第 5 章 Dream Booth

5.1 任务简介

本章的任务名字叫作 Dream Booth，这是一种让 Diffusion 模型认识新的个体对象的训练方法。作为使用大数据量训练的 Diffusion 模型来讲，它的知识量是比较大的，但是还缺乏对具体个体的认知，例如它知道什么是哈士奇，什么是吉娃娃，但是并不知道我家邻居的狗是什么样的狗。有时也希望 Diffusion 模型能对具体的个体产生认知，例如我想整蛊公司的同事，但是 Diffusion 模型并不认识我的同事，它也不知道“小马哥”具体指的是谁。

本章要介绍的 Dream Booth 训练法就是让 Diffusion 模型认识具体的个体的训练方法。通过 Dream Booth 训练，Diffusion 将认识到当我说“小马哥”时指的是谁，或者“小点点”其实是指我邻居家的狗，并且它是一只斑点狗。

具体到本章的任务，将训练 Diffusion 模型认识一只小狗，它没有特定的名字，这里就叫它 little dog 好了。通过本章的训练，我希望当我说 little dog 时，Diffusion 能意识到我说的具体是哪条狗，它长什么样子等，具体的输入和输出在测试阶段会看得更加清楚。

5.2 数据集介绍

既然要让 Diffusion 模型认识具体的个例，当然要准备关于这个个例的图像数据，Dream Booth 对数据量的要求很低，即使只有很少的几张图像也能让 Diffusion 模型学习得很好。这是个好消息，因为关于个例的图像数据往往不那么容易收集。作为演示，本章关于个例的图像数据将只有 5 张，以此演示即使在数据量很少的情况下，依然可以得到较好的训练效果。

本章要让 Diffusion 模型认识到 little dog 的模样，使用 5 张图片来定义，这 5 张图片如图 5-1 所示。

图 5-1 中的 5 张图片定义了 little dog 是一只怎样的狗，它是一只橘黄色混杂白色可爱的小狗，经过本章的训练，期待 Diffusion 模型能记住它的模样。

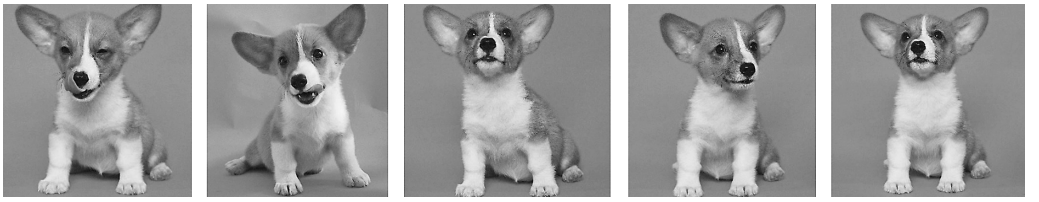

图 5-1　little dog 的图片

5.3　测试部分

和以往的任务一样，在训练之前先进行测试部分的工作，通过测试能更明确模型的输入、输出，也能更直观地理解本章任务的目的。

5.3.1　测试函数

上面已经通过图片定义了 little dog 的模样，但是 Diffusion 模型仅仅记住 little dog 的模样是不够的，还要能够控制 little dog 执行各种各样的动作，还能让它去全世界旅行，这样才能整蛊或者进行创作。

综上所述，写出如下测试函数，代码如下：

```
#第 5 章/定义测试函数
from diffusers import DiffusionPipeline
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)

    texts = [
        'A photo of little dog in a bucket', 'A photo of little dog swimming',
        'A photo of little dog sleeping', 'A photo of little dog in a doghouse'
    ]

    images = [pipeline(i).images[0] for i in texts]

    plt.figure(figsize=(20, 10))
    for i in range(4):
        plt.subplot(1, 4, i + 1)
        plt.imshow(images[i])
```

```
        plt.axis('off')

    plt.show()
```

