In [85]:
```
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from torch.utils.data import Dataset, DataLoader, random_split
from PIL import Image
import torchvision
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
import torch.nn as nn
from PIL import Image
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torch.optim as optim
from tqdm import tqdm

manualSeed = 999
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
torch.use_deterministic_algorithms(True)
```

```
Random Seed:  999
```

Importing the data from the folders to get a look at what we are working with, we will be using the "photos" below to make them look more like the "monet" images

In [87]:
```
dataroot1 = "/Users/evelynhaskins/Downloads/gan-getting-started-monet"
```

In [88]:
```
workers = 4
batch_size = 64
image_size = 256
nc = 3
nz = 100
ngf = 128
ndf = 128
num_epochs = 20
lr = 0.0001
beta1 = 0.5
ngpu = 1
```

```
print(os.listdir(dataroot1))
```

['.DS_Store', 'monet_jpg']

Here are the "monet" images

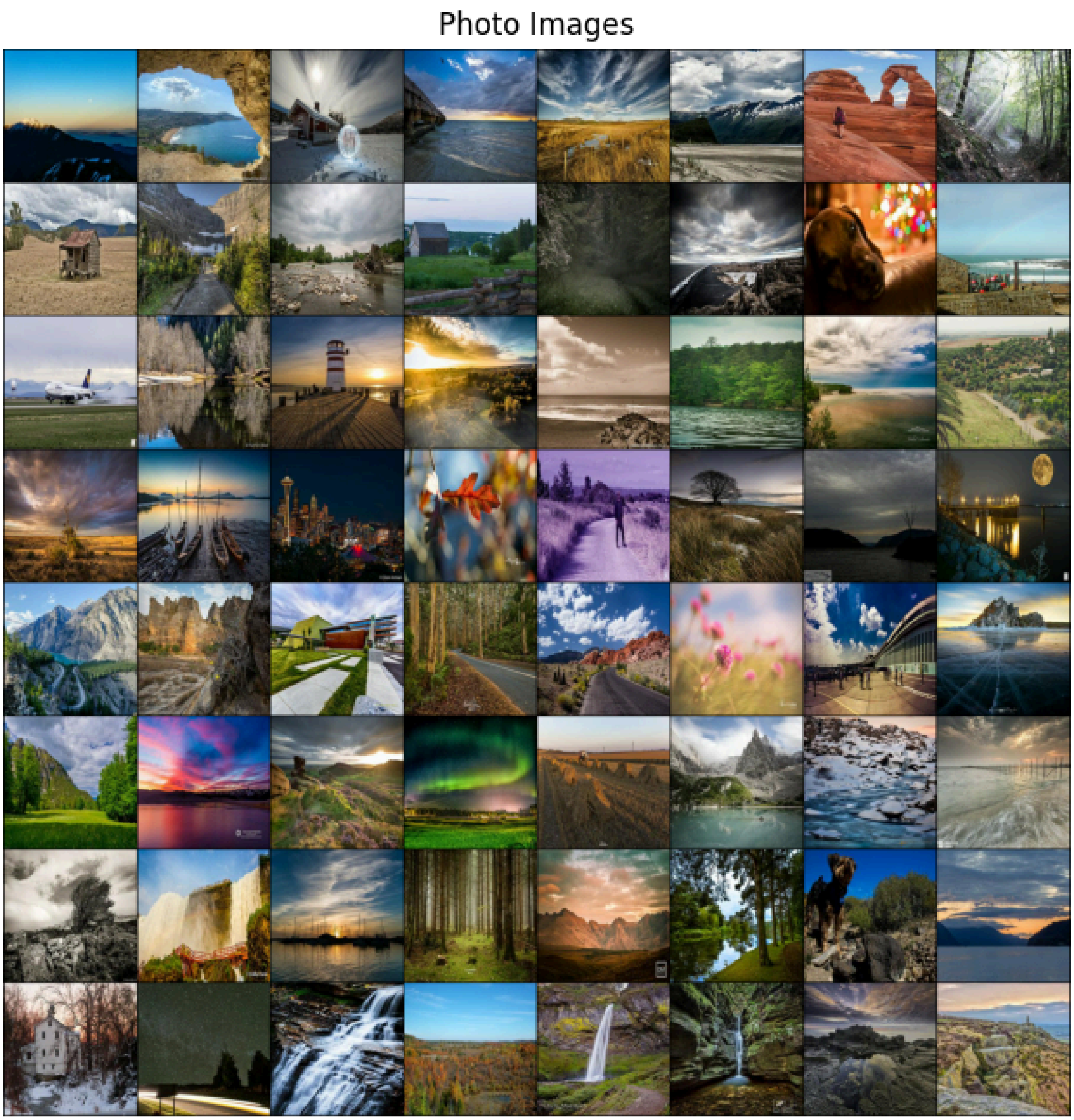
In [89]:
```
dataset_monet = dset.ImageFolder(root=dataroot1,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0
                           ]))

dataloader = torch.utils.data.DataLoader(dataset_monet, batch_size=batch_siz
                                         shuffle=True, num_workers=workers)


device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) e


real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Monet Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padd
plt.show()
```

