Project Description

This project uses 1510 oil paintings in the Symbolic Art style as a dataset. As a contemporary abstract oil painting artist, I wanted to use machine learning methods to gain more inspiration for my artwork. Almost all art creators work from their own experiences and bring these to their work, but computers don't work like this, they don't have human experiences and emotions, so I created this art project to gain more inspiration from it.

I tried to use a variety of models and learning degrees to make my project perfect. I used DCGAN and styleGan2 to complete my project and compared their outputs. After the coding3 module, I have become proficient in using machine learning to inspire my artwork. With the help of Jasper, I have learned more about python applications such as how to use it to process image data and some tips on how to use Colab, and most excitingly, I have been able to use different datasets to produce some interesting artwork after this module.

Github Link: 22010385/Coding3Fianl-project (arts.ac.uk)

Datasate: WikiArt Art Movements/Styles | Kaggle

Code reference:

1.Art-ViT | Kaggle
2.New images generated through GAN | Kaggle
3.Pytorch DCGAN Art Generation | Kaggle
4.GANart | Kaggle
5. dvschultz/stylegan2-ada-pytorch: StyleGAN2-ADA - Official PyTorch implementation (github.com)
6. Abstract Art Generation using DCGAN and PyTorch | Kaggle

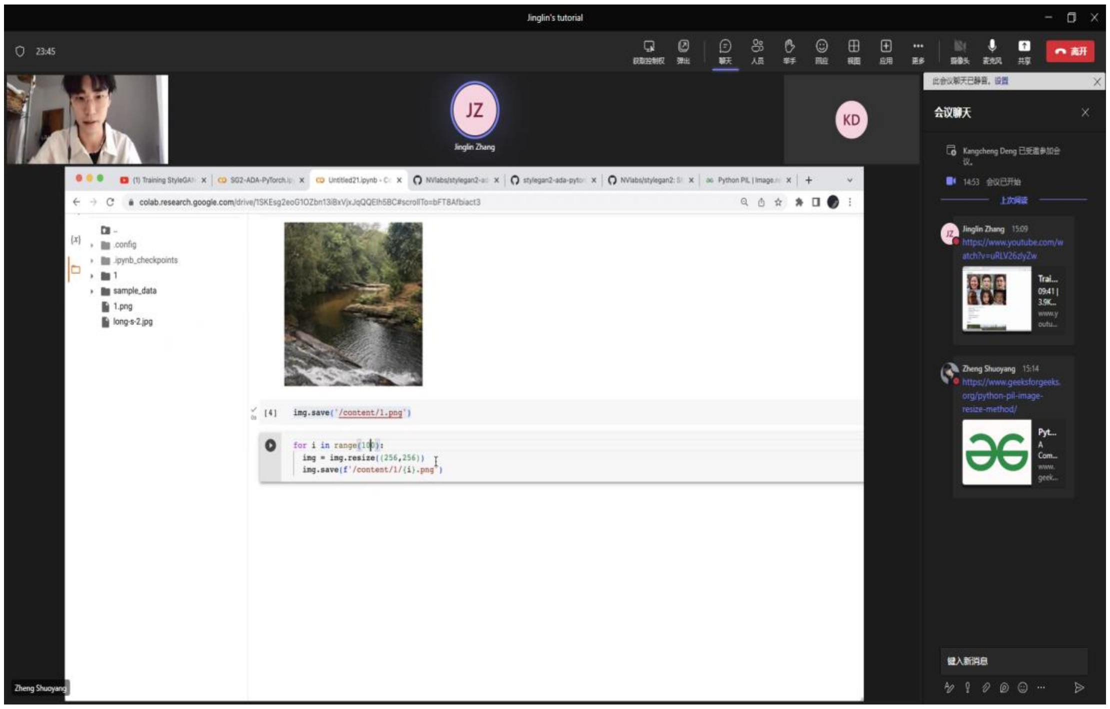
Learning materials:

1.【不要再看那些过时的GAN老教程了】耗时半年，GAN生成对抗网络⭐2022新版教程全15讲！GAN/生成对抗/深度学习/神经网络/卷积_哔哩哔哩_bilibili
2.基于Pytorch的DCGAN【深度卷积生成对抗网络】生成二次元头像_哔哩哔哩_bilibili
3. DC-GAN Explained! - YouTube
4. Training a GAN from your Own Images: StyleGAN2 ADA - YouTube

5. [DCGAN implementation from scratch - YouTube]

Difficulties:

At the beginning I wanted to use stylegan2-ADA for training, but to be compatible with stylegan2 I had to standardize the size of all the photos in my dataset. At the same time, I was also having problems importing datasets so I booked a tutorial with Jasper and asked for his help



```
[4] img.save('/content/1.png')
```

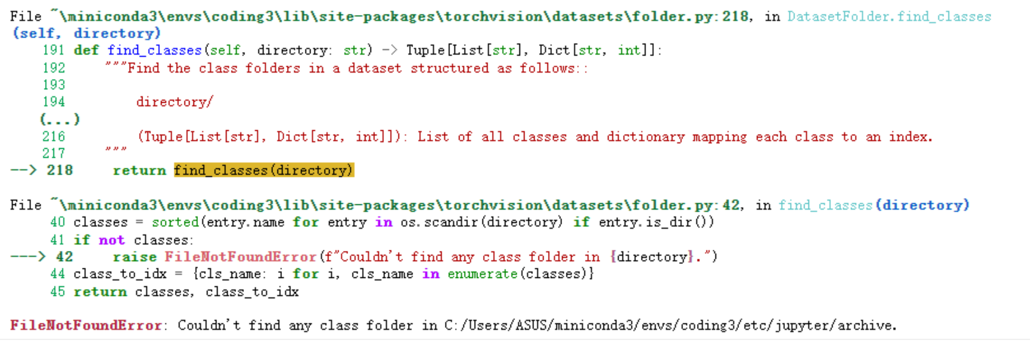
```
for i in range(100):
  img = img.resize((256,256))
  img.save(f'/content/1/{i}.png')
```

With the help of jasper, I completed the pre-processing of the dataset, The process was fun because it was the first time I used python for data processing besides machine learning

