第 3 章 不定图像的生成

3.1 任务简介

不定图像的生成，顾名思义在这个任务中生成的图像是不确定的，这里的不确定指的是不使用文字控制生成图像的内容。

在一般的 Diffusion 模型图像生成任务中会使用一段文本控制图像的内容，例如以“一匹高大的白色骏马”为文本来生成图像，期望的生成图像内容和文字描述是一致的。

在有文本控制的图像生成任务中需要考虑文本的特征，相对来讲会更复杂一些，为了由浅入深地进行学习，在本章只考虑无文本控制的图像生成，也就是说在本章只是让模型生成图像，不控制它生成的图像内容。有文本控制的图像生成任务在后续章节中再实现。

虽然在本章的图像生成任务中没有文本控制，但并不代表模型会画出任何内容，正如人类无法想象自己认知以外的事物一样，Diffusion 模型也无法生成它认知之外的图像，所以 Diffusion 模型在没有文本控制的情况下会生成什么样的图像取决于它见过什么样的训练图像。也就是说可以通过调整不同的训练数据来影响 Diffusion 生成的图像。

举个例子来讲，如果希望 Diffusion 生成的图像都是猫咪的图片，就可以使用大量的猫咪的图片来训练它，让它见识各种各样的猫咪，这样它在生成时也会生成猫咪的图片，这是显而易见的，因为在它的认知之中，这个世界上存在的事物就是训练数据集中的事物，不存在训练数据集以外的任何事物，所以在让它生成图像时，它当然只会生成训练数据集中的事物。

具体到本章的任务，在本章中会训练一个没有任何先验知识的 Diffusion 模型去生成鲜花的图像。没有先验知识意味着不是微调(Fine Tuning)，而是从随机参数开始训练，由于本章使用的实验模型体量较小，数据集的复杂度也较低，所以从随机参数训练也可以得到较好的结果。

3.2 数据集介绍

本章所使用的数据集已经上传到了 HuggingFace。这个数据集转载自 HugGAN Community 的 flowers-102-categories 数据集，感谢相关人员的无私奉献。

数据集的部分样例如图 3-1 所示。

图 3-1　数据集的样例

从图 3-1 可以看出，数据集中的数据就是简单的图片，图片的内容就是各种各样的鲜花，在本章中会向 Diffusion 模型输入很多这样的鲜花图片，训练完成后期望 Diffusion 模型能生成鲜花的图片。

3.3　测试部分

在训练之前先来进行测试部分的工作，正所谓不会测试就不会开发，这是笔者的小小心得，笔者个人甚至认为测试比开发更重要，测试是为了确保一切都走在正确的道路上，而不是跑偏了却不自知。

3.3.1　测试函数

为了对 Diffusion 模型的生成效果进行测试，需要定义测试函数，代码如下：

```
#第 3 章/定义测试函数
from matplotlib import pyplot as plt
%matplotlib inline
import torch


def test(pipeline):
```

```
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    pipeline = pipeline.to(device)

    images = pipeline(batch_size=8,
                      num_inference_steps=1000,
                      output_type='NumPy').images

    pipeline.to('cpu')
    torch.cuda.empty_cache()

    images = (images * 255).round().astype('uint8')

    plt.figure(figsize=(10, 5))
    for i in range(8):
        plt.subplot(2, 4, i + 1)
        plt.imshow(images[i])
        plt.axis('off')

    plt.show()
```

测试函数的入参是 pipeline，pipeline 代表着一个 Diffusion 模型，在测试函数中会自动判断可用的计算设备，支持 CUDA 的 GPU 或者 CPU，优先使用支持 CUDA 的 GPU。随后测试函数会用 pipeline 生成 8 张图像，并绘制到 Jupyter Notebook，方便开发者观察。

3.3.2　未训练模型的测试结果

先来测试未训练的模型，这是为了给本次实验定下基线（Baseline），未训练的模型代表着最差的表现，如果训练后的模型表现不能明显好于基线，则代表实验失败，所以为了验证训练的有效性需要先明确这条基线，代码如下：

```
#第 3 章/测试未训练的模型
from diffusers import DDPMPipeline, UNet2DModel, DDPMScheduler

#定义模型，随机初始化参数
model = UNet2DModel(
    sample_size=64,
    in_channels=3,
    out_channels=3,
    layers_per_block=2,
    block_out_channels=(128, 128, 256, 256, 512, 512),
    down_block_types=(
        'DownBlock2D',
        'DownBlock2D',
        'DownBlock2D',
```

```
            'DownBlock2D',
            'AttnDownBlock2D',
            'DownBlock2D',
        ),
        up_block_types=(
            'UpBlock2D',
            'AttnUpBlock2D',
            'UpBlock2D',
            'UpBlock2D',
            'UpBlock2D',
            'UpBlock2D',
        ),
    )

    #初始化工具类
    scheduler = DDPMScheduler(num_train_timesteps=1000,
                              beta_schedule='linear',
                              prediction_type='epsilon')

    test(DDPMPipeline(unet=model, scheduler=scheduler))
```

