```
import os
import numpy as np
import torch
import torch.nn as nn
from torch.nn.utils import clip_grad_norm_
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from tqdm.notebook import tqdm
from torchvision.transforms.functional import InterpolationMode

import matplotlib.pyplot as plt
from PIL import Image

import shutil


seed = 142

np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.backends.cudnn.deterministic = True

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

```
class ImageDataset(Dataset):
    def __init__(self, data_dir, transforms=None):
        monet_dir = os.path.join(data_dir, 'monet_jpg')
        photo_dir = os.path.join(data_dir, 'photo_jpg')

        self.files_monet = [os.path.join(monet_dir, name) for name in sorted(os.listdir(monet
        self.files_photo = [os.path.join(photo_dir, name) for name in sorted(os.listdir(photo

        self.transforms = transforms

    def __len__(self):
        # we know that len(files_monet) = 300 < 7038 = len(files_photo)
        return len(self.files_monet)

    def __getitem__(self, index):
        # we will use only 300 (=len(files_monet)) photos during training
        # randomly picking them from the first 300 photos
        random_index = np.random.randint(0, len(self.files_monet))
        file_monet = self.files_monet[index]
        file_photo = self.files_photo[random_index]
```

```
        image_monet = Image.open(file_monet)
        image_photo = Image.open(file_photo)

        if self.transforms is not None:
            image_monet = self.transforms(image_monet)
            image_photo = self.transforms(image_photo)

        return image_monet, image_photo

data_dir = '/content/drive/MyDrive/Colab Notebooks/kaggle/data'
batch_size = 5


transforms_ = transforms.Compose([
    #transforms.Resize((256, 256)), # photos already have the same size
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
])

dataloader = DataLoader(
    ImageDataset(data_dir, transforms=transforms_),
    batch_size=batch_size,
    shuffle=True,
    num_workers=2,
)

def unnorm(img, mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]):
    for t, m, s in zip(img, mean, std):
        t.mul_(s).add_(s)

    return img
```

Build CycleGAN

Auxiliary blocks

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=0,
                 transpose=False, use_leaky=True, use_dropout=False, normalize=True):

        super(ConvBlock, self).__init__()
        self.block = []
```

```
            if transpose:
                self.block += [nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
                                                  stride, padding, output_padding=1)]
            else:
                self.block += [nn.Conv2d(in_channels, out_channels, kernel_size,
                                         stride, padding, bias=True)]

            if normalize:
                self.block += [nn.InstanceNorm2d(out_channels)]

            if use_dropout:
                self.block += [nn.Dropout(0.5)]

            if use_leaky:
                self.block += [nn.LeakyReLU(negative_slope=0.2, inplace=True)]
            else:
                self.block += [nn.ReLU(inplace=True)]

            self.block = nn.Sequential(*self.block)


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



    class ResidualBlock(nn.Module):
        def __init__(self, channels):
            super(ResidualBlock, self).__init__()
            self.block = nn.Sequential(
                nn.ReflectionPad2d(1),
                ConvBlock(in_channels=channels, out_channels=channels,
                          kernel_size=3, use_leaky=False, use_dropout=True),
                nn.ReflectionPad2d(1),
                nn.Conv2d(in_channels=channels, out_channels=channels, kernel_size=3),
                nn.InstanceNorm2d(channels)
            )


        def forward(self, x):
            return x + self.block(x)

    class ConvBlock(nn.Module):
        def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1,
                     transpose=False, use_leaky=True, use_dropout=False, normalize=True):

            super(ConvBlock, self).__init__()
            self.block = []

            if transpose:
                self.block += [nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
```

```
                                            stride, padding, output_padding=1)]
        else:
            self.block += [nn.Conv2d(in_channels, out_channels, kernel_size,
                                     stride, padding, bias=True)]

        if normalize:
            self.block += [nn.InstanceNorm2d(out_channels,affine=True)]

        if use_dropout:
            self.block += [nn.Dropout(0.5)]

        if use_leaky:
            self.block += [nn.LeakyReLU(negative_slope=0.2, inplace=True)]
        else:
            self.block += [nn.ReLU(inplace=True)]

        self.block = nn.Sequential(*self.block)


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


class ResidualBlock(nn.Module):
    def __init__(self, channels,kernel_size=3):
        super(ResidualBlock, self).__init__()

        #doesn't change shape of input
        self.block = nn.Sequential(
            nn.ReflectionPad2d(int((kernel_size - 1) / 2)),
            ConvBlock(in_channels=channels, out_channels=channels,padding=0,
                      kernel_size=kernel_size, use_leaky=False, use_dropout=True),
            nn.ReflectionPad2d(int((kernel_size - 1) / 2)),
            nn.Conv2d(in_channels=channels, out_channels=channels, kernel_size=kernel_size,pa
            nn.InstanceNorm2d(channels,affine=True)
        )


    def forward(self, x):
        return x + self.block(x)


class transformer_(nn.Module):

    def __init__(self,transformer_kernel_size=9):
        super(transformer_, self).__init__()
        self.model=nn.AvgPool2d(transformer_kernel_size, stride=1,padding=transformer_kernel_

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

Define Generator