Here are the "photo" images

In [90]:
```
dataroot2 = "/Users/evelynhaskins/Downloads/gan-getting-started-photo"
```

In [91]:
```
dataset_photos = dset.ImageFolder(root=dataroot2,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0
                           ]))

dataloader = torch.utils.data.DataLoader(dataset_photos, batch_size=batch_si
                                         shuffle=True, num_workers=workers)


device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) e
```

```
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Photo Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padd
plt.show()
```



Here is where we start creating the model, starting with loading those specific photos and monets

In [92]:
```
photo_path = '/Users/evelynhaskins/Downloads/gan-getting-started/photo_jpg/'
monet_path = '/Users/evelynhaskins/Downloads/gan-getting-started/monet_jpg/'
```

In [93]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
```

```
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
```

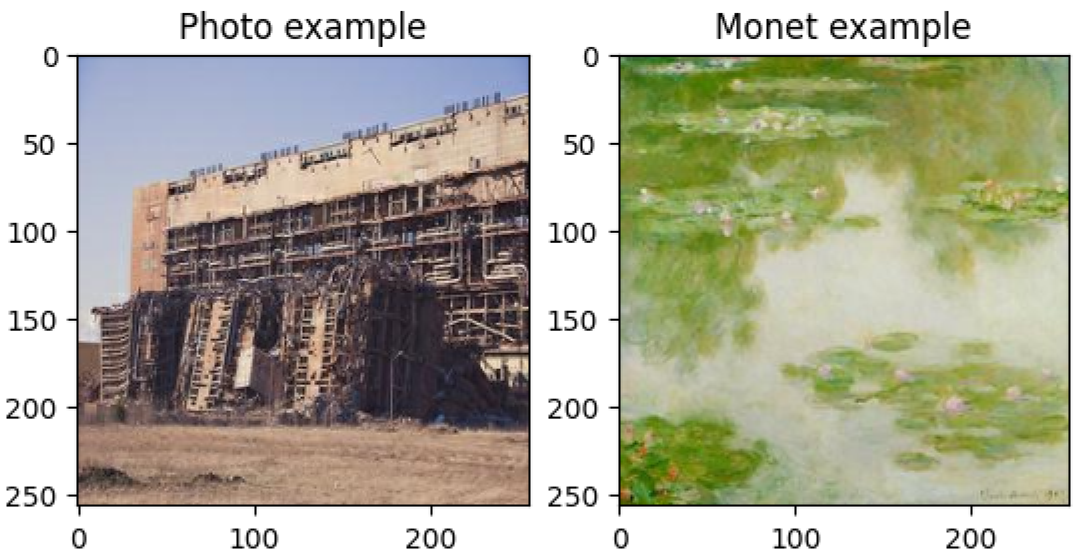
In [94]:
```
from PIL import Image
import os
from torch.utils.data import Dataset
import numpy as np

class Images(Dataset):
    def __init__(self, photo_path, monet_path, transform):
        self.photo_path = photo_path
        self.monet_path = monet_path
        self.transform = transform

        self.photos = os.listdir(photo_path)
        self.monets = os.listdir(monet_path)
        self.l_photo = len(self.photos)
        self.l_monet = len(self.monets)

    def __len__(self):
        return max(len(self.photos), len(self.monets))

    def __getitem__(self, idx):
        photo = Image.open(self.photo_path + self.photos[idx % self.l_photo]
        monet = Image.open(self.monet_path + self.monets[idx % self.l_monet]

        photo = self.transform(photo)
        monet = self.transform(monet)

        return photo, monet
```

In [95]:
```
dataset = Images(photo_path, monet_path, transform)
```

In [96]:
```
dataloader = DataLoader(dataset, batch_size=8, shuffle=True)
```

In [97]:
```
example = next(iter(dataloader))