在以上代码中创建了一个随机初始化参数的 Diffusion 模型，它是一个没有训练过的模型，使用它得到的测试结果代表着本次实验的基线。运行结果如图 3-2 所示。

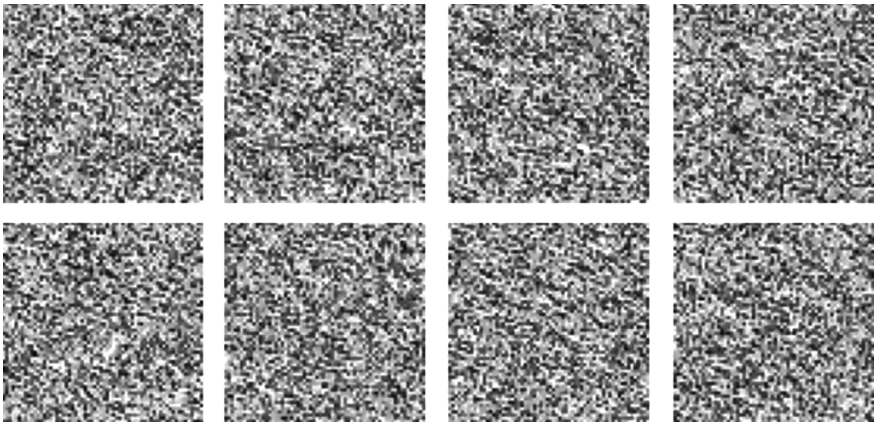

图 3-2　未训练模型的测试结果

图 3-2 表明未训练模型的测试结果是无内容的噪声图，这是符合预期的，因为模型还没有被训练，所以它当然不知道任何事物，更不知道该如何生成图像，所以它只是胡乱地输出了一些噪声作为图像。

而这些噪声图就是本次实验的基线，作为第 1 个实验项目会比较简单，只要输出的图像不是单纯的噪声就算成功。

3.3.3　训练后模型的测试结果

虽然本书到目前为止还没有讲解模型的训练，但是可以假设已经训练完毕了，因为笔者已经把训练好的模型上传到了 HuggingFace，所以可以使用笔者训练好的模型先进行测试，查看训练后的模型的测试结果，看一看它能实现什么样的效果，以做到心中有数，读者可以在自己训练完成后和笔者训练好的模型进行比较，以验证自己的训练结果是否符合预期。

使用训练好的模型进行测试的代码如下：

```
#第 3 章/在线加载笔者训练好的模型并测试
test(DDPMPipeline.from_pretrained('lansinuote/diffusion.1.unconditional'))
```

运行结果如图 3-3 所示。

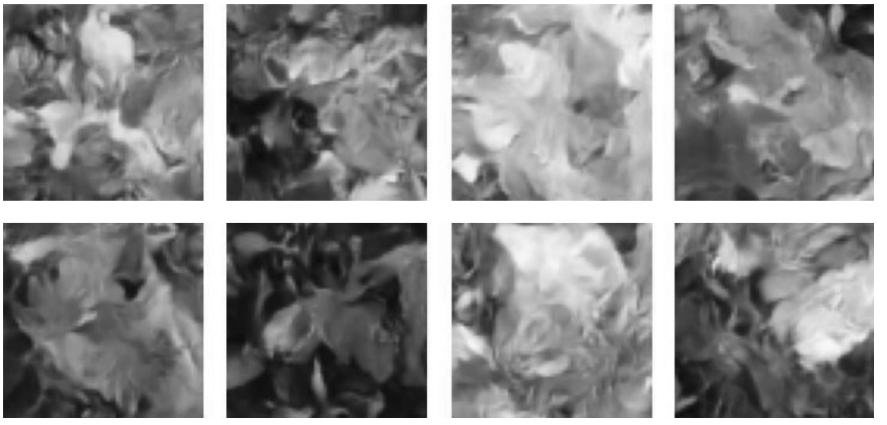

图 3-3　训练后模型的测试结果

从图 3-3 可以看出，训练后的模型生成的图像虽然质量不是很好，但是可以看出它试图画出一些鲜花的图像，这验证了训练的有效性。

在训练的部分，会把训练好的模型保存到代码文件同一目录的 save 文件夹下，当然这个路径是可以自定义的，这里先假设保存到代码文件目录的 save 文件夹下，测试保存的模型，代码如下：

```
#第 3 章/测试训练好的模型
test(DDPMPipeline.from_pretrained('./save'))
```