```
class Generator(nn.Module):
    def __init__(self, in_channels, out_channels, num_residual_blocks=9):
        super(Generator, self).__init__()

        ''' Encoder '''
        # Inital layer:  3*256*256 -> 64*256*256
        self.initial = [
            nn.ReflectionPad2d(in_channels),
            ConvBlock(in_channels=in_channels, out_channels=64,
                      kernel_size=2*in_channels+1, use_leaky=False),
        ]
        self.initial = nn.Sequential(*self.initial)

        # Downsampling:  64*256*256 -> 128*128*128 -> 256*64*64
        self.down = [
            ConvBlock(in_channels=64, out_channels=128, kernel_size=3,
                      stride=2, padding=1, use_leaky=False),
            ConvBlock(in_channels=128, out_channels=256, kernel_size=3,
                      stride=2, padding=1, use_leaky=False),
        ]
        self.down = nn.Sequential(*self.down)


        """ Transformer """
        # ResNet:  256*64*64 -> 256*64*64
        self.transform = [ResidualBlock(256) for _ in range(num_residual_blocks)]
        self.transform = nn.Sequential(*self.transform)


        """ Decoder """
        # Upsampling:  256*64*64 -> 128*128*128 -> 64*256*256
        self.up = [
            ConvBlock(in_channels=256, out_channels=128, kernel_size=3, stride=2,
                      padding=1, transpose=True, use_leaky=False),
            ConvBlock(in_channels=128, out_channels=64, kernel_size=3, stride=2,
                      padding=1, transpose=True, use_leaky=False),
        ]
        self.up = nn.Sequential(*self.up)

        # Out layer:  64*256*256 -> 3*256*256
        self.out = nn.Sequential(
            nn.ReflectionPad2d(out_channels),
```

```
            nn.Conv2d(in_channels=64, out_channels=out_channels, kernel_size=2*out_channels+1
            nn.Tanh()
        )


    def forward(self, x):
        x = self.down(self.initial(x))
        x = self.transform(x)
        x = self.out(self.up(x))
        return x


class encoder_(nn.Module):

    def __init__(self,in_channels=3,n_filter_generator=32,pad_type=0):
        super(encoder_, self).__init__()

        self.model=nn.Sequential(

            nn.InstanceNorm2d(in_channels,affine=True),

            #3*256*256 - 3*286*286
            nn.ReflectionPad2d(15),

            #3*286*286 - 32*284*284
            ConvBlock(in_channels=in_channels, out_channels=n_filter_generator, kernel_size=3
                      stride=1, padding=pad_type, normalize=True,use_leaky=False),
            #32*284*284 - 32*141*141
            ConvBlock(in_channels=n_filter_generator, out_channels=n_filter_generator, kernel
                      stride=2, padding=pad_type, normalize=True,use_leaky=False),
            #32*141*141 - 64*70*70
            ConvBlock(in_channels=n_filter_generator, out_channels=n_filter_generator*2, kern
                      stride=2, padding=pad_type, normalize=True,use_leaky=False),
            #64*70*70 - 128*34*34
            ConvBlock(in_channels=n_filter_generator*2, out_channels=n_filter_generator*4, ke
                      stride=2, padding=pad_type, normalize=True,use_leaky=False),
            #128*34*34 - 256*32*32
            ConvBlock(in_channels=n_filter_generator*4, out_channels=n_filter_generator*8, ke
                      stride=1, padding=pad_type, normalize=True,use_leaky=False)
        )

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

class decoder_(nn.Module):

    """decoder model following https://arxiv.org/pdf/1807.10201.pdf
    Returns: decoder model
    """
    def __init__(self, input_shape,n_filter_generator=32):
        super(decoder_, self).__init__()
```

```
        num_kernels = n_filter_generator * 8
        self.model=[]
        for i in range(9):
            self.model.append(ResidualBlock(num_kernels))