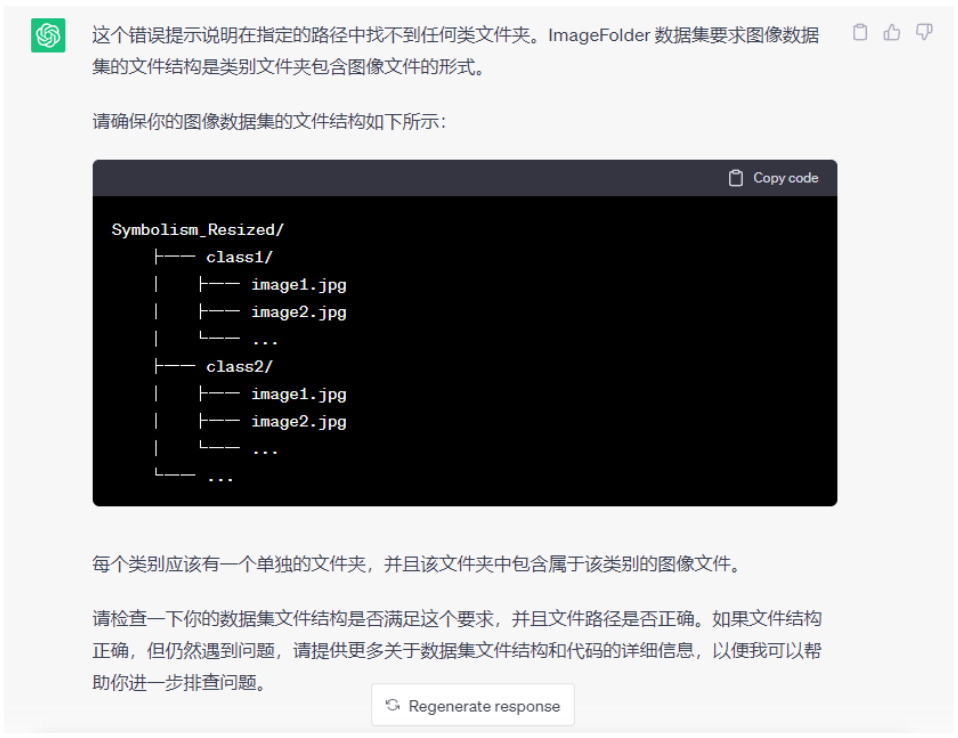
```
15 lines (13 sloc) | 654 Bytes
```

```python
#This code is used to batch process photo sizes, I set all images in the dataset to 256*256
from PIL import Image
import os

folder_path = 'D:/Programming/Projects/Python/DKC ImageResize/Symbolism'
folder_path_resized = 'D:/Programming/Projects/Python/DKC ImageResize/Symbolism_Resized'

for root, dirs, files in os.walk(folder_path):
    for file in files:
        file_path = os.path.join(root, file)
        file_name = os.path.basename(file_path)
        image = Image.open(file_path)
        img = image.resize((256,256))
        img.save(folder_path_resized + '/' + file_name)
    print('All graphs have been resized successfully.')
```

The second problem I encountered was corruption of the dataset as I had selected a portion of the entire dataset, resulting in a confusing classification of the data in the dataset and I had to rearrange my dataset and modify the import path of the dataset and pre-process them again. I asked ChatGPT and followed the steps ChatGPT taught me to solve this part of the problem.



```
File ~\miniconda3\envs\coding3\lib\site-packages\torchvision\datasets\folder.py:218, in DatasetFolder.find_classes
(self, directory)
    191 def find_classes(self, directory: str) -> Tuple[List[str], Dict[str, int]]:
    192     """Find the class folders in a dataset structured as follows::
    193 
    194         directory/
   (...)
    216         (Tuple[List[str], Dict[str, int]]): List of all classes and dictionary mapping each class to an index.
    217     """
--> 218     return find_classes(directory)

File ~\miniconda3\envs\coding3\lib\site-packages\torchvision\datasets\folder.py:42, in find_classes(directory)
     40 classes = sorted(entry.name for entry in os.scandir(directory) if entry.is_dir())
     41 if not classes:
---> 42     raise FileNotFoundError(f"Couldn't find any class folder in {directory}.")
     44 class_to_idx = {cls_name: i for i, cls_name in enumerate(classes)}
     45 return classes, class_to_idx

FileNotFoundError: Couldn't find any class folder in C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/archive.
```



这个错误提示说明在指定的路径中找不到任何类文件夹。ImageFolder 数据集要求图像数据集的文件结构是类别文件夹包含图像文件的形式。

请确保你的图像数据集的文件结构如下所示：

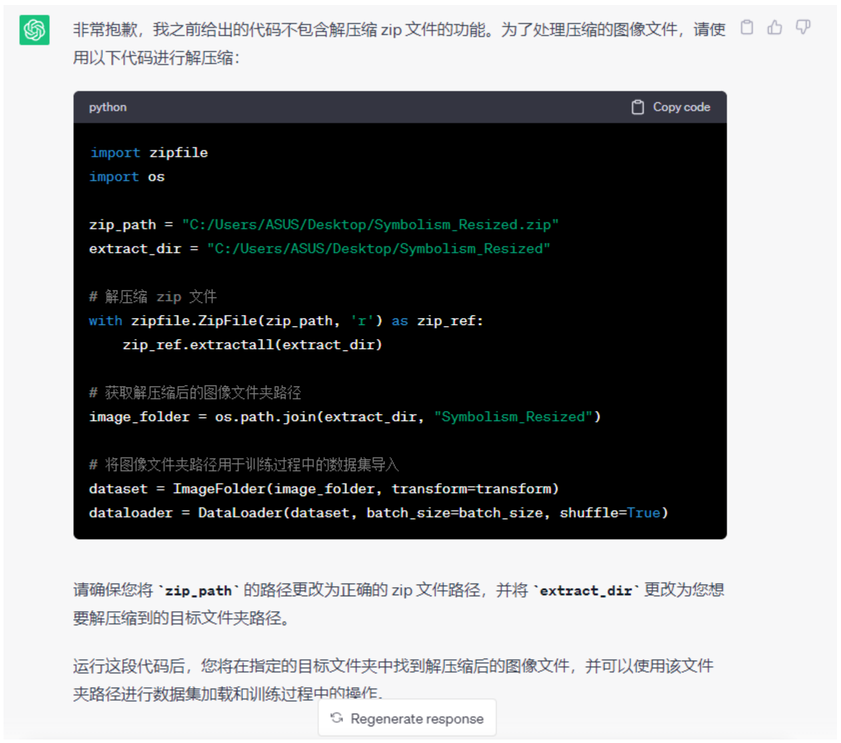
```
Symbolism_Resized/
    ├── class1/
    │   ├── image1.jpg
    │   ├── image2.jpg
    │   └── ...
    ├── class2/
    │   ├── image1.jpg
    │   ├── image2.jpg
    │   └── ...
    └── ...
```

每个类别应该有一个单独的文件夹，并且该文件夹中包含属于该类别的图像文件。

请检查一下你的数据集文件结构是否满足这个要求，并且文件路径是否正确。如果文件结构正确，但仍然遇到问题，请提供更多关于数据集文件结构和代码的详细信息，以便我可以帮助你进一步排查问题。

After completing the first training session, I was disappointed that the parameters resulted in a poorly outputted image that did not hit my standards.

I also encountered a lot of problems during the training process, such as how to use CUDA to complete the computational tasks, how to set the size, magnitude and delay of the output gifs……These questions were answered for me by ChatGPT, but the main technical questions were answered for me by Jasper on tutorial.