plt.subplot(1, 2, 1)
plt.title('Photo example')
plt.imshow(example[0][0].permute(1, 2, 0) * 0.5 + 0.5)

plt.subplot(1, 2, 2)
plt.title('Monet example')
plt.imshow(example[1][0].permute(1, 2, 0) * 0.5 + 0.5)
```

Out[97]: <matplotlib.image.AxesImage at 0x1288cdd50>



Here we are creating the discriminator

In [98]:
```
import torch
import torch.nn as nn


class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                4,
                stride,
                1,
                bias=True,
                padding_mode="reflect",
            ),
            nn.InstanceNorm2d(out_channels),
            nn.LeakyReLU(0.2, inplace=True),
        )

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


class Discriminator(nn.Module):
    def __init__(self, in_channels=3, features=[64, 128, 256, 512]):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(
                in_channels,
                features[0],
                kernel_size=4,
                stride=2,
                padding=1,
```

```
                padding_mode="reflect",
            ),
            nn.LeakyReLU(0.2, inplace=True),
        )

        layers = []
        in_channels = features[0]
        for feature in features[1:]:
            layers.append(
                Block(in_channels, feature, stride=1 if feature == features[
            )
            in_channels = feature
        layers.append(
            nn.Conv2d(
                in_channels,
                1,
                kernel_size=4,
                stride=1,
                padding=1,
                padding_mode="reflect",
            )
        )
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = self.initial(x)
        return torch.sigmoid(self.model(x))


def test():
    x = torch.randn((5, 3, 256, 256))
    model = Discriminator(in_channels=3)
    preds = model(x)
    print(preds.shape)


if __name__ == "__main__":
    test()
```

```
torch.Size([5, 1, 30, 30])
```

Creating the generator....

In [99]:
```
import torch
import torch.nn as nn


class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down=True, use_act=True, *
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, padding_mode="reflect", **k
            if down
            else nn.ConvTranspose2d(in_channels, out_channels, **kwargs),
            nn.InstanceNorm2d(out_channels),
```

```
            nn.ReLU(inplace=True) if use_act else nn.Identity(),
        )

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


class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.block = nn.Sequential(
            ConvBlock(channels, channels, kernel_size=3, padding=1),
            ConvBlock(channels, channels, use_act=False, kernel_size=3, padd
        )

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


class Generator(nn.Module):
    def __init__(self, img_channels, num_features=64, num_residuals=9):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(
                img_channels,
                num_features,
                kernel_size=7,
                stride=1,
                padding=3,
                padding_mode="reflect",
            ),
            nn.InstanceNorm2d(num_features),
            nn.ReLU(inplace=True),
        )
        self.down_blocks = nn.ModuleList(
            [
                ConvBlock(
                    num_features, num_features * 2, kernel_size=3, stride=2,
                ),
                ConvBlock(
                    num_features * 2,
                    num_features * 4,
                    kernel_size=3,
                    stride=2,
                    padding=1,
                ),
            ]
        )
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(num_features * 4) for _ in range(num_residuals)]
        )
        self.up_blocks = nn.ModuleList(
            [
                ConvBlock(
                    num_features * 4,
                    num_features * 2,
```

```
                    down=False,
                    kernel_size=3,
                    stride=2,
                    padding=1,
                    output_padding=1,
                ),
                ConvBlock(
                    num_features * 2,
                    num_features * 1,
                    down=False,
                    kernel_size=3,
                    stride=2,
                    padding=1,
                    output_padding=1,
                ),
            ]
        )

        self.last = nn.Conv2d(
            num_features * 1,
            img_channels,
            kernel_size=7,
            stride=1,
            padding=3,
            padding_mode="reflect",
        )

    def forward(self, x):
        x = self.initial(x)
        for layer in self.down_blocks:
            x = layer(x)
        x = self.res_blocks(x)
        for layer in self.up_blocks:
            x = layer(x)
        return torch.tanh(self.last(x))


def test():
    img_channels = 3
    img_size = 256
    x = torch.randn((2, img_channels, img_size, img_size))
    gen = Generator(img_channels, 9)
    print(gen(x).shape)


if __name__ == "__main__":
    test()
```

```
torch.Size([2, 3, 256, 256])
```

Training the data

In [106…

```
device = "cuda" if torch.cuda.is_available() else ("mps" if torch.backends.m
print(device)