        #i=0: 256*32*32 - 256*32*32 - 256*32*32
        #i=1: 256*32*32 - 256*64*64 - 128*64*64
        #i=2: 128*64*64 - 128*128*128 - 64*128*128
        #i=3: 64*128*128 - 64*256*256 - 32*256*256
        in_channels=[num_kernels,8*n_filter_generator,4*n_filter_generator,2*n_filter_generat
        for i in range(4):
            self.model.append(transforms.Resize(size=(input_shape[2]*2**(i),input_shape[3]*2*
            self.model.append(nn.Conv2d(in_channels=in_channels[i], out_channels=n_filter_gen
                                        stride=1, padding=1, bias=True))
            self.model.append(nn.InstanceNorm2d(n_filter_generator * 2 ** (3 - i),affine=True
            self.model.append(nn.ReLU(inplace=True))

        self.model.append(nn.ReflectionPad2d(3))

        #32*256*256 - 3*256*256
        self.model.append(nn.Conv2d(in_channels=n_filter_generator, out_channels=3, kernel_si
                                    stride=1, padding=0, bias=True))
        self.model.append(nn.Tanh())
        self.model = nn.Sequential(*self.model)
        #x = CenterLayer()(x)

    def forward(self,x):
        return self.model(x)#*2-1
```

Define Discriminator

```
class discriminator_(nn.Module):
    def __init__(self, in_channels=3,pad_type='valid',n_filter_discriminator=64):#ASA code us
        super(discriminator_, self).__init__()

        #3*256*256 - 128*128*128 -> 1*128*128
        self.conv0=ConvBlock(in_channels=in_channels, out_channels=n_filter_discriminator*2,
                     stride=2, padding=2, normalize=True)
        self.conv0_pred=nn.Conv2d(in_channels=n_filter_discriminator*2, out_channels=1, kerne
                                    stride=1, padding=2, bias=True)

        #128*128*128 - 128*64*64 -> 1*65*65
        self.conv1=ConvBlock(in_channels=n_filter_discriminator*2, out_channels=n_filter_disc
                     stride=2, padding=2, normalize=True)
        self.conv1_pred=nn.Conv2d(in_channels=n_filter_discriminator*2, out_channels=1, kerne
                                    stride=1, padding=5, bias=True)
```

```
        #128*64*64 - 256*32*32
        self.conv2=ConvBlock(in_channels=n_filter_discriminator*2, out_channels=n_filter_disc
                     stride=2, padding=2, normalize=True)

        #256*32*32 - 512*16*16 -> 1*17*17
        self.conv3=ConvBlock(in_channels=n_filter_discriminator*4, out_channels=n_filter_disc
                     stride=2, padding=2, normalize=True)
        self.conv3_pred=nn.Conv2d(in_channels=n_filter_discriminator*8, out_channels=1, kerne
                                  stride=1, padding=5, bias=True)

        #512*16*16 - 512*8*8
        self.conv4=ConvBlock(in_channels=n_filter_discriminator*8, out_channels=n_filter_disc
                     stride=2, padding=2, normalize=True)

        #512*4*4 - 1024*4*4 -> 1*5*5
        self.conv5=ConvBlock(in_channels=n_filter_discriminator*8, out_channels=n_filter_disc
                     stride=2, padding=2, normalize=True)
        self.conv5_pred=nn.Conv2d(in_channels=n_filter_discriminator*16, out_channels=1, kern
                                  stride=1, padding=3, bias=True)

        #1024*1*1 - 1024*2*2 -> 1*2*2
        self.conv6=ConvBlock(in_channels=n_filter_discriminator*16, out_channels=n_filter_dis
                     stride=2, padding=0, normalize=True)
        self.conv6_pred=nn.Conv2d(in_channels=n_filter_discriminator*16, out_channels=1, kern
                                  stride=1, padding=1, bias=True)



    def forward(self, x):

        h0=self.conv0(x)
        h0_pred=self.conv0_pred(h0)

        h1=self.conv1(h0)
        h1_pred=self.conv1_pred(h1)

        h2=self.conv2(h1)

        h3=self.conv3(h2)
        h3_pred=self.conv3_pred(h3)

        h4=self.conv4(h3)

        h5=self.conv5(h4)
        h5_pred=self.conv5_pred(h5)

        h6=self.conv6(h5)
        h6_pred=self.conv6_pred(h6)

        #print((h0.size(),h1.size(),h2.size(),h3.size(),h4.size(),h5.size(),h6.size()))
```

```
        #print((h0_pred.size(),h1_pred.size(),h3_pred.size(),h5_pred.size(),h6_pred.size()))
        return [h0_pred, h1_pred, h3_pred, h5_pred, h6_pred]