在测试函数中定义了四条语句，分别是 little dog 在桶里、little dog 在游泳、little dog 在睡觉、little dog 在狗屋里。这四条语句的开头部分都是一样的，这和训练的过程有关系，在训练中是以一个固定的开头来训练的，所以在测试时也使用这个开头进行计算。这样 Diffusion 模型在看到这个固定的开头时，就意识到这里的 little dog 指的是数据集中定义的小狗，能够帮助它更好地完成任务。

5.3.2 未训练模型的测试结果

先来测试没有微调的模型，代码如下：

```
#第 5 章/测试训练前的模型
pipeline = DiffusionPipeline.from_pretrained('CompVis/stable-diffusion-v1-4',
                                             safety_checker=None)

test(pipeline)
```

在以上代码中从一个 checkpoint 加载了一个预训练的 Diffusion 模型。运行结果如图 5-2 所示。

图 5-2 未训练模型的测试结果

很显然此时的 Diffusion 模型还不知道这里的 little dog 具体指的是哪条狗，所以它只是胡乱地画了 4 只小狗，这并不符合训练数据集的定义。

5.3.3 训练后模型的测试结果

使用笔者训练好的模型进行测试的代码如下：

```
#第 5 章/在线加载笔者训练好的模型并测试
pipeline = DiffusionPipeline.from_pretrained(
    'lansinuote/diffusion.3.dream_booth', safety_checker=None)

test(pipeline)
```

上面这段代码从另一个 checkpoint 加载了一个 Diffusion 模型，这个模型就是笔者训练完成的模型，使用这个模型执行同样的测试，运行结果如图 5-3 所示。

图 5-3　训练后模型的测试结果

从图 5-3 可以看出，训练完成后的模型很好地认识到了 little dog 指的是哪条狗，它长什么样子等，并且能控制 little dog 做各种动作，去任何地方，虽然在训练数据集中 little dog 从未做过这些动作也从未去过这些地方。

以上就是本章的测试部分，通过测试读者能更直观地理解本章任务的输入和输出，并且可以使用测试函数检验模型训练的有效性、正确性，以确保整个实验不会跑偏。接下来进入训练部分。

5.4　训练部分

5.4.1　全局常量

首先需要定义两个全局常量，分别代表笔者上传到 HuggingFace 的数据集的 id 和预训练模型的 checkpoint，代码如下：

```
#第 5 章/全局常量
repo_id = 'lansinuote/diffusion.3.dream_booth'
checkpoint = 'CompVis/stable-diffusion-v1-4'
```

5.4.2　定义数据集

1. 加载数据集

由于本章的数据集体量很小，所以可以使用本地化加载，代码如下：

```
#第 5 章/加载数据集
from datasets import Dataset
import PIL.Image


def get_dataset():
    images = [{
```

```
        'image': PIL.Image.open('images/%d.jpeg' % i),
        'text': 'a photo of little dog',
    } for i in range(5)]

    return Dataset.from_list(images)


get_dataset()
```

上面这段代码将从本地文件夹 images 中加载的每张图片作为一条数据，数据集的体量非常小，只有 5 条，这 5 张图片就是本章想要教给 Diffusion 模型的 little dog 的图片。

Diffusion 模型是一个输入文字和输出图像的模型，因此每条数据中不仅要有图片，还要有一段对应的描述文本，这里使用了固定的文本，固定的文本和测试中的文本要相互对应，对于 Diffusion 模型来讲，以后看到这段文本就会意识到这段文本指代的对象就是这 5 张图片描述的对象。

以上代码的运行结果如下：

```
Dataset({
    features: ['image', 'text'],
    num_rows: 5
})
```

上面演示了本地化加载数据集的方法，这个数据集笔者也已上传到 HuggingFace 的数据集仓库中，可以使用 HuggingFace 的在线加载功能，代码如下：

```
#第 5 章/在线加载数据集
from datasets import load_dataset

dataset = load_dataset(path=repo_id, split='train')

dataset, dataset[0]
```