lr = 2e-4
```

```
lambda_cycle = 10
img_channels = 3

disc_photo = Discriminator().to(device)
disc_monet = Discriminator().to(device)

gen_photo = Generator(img_channels=img_channels).to(device)
gen_monet = Generator(img_channels=img_channels).to(device)

disc_optimizer = optim.Adam(
    list(disc_photo.parameters()) + list(disc_monet.parameters()),
    lr=lr,
    betas=(0.5, 0.999)
)

gen_optimizer = optim.Adam(
    list(gen_photo.parameters()) + list(gen_monet.parameters()),
    lr=lr,
    betas=(0.5, 0.999)
)

dis_scaler = GradScaler()
gen_scaler = GradScaler()

# Loss functions
MSE = nn.MSELoss()
L1 = nn.L1Loss()

def test():
    img_size = 256
    x = torch.randn((2, img_channels, img_size, img_size)).to(device)
    gen = Generator(img_channels=img_channels).to(device)
    print(gen(x).shape)

if __name__ == "__main__":
    test()
```

```
mps
/var/folders/xh/ljc5kpqs6959s_908qkvm5kc0000gn/T/ipykernel_93429/2686987222.
py:29: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is deprecated. Pl
ease use `torch.amp.GradScaler('cuda', args...)` instead.
  dis_scaler = GradScaler()  # Use without specifying 'cuda'
/var/folders/xh/ljc5kpqs6959s_908qkvm5kc0000gn/T/ipykernel_93429/2686987222.
py:30: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is deprecated. Pl
ease use `torch.amp.GradScaler('cuda', args...)` instead.
  gen_scaler = GradScaler()  # Use without specifying 'cuda'
torch.Size([2, 3, 256, 256])
```

And finally running the model

In [107…

```
epoches = 1

for epoch in range(epoches):
    running_dis_loss = 0.0
    running_gen_loss = 0.0
    for photo, monet in tqdm(dataloader, leave=True):
```

```
        photo = photo.to(device)
        monet = monet.to(device)

        # Train discriminator:
        fake_photo = gen_photo(monet)
        Dis_photo_real = disc_photo(photo)
        Dis_photo_fake = disc_photo(fake_photo.detach())

        Dis_photo_loss = MSE(Dis_photo_real, torch.ones_like(Dis_photo_real)
                         MSE(Dis_photo_fake, torch.zeros_like(Dis_photo_fake

        fake_monet = gen_monet(photo)
        Dis_monet_real = disc_monet(monet)
        Dis_monet_fake = disc_monet(fake_monet.detach())

        Dis_monet_loss = MSE(Dis_monet_real, torch.ones_like(Dis_monet_real)
                         MSE(Dis_monet_fake, torch.zeros_like(Dis_monet_fake

        Dis_loss = (Dis_photo_loss + Dis_monet_loss) / 2.0
        running_dis_loss += Dis_loss / len(dataloader)

        disc_optimizer.zero_grad()
        dis_scaler.scale(Dis_loss).backward()
        dis_scaler.step(disc_optimizer)
        dis_scaler.update()

        # Train Generator:
        Dis_photo_fake = disc_photo(fake_photo)
        Dis_monet_fake = disc_monet(fake_monet)

        Gen_photo_loss = MSE(Dis_photo_fake, torch.ones_like(Dis_photo_fake)
        Gen_monet_loss = MSE(Dis_monet_fake, torch.ones_like(Dis_monet_fake)

        Cycled_monet = gen_monet(fake_photo)
        Cycled_photo = gen_photo(fake_monet)

        Cycled_loss = L1(monet, Cycled_monet) + L1(photo, Cycled_photo)

        Gen_loss = Gen_photo_loss + Gen_monet_loss + Cycled_loss * lambda_cy
        running_gen_loss += Gen_loss / len(dataloader)