非常抱歉，我之前给出的代码不包含解压缩 zip 文件的功能。为了处理压缩的图像文件，请使用以下代码进行解压缩：

```python
import zipfile
import os

zip_path = "C:/Users/ASUS/Desktop/Symbolism_Resized.zip"
extract_dir = "C:/Users/ASUS/Desktop/Symbolism_Resized"

# 解压缩 zip 文件
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

# 获取解压缩后的图像文件夹路径
image_folder = os.path.join(extract_dir, "Symbolism_Resized")

# 将图像文件夹路径用于训练过程中的数据集导入
dataset = ImageFolder(image_folder, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

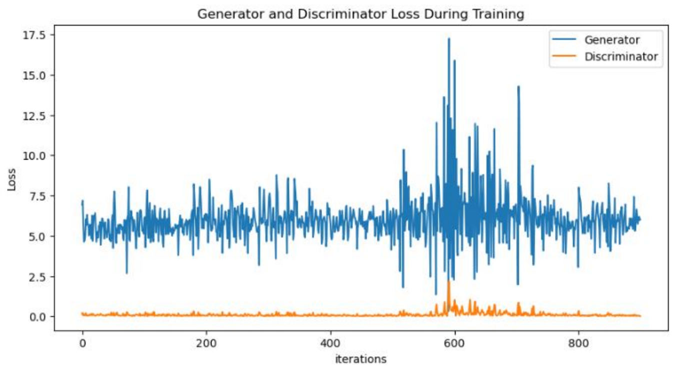
请确保您将 `zip_path` 的路径更改为正确的 zip 文件路径，并将 `extract_dir` 更改为您想要解压缩到的目标文件夹路径。

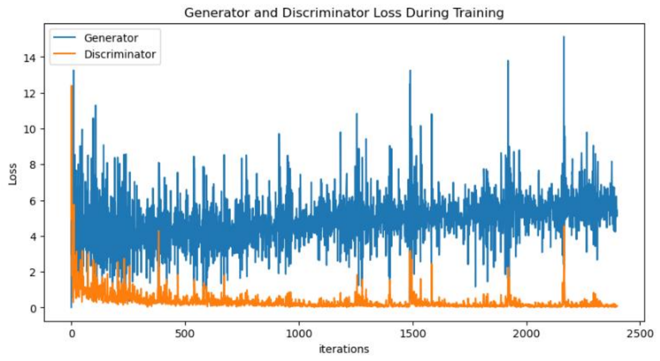
运行这段代码后，您将在指定的目标文件夹中找到解压缩后的图像文件，并可以使用该文件夹路径进行数据集加载和训练过程中的操作。

About my works:

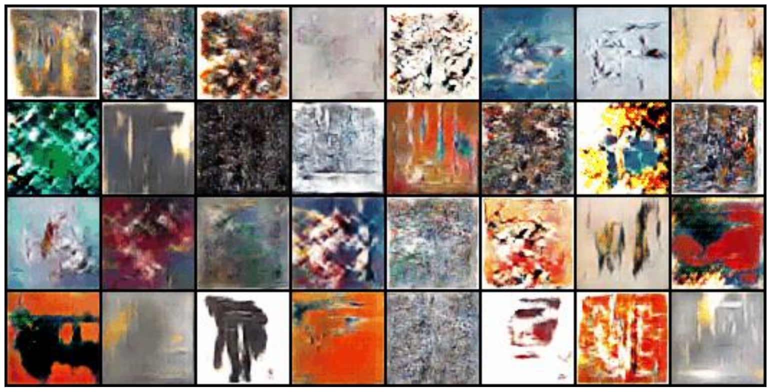
After the initial training, I was not satisfied with the output as it did not change significantly although I could see the training process, but the output was not up to my standard.

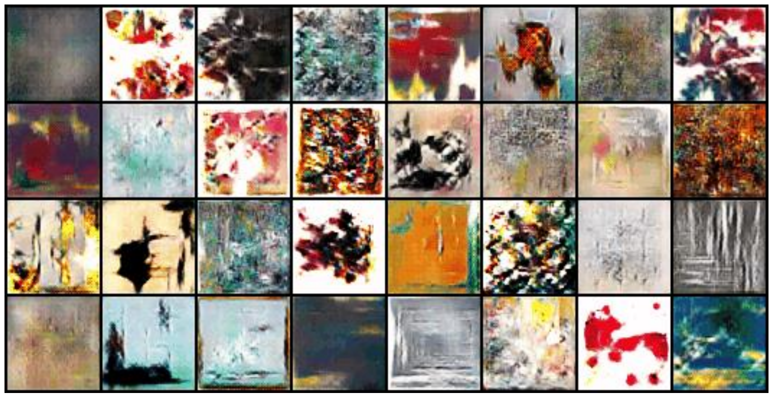
I think this happens because there are not enough loops and a small amount of learning. But in my later attempts I used another optimiser (ADAM) to control the decay rate and give it a stable value to allow the computer to stabilise during the training process.

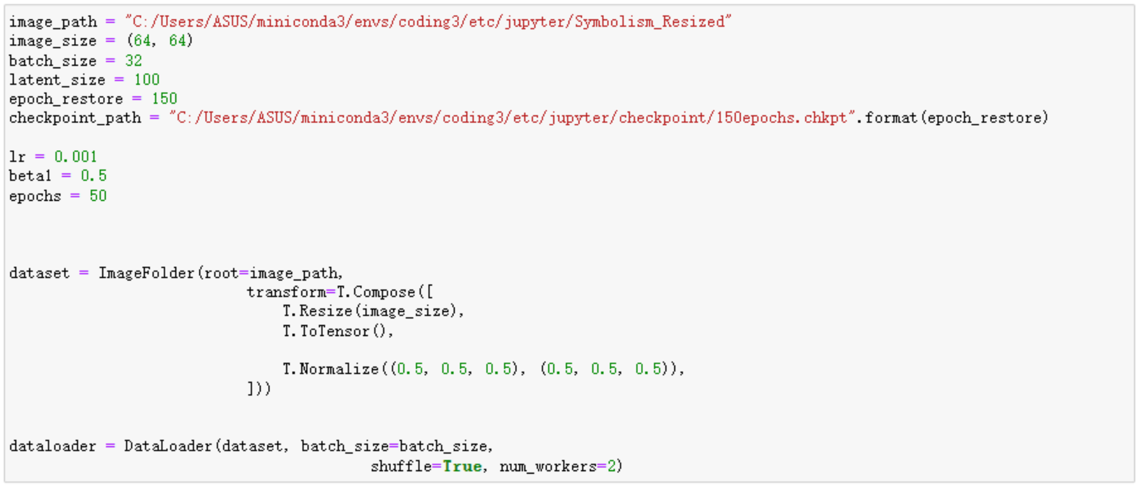




After this, I made adjustments to the parameters for the output gifs. You can see the obvious changes in the two graphs below.







```
image_path = "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/Symbolism_Resized"
image_size = (64, 64)
batch_size = 32
latent_size = 100
epoch_restore = 150
checkpoint_path = "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/checkpoint/150epochs.chkpt".format(epoch_restore)

lr = 0.001
beta1 = 0.5
epochs = 50



dataset = ImageFolder(root=image_path,
                           transform=T.Compose([
                               T.Resize(image_size),
                               T.ToTensor(),

                               T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))


