.4711 Loss_G: 6.3956 Loss_G_identity: 1.9332 loss_G_GAN: 0.48
[1/149][43/43] Loss_D: 0.4938 Loss_G: 7.9936 Loss_G_identity: 2.4837 loss_G_GAN: 0.48
[2/149][43/43] Loss_D: 0.4594 Loss_G: 4.5665 Loss_G_identity: 1.2560 loss_G_GAN: 0.50
[3/149][43/43] Loss_D: 0.4898 Loss_G: 5.6100 Loss_G_identity: 1.5883 loss_G_GAN: 0.52
[4/149][43/43] Loss_D: 0.5166 Loss_G: 5.4828 Loss_G_identity: 1.6619 loss_G_GAN: 0.47
[5/149][43/43] Loss_D: 0.3918 Loss_G: 7.0219 Loss_G_identity: 2.0781 loss_G_GAN: 0.59
[6/149][43/43] Loss_D: 0.4742 Loss_G: 6.3384 Loss_G_identity: 1.7013 loss_G_GAN: 0.53
[7/149][43/43] Loss_D: 0.4414 Loss_G: 5.1373 Loss_G_identity: 1.4719 loss_G_GAN: 0.61
[8/149][43/43] Loss_D: 0.4093 Loss_G: 7.3040 Loss_G_identity: 2.2564 loss_G_GAN: 0.60
[9/149][43/43] Loss_D: 0.5221 Loss_G: 5.1533 Loss_G_identity: 1.4363 loss_G_GAN: 0.49
[10/149][43/43] Loss_D: 0.4708 Loss_G: 6.5199 Loss_G_identity: 1.8673 loss_G_GAN: 0.5
[11/149][43/43] Loss_D: 0.4635 Loss_G: 5.3050 Loss_G_identity: 1.5569 loss_G_GAN: 0.4
[12/149][43/43] Loss_D: 0.4547 Loss_G: 6.0913 Loss_G_identity: 1.7537 loss_G_GAN: 0.5
[13/149][43/43] Loss_D: 0.5207 Loss_G: 4.9973 Loss_G_identity: 1.3276 loss_G_GAN: 0.4
[14/149][43/43] Loss_D: 0.3843 Loss_G: 5.7126 Loss_G_identity: 1.6882 loss_G_GAN: 0.4
[15/149][43/43] Loss_D: 0.4489 Loss_G: 5.6740 Loss_G_identity: 1.6323 loss_G_GAN: 0.4
[16/149][43/43] Loss_D: 0.4719 Loss_G: 5.5147 Loss_G_identity: 1.6345 loss_G_GAN: 0.5
[17/149][43/43] Loss_D: 0.3861 Loss_G: 4.9512 Loss_G_identity: 1.3235 loss_G_GAN: 0.6
[18/149][43/43] Loss_D: 0.4571 Loss_G: 6.5705 Loss_G_identity: 1.6643 loss_G_GAN: 0.5
[19/149][43/43] Loss_D: 0.4880 Loss_G: 4.6497 Loss_G_identity: 1.3611 loss_G_GAN: 0.5
[20/149][43/43] Loss_D: 0.4170 Loss_G: 4.4679 Loss_G_identity: 1.0639 loss_G_GAN: 0.5
[21/149][43/43] Loss_D: 0.4817 Loss_G: 5.9181 Loss_G_identity: 1.5762 loss_G_GAN: 0.6
[22/149][43/43] Loss_D: 0.4676 Loss_G: 3.8965 Loss_G_identity: 1.0895 loss_G_GAN: 0.5
[23/149][43/43] Loss_D: 0.4677 Loss_G: 4.7349 Loss_G_identity: 1.4154 loss_G_GAN: 0.5
[24/149][43/43] Loss_D: 0.4689 Loss_G: 5.2312 Loss_G_identity: 1.4191 loss_G_GAN: 0.5
[25/149][43/43] Loss_D: 0.4122 Loss_G: 5.9940 Loss_G_identity: 1.8293 loss_G_GAN: 0.5
[26/149][43/43] Loss_D: 0.4905 Loss_G: 4.0712 Loss_G_identity: 1.1806 loss_G_GAN: 0.4
[27/149][43/43] Loss_D: 0.5105 Loss_G: 4.1637 Loss_G_identity: 1.2411 loss_G_GAN: 0.5
[28/149][43/43] Loss_D: 0.4562 Loss_G: 4.8327 Loss_G_identity: 1.2793 loss_G_GAN: 0.5
[29/149][43/43] Loss_D: 0.5144 Loss_G: 3.8339 Loss_G_identity: 1.0502 loss_G_GAN: 0.6