        gen_optimizer.zero_grad()
        gen_scaler.scale(Gen_loss).backward()
        gen_scaler.step(gen_optimizer)
        gen_scaler.update()
    print(f"Epoch {epoch + 1}. Generator loss by epoch: {running_gen_loss},
```

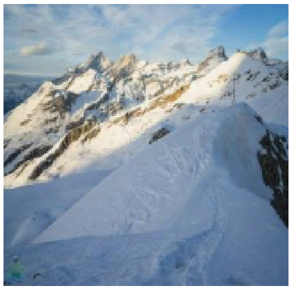
```
100%|████████████████████████████████████████| 880/880 [2:07:06<00:00,  8.67
s/it]
Epoch 1. Generator loss by epoch: 5.680804252624512, discriminator loss by e
poch: 0.3519006669521332
```

In [108…
```
torch.save(disc_photo.state_dict(), '/Users/evelynhaskins/disc_photo.pth')
torch.save(disc_monet.state_dict(), '/Users/evelynhaskins/disc_monet.pth')
torch.save(gen_photo.state_dict(), '/Users/evelynhaskins/gen_photo.pth')
torch.save(gen_monet.state_dict(), '/Users/evelynhaskins/gen_monet.pth')
```

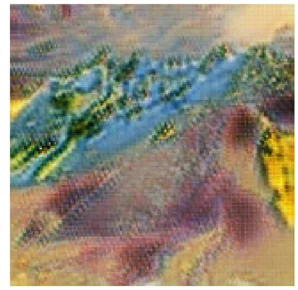
Let's see what it does, it takes in the photo on the left, aka "origonal photo" and makes it look more like a monet

In [109…
```
batch = next(iter(dataloader))[0]

_, ax = plt.subplots(5, 2, figsize=(12, 12))

for i in range(5):
    original_img = batch[i]
    predicted_img = None
    with torch.no_grad():
        predicted_img = gen_monet(original_img.unsqueeze(0).to(device))

    ax[i, 0].imshow(original_img.permute(1, 2, 0) * 0.5 + 0.5)
    ax[i, 1].imshow(predicted_img.squeeze(0).permute(1, 2, 0).cpu() * 0.5 +

    ax[i, 0].set_title("Original photo")
    ax[i, 1].set_title("Monet like")

    ax[i, 0].axis("off")
    ax[i, 1].axis("off")
plt.show()
```

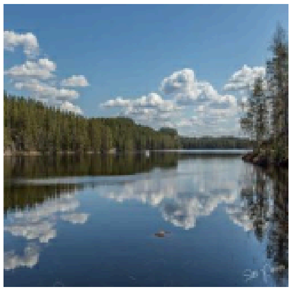
Original photo



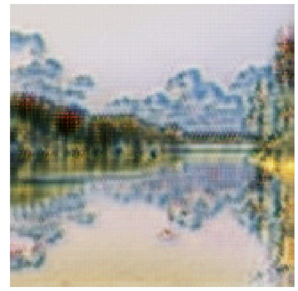
Monet like



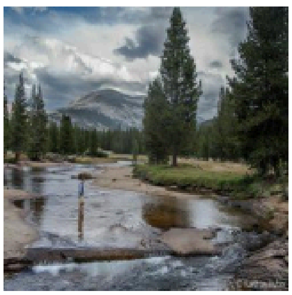
Original photo



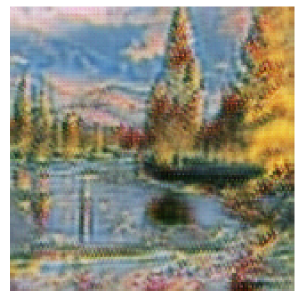
Monet like



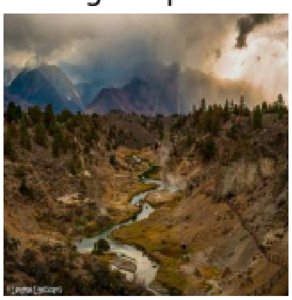
Original photo



Monet like

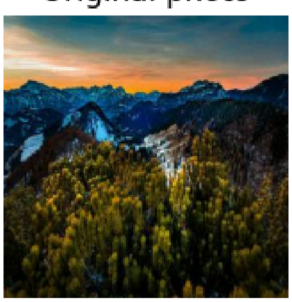


Original photo

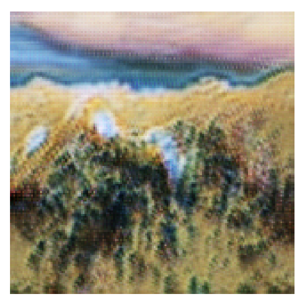


Monet like

Original photo



Monet like



Conclusion

This project successfully demonstrated the application of CycleGAN for unpaired image-to-image translation by transforming everyday photographs into Monet-style paintings. Through careful implementation of the CycleGAN architecture and loss functions, the model was able to produce high-quality, visually compelling images while preserving the content of the original photos. This project showcases the power of generative models in creating art, bridging the gap between technology and creativity.