dataloader = DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=2)
```

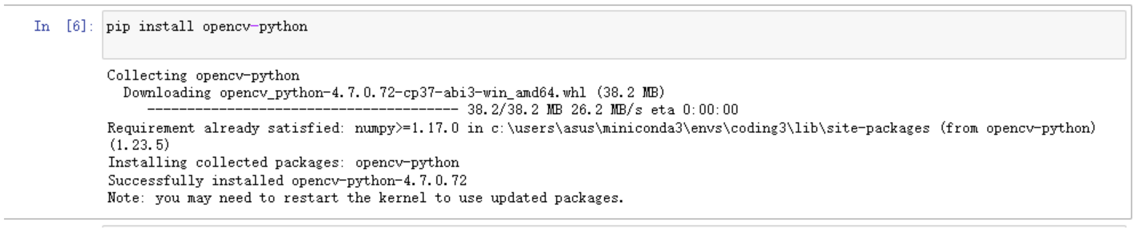
In the first attempt I used an lr of 0.0002, Beta1=0.9, epochs=30, This section made it clear to me that this parameter can have a huge impact on the training results.

Having familiarised myself with how to use DCgan, I'm still not too happy with the results so far as I feel I've used the best dataset in the world and it shouldn't be what it is now, and an obsession with symbolism has led me to continue experimenting with my project.

For my next work, I tried to use the symbolism dataset to make a state dictionary of generators and discriminators, and loaded the model with these two files. I wanted to try to make a new project in this way, but unfortunately I failed.

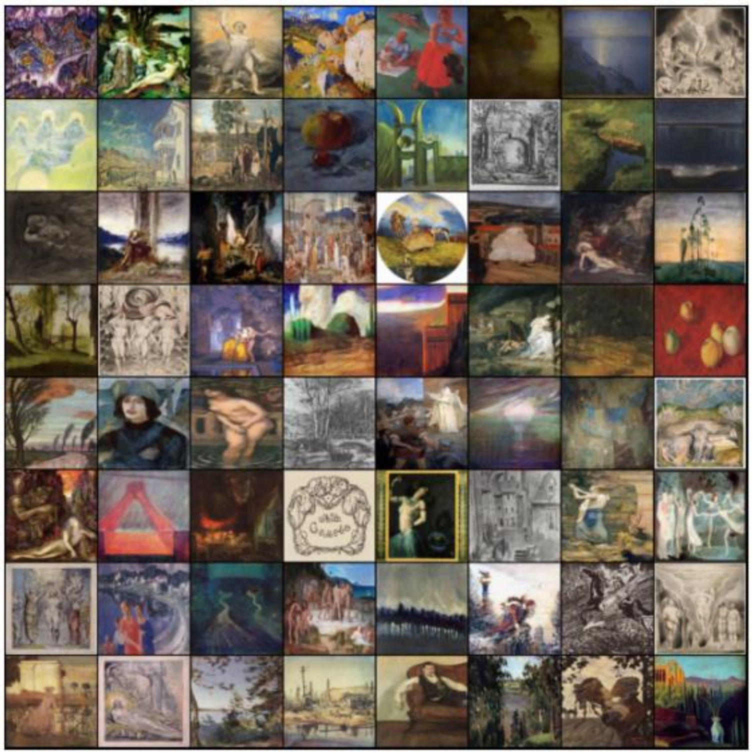
I imported OpenCv's libraries into my jupyter notebook and used models already trained on kaggle to create them. I am happy to share my work with you, although it is not that original and perfect.

First I imported the Opencv library and imported my favourite symbolism dataset.



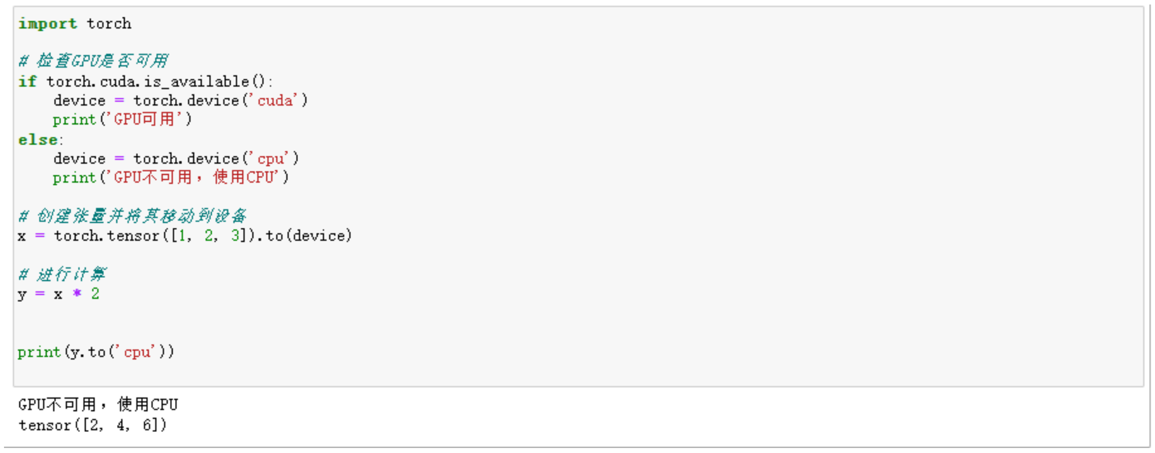
```
In [6]: pip install opencv-python

        Collecting opencv-python
          Downloading opencv_python-4.7.0.72-cp37-abi3-win_amd64.whl (38.2 MB)
             ---------------------------------------- 38.2/38.2 MB 26.2 MB/s eta 0:00:00
        Requirement already satisfied: numpy>=1.17.0 in c:\users\asus\miniconda3\envs\coding3\lib\site-packages (from opencv-python)
        (1.23.5)
        Installing collected packages: opencv-python
        Successfully installed opencv-python-4.7.0.72
        Note: you may need to restart the kernel to use updated packages.
```

After this I visualized it in my jupyter notebook, It looks like that~~~



Gorgeous!!!!

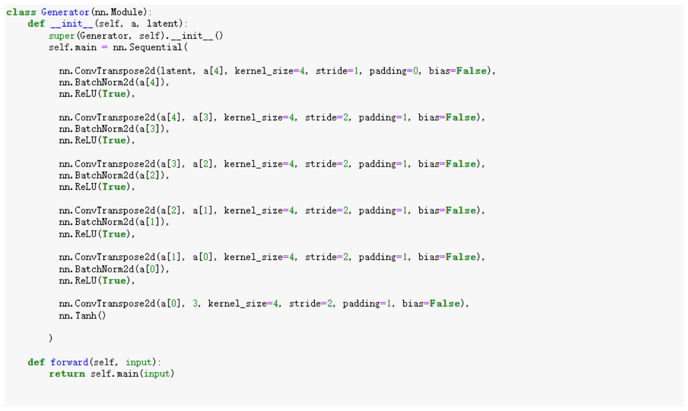
But after creating the generators and discriminators, the code gave an error because, according to the tutorial, I couldn't use CUDA to accelerate my GPU and I wasn't even sure if my computer was using a GPU or a CPU for training, Based on the answer provided by chatgpt, I wrote a piece of code to check if my computer's GPU was available. The answer I got was that the GPU was unavailable and the CPU was available.



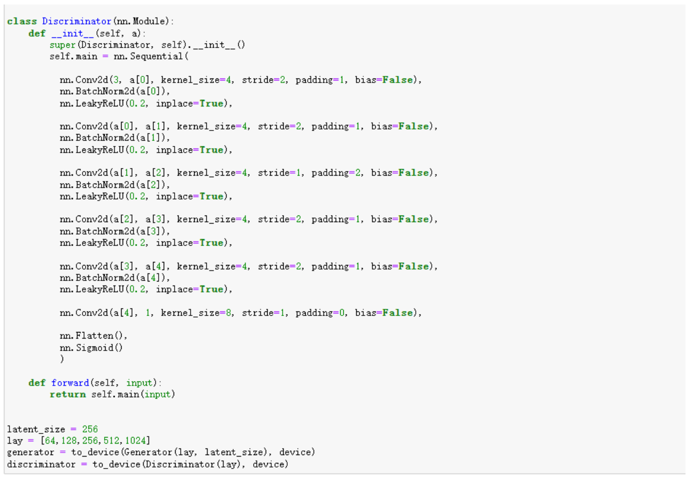
```
import torch

# 检查GPU是否可用
if torch.cuda.is_available():
    device = torch.device('cuda')
    print('GPU可用')
else:
    device = torch.device('cpu')
    print('GPU不可用, 使用CPU')