[30/149][43/43] Loss_D: 0.4660 Loss_G: 4.7142 Loss_G_identity: 1.3900 loss_G_GAN: 0.5
[31/149][43/43] Loss_D: 0.4864 Loss_G: 4.5024 Loss_G_identity: 1.2201 loss_G_GAN: 0.6
[32/149][43/43] Loss_D: 0.4474 Loss_G: 4.0003 Loss_G_identity: 1.2440 loss_G_GAN: 0.4
[33/149][43/43] Loss_D: 0.4703 Loss_G: 5.2638 Loss_G_identity: 1.4703 loss_G_GAN: 0.7
[34/149][43/43] Loss_D: 0.3721 Loss_G: 4.7673 Loss_G_identity: 1.4463 loss_G_GAN: 0.5
[35/149][43/43] Loss_D: 0.4784 Loss_G: 4.3706 Loss_G_identity: 1.1335 loss_G_GAN: 0.7
[36/149][43/43] Loss_D: 0.4452 Loss_G: 4.5905 Loss_G_identity: 1.2557 loss_G_GAN: 0.6
[37/149][43/43] Loss_D: 0.4168 Loss_G: 3.8360 Loss_G_identity: 1.0165 loss_G_GAN: 0.5
[38/149][43/43] Loss_D: 0.5181 Loss_G: 4.1638 Loss_G_identity: 1.1158 loss_G_GAN: 0.5
[39/149][43/43] Loss_D: 0.4759 Loss_G: 3.5514 Loss_G_identity: 0.8662 loss_G_GAN: 0.5
[40/149][43/43] Loss_D: 0.5343 Loss_G: 4.1597 Loss_G_identity: 1.3924 loss_G_GAN: 0.4
[41/149][43/43] Loss_D: 0.3856 Loss_G: 5.0852 Loss_G_identity: 1.5239 loss_G_GAN: 0.6
[42/149][43/43] Loss_D: 0.4364 Loss_G: 3.8472 Loss_G_identity: 0.9904 loss_G_GAN: 0.6
[43/149][43/43] Loss_D: 0.3960 Loss_G: 4.7242 Loss_G_identity: 1.3693 loss_G_GAN: 0.5
[44/149][43/43] Loss_D: 0.5075 Loss_G: 4.2655 Loss_G_identity: 1.0897 loss_G_GAN: 0.6
[45/149][43/43] Loss_D: 0.4322 Loss_G: 4.1021 Loss_G_identity: 1.2192 loss_G_GAN: 0.5
[46/149][43/43] Loss_D: 0.4005 Loss_G: 4.0132 Loss_G_identity: 1.0224 loss_G_GAN: 0.6
[47/149][43/43] Loss_D: 0.4586 Loss_G: 3.9912 Loss_G_identity: 1.2514 loss_G_GAN: 0.4
[48/149][43/43] Loss_D: 0.5446 Loss_G: 5.1077 Loss_G_identity: 1.2376 loss_G_GAN: 0.5
[49/149][43/43] Loss_D: 0.4821 Loss_G: 3.5838 Loss_G_identity: 0.9781 loss_G_GAN: 0.4
[50/149][43/43] Loss_D: 0.4868 Loss_G: 3.6844 Loss_G_identity: 0.9592 loss_G_GAN: 0.7
[51/149][43/43] Loss_D: 0.4674 Loss_G: 4.3521 Loss_G_identity: 1.3118 loss_G_GAN: 0.5
[52/149][43/43] Loss_D: 0.4586 Loss_G: 4.0157 Loss_G_identity: 1.1089 loss_G_GAN: 0.5
[53/149][43/43] Loss_D: 0.4086 Loss_G: 4.2251 Loss_G_identity: 1.2792 loss_G_GAN: 0.6
[54/149][43/43] Loss_D: 0.4808 Loss_G: 3.7984 Loss_G_identity: 1.0846 loss_G_GAN: 0.5
[55/149][43/43] Loss_D: 0.3482 Loss_G: 4.3015 Loss_G_identity: 1.4217 loss_G_GAN: 0.5
```

```
[56/149][43/43] Loss_D: 0.4891 Loss_G: 3.9655 Loss_G_identity: 0.9819 loss_G_GAN: 0.7
[57/149][43/43] Loss_D: 0.4399 Loss_G: 4.2165 Loss_G_identity: 1.2332 loss_G_GAN: 0.5
[58/149][43/43] Loss_D: 0.4864 Loss_G: 4.2851 Loss_G_identity: 1.2538 loss_G_GAN: 0.6