上面的代码和本地加载的数据集是等价的，运行结果如下：

```
(Dataset({
    features: ['image', 'text'],
    num_rows: 5
}),
{'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=2469x2558>,
  'text': 'a photo of little dog'})
```

以上两个加载数据集的方法读者只要选择其中之一即可，对于后续的任务来讲这两个数据集是等价的，没有区别。建议使用在线加载的方法，会更加简单，避免出现环境问题。

2. 数据集预处理

数据集加载完成了，但是现在的数据还都是抽象的图像和文本，还没有数字化，按照以往任务的检验可知这里应该对图像和文本进行编码了。

此外，由于训练的数据集规模非常小，只有 5 张图片，所以进行一定的数据增强是很有必要的，可以在数据集编码的同时进行数据增强，代码如下：

```
#第 5 章/数据集预处理
import torchvision
from transformers import AutoTokenizer

#数据增强
compose = torchvision.transforms.Compose([
    torchvision.transforms.Resize(
            512,
        interpolation=torchvision.transforms.InterpolationMode.BILINEAR),
    torchvision.transforms.RandomCrop(512),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.5], [0.5]),
])

#文字编码
tokenizer = AutoTokenizer.from_pretrained(checkpoint,
                                          subfolder='tokenizer',
                                          use_fast=False)


def f(data):
    #图像编码
    pixel_values = compose(data['image'][0]).unsqueeze(dim=0)

    #文字编码
    #77 = tokenizer.model_max_length
    tokens = tokenizer(data['text'][0],
                       truncation=True,
                       padding='max_length',
                       max_length=77,
                       return_tensors='pt')

    return {
        'pixel_values': pixel_values,
        'input_ids': tokens.input_ids,
        'attention_mask': tokens.attention_mask
    }
```

```
dataset = dataset.with_transform(f)

for k, v in dataset[0].items():
    print(k, v.shape, v.dtype)

dataset
```

运行结果如下：

```
pixel_values torch.Size([3, 512, 512]) torch.float32
input_ids torch.Size([77]) torch.int64
attention_mask torch.Size([77]) torch.int64

Dataset({
    features: ['image', 'text'],
    num_rows: 5
})
```

在上面的代码中首先对图形进行数据增强，然后把图像和文本都进行编码，编码的结果如下。

(1) pixel_values：图像的编码结果很明确地就是 3×512×512 的张量，很显然它代表了一张彩色的 512×512 尺寸的图像。

(2) input_ids：对文本进行分词、添加特殊符号、固定长度后把每个词映射为数字，也就是一串代表了原文本的数字。

(3) attention_mask：由于原文本需要固定长度，所以编码后的句子中可能会有很多 PAD，此时需要使用一个 MASK 来描述哪些位置是 PAD，MASK 的内容只有 0 和 1 两种值，它和 input_ids 内容是一一对应的。

3. 定义 loader

到这里为止，数据集准备好了，为了便于遍历，还需要定义一个 loader 工具，代码如下：

```
#第 5 章/定义 loader
import torch

loader = torch.utils.data.DataLoader(dataset,
                                     batch_size=1,
                                     shuffle=True,
                                     collate_fn=None)

for k, v in next(iter(loader)).items():
    print(k, v.shape, v.dtype)

len(loader)
```

运行结果如下：

```
pixel_values torch.Size([1, 3, 512, 512]) torch.float32
input_ids torch.Size([1, 77]) torch.int64
attention_mask torch.Size([1, 77]) torch.int64

5
```

可以看到 loader 使用的 batch size 很小，只有 1，这主要因为受限于显卡的显存，如果读者的运行环境有更大的显存，则可以适当调大 batch size。

