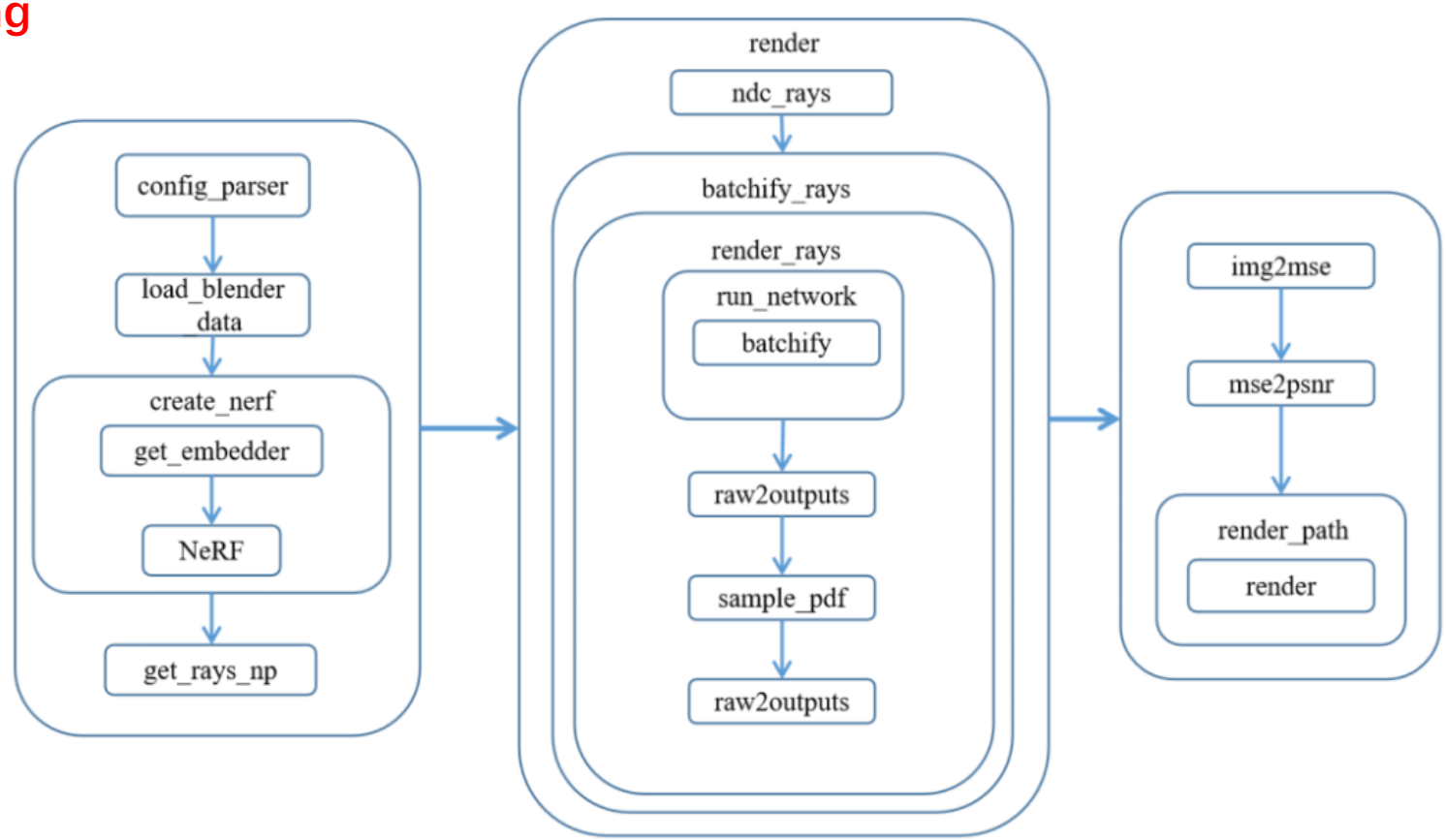
NeRF源码解读-dezeming



本源码解读包含了多个网上解读博客，以及本人的解读和一些对一些逻辑和细节的补充说明。后面部分关于NeRF部分的基本都是来自本人的解读。

源码地址： https://github.com/yenchenlin/nerf-pytorch

参考1： https://zhuanlan.zhihu.com/p/593204605/

参考2： https://blog.csdn.net/qq_35831906/article/details/138251427

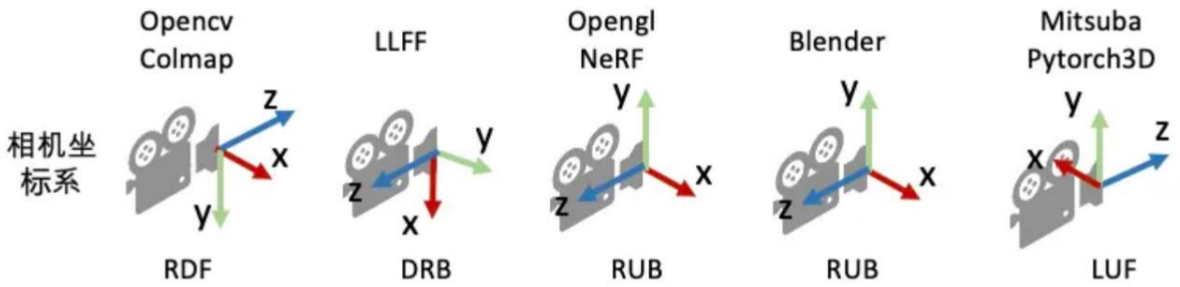
参考3： 【三维重建】NeRF原理+代码讲解_深度学习_杀生丸学AI-GitCode 开源社区 (csdn.net)

理解相机变换矩阵c2w

若要训练NeRF，我们需要知道相机的位姿，这样，当相机发射一条射线以后才能知道这个射线如何变换到世界坐标系中进行采样。

相机的位姿和内参信息如下：

坐标系定义： 为了唯一地描述每一个空间点的坐标，以及相机的位置和朝向，我们需要先定义一个世界坐标系。一个坐标系其实就是由原点的位置与XYZ轴的方向决定。接着，为了建立3D空间点到相机平面的映射关系以及多个相机之间的相对关系，我们会对每一个相机定义一个局部的相机坐标系。下图为常见的坐标系定义习惯。



常见的相机坐标系定义习惯（右手坐标系）。注意：在OpenCV/COLMAP的相机坐标系里相机朝向+z轴，在LLFF/NeRF的相机坐标系中里相机朝向-z轴。有时我们会按坐标系的xyz朝向描述坐标系，如OpenCV/COLMAP里使用的RDF表述X轴指向right，Y轴指向Down，Z轴指向Foward。

相机外参的逆矩阵被称为**camera-to-world (c2w)矩阵**，其作用是把相机坐标系的点变换到世界坐标系。因为NeRF主要使用c2w，这里详细介绍一下c2w的含义。c2w矩阵是一个4x4的矩阵，左上角3x3是旋转矩阵R，右上角的3x1向量是平移向量T。有时写的时候可以忽略最后一行[0,0,0,1]。

$$\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad [R \quad T] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$$