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

        self.model = nn.Sequential(
            # 3*256*256 -> 64*128*128
            ConvBlock(in_channels=in_channels, out_channels=64, kernel_size=4,
                      stride=2, padding=1, normalize=False),

            # 64*128*128 -> 128*64*64
            ConvBlock(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1),

            # 128*64*64 -> 256*32*32
            ConvBlock(in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=1),

            # 256*32*32 -> 512*31*31
            ConvBlock(in_channels=256, out_channels=512, kernel_size=4, stride=1, padding=1),

            # 512*31*31 -> 1*30*30
            nn.Conv2d(in_channels=512, out_channels=1, kernel_size=4, stride=1, padding=1),
        )


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

Training tools

Model initialization

```
generator_monet2photo = Generator(in_channels=3, out_channels=3, num_residual_blocks=9).to(de
generator_photo2monet = Generator(in_channels=3, out_channels=3, num_residual_blocks=9).to(de

discriminator_monet = Discriminator(in_channels=3).to(device)
discriminator_photo = Discriminator(in_channels=3).to(device)


generator_monet2photo = nn.Sequential(encoder_(),decoder_(input_shape=(batch_size,256,32,32))
generator_photo2monet = nn.Sequential(encoder_(),decoder_(input_shape=(batch_size,256,32,32))
```

```
discriminator_monet = Discriminator(in_channels=3).to(device)
```

Define Losses

```
criterion_GAN = nn.MSELoss()
criterion_cycle = nn.L1Loss()
criterion_identity = nn.L1Loss()
```

Define Optimizers

```
lr = 2e-4
b1 = 0.5
b2 = 0.999

optim_generators = torch.optim.Adam(
    list(generator_monet2photo.parameters()) + list(generator_photo2monet.parameters()),
    lr=lr, betas=(b1, b2)
)

optim_discriminators = torch.optim.Adam(
    list(discriminator_monet.parameters()) + list(discriminator_photo.parameters()),
    lr=lr, betas=(b1, b2)
)
```

Define Learning rate schedulers

```
num_epochs = 80
decay_epoch = 25

lr_sched_step = lambda epoch: 1 - max(0, epoch - decay_epoch) / (num_epochs - decay_epoch)

lr_sched_generators = torch.optim.lr_scheduler.LambdaLR(optim_generators, lr_lambda=lr_sched_
lr_sched_discriminators = torch.optim.lr_scheduler.LambdaLR(optim_discriminators, lr_lambda=l
```

Auxiliary tools

```
class History:
    def __init__(self):
        self.generators_loss = []
        self.discriminators_loss = []
```

```
        def update(self, gen_loss, discr_loss):
            self.generators_loss.append(gen_loss)
            self.discriminators_loss.append(discr_loss)

        def show(self, title='Losses'):
            fig = plt.figure(figsize=(20, 8))
            plt.title(title)
            plt.plot(self.generators_loss, 'o-', color='r',
                     linewidth=2, markersize=3, label='Generators Loss')
            plt.plot(self.discriminators_loss, 'o-', color='b',
                     linewidth=2, markersize=3, label='Discriminators Loss')
            plt.legend(loc='best')
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.grid(True)
            plt.show()

    class Buffer:
        def __init__(self, max_images=50):
            self.max_images = max_images
            self.images = []

        def update(self, images):
            images = images.detach().cpu().data.numpy()
            for image in images:
                if len(self.images) < self.max_images:
                    self.images.append(image)
                else:
                    if np.random.rand() > 0.5:
                        index = np.random.randint(0, self.max_images)
                        self.images[index] = image

        def sample(self, num_images):
            samples = np.array([self.images[np.random.randint(0, len(self.images))]
                                for _ in range(num_images)])
            return torch.tensor(samples)
    def update_req_grad(models, requires_grad=True):
        for model in models:
            for param in model.parameters():
                param.requires_grad = requires_grad
```

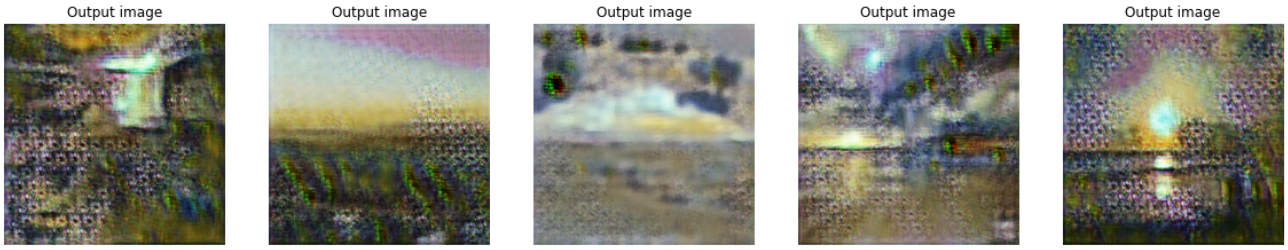
Training CycleGAN

```
    history = History()
    buffer_monet = Buffer()
    buffer_photo = Buffer()

    for epoch in range(num_epochs):
```

