大作业NeRF参考代码说明

环境配置

Python

本次作业需要使用python实现，这里推荐使用python3.8，可以登录官网安装。如果你有多个版本的python，选择3.x以上的版本即可。

依赖库

所需要的依赖库列在了requirement.txt当中，可以使用pip安装，pip是python自带的依赖库管理与安装工具。在项目目录下打开命令行或者powershell，运行下面的命令即可安装依赖库。

```
pip install -r requirement.txt
```

虽然本次作业依赖pytorch，但是并不需要cuda，所以可以安装只有cpu功能的pytorch。如果希望使用cuda编程来加速渲染速度，可以自行安装带cuda的pytorch版本。

Python IDE

没有严格的限制，使用vs code或者pycharm都可以。最好熟悉一下命令行的操作，不论用哪个IDE，命令行都是通用的。

测试渲染程序

安装好环境后，接下来可以测试渲染程序。

数据集和模型位置

数据集文件为cameras_sphere.npz，包含了渲染时的相机参数。模型文件为nerf_model.pth，包含了训练好的nerf模型。接下来需要新建dataset文件夹，在dataset下再新建test文件夹，把数据集文件放进去即可。模型文件可直接放在项目目录下:

```
Project:.
|  .gitignore
|  nerf_model.pth
|  run_nerf.py
|
├─confs
|      nerf.conf
|
├─dataset
|  └─test
|          cameras_sphere.npz
|
└─models
        embedder.py
        fields.py
        my_dataset.py
        my_nerf.py
        my_renderer.py
```

然后在 `run_nerf.py` 中将50行附近的载入模型路径改为 `nerf_model.pth` 的绝对路径。

```
self.load_checkpoint(r'D:\share\WYT-C\nerf_ds\stu\nerf_model.pth', absolute=True)
```

运行代码

运行代码指令为

```
python run_nerf.py --mode test --conf confs/nerf.conf --case test
```

在cpu模式下，等待1-10分钟后，将在 `exp/test/render` 文件夹下生成渲染图片 `0.jpg`：

该过程总共会渲染90个视角的图片，连起来可形成一个自转一圈的视频。由于渲染一张图片后就说明程序可以跑通，所以可以渲染成功后可直接中断程序。

代码原理

在test模式下，程序会调用 `CheatNeRF` 类，这个类会直接调用NeRF来渲染，所以渲染速度会非常慢。

基础知识

接下来介绍python以及pytorch一些基础的操作，方便实现后续功能。

声明数组

pytorch和numpy等python库的数组可以直接声明，且是固定大小，例如：

```
a = torch.zeros(128, 32, 3, device=xxx)
a = torch.zeros((128, 32, 3), device=xxx)
a = torch.zeros((128, 32, 3))
```

这两种声明方式等价，都是声明一个全0的128x32x3的三维浮点类型数组，device可以不管，因为我们这里只用cpu，不涉及gpu和cpu的转换，可以留空（默认cpu）。

在MyNeRF中，我们可以直接声明2个数组来保存密度和颜色：

```
self.volume_sigma = torch.zeros((128, 128, 128))
self.volume_color = torch.zeros((128, 128, 128, 3))
```

这里关于类为什么用self可以看这个教程。 我们可以通过 `.shape` 来获取一个pytroch数组的维度大小：

```
X = self.volume_sigma.shape[0] # 128
X, Y, Z, D = self.volume_color.shape # 128 128 128 3
```

此外，我们可以使用torch.XXXTensor来将某个列表转换成Pytorch的数组，在Pytorch中，转换是必须的，因为所有的Pytorch运算都要求其类型为pytorch数组。

```
a = [0, 1, 3, 5]
b = torch.FloatTensor(a)
c = torch.LongTensor(a)
d = b + c
```

数组运算

这一部分内容具体可以看pytorch的教程，这里简单概括一下就是当数组维度相同时， `+-*/` 这四类运算都是逐个对应元素去做的，以加法为例:

```
a = np.zeros(128, 128)
b = np.zeros(128, 128)
c = np.zeros(128, 128)
# C++
for i in range(128):
    for j in range(128):
        c[i, j] = a[i, j] + b[i, j]
# Pytorch
c = a + b
```

Pytorch的一次整体的数组运算通常要比C++写法快10~100倍，所以如果可以用整体的数组运算实现，就尽量用pytorch的方式计算。

快速索引

python和c++最大的不同就是支持快速索引，以下有两种方式可以实现数组a加1

```
# C++
a = torch.zeros(512)
for i in range(512):
    a[i] = a[i] + 1
# Pytorch
a[:] += 1

# C++
a = torch.zeros(512, 512)
for i in range(512):
    for j in range(512):
        a[i, j] = a[i, j] + 1
# Pytorch
a[:, :] += 1


# C++
a = torch.zeros(512, 512)
for i in range(66, 400):
    a[i, 3] = a[i, 3] + 1
# Pytorch
a[66:400, 3] += 1
```