经过 loader 的遍历，每条数据都在前面加上了一个 batch size 的维度，这样就可以放到 Diffusion 模型中进行计算了。

5.4.3 定义模型

数据集准备完毕后，现在可以加载 Diffusion 模型了，如前所述，本章使用的也是迁移学习，所以从一个 checkpoint 加载一个预训练的 Diffusion 模型，作为后续训练的基础，代码如下：

```
#第 5 章/加载模型
from transformers.models.clip.modeling_clip import CLIPTextModel
from diffusers import AutoencoderKL, UNet2DConditionModel

encoder = CLIPTextModel.from_pretrained(checkpoint, subfolder='text_encoder')
vae = AutoencoderKL.from_pretrained(checkpoint, subfolder='vae')
unet = UNet2DConditionModel.from_pretrained(checkpoint, subfolder='unet')

vae.requires_grad_(False)
encoder.requires_grad_(False)


def print_model_size(name, model):
    print(name, sum(i.numel() for i in model.parameters()) / 10000)


print_model_size('encoder', encoder)
print_model_size('vae', vae)
print_model_size('unet', unet)
```

运行结果如下：

```
encoder 12306.048
vae 8365.3863
unet 85952.0964
```

上面这段代码读者应该已经见过好几次了，相信读者已经不再陌生，本章要训练的模块是 U-Net 模型，而不是 Encoder 模型和 VAE 模型，所以把 Encoder 模型和 VAE 模型的参数锁定，不计算它们的梯度，只计算 U-Net 模型的梯度。

虽然 U-Net 模型的体量比较大，它的参数量多达 8 亿多个，但是本章的训练难度并不大，计算量也不算太大，读者不用太担心。

5.4.4 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 5 章/初始化工具类
from diffusers import DDPMScheduler

scheduler = DDPMScheduler.from_pretrained(checkpoint, subfolder='scheduler')

optimizer = torch.optim.AdamW(unet.parameters(),
                              lr=5e-6,
                              betas=(0.9, 0.999),
                              weight_decay=0.01,
                              eps=1e-8)

criterion = torch.nn.MSELoss()

scheduler, optimizer, criterion
```

初始化如下工具类：

(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) criterion 是计算 loss 的工具类。

5.4.5 计算 loss

本章计算 loss 的函数的代码如下：

```
#第 5 章/定义计算 loss 的函数
def get_loss(data):
    #编码文字
    #[1, 77] -> [1, 77, 768]
    out_encoder = encoder(input_ids=data['input_ids'],
                          attention_mask=data['attention_mask'])[0]

    #计算特征图
    #[1, 3, 512, 512] -> [1, 4, 64, 64]
    out_vae = vae.encode(data['pixel_values']).latent_dist.sample()
```

```
    #0.18215 = vae.config.scaling_factor
    out_vae = out_vae * 0.18215

    #随机噪声
    #[1, 4, 64, 64]
    noise = torch.randn_like(out_vae)

    #随机噪声步
    #1000 = scheduler.config.num_train_timesteps
    #1 = b
    noise_step = torch.randint(0, 1000, (1, ),
                               device=data['input_ids'].device).long()

    #添加噪声
    #[1, 4, 64, 64]
    out_vae_noise = scheduler.add_noise(out_vae, noise, noise_step)

    #从噪声图中把噪声计算出来
    #[1, 4, 64, 64],[1, 77, 768] -> [1, 4, 64, 64]
    out_unet = unet(out_vae_noise, noise_step, out_encoder).sample

    return criterion(out_unet, noise)


get_loss({
    'pixel_values': torch.randn(1, 3, 512, 512),
    'input_ids': torch.ones(1, 77).long(),
    'attention_mask': torch.ones(1, 77).long()
})
```

这个函数相信读者相当熟悉，本章计算 loss 的函数和上一个章节的函数非常类似，只有一处很小的修改，入参多了一个键 attention_mask，在文字编码时把这个键输入 Encoder 模型即可。Encoder 模型在编码过程中会忽略 PAD 的词的注意力，从而把注意力集中在文本本身。

代码的运行结果如下：

```
tensor(0.0036, grad_fn=<MseLossBackward0>)
```

5.4.6 训练

有了计算 loss 的函数，现在就可以进行训练了，代码如下：

```
#第 5 章/训练
```