Camera-to-world (c2w) 矩阵

刚刚接触的时候，对这个c2w矩阵的值可能会比较陌生。其实c2w矩阵的值直接描述了相机坐标系的朝向和原点：

$$[R \quad T] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$$

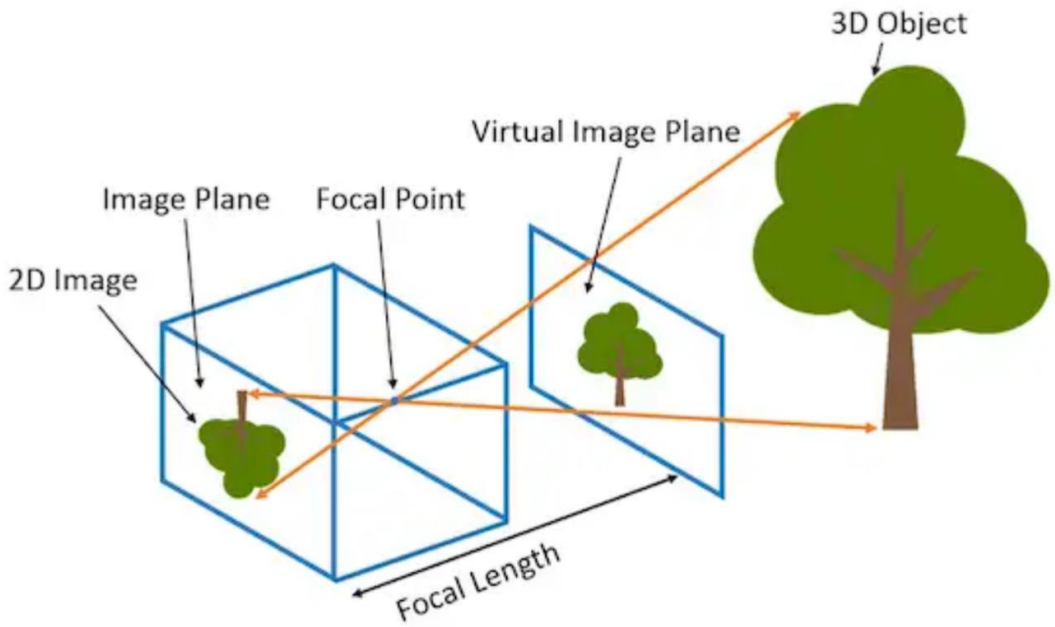
X axis　Y axis　Z axis　camera center

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

理解相机内参

相机内参

刚刚介绍了相机的外参，现在简单介绍一下相机的内参。



解读来自：
https://zhuanlan.zhihu.com/p/593204605/

相机的内参矩阵将相机坐标系下的3D坐标映射到2D的图像平面，这里以针孔相机(Pinhole camera)为例介绍相机的内参矩阵K：

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

针孔相机的内参矩阵

内参矩阵K包含4个值，其中fx和fy是相机的水平和垂直**焦距**（对于理想的针孔相机，fx=fy）。焦距的物理含义是相机中心到成像平面的距离，长度以像素为单位。cx和cy是图像原点相对于相机光心的水平和垂直偏移量。cx，cy有时候可以用图像宽和高的1/2近似：

```
#  NeRF run_nerf.py有这么一段构造K的代码
    if K is None:
        K = np.array([
            [focal, 0, 0.5*W],
            [0, focal, 0.5*H],
            [0, 0, 1]
        ])
```

后面还会介绍这个内参矩阵K

理解COLMAP到LLFF的过程

LLFF真实数据格式

NeRF代码里用load_llff.py这个文件来读取真实的数据，第一次看到LLFF这个词可能不知道是什么意思。其实LLFF GitHub - Fyusion/LLFF: Code release for Local Light Field Fusion at SIGGRAPH 2019 是NeRF作者的上一篇做新视角合成的工作。为了和LLFF方法保持一致的数据格式，NeRF使用load_llff.py读取LLFF格式的真实数据，并建议大家使用LLFF提供的的imgs2poses.py文件获取所需相机参数。

COLMAP到LLFF数据格式

imgs2poses.py这个文件其实很简单，就干了两件事。

- 第一件事是调用colmap软件估计相机的参数，在sparse/0/文件夹下生成一些二进制文件：cameras.bin, images.bin, points3D.bin, project.ini。
- 第二件事是读取上一步得到的二进制文件，保存成一个poses_bounds.npy文件。