使用 : 符号可实现快速索引，两边不带数字为全部索引，左边带数字规定起始，右边带数字规定末尾（但不包括末尾）。快速索引在python中会大大提升代码运行速度，提升幅度约为10~100倍，所以如果能用快速索引就不要用for循环。

另外快速索引可以结合数组运算一起进行，可以实现两个数组间指定元素的加减乘除。

高级快速索引

在Pytorch中，除了使用 : 外，还可以自定义索引的值：

```
index = torch.LongTensor([0, 1, 3, 5, 7])
value = torch.FloatTensor([0, 1, 2, 3, 4])
a = torch.zeros(10)
# c++
for i in range(index.shape[0]):
    a[index[i]] = value[i]
# Pytorch
a[index] = value
```

这里就可以把数组a中元素位置在0,1,3,5,7的值改为0,1,2,3,4。同样的，这种索引方式要远远快于for循环。但是要注意这种索引只有index类型为Long时才可行。运用高级快速索引实现快速查询的参考代码如下：

```
N, _ = pts_xyz.shape
sigma = torch.zeros(N, 1, device=pts_xyz.device)
color = torch.zeros(N, 3, device=pts_xyz.device)
X_index = ((pts_xyz[:, 0] + 0.125) * 4 * 128).clamp(0, 127).long()
Y_index = ((pts_xyz[:, 1] - 0.75) * 4 * 128).clamp(0, 127).long()
Z_index = ((pts_xyz[:, 2] + 0.125) * 4 * 128).clamp(0, 127).long()
sigma[:, 0] = self.volumes_sigma[X_index, Y_index, Z_index]
color[:, :] = self.volumes_color[X_index, Y_index, Z_index]
return sigma, color
```

这里clamp会将所有小于0的值变成0，所有大于127的值变成127。

渲染过程

测试渲染通过后，我们需要实现本作业的基础目标。基础目标渲染代码的运行方式为:

```
python run_nerf.py --mode render --conf confs/nerf.conf --case test
```

整个渲染分成两个过程，一个是将nerf的属性保存到数据结构中的save过程，一个是给定点的位置，查询nerf属性的查询过程。

save过程

save过程的主体在 `run_nerf.py` 当中，它会密集地在[-0.125, 0.75, -0.125]到[0.125, 1.0, 0.125]这个范围内采集 `RSxRSxRS` 个点，并询问神经网络每个点的属性。询问后会调用 `models/my_nerf.py` 中 `MyNeRF` 的save函数，你需要实现save函数，将这些点的属性保存到你的数据结构中。初始时 `RS` 为64，此时由于采样分辨率过低，按照这样渲染出来的图像会有很多噪声，不过这样运行速度会很快，方便调试。等跑通后可以将 `RS` 改为128，就可以渲染一个正常的结果。

query过程

你需要实现 `models/my_nerf.py` 中MyNeRF的query函数，它的输入pts_xyz为Nx3的浮点类型数组代表每个点的位置，你需要输出一个Nx1的数组sigma代表每个点的密度，Nx3的数组color代表每个点的颜色。注意这里点的位置是实数，所以很大概率和save过程采的点不一样，可以按照高级快速索引中给的代码，求取整后的结果。当然也可以找最近点。

渲染参数

渲染参数当中一个比较重要的参数是 `resolution_level` ，它可以填 `1,2,4` ，填1时会渲染512x512分辨率的图像，填2时是256x256，填4时是128x128。初始调试的时候尽量填4，渲染更低分辨率的图像速度会快很多。等没有bug时再调成1或者2来提高渲染质量。另外如果没有实现进阶目标，因为我们数据结构的数组大小只有128x128x128，所以填2或者1可能也没有实质作用。

保存和读取

如果每次主程序的save过程都很耗时，也可以通过torch.save来直接保存数据结构。保存数据结构到硬盘以后，可以跳过主程序的save过程，从硬盘读取来做渲染。

```
# save to the disk
checkpoint = {
    "volume_sigma": self.volume_sigma,
    "volume_color": self.volume_color
}
torch.save(checkpoint, "temp.pth")
# load from the disk
checkpoint = torch.load("temp.pth")
self.volume_sigma = checkpoint["volume_sigma"]
self.volume_color = checkpoint["volume_color"]
```