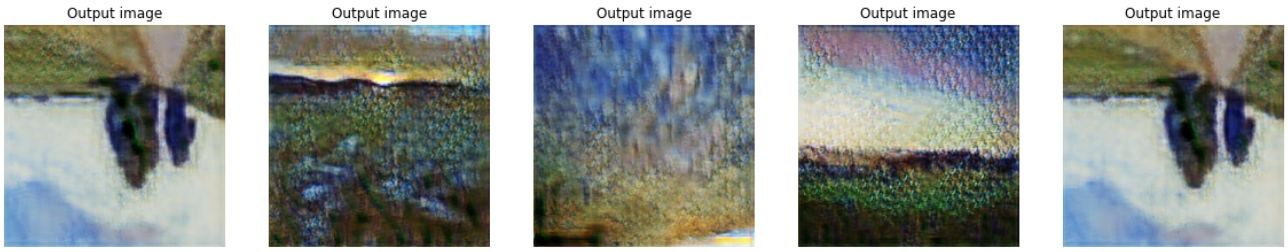
```
    avg_generators_loss = 0
    avg_discriminators_loss = 0

    for i, (real_monet, real_photo) in enumerate(tqdm(dataloader, leave=False, total=len(data
        real_monet, real_photo = real_monet.to(device), real_photo.to(device)

        """ Train Generators """
        # switching models parameters so that only generators are trained
        update_req_grad([generator_monet2photo, generator_photo2monet], True)
        update_req_grad([discriminator_monet, discriminator_photo], False)

        # zero the parameters gradients
        optim_generators.zero_grad()

        # forward-pass
        fake_photo = generator_monet2photo(real_monet)
        fake_monet = generator_photo2monet(real_photo)

        cycle_photo = generator_monet2photo(fake_monet)
        cycle_monet = generator_photo2monet(fake_photo)

        identity_photo = generator_monet2photo(real_photo)
        identity_monet = generator_photo2monet(real_monet)

        # update photos that are used to feed up discriminators
        buffer_photo.update(fake_photo)
        buffer_monet.update(fake_monet)

        # discriminators outputs that are used in adversarial loss
        discriminator_outputs_photo = discriminator_photo(fake_photo)
        discriminator_outputs_monet = discriminator_monet(fake_monet)

        # labels that are used as ground truth
        labels_real = torch.ones(discriminator_outputs_monet.size()).to(device)
        labels_fake = torch.zeros(discriminator_outputs_monet.size()).to(device)

        # adversarial loss - enforces that the generated output be of the appropriate domain
        loss_GAN_monet2photo = criterion_GAN(discriminator_outputs_photo, labels_real)
        loss_GAN_photo2monet = criterion_GAN(discriminator_outputs_monet, labels_real)
        loss_GAN = (loss_GAN_monet2photo + loss_GAN_photo2monet) / 2

        # cycle consistency loss - enforces that the input and output are recognizably the sa
        loss_cycle_photo = criterion_cycle(cycle_photo, real_photo)
        loss_cycle_monet = criterion_cycle(cycle_monet, real_monet)
        loss_cycle = (loss_cycle_photo + loss_cycle_monet) / 2

        # identity mapping loss - helps preserve the color of the input images
        loss_identity_photo = criterion_identity(identity_photo, real_photo)
        loss_identity_monet = criterion_identity(identity_monet, real_monet)
        loss_identity = (loss_identity_photo + loss_identity_monet) / 2

        # total loss
```

```
        loss_generators_total = loss_GAN + 10 * loss_cycle + 5 * loss_identity

        # backward-pass
        loss_generators_total.backward()
        optim_generators.step()

        # limiting gradient norms - if they exceed 100, something went wrong
        clip_grad_norm_(generator_photo2monet.parameters(), 100)
        clip_grad_norm_(generator_monet2photo.parameters(), 100)


        """ Train Discriminators """
        # switching models parameters so that only discriminators are trained
        update_req_grad([discriminator_monet, discriminator_photo], True)
        update_req_grad([generator_monet2photo, generator_photo2monet], False)

        # zero the parameters gradients
        optim_discriminators.zero_grad()

        # sample images from 50 stored
        fake_photo = buffer_photo.sample(num_images=batch_size).to(device)
        fake_monet = buffer_monet.sample(num_images=batch_size).to(device)

        # making labels noisy for discriminators so that they don't prevail over generators
        threshold = min(1, 0.85 + (1 - 0.85) * epoch / (num_epochs // 2))
        noisy_labels_real = (torch.rand(discriminator_outputs_monet.size()) < threshold).floa

        # forward-pass + losses
        loss_real_photo = criterion_GAN(discriminator_photo(real_photo), noisy_labels_real)
        loss_fake_photo = criterion_GAN(discriminator_photo(fake_photo.detach()), labels_fake
        loss_photo = (loss_real_photo + loss_fake_photo) / 2

        loss_real_monet = criterion_GAN(discriminator_monet(real_monet), noisy_labels_real)
        loss_fake_monet = criterion_GAN(discriminator_monet(fake_monet.detach()), labels_fake
        loss_monet = (loss_real_monet + loss_fake_monet) / 2

        loss_discriminators_total = loss_monet + loss_photo

        # backward-pass
        loss_discriminators_total.backward()
        optim_discriminators.step()

        # clipping gradients to avoid gradients explosion
        clip_grad_norm_(discriminator_monet.parameters(), 100)
        clip_grad_norm_(discriminator_photo.parameters(), 100)

        # updating intermediate results
        avg_generators_loss += loss_generators_total.item()
        avg_discriminators_loss += loss_discriminators_total.item()

    # saving intermediate results
    avg_generators_loss /= len(dataloader)
```