# 创建张量并将其移动到设备
x = torch.tensor([1, 2, 3]).to(device)

# 进行计算
y = x * 2


print(y.to('cpu'))
```

```
GPU不可用, 使用CPU
tensor([2, 4, 6])
```

Next I defined generators and discriminators, assigned them to my devices, and initialized them.

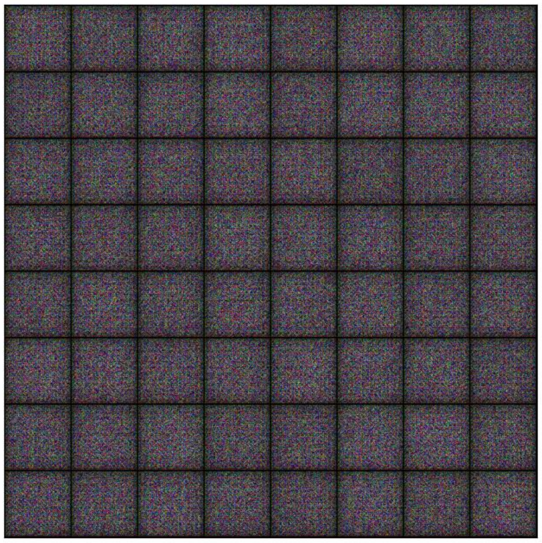


```
class Generator(nn.Module):
    def __init__(self, a, latent):
        super(Generator, self).__init__()
        self.main = nn.Sequential(

          nn.ConvTranspose2d(latent, a[4], kernel_size=4, stride=1, padding=0, bias=False),
          nn.BatchNorm2d(a[4]),
          nn.ReLU(True),

          nn.ConvTranspose2d(a[4], a[3], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[3]),
          nn.ReLU(True),

          nn.ConvTranspose2d(a[3], a[2], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[2]),
          nn.ReLU(True),

          nn.ConvTranspose2d(a[2], a[1], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[1]),
          nn.ReLU(True),

          nn.ConvTranspose2d(a[1], a[0], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[0]),
          nn.ReLU(True),

          nn.ConvTranspose2d(a[0], 3, kernel_size=4, stride=2, padding=1, bias=False),
          nn.Tanh()

        )

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

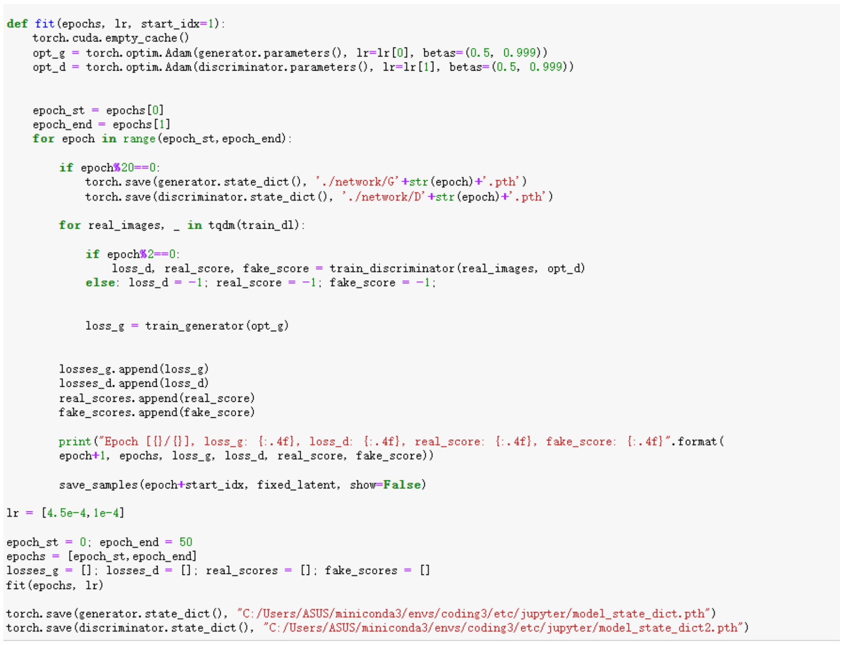


```
class Discriminator(nn.Module):
    def __init__(self, a):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(

          nn.Conv2d(3, a[0], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[0]),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(a[0], a[1], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[1]),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(a[1], a[2], kernel_size=4, stride=1, padding=2, bias=False),
          nn.BatchNorm2d(a[2]),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(a[2], a[3], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[3]),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(a[3], a[4], kernel_size=4, stride=2, padding=1, bias=False),
          nn.BatchNorm2d(a[4]),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(a[4], 1, kernel_size=8, stride=1, padding=0, bias=False),

          nn.Flatten(),
          nn.Sigmoid()
          )

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


latent_size = 256
lay = [64,128,256,512,1024]
generator = to_device(Generator(lay, latent_size), device)
discriminator = to_device(Discriminator(lay), device)
```

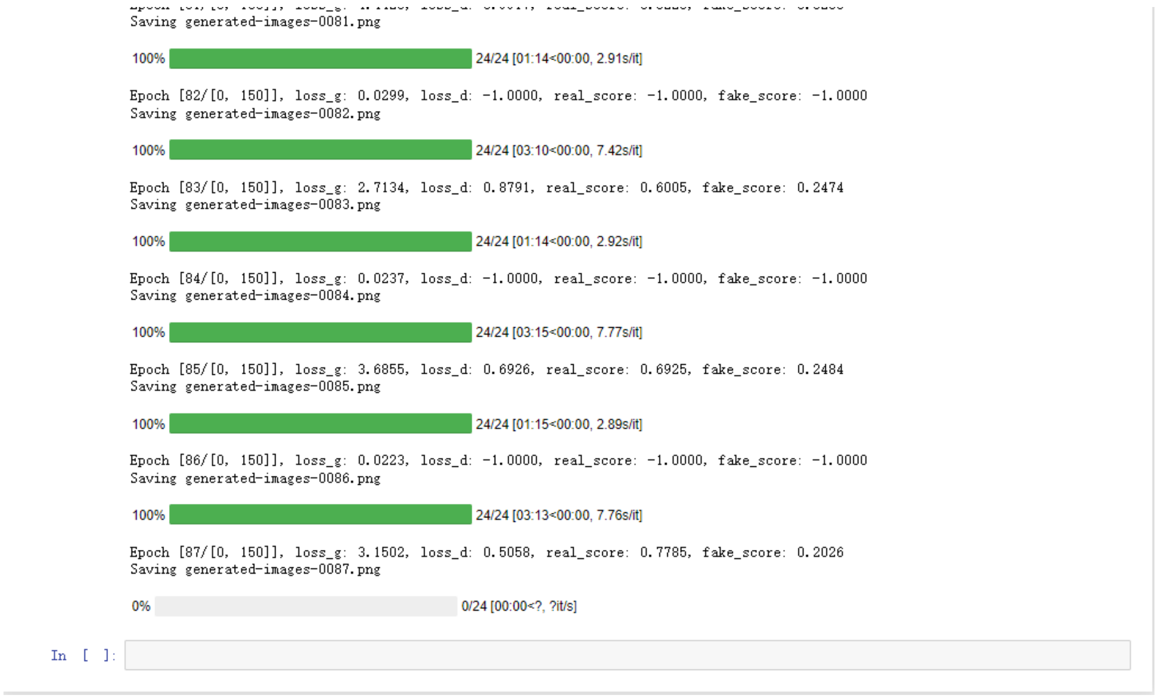
After this, I preview the sample images generated by the generator, I use the save_samples function to accept an index value index and the latent vector latent_tensors as input. It uses the generator model generator to generate the fake image so that I can preview it.



Next I created optimizers for the generator and discriminator and defined a training loop during which I transmitted the epochs and lr for the computer to be trained in each training round.