测试的方法很简单，调用 DDPMPipeline 的 from_pretrained()函数得到 pipeline，参数为模型保存的路径，有了 pipeline 以后就可以调用 test()函数得到测试结果了。

到目前为止，测试方法已经讲述完毕，有了测试方法，可以随时验证模型训练的有效性，以确保整个实验不会跑偏，接下来就可以开始讲解训练部分了。

3.4 训练部分

3.4.1 全局常量

需要定义一些全局常量，本章中由于任务本身比较简单，所以全局常量也比较简单，只有一个，定义的代码如下：

```
#第 3 章/全局常量
repo_id = 'lansinuote/diffusion.1.unconditional'
```

代码中定义了一个全局常量 repo_id，标识了上传到 HuggingFace 的数据集和模型的 id，在后续的代码中要多次用到这个常量，所以把它定义为全局常量。

3.4.2 定义数据集

1. 加载数据集

要训练模型，当然需要一个数据集，在之前的章节中已经介绍过本章使用的数据集的内容，这里讲解数据集的加载方法，出于简洁起见，使用处理好的数据集是最简单的，代码如下：

```
#第 3 章/使用笔者转载的数据集
dataset = load_dataset(path=repo_id, split='train')

dataset, dataset[0]
```

代码中加载了上传到 HuggingFace 的数据集，这个数据集只有一个子集 train，这里直接加载了 train 子集，这段代码的运行结果如下：

```
(Dataset({
    features: ['image'],
    num_rows: 8189
}),
{'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=752x500>})
```

从上面的输出可以看到，一共有 8000 多条数据，每条数据只有一个字段 image，显而易见，这个数据集其实就是 8000 多张图片。

上传到 HuggingFace 的这个数据集是转载自 HugGAN Community 的 flowers-102-categories 数据集，出于代码完整性考虑，这里给出使用原数据集的方式，代码如下：

```
#第 3 章/加载数据集
from datasets import load_dataset
```

```
load_dataset('huggan/flowers-102-categories', split='train')
```

运行结果如下：

```
Dataset({
    features: ['image'],
    num_rows: 8189
})
```

从上面的输出可以看出，和转载的数据集没有什么区别，输出结果是完全一致的。因为本章的任务比较简单，所以数据集部分没有什么太多的预处理，只是简单的转载，在后续的任务中，就可以看到使用预处理数据集的便利性。两种方法只要选择其中一种即可，建议使用转载的数据集进行训练，加载原数据集的方法可以作为笔记记录。

2. 数据集预处理

现在数据集加载好了，但是这个数据集还不能直接使用，需要一些预处理和数据增强，首要的任务是缩小并统一图片的尺寸，太大的图片尺寸对计算力的要求太高，训练的难度也会加大，对于本章这个简单的实验来讲 64×64 的图像尺寸就足够了，数据增强部分使用简单的左右翻转增强即可，对于鲜花图片来讲，左右翻转不影响图像的内容表达。综上所述，数据集预处理部分的代码如下：

```
#第 3 章/数据集预处理
import torchvision

#图像增强和编码
compose = torchvision.transforms.Compose([
    torchvision.transforms.Resize(
        64,
interpolation=torchvision.transforms.InterpolationMode.BILINEAR),
    torchvision.transforms.RandomCrop(64),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.5], [0.5])
])


def f(data):
    image = [compose(i) for i in data['image']]
    return {'image': image}


#因为图像增强在每个 epoch 中要动态计算，所以不能简单地用 map 处理
dataset.set_transform(f)
```

```
dataset, dataset[0]['image'].shape
```

代码中的注释已经表明了为什么需要使用动态计算，而不是直接在数据集预处理时解决这些问题，因为数据增强的部分是随机地左右翻转，所以每张图片被多次读取时翻转的结果可能是不一样的，所以需要动态计算，运行结果如下：

```
(Dataset({
    features: ['image'],
    num_rows: 8189
}),
torch.Size([3, 64, 64]))
```