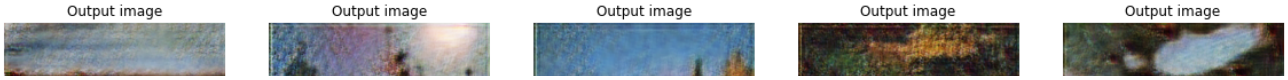
```
        avg_generators_loss /= len(dataloader)
        avg_discriminators_loss /= len(dataloader)
        history.update(avg_generators_loss, avg_discriminators_loss)

        # showing intermediate results
        print("Epoch: %d/%d | Generators Loss: %.4f | Discriminators Loss: %.4f"
                  % (epoch+1, num_epochs, avg_generators_loss, avg_discriminators_loss))

        # showing generated images
        if (epoch + 1) % 10 == 0:
            _, sample_real_photo = next(iter(dataloader))

            sample_fake_monet = generator_photo2monet(sample_real_photo.to(device)).detach().cpu(

            num_photos = min(batch_size, 5)
            plt.figure(figsize=(20, 8))
            for k in range(num_photos):
                plt.subplot(2, num_photos, k + 1)
                plt.imshow(unnorm(sample_real_photo[k]).permute(1, 2, 0))
                plt.title('Input photo')
                plt.axis('off')

                plt.subplot(2, num_photos, k + num_photos + 1)
                plt.imshow(unnorm(sample_fake_monet[k]).permute(1, 2, 0))
                plt.title('Output image')
                plt.axis('off')
            plt.show()

        lr_sched_generators.step()
        lr_sched_discriminators.step()
```



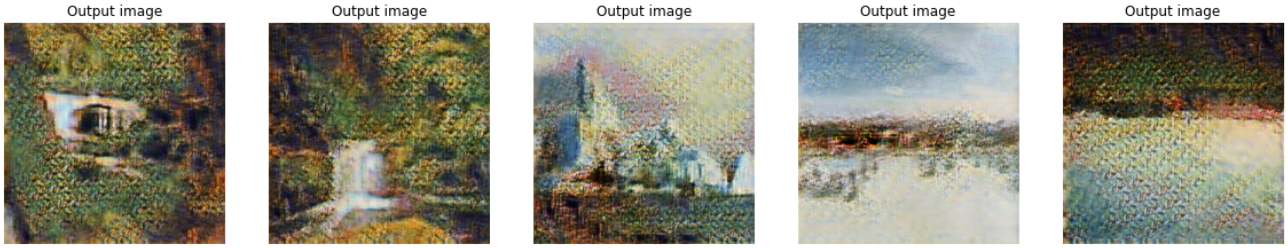

```
Epoch: 21/80 | Generators Loss: 3.5662 | Discriminators Loss: 0.2668
Epoch: 22/80 | Generators Loss: 3.5757 | Discriminators Loss: 0.2145
Epoch: 23/80 | Generators Loss: 3.4235 | Discriminators Loss: 0.2079
Epoch: 24/80 | Generators Loss: 3.5432 | Discriminators Loss: 0.2116
Epoch: 25/80 | Generators Loss: 3.5553 | Discriminators Loss: 0.2033
Epoch: 26/80 | Generators Loss: 3.4547 | Discriminators Loss: 0.2041
Epoch: 27/80 | Generators Loss: 3.4947 | Discriminators Loss: 0.1875
Epoch: 28/80 | Generators Loss: 3.4394 | Discriminators Loss: 0.1866
Epoch: 29/80 | Generators Loss: 3.4496 | Discriminators Loss: 0.1889
Epoch: 30/80 | Generators Loss: 3.3074 | Discriminators Loss: 0.1774
```