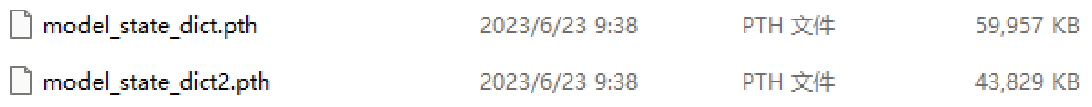


```
def fit(epochs, lr, start_idx=1):
    torch.cuda.empty_cache()
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr[0], betas=(0.5, 0.999))
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr[1], betas=(0.5, 0.999))


    epoch_st = epochs[0]
    epoch_end = epochs[1]
    for epoch in range(epoch_st,epoch_end):

        if epoch%20==0:
            torch.save(generator.state_dict(), './network/G'+str(epoch)+'.pth')
            torch.save(discriminator.state_dict(), './network/D'+str(epoch)+'.pth')

        for real_images, _ in tqdm(train_dl):

            if epoch%2==0:
                loss_d, real_score, fake_score = train_discriminator(real_images, opt_d)
            else: loss_d = -1; real_score = -1; fake_score = -1;


            loss_g = train_generator(opt_g)


        losses_g.append(loss_g)
        losses_d.append(loss_d)
        real_scores.append(real_score)
        fake_scores.append(fake_score)

        print("Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, real_score: {:.4f}, fake_score: {:.4f}".format(
        epoch+1, epochs, loss_g, loss_d, real_score, fake_score))

        save_samples(epoch+start_idx, fixed_latent, show=False)

lr = [4.5e-4,1e-4]

epoch_st = 0; epoch_end = 50
epochs = [epoch_st,epoch_end]
losses_g = []; losses_d = []; real_scores = []; fake_scores = []
fit(epochs, lr)

torch.save(generator.state_dict(), "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/model_state_dict.pth")
torch.save(discriminator.state_dict(), "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/model_state_dict2.pth")
```



```
Saving generated-images-0081.png

100% 24/24 [01:14<00:00, 2.91s/it]

Epoch [82/[0, 150]], loss_g: 0.0299, loss_d: -1.0000, real_score: -1.0000, fake_score: -1.0000
Saving generated-images-0082.png

100% 24/24 [03:10<00:00, 7.42s/it]

Epoch [83/[0, 150]], loss_g: 2.7134, loss_d: 0.8791, real_score: 0.6005, fake_score: 0.2474
Saving generated-images-0083.png

100% 24/24 [01:14<00:00, 2.92s/it]

Epoch [84/[0, 150]], loss_g: 0.0237, loss_d: -1.0000, real_score: -1.0000, fake_score: -1.0000
Saving generated-images-0084.png

100% 24/24 [03:15<00:00, 7.77s/it]

Epoch [85/[0, 150]], loss_g: 3.6855, loss_d: 0.6926, real_score: 0.6925, fake_score: 0.2484
Saving generated-images-0085.png

100% 24/24 [01:15<00:00, 2.89s/it]

Epoch [86/[0, 150]], loss_g: 0.0223, loss_d: -1.0000, real_score: -1.0000, fake_score: -1.0000
Saving generated-images-0086.png

100% 24/24 [03:13<00:00, 7.76s/it]

Epoch [87/[0, 150]], loss_g: 3.1502, loss_d: 0.5058, real_score: 0.7785, fake_score: 0.2026
Saving generated-images-0087.png

0% 0/24 [00:00<?, ?it/s]

In [ ]:
```

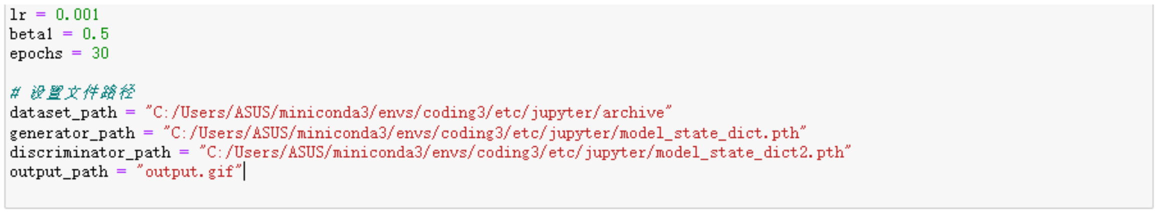
At the end I saved the output to my local folder, It is two such pth files.



model_state_dict.pth	2023/6/23 9:38	PTH 文件	59,957 KB
model_state_dict2.pth	2023/6/23 9:38	PTH 文件	43,829 KB

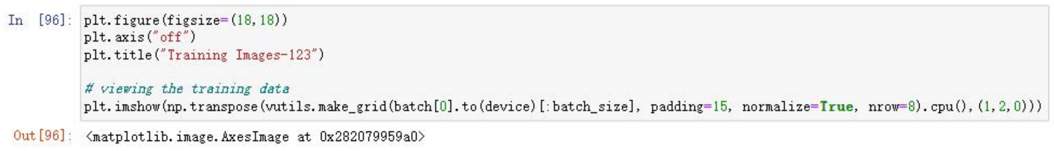
After this I tried to train the new dataset with my saved state dictionary and model, and I downloaded a dataset on kaggle with abstract art After this I tried to train the new dataset with my saved state dictionary and model. I downloaded a dataset on kaggle with abstract art and started a new round of machine training.

I set the training parameters and imported the new dataset and file paths, as well as defining the model dictionaries for the generators and discriminators.

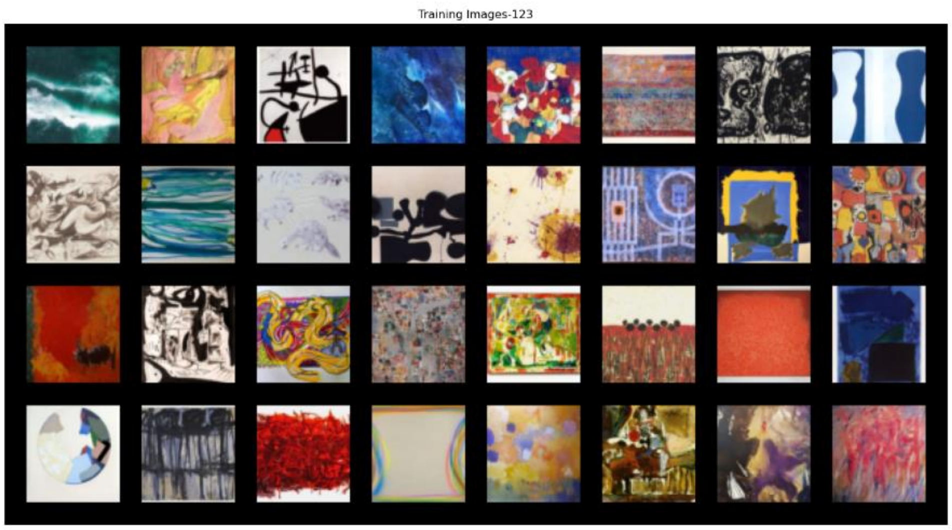


```
lr = 0.001
beta1 = 0.5
epochs = 30

# 设置文件路径
dataset_path = "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/archive"
generator_path = "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/model_state_dict.pth"
discriminator_path = "C:/Users/ASUS/miniconda3/envs/coding3/etc/jupyter/model_state_dict2.pth"
output_path = "output.gif"
```

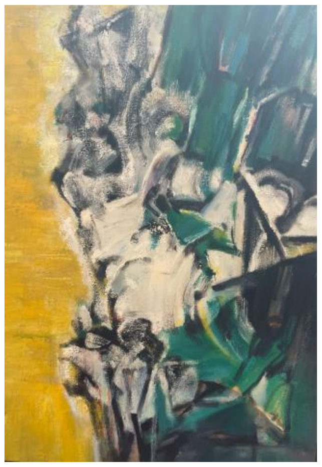
I have selected 1200 of my favourite abstract artworks from the Abstract Art dataset and put my own abstract oil paintings into the dataset. I have about 20 of my own works and although they are not many, I would like to see some variation in the output as I use this particular dataset to create a companion piece of art that is just for me.