这里有一个细节需要注意，就是在pose_utils.py文件里load_colmap_data()函数的倒数第二行，有一个操作将colmap得到的c2w旋转矩阵中的第一列和第二列互换，第三列乘以负号：

```
# LLFF/llff/poses/pose_utils.py
def load_colmap_data(realdir):
    ...
    # must switch to [-u, r, -t] from [r, -u, t], NOT [r, u, -t]
    poses = np.concatenate([poses[:, 1:2, :], poses[:, 0:1, :], -poses[:, 2:3, :], pos
    return poses, pts3d, perm
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

还记得刚刚提到c2w旋转矩阵的三列向量分别代表XYZ轴的朝向，上述操作实际上就是把相机坐标系轴的朝向进行了变换：X和Y轴调换，Z轴取反，如下图所示：

从Colmap的坐标系转到LLFF的坐标系

poses_bounds.npy里有什么

load_llff.py会直接读取poses_bounds.npy文件获得相机参数。poses_bounds.npy是一个Nx17的矩阵，其中N是图像的数量，即每一张图像有17个参数。其中前面15个参数可以重排成3x5的矩阵形式：

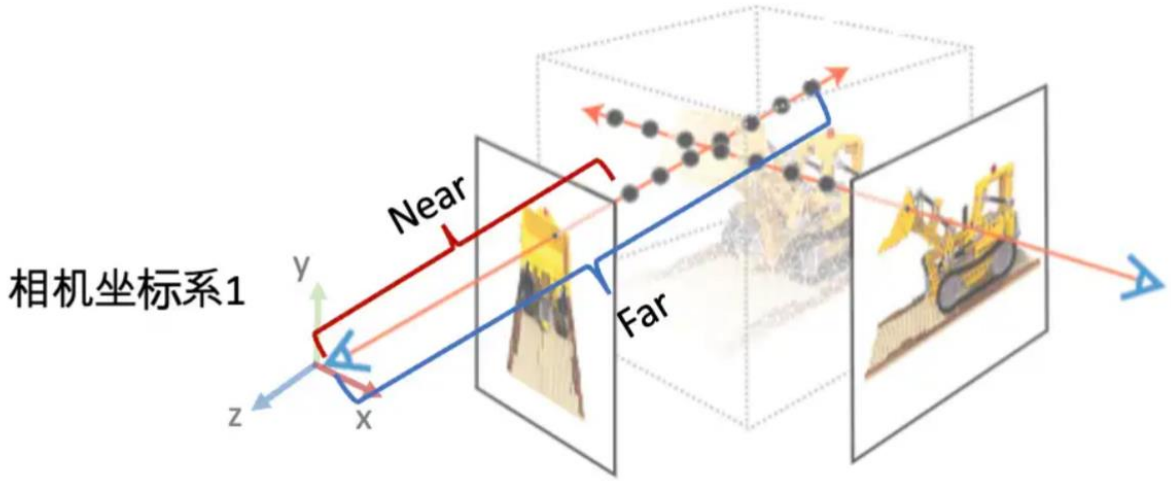
$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 & H \\ r_{21} & r_{22} & r_{23} & t_2 & W \\ r_{31} & r_{32} & r_{33} & t_3 & f \end{bmatrix}$$

poses_bounds.npy的前15维参数。左边3x3矩阵是c2w的旋转矩阵，第四列是c2w的平移向量，第五列分别是图像的高H、宽W和相机的焦距f

最后两个参数用于表示场景的范围**Bounds (bds)**，是该相机视角下场景点离相机中心最近(near)和最远(far)的距离，所以near/far肯定是大于0的。

理解COLMAP到LLFF的过程

- 这两个值是怎么得到的？是在imgs2poses.py中，计算colmap重建的**3D稀疏点**在各个相机视角下最近和最远的距离得到的。
- 这两个值有什么用？之前提到体素渲染需要在一条射线上采样3D点，这就需要一个采样区间，而near和far就是定义了采样区间的最近点和最远点。贴近场景边界的near/far可以使采样点分布更加密集，从而有效地提升收敛速度和渲染质量。



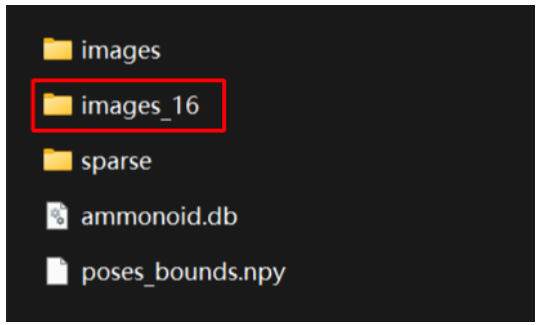
poses_bounds.npy里最后两个参数（near/far）的作用示意图

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

load_llff.py文件：_minify函数