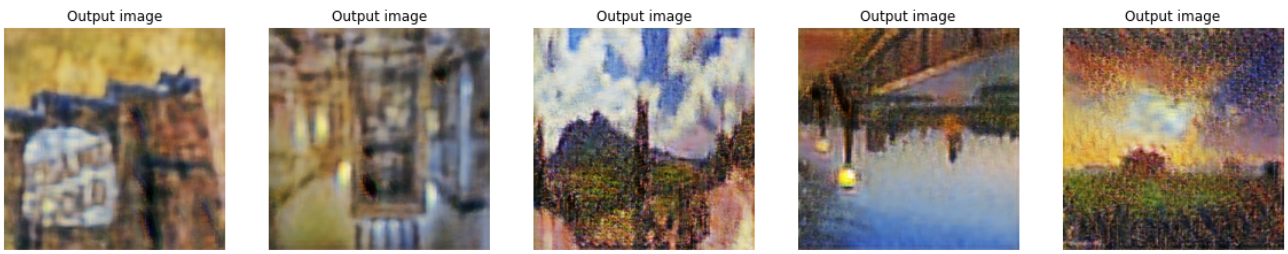

```
Epoch: 31/80 | Generators Loss: 3.4079 | Discriminators Loss: 0.1644
Epoch: 32/80 | Generators Loss: 3.2475 | Discriminators Loss: 0.1778
Epoch: 33/80 | Generators Loss: 3.2962 | Discriminators Loss: 0.1484
Epoch: 34/80 | Generators Loss: 3.2317 | Discriminators Loss: 0.1593
Epoch: 35/80 | Generators Loss: 3.3812 | Discriminators Loss: 0.6244
Epoch: 36/80 | Generators Loss: 3.1057 | Discriminators Loss: 0.3031
Epoch: 37/80 | Generators Loss: 3.1574 | Discriminators Loss: 0.2748
Epoch: 38/80 | Generators Loss: 3.1272 | Discriminators Loss: 0.2352
Epoch: 39/80 | Generators Loss: 3.1643 | Discriminators Loss: 0.2233
Epoch: 40/80 | Generators Loss: 3.0134 | Discriminators Loss: 0.1783
```



```
Epoch: 41/80 | Generators Loss: 3.0460 | Discriminators Loss: 0.1618
Epoch: 42/80 | Generators Loss: 3.0042 | Discriminators Loss: 0.1583
Epoch: 43/80 | Generators Loss: 3.0283 | Discriminators Loss: 0.1632
Epoch: 44/80 | Generators Loss: 2.9768 | Discriminators Loss: 0.1493
Epoch: 45/80 | Generators Loss: 2.9740 | Discriminators Loss: 0.1524
Epoch: 46/80 | Generators Loss: 2.9119 | Discriminators Loss: 0.1491
Epoch: 47/80 | Generators Loss: 2.9590 | Discriminators Loss: 0.1415
Epoch: 48/80 | Generators Loss: 2.9422 | Discriminators Loss: 0.1486
Epoch: 49/80 | Generators Loss: 2.9141 | Discriminators Loss: 0.1401
Epoch: 50/80 | Generators Loss: 2.8690 | Discriminators Loss: 0.1327
```