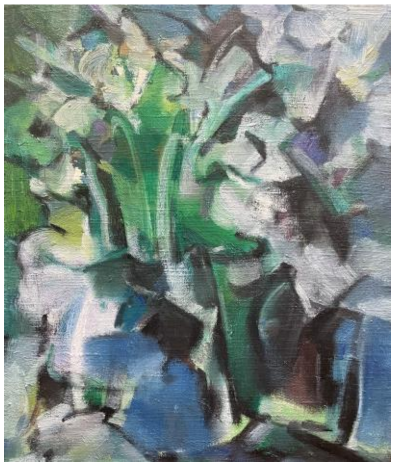


```
In [96]: plt.figure(figsize=(18,18))
         plt.axis("off")
         plt.title("Training Images-123")

         # viewing the training data
         plt.imshow(np.transpose(vutils.make_grid(batch[0].to(device)[:batch_size], padding=15, normalize=True, nrow=8).cpu(),(1,2,0)))

Out[96]: <matplotlib.image.AxesImage at 0x282079959a0>
```

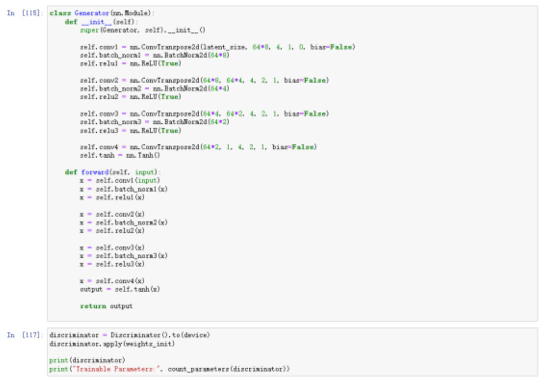


This is my own data that I added to the dataset,   my abstract paintings

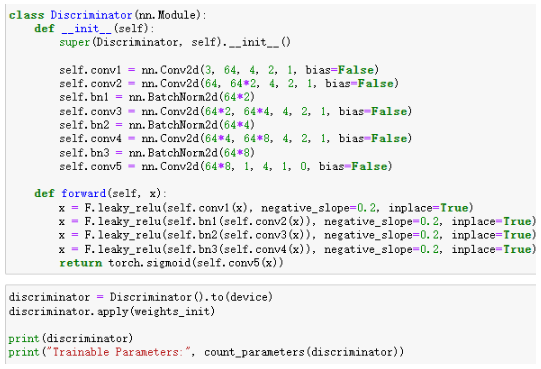




In the next process I redefined the generator and discriminator.

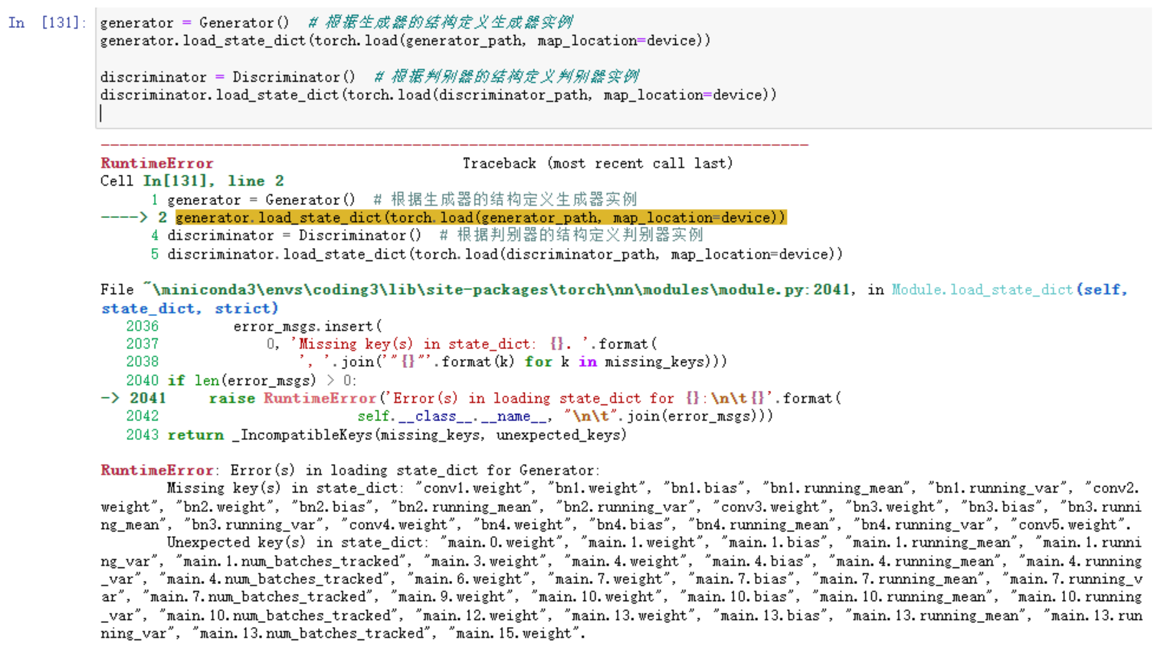


```
In [115]: class Generator(nn.Module):
              def __init__(self):
                  super(Generator, self).__init__()

                  self.conv1 = nn.ConvTranspose2d(latent_size, 64*8, 4, 1, 0, bias=False)
                  self.batch_norm1 = nn.BatchNorm2d(64*8)
                  self.relu1 = nn.ReLU(True)

                  self.conv2 = nn.ConvTranspose2d(64*8, 64*4, 4, 2, 1, bias=False)
                  self.batch_norm2 = nn.BatchNorm2d(64*4)
                  self.relu2 = nn.ReLU(True)

                  self.conv3 = nn.ConvTranspose2d(64*4, 64*2, 4, 2, 1, bias=False)
                  self.batch_norm3 = nn.BatchNorm2d(64*2)
                  self.relu3 = nn.ReLU(True)

                  self.conv4 = nn.ConvTranspose2d(64*2, 1, 4, 2, 1, bias=False)
                  self.tanh = nn.Tanh()

              def forward(self, input):
                  x = self.conv1(input)
                  x = self.batch_norm1(x)
                  x = self.relu1(x)

                  x = self.conv2(x)
                  x = self.batch_norm2(x)
                  x = self.relu2(x)

                  x = self.conv3(x)
                  x = self.batch_norm3(x)
                  x = self.relu3(x)

                  x = self.conv4(x)
                  output = self.tanh(x)

                  return output

In [117]: discriminator = Discriminator().to(device)
          discriminator.apply(weights_init)

          print(discriminator)
          print("Trainable Parameters:", count_parameters(discriminator))
```