[59/149][43/43] Loss_D: 0.4346 Loss_G: 3.9876 Loss_G_identity: 1.0926 loss_G_GAN: 0.6
[60/149][43/43] Loss_D: 0.3576 Loss_G: 4.5084 Loss_G_identity: 1.4871 loss_G_GAN: 0.4
[61/149][43/43] Loss_D: 0.4471 Loss_G: 3.4984 Loss_G_identity: 1.0309 loss_G_GAN: 0.4
[62/149][43/43] Loss_D: 0.3873 Loss_G: 4.1285 Loss_G_identity: 1.2810 loss_G_GAN: 0.6
[63/149][43/43] Loss_D: 0.3896 Loss_G: 4.8070 Loss_G_identity: 1.4214 loss_G_GAN: 0.5
[64/149][43/43] Loss_D: 0.4603 Loss_G: 3.4466 Loss_G_identity: 0.9811 loss_G_GAN: 0.6
[65/149][43/43] Loss_D: 0.4226 Loss_G: 4.1359 Loss_G_identity: 1.2979 loss_G_GAN: 0.6
[66/149][43/43] Loss_D: 0.4990 Loss_G: 3.7595 Loss_G_identity: 1.0556 loss_G_GAN: 0.5
[67/149][43/43] Loss_D: 0.4370 Loss_G: 3.5491 Loss_G_identity: 1.0771 loss_G_GAN: 0.4
[68/149][43/43] Loss_D: 0.4411 Loss_G: 3.4690 Loss_G_identity: 0.9879 loss_G_GAN: 0.7
[69/149][43/43] Loss_D: 0.4349 Loss_G: 3.4782 Loss_G_identity: 1.1010 loss_G_GAN: 0.5
[70/149][43/43] Loss_D: 0.4837 Loss_G: 3.6511 Loss_G_identity: 1.1124 loss_G_GAN: 0.3
[71/149][43/43] Loss_D: 0.4781 Loss_G: 3.2513 Loss_G_identity: 0.7992 loss_G_GAN: 0.5
[72/149][43/43] Loss_D: 0.4889 Loss_G: 3.9234 Loss_G_identity: 1.0697 loss_G_GAN: 0.7
[73/149][43/43] Loss_D: 0.4085 Loss_G: 3.9012 Loss_G_identity: 1.0013 loss_G_GAN: 0.7
[74/149][43/43] Loss_D: 0.3890 Loss_G: 3.6650 Loss_G_identity: 1.1858 loss_G_GAN: 0.5
[75/149][43/43] Loss_D: 0.4260 Loss_G: 3.5998 Loss_G_identity: 0.9068 loss_G_GAN: 0.7
[76/149][43/43] Loss_D: 0.4781 Loss_G: 3.4985 Loss_G_identity: 1.0085 loss_G_GAN: 0.5
[77/149][43/43] Loss_D: 0.3621 Loss_G: 3.6343 Loss_G_identity: 0.9498 loss_G_GAN: 0.8
[78/149][43/43] Loss_D: 0.3921 Loss_G: 3.2594 Loss_G_identity: 0.8412 loss_G_GAN: 0.5
[79/149][43/43] Loss_D: 0.4977 Loss_G: 3.2360 Loss_G_identity: 1.0604 loss_G_GAN: 0.4
[80/149][43/43] Loss_D: 0.4386 Loss_G: 3.2492 Loss_G_identity: 0.7376 loss_G_GAN: 0.7
[81/149][43/43] Loss_D: 0.3930 Loss_G: 3.0415 Loss_G_identity: 0.7943 loss_G_GAN: 0.5
[82/149][43/43] Loss_D: 0.4431 Loss_G: 4.1876 Loss_G_identity: 1.4447 loss_G_GAN: 0.5
[83/149][43/43] Loss_D: 0.4343 Loss_G: 4.4590 Loss_G_identity: 1.3221 loss_G_GAN: 0.6
[84/149][43/43] Loss_D: 0.4666 Loss_G: 3.6194 Loss_G_identity: 1.1302 loss_G_GAN: 0.6
[85/149][43/43] Loss_D: 0.4230 Loss_G: 3.2900 Loss_G_identity: 0.8490 loss_G_GAN: 0.6
[86/149][43/43] Loss_D: 0.4454 Loss_G: 3.2225 Loss_G_identity: 0.7715 loss_G_GAN: 0.5
[87/149][43/43] Loss_D: 0.5358 Loss_G: 3.6666 Loss_G_identity: 1.0316 loss_G_GAN: 0.6
[88/149][43/43] Loss_D: 0.3651 Loss_G: 3.5815 Loss_G_identity: 0.9082 loss_G_GAN: 0.7