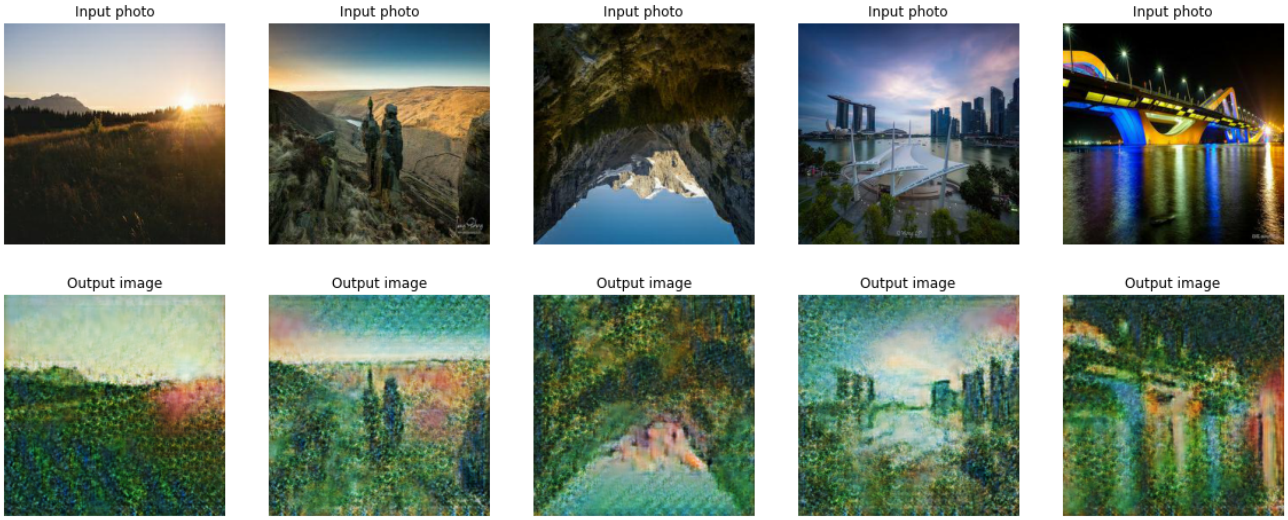



```
Epoch: 51/80 | Generators Loss: 2.9565 | Discriminators Loss: 0.1325
Epoch: 52/80 | Generators Loss: 2.8789 | Discriminators Loss: 0.1458
Epoch: 53/80 | Generators Loss: 2.8637 | Discriminators Loss: 0.1298
Epoch: 54/80 | Generators Loss: 2.8644 | Discriminators Loss: 0.1362
Epoch: 55/80 | Generators Loss: 2.8138 | Discriminators Loss: 0.1363
Epoch: 56/80 | Generators Loss: 2.7655 | Discriminators Loss: 0.1280
Epoch: 57/80 | Generators Loss: 2.8331 | Discriminators Loss: 0.1274
Epoch: 58/80 | Generators Loss: 2.7315 | Discriminators Loss: 0.1297
Epoch: 59/80 | Generators Loss: 2.8161 | Discriminators Loss: 0.1288
Epoch: 60/80 | Generators Loss: 2.7957 | Discriminators Loss: 0.1216
```




```
Epoch: 61/80 | Generators Loss: 2.7717 | Discriminators Loss: 0.1193
```

```
Epoch: 62/80 | Generators Loss: 2.7705 | Discriminators Loss: 0.1208
Epoch: 63/80 | Generators Loss: 2.7081 | Discriminators Loss: 0.1301
Epoch: 64/80 | Generators Loss: 2.7171 | Discriminators Loss: 0.1222
Epoch: 65/80 | Generators Loss: 2.6859 | Discriminators Loss: 0.1214
Epoch: 66/80 | Generators Loss: 2.6967 | Discriminators Loss: 0.1247
Epoch: 67/80 | Generators Loss: 2.6797 | Discriminators Loss: 0.1143
Epoch: 68/80 | Generators Loss: 2.7060 | Discriminators Loss: 0.1239
Epoch: 69/80 | Generators Loss: 2.6109 | Discriminators Loss: 0.1273
Epoch: 70/80 | Generators Loss: 2.6857 | Discriminators Loss: 0.1276
```




```
Epoch: 71/80 | Generators Loss: 2.6193 | Discriminators Loss: 0.1233
Epoch: 72/80 | Generators Loss: 2.6080 | Discriminators Loss: 0.1272
Epoch: 73/80 | Generators Loss: 2.5859 | Discriminators Loss: 0.1219
Epoch: 74/80 | Generators Loss: 2.7110 | Discriminators Loss: 0.1295
Epoch: 75/80 | Generators Loss: 2.6496 | Discriminators Loss: 0.1295
Epoch: 76/80 | Generators Loss: 2.6237 | Discriminators Loss: 0.1361
Epoch: 77/80 | Generators Loss: 2.6987 | Discriminators Loss: 0.1340
Epoch: 78/80 | Generators Loss: 2.6473 | Discriminators Loss: 0.1263
Epoch: 79/80 | Generators Loss: 2.7237 | Discriminators Loss: 0.1465
Epoch: 80/80 | Generators Loss: 2.7680 | Discriminators Loss: 0.1526
```