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

        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1, bias=False)
        self.conv2 = nn.Conv2d(64, 64*2, 4, 2, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(64*2)
        self.conv3 = nn.Conv2d(64*2, 64*4, 4, 2, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(64*4)
        self.conv4 = nn.Conv2d(64*4, 64*8, 4, 2, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(64*8)
        self.conv5 = nn.Conv2d(64*8, 1, 4, 1, 0, bias=False)

    def forward(self, x):
        x = F.leaky_relu(self.conv1(x), negative_slope=0.2, inplace=True)
        x = F.leaky_relu(self.bn1(self.conv2(x)), negative_slope=0.2, inplace=True)
        x = F.leaky_relu(self.bn2(self.conv3(x)), negative_slope=0.2, inplace=True)
        x = F.leaky_relu(self.bn3(self.conv4(x)), negative_slope=0.2, inplace=True)
        return torch.sigmoid(self.conv5(x))

discriminator = Discriminator().to(device)
discriminator.apply(weights_init)

print(discriminator)
print("Trainable Parameters:", count_parameters(discriminator))
```

After defining the structure of the generator and discriminator, I tried to load the state dictionary, but it failed.



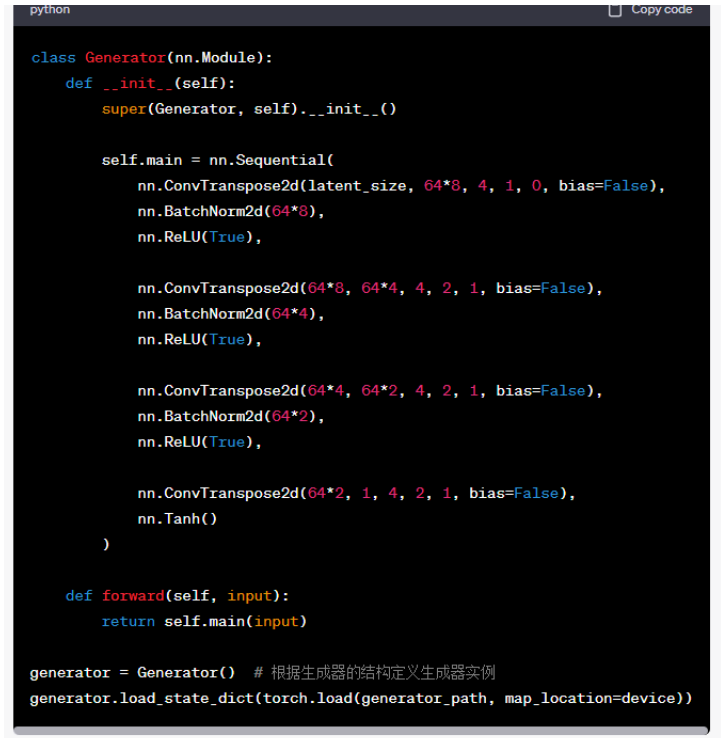
```
In  [131]: generator = Generator()  # 根据生成器的结构定义生成器实例
           generator.load_state_dict(torch.load(generator_path, map_location=device))

           discriminator = Discriminator()  # 根据判别器的结构定义判别器实例
           discriminator.load_state_dict(torch.load(discriminator_path, map_location=device))

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[131], line 2
      1 generator = Generator()  # 根据生成器的结构定义生成器实例
----> 2 generator.load_state_dict(torch.load(generator_path, map_location=device))
      4 discriminator = Discriminator()  # 根据判别器的结构定义判别器实例
      5 discriminator.load_state_dict(torch.load(discriminator_path, map_location=device))

File ~\miniconda3\envs\coding3\lib\site-packages\torch\nn\modules\module.py:2041, in Module.load_state_dict(self,
state_dict, strict)
   2036         error_msgs.insert(
   2037             0, 'Missing key(s) in state_dict: {}. '.format(
   2038                 ', '.join('"{}"'.format(k) for k in missing_keys)))
   2040 if len(error_msgs) > 0:
-> 2041     raise RuntimeError('Error(s) in loading state_dict for {}:\n\t{}'.format(
   2042                        self.__class__.__name__, "\n\t".join(error_msgs)))
   2043 return _IncompatibleKeys(missing_keys, unexpected_keys)

RuntimeError: Error(s) in loading state_dict for Generator:
	Missing key(s) in state_dict: "conv1.weight", "bn1.weight", "bn1.bias", "bn1.running_mean", "bn1.running_var", "conv2.
weight", "bn2.weight", "bn2.bias", "bn2.running_mean", "bn2.running_var", "conv3.weight", "bn3.weight", "bn3.bias", "bn3.runni
ng_mean", "bn3.running_var", "conv4.weight", "bn4.weight", "bn4.bias", "bn4.running_mean", "bn4.running_var", "conv5.weight".
	Unexpected key(s) in state_dict: "main.0.weight", "main.1.weight", "main.1.bias", "main.1.running_mean", "main.1.runni
ng_var", "main.1.num_batches_tracked", "main.3.weight", "main.4.weight", "main.4.bias", "main.4.running_mean", "main.4.running
_var", "main.4.num_batches_tracked", "main.6.weight", "main.7.weight", "main.7.bias", "main.7.running_mean", "main.7.running_v
ar", "main.7.num_batches_tracked", "main.9.weight", "main.10.weight", "main.10.bias", "main.10.running_mean", "main.10.running
_var", "main.10.num_batches_tracked", "main.12.weight", "main.13.weight", "main.13.bias", "main.13.running_mean", "main.13.run
ning_var", "main.13.num_batches_tracked", "main.15.weight".
```

I asked my teacher how this error happened, and he said that the state dictionary I loaded did not match the structure in my generator model, and that the structure of the existing state dictionary did not match the class structure of the generator. This is what my teacher suggested to me:

1. If you have access to the original code used to define the Generator class, you can modify the code so that it matches the existing state dictionary structure. This means that you need to change the name of the layer or module in the model so that it matches the keys in the state dictionary.

2. If you cannot access the original code or cannot modify it, you can create a new model class so that it matches the existing state dictionary structure. You can use state_dict.keys() to see the keys in the state dictionary and define the appropriate layer or module in the new model class.

3. If you have access to the original code and you are confident that the existing state dictionary contains compatible weights, you can manually load the weights from the state dictionary into the corresponding layer or module of the Generator model. This requires writing custom code to handle the loading process.


python Copy code

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

        self.main = nn.Sequential(
            nn.ConvTranspose2d(latent_size, 64*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64*8),
            nn.ReLU(True),

            nn.ConvTranspose2d(64*8, 64*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64*4),
            nn.ReLU(True),

            nn.ConvTranspose2d(64*4, 64*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64*2),
            nn.ReLU(True),

            nn.ConvTranspose2d(64*2, 1, 4, 2, 1, bias=False),
            nn.Tanh()
        )

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

generator = Generator()  # 根据生成器的结构定义生成器实例
generator.load_state_dict(torch.load(generator_path, map_location=device))
```

I did try to ask chatgpt to explain it to me and analyse it, but it was too difficult for me with an art background, so unfortunately I could only try so far, but I will continue to talk to my teacher to solve this problem, which is a problem I have been thinking about and trying to solve in the past few days.

In summary

I made a total of three attempts, the first to test the effect of DCgan and the impact of the data on the output of the results, here is a link to the first attempt(22010385/Coding3Fianl-project (arts.ac.uk)), After trying and learning, I modified the training parameters and the code to make my work look perfect. here is a link to the final works(22010385/Coding3Fianl-project (arts.ac.uk)), After this I wanted to go further, I tried to train my own model and keep a state dictionary of generators and discriminators, and even though I failed, I was still enthusiastic about the project and wanted to continue to learn enough to solve it.(22010385/Coding3Fianl-project (arts.ac.uk)).