[89/149][43/43] Loss_D: 0.4595 Loss_G: 3.8905 Loss_G_identity: 1.2094 loss_G_GAN: 0.6
[90/149][43/43] Loss_D: 0.4140 Loss_G: 3.4978 Loss_G_identity: 0.8752 loss_G_GAN: 0.7
[91/149][43/43] Loss_D: 0.4578 Loss_G: 3.6478 Loss_G_identity: 0.9614 loss_G_GAN: 0.6
[92/149][43/43] Loss_D: 0.4739 Loss_G: 3.4900 Loss_G_identity: 0.9788 loss_G_GAN: 0.5
[93/149][43/43] Loss_D: 0.4444 Loss_G: 3.2602 Loss_G_identity: 0.9623 loss_G_GAN: 0.5
[94/149][43/43] Loss_D: 0.4373 Loss_G: 3.2711 Loss_G_identity: 0.8936 loss_G_GAN: 0.5
[95/149][43/43] Loss_D: 0.3783 Loss_G: 3.6698 Loss_G_identity: 1.1651 loss_G_GAN: 0.6
[96/149][43/43] Loss_D: 0.3585 Loss_G: 3.6279 Loss_G_identity: 1.2201 loss_G_GAN: 0.5
[97/149][43/43] Loss_D: 0.4253 Loss_G: 3.3328 Loss_G_identity: 0.9539 loss_G_GAN: 0.4
[98/149][43/43] Loss_D: 0.3123 Loss_G: 3.9401 Loss_G_identity: 1.1537 loss_G_GAN: 0.7
[99/149][43/43] Loss_D: 0.4878 Loss_G: 4.1115 Loss_G_identity: 1.3227 loss_G_GAN: 0.7
[100/149][43/43] Loss_D: 0.3893 Loss_G: 3.5209 Loss_G_identity: 0.9424 loss_G_GAN: 0.
[101/149][43/43] Loss_D: 0.2640 Loss_G: 3.8010 Loss_G_identity: 1.2601 loss_G_GAN: 0.
[102/149][43/43] Loss_D: 0.4099 Loss_G: 3.0883 Loss_G_identity: 0.9100 loss_G_GAN: 0.
[103/149][43/43] Loss_D: 0.4025 Loss_G: 3.4205 Loss_G_identity: 0.9993 loss_G_GAN: 0.
[104/149][43/43] Loss_D: 0.4454 Loss_G: 2.9031 Loss_G_identity: 0.7729 loss_G_GAN: 0.
[105/149][43/43] Loss_D: 0.4796 Loss_G: 3.8393 Loss_G_identity: 1.0742 loss_G_GAN: 0.
[106/149][43/43] Loss_D: 0.5441 Loss_G: 3.2684 Loss_G_identity: 0.8017 loss_G_GAN: 0.
[107/149][43/43] Loss_D: 0.5266 Loss_G: 2.7711 Loss_G_identity: 0.6985 loss_G_GAN: 0.
[108/149][43/43] Loss_D: 0.3892 Loss_G: 3.6114 Loss_G_identity: 1.1945 loss_G_GAN: 0.
[109/149][43/43] Loss_D: 0.5433 Loss_G: 3.0715 Loss_G_identity: 0.8403 loss_G_GAN: 0.
[110/149][43/43] Loss_D: 0.4103 Loss_G: 2.9816 Loss_G_identity: 0.8522 loss_G_GAN: 0.
[111/149][43/43] Loss_D: 0.4043 Loss_G: 3.0771 Loss_G_identity: 0.9327 loss_G_GAN: 0.
[112/149][43/43] Loss_D: 0.3269 Loss_G: 2.9158 Loss_G_identity: 0.8269 loss_G_GAN: 0.
[113/149][43/43] Loss_D: 0.4543 Loss_G: 2.8956 Loss_G_identity: 0.8196 loss_G_GAN: 0.
[114/149][43/43] Loss_D: 0.4344 Loss_G: 3.2946 Loss_G_identity: 0.9779 loss_G_GAN: 0.
[115/149][43/43] Loss_D: 0.4877 Loss_G: 3.0123 Loss_G_identity: 0.7945 loss_G_GAN: 0.
[116/149][43/43] Loss_D: 0.3909 Loss_G: 3.6532 Loss_G_identity: 1.1643 loss_G_GAN: 0.
[117/149][43/43] Loss_D: 0.4999 Loss_G: 3.0282 Loss_G_identity: 0.6732 loss_G_GAN: 0.
```