可以看到数据集中的图片已经被转换成张量的形式。

3. 定义 loader

接下来定义一个数据集加载器，方便数据集的遍历，代码如下：

```
#第 3 章/定义 loader
import torch

loader = torch.utils.data.DataLoader(dataset,
                                     batch_size=16,
                                     shuffle=True,
                                     drop_last=False)

len(loader), next(iter(loader))['image'].shape
```

运行结果如下：

```
(512, torch.Size([16, 3, 64, 64]))
```

可以看到一共有 512 个批次，每个批次中有 16 张图像的数据。

3.4.3 定义模型

由于本章的任务比较简单，所以可以不使用预训练模型的参数，而是直接从随机参数开始训练模型，也可以得到较好的训练结果，定义随机参数的模型的代码如下：

```
#第 3 章/定义模型
from diffusers import UNet2DModel

#定义模型,随机初始化参数
model = UNet2DModel(
    sample_size=64,
```

```
    in_channels=3,
    out_channels=3,
    layers_per_block=2,
    block_out_channels=(128, 128, 256, 256, 512, 512),
    down_block_types=(
        'DownBlock2D',
        'DownBlock2D',
        'DownBlock2D',
        'DownBlock2D',
        'AttnDownBlock2D',
        'DownBlock2D',
    ),
    up_block_types=(
        'UpBlock2D',
        'AttnUpBlock2D',
        'UpBlock2D',
        'UpBlock2D',
        'UpBlock2D',
        'UpBlock2D',
    ),
)

sum(i.numel() for i in model.parameters()) / 10000
```

这里使用了 diffusers 库提供的工具类来定义模型，暂时不需要太关注模型实现的细节和原理，在后续的章节会展开详解，这里只需先跟随代码简单地把模型定义出来，而不需要理解工具类的实现细节。

在代码的最后部分计算了模型中的参数数量，代码的运行结果如下：

```
11367.3219
```

可以看到，在这个模型中参数量约为 1.1 亿，相比以往的深度学习模型，Diffusion 模型的体量普遍较大。

3.4.4　初始化工具类

在训练过程中需要用到一些工具类，这里统一把这些工具类定义出来，代码如下：

```
#第 3 章/初始化工具类
from diffusers import DDPMScheduler
from diffusers.optimization import get_scheduler

scheduler = DDPMScheduler(num_train_timesteps=1000,
                          beta_schedule='linear',
```

```
                          prediction_type='epsilon')

    optimizer = torch.optim.AdamW(model.parameters(),
                                  lr=1e-4,
                                  betas=(0.95, 0.999),
                                  weight_decay=1e-6,
                                  eps=1e-8)

    scheduler_lr = get_scheduler('cosine',
                                 optimizer=optimizer,
                                 num_warmup_steps=500,
                                 num_training_steps=len(loader) * 100)

    criterion = torch.nn.MSELoss()

    scheduler, optimizer, scheduler_lr, criterion
```

在这段代码中主要初始化了如下工具类：

(1) scheduler 是往图像中添加噪声的工具类。

(2) optimizer 是根据梯度调整模型参数的工具类。

(3) scheduler_lr 是调整学习率(Learning Rate)的工具类。

(4) criterion 是计算 loss 的工具类。

3.4.5 计算 loss

到目前为止，数据集、模型和工具类都已经准备完毕，还需要一个计算 loss 的函数。本章计算 loss 的过程比较简单，可以简单地使用 3 张图总结，第 1 步如图 3-4 所示。

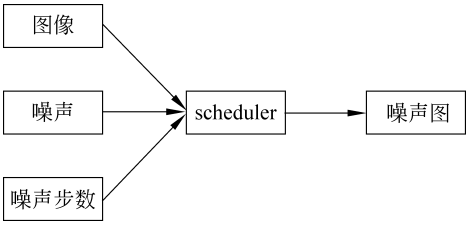

图 3-4 向图像中添加噪声

图 3-4 概述了向图像中添加噪声的方法，其中噪声步数是一个 0～999 的随机整数，这个数值衡量了噪声图中噪声的比例，999 代表是完全是噪声，几乎不包括原始图像中的数据，0 代表完全是原始图像中的数据，几乎不包括噪声。

之所以需要这个数值，是为了模拟图像生成的不同阶段，因为在生成图像时是从完全的噪声开始生成的，此时的噪声步数应该是 999，生成的过程就是不断地进行降噪的过程，在生成的过程中，逐渐降低噪声步数，当噪声步数降到 0 时，图像也就生成完毕了，这个过程在

后续的 Diffusion 底层原理部分会看得更详细，这里读者只需知道噪声步数这个数值是 0～999 的一个整数，并且它代表了噪声图中数据来自噪声的比例。

图 3-5 概述了模型计算的过程，模型的入参是添加了噪声的噪声图和噪声步数两份数据，模型的任务是从噪声图中根据噪声步数把原始的噪声预测出来。

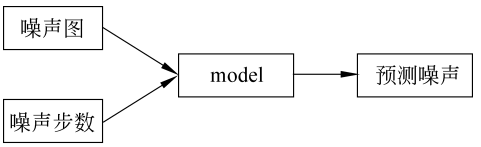

图 3-5　从噪声图中预测出噪声

图 3-6 概述了 loss 的计算方法，有了模型预测出来的噪声，想要计算 loss 就比较简单了，用预测出来的 loss 和原始的噪声求误差即可，一般计算均方误差损失（Mean Squared Error Loss，MSELoss）。

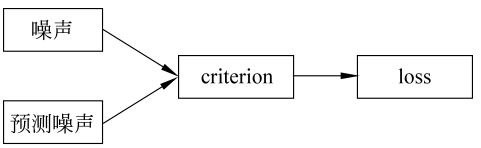

图 3-6　计算 loss

上面通过 3 张图向读者大概介绍了在本章任务中计算 loss 的方法，下面来看实际计算 loss 的代码，代码如下：

```
#第 3 章/定义计算 loss 的函数
def get_loss(image):
    device = image.device

    #随机噪声
    #[b, 3, 64, 64]
    noise = torch.randn(image.shape).to(device)

    #随机 b 个噪声步数
    #1000 = scheduler.config.num_train_timesteps
    #[b]
    noise_step = torch.randint(0, 1000, (image.shape[0], ),
                               device=device).long()

    #往图片中添加噪声
    #[b, 3, 64, 64]
    image_noise = scheduler.add_noise(image, noise, noise_step)

    #把图片里的噪声计算出来
    #[b, 3, 64, 64]
```

```
    out = model(image_noise, noise_step).sample

    #求 mse loss
    return criterion(out, noise)


get_loss(torch.randn(16, 3, 64, 64))
```

代码中使用随机数生成了噪声数据和噪声步数，如上面的图中所叙述的，首先使用工具类向图像数据中添加噪声得到噪声图，然后使用模型预测出原始的噪声数据，最后用两份噪声计算误差即可得到 loss，运行结果如下：

```
tensor(1.0573, grad_fn=<MseLossBackward0>)
```

3.4.6 训练

到这里为止，训练所需要的一切要素都已经准备完毕，可以开始训练了，训练的过程也已整理成了函数，简单调用即可，代码如下：

```
#第 3 章/训练
from diffusers import DDPMPipeline


def train():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model.to(device)
    model.train()

    loss_sum = 0
    for epoch in range(10):
        for i, data in enumerate(loader):
            loss = get_loss(data['image'].to(device))

            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler_lr.step()
            optimizer.zero_grad()

            loss_sum += loss.item()

        if epoch % 1 == 0:
            print(epoch, loss_sum)
            loss_sum = 0
```

```
        #save
        DDPMPipeline(unet=model, scheduler=scheduler).save_pretrained('./save')


    train()
```

由于训练的过程还是非常消耗资源的，程序会判断运行环境中是否有支持 CUDA 计算的设备，如果有，则优先使用 CUDA 设备计算，否则使用 CPU 计算。

训练的过程非常标准化，和大多数深度学习的方法一样，简单地从 loader 中得到一批一批的数据，然后计算 loss，根据 loss 计算梯度，根据梯度调整模型参数。

所有数据循环遍历 10 次，即可完成训练任务，把模型保存到本地磁盘，准备到测试的代码文件中进行测试，测试的方法在之前的章节中已经介绍，不再赘述。

3.5　小结

本章通过训练一个不定图像生成的模型向读者介绍了 Diffusion 模型一般的训练方法，包括数据集的加载方法、数据集的预处理、定义模型、训练模型、测试模型的方法等。

本章介绍的训练方法不是迁移学习，模型没有使用预训练参数而是从随机参数初始化的，这决定了任务的复杂度不能太高，否则计算量和训练难度会超出大多数计算设备的极限，在后续章节中会陆续介绍使用迁移学习训练的方法。

本章的任务顾名思义是不定图像的生成，图像的生成过程没有考虑文本特征，不符合以文生图、以图生图的一般定义，在后续章节中会陆续介绍以文生图、以图生图和以文改图等任务。