```
3 usages   SirinL *
def _minify(basedir, factors=[], resolutions=[]):
```

该函数顾名思义，将目录下的图片降分辨率，降低X倍，则存储到image_X目录下。比如降低16倍分辨率：



load_llff.py文件：_load_data函数

```
1 usage  SirinL
def _load_data(basedir, factor=None, width=None, height=None, load_imgs=True):
```

该函数就是从poses_bounds.npy文件中获得各种相机位姿和图像信息参数。

然后根据factor参数或者指定的图像宽和高来选择是否降低分辨采样。

之后根据load_imgs参数选择是否把图像加载进来。如果确定要加载图像，就加载，否则就不加载。

注意，images目录下的图像在使用Colmap标定后便不能增加或减少，也不能改名。我们读到的相机位姿都是按图像名称顺序排列的，因此不能打乱顺序。图像数不一致就会报错：

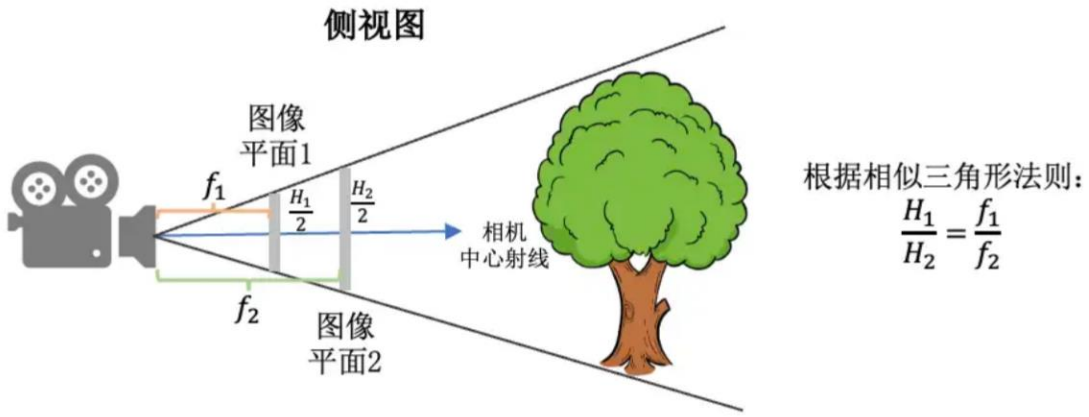
```
imgfiles = [os.path.join(imgdir, f) for f in sorted(os.listdir(imgdir)) if f.endswith('JPG') or f.endswith('jpg') or f.endswith('png')]
if poses.shape[-1] != len(imgfiles):
    print( 'Mismatch between imgs {} and poses {} !!!!'.format(len(imgfiles), poses.shape[-1]) )
    return
```

load_llff.py文件：_load_data函数

缩放图像需要修改什么相机参数?

在_load_data()函数里，有一个用于图像缩放的factor比例参数，将HxW的图像缩放成(H/factor)x(W/factor)。这里面有一个问题是如果缩放了图像尺寸，相机的参数需要相应的做什么变化?

- 做法是：**外参（位置和朝向）不变，相机的焦距f，cx, 和cy等比例缩放。**下图的示意图展示了当相机位置不变，相机视野(Field of view, FOV)不变的情况下，图像的高和焦距长短的关系。



根据相似三角形法则：

$$\frac{H_1}{H_2} = \frac{f_1}{f_2}$$

图像平面1与图像平面2拍摄的图像内容是一样的，只是分辨率不同

下面这段代码就是把下图的H和W换成降分辨率以后的，然后再把focal除以factor:

```
sh = imageio.imread(imgfiles[0]).shape # resize以后的图像的 (高,宽,通道数)
# poses.shape: (3,5,图像数)
poses[:2, 4, :] = np.array(sh[:2]).reshape([2, 1]) # 把 高X宽 变为2X1矩阵，赋值给poses
poses[2, 4, :] = poses[2, 4, :] * 1./factor
```

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 & H \\ r_{21} & r_{22} & r_{23} & t_2 & W \\ r_{31} & r_{32} & r_{33} & t_3 & f \end{bmatrix}$$

加载以后得到poses，bds和根据load_imgs参数决定是否读取的imgs:

```
print('Loaded image data', imgs.shape, poses[:,-1,0])
return poses, bds, imgs
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

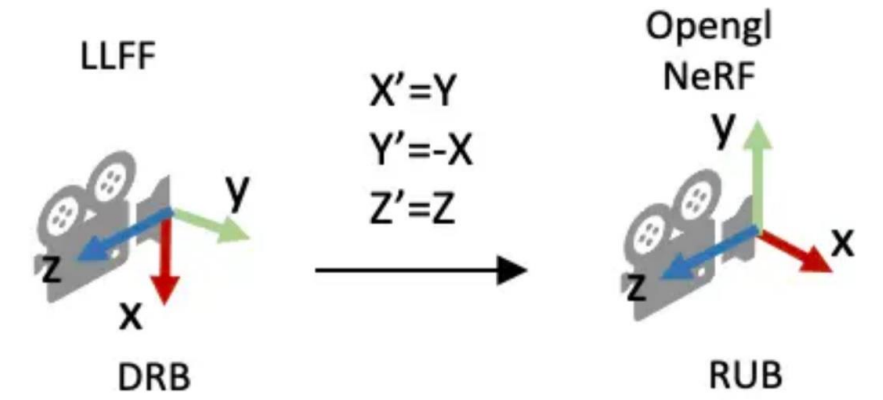
load_llff.py文件

DRB到RUB的变换

第一个疑问是，为什么读进poses_bounds.npy里的c2w矩阵之后，对c2w的旋转矩阵又做了一些列变换?

```
# load_llff.py文件
def load_llff_data(basedir, factor=8, recenter=True, bd_factor=.75, spherify=False, pa

    poses, bds, imgs = _load_data(basedir, factor=factor) # factor=8 downsamples origi
    print('Loaded', basedir, bds.min(), bds.max())

    # Correct rotation matrix ordering and move variable dim to axis 0
    poses = np.concatenate([poses[:, 1:2, :], -poses[:, 0:1, :], poses[:, 2:, :]], 1)
    ...
```

上面的代码段的最后一行实际上是把旋转矩阵的第一列（X轴）和第二列（Y轴）互换，并且对第二列（Y轴）做了一个反向。这样做的目的是将LLFF的相机坐标系变成OpenGL/NeRF的相机坐标系，如下图所示。



poses = np.concatenate([poses[:, 1:2, :], -poses[:, 0:1, :], poses[:, 2:, :]], 1)

然后使用moveaxis改变一下形状，从(高X宽X图像数)变为(图像数X高X宽)：

```
# 将 poses 数组的最后一个轴移动到第一个位置
poses = np.moveaxis(poses, -1, 0).astype(np.float32) # 变为(图像数 X 3 X 5)
imgs = np.moveaxis(imgs, -1, 0).astype(np.float32)
images = imgs
bds = np.moveaxis(bds, -1, 0).astype(np.float32)
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

load_llff.py文件

viewmatrix()

view_matrix是一个构造相机矩阵的的函数，输入是相机的**Z轴朝向**、**up轴的朝向**(即相机平面朝上的方向Y)、以及**相机中心**。输出下图所示的camera-to-world (c2w)矩阵。因为Z轴朝向，Y轴朝向，和相机中心都已经给定，所以只需求X轴的方向即可。又由于X轴同时和Z轴和Y轴垂直，我们可以用Y轴与Z轴的叉乘得到X轴方向。

$$[R \quad T] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$$

X axis　Y axis　Z axis　camera center

camera-to-world matrix

下面是load_llff.py里关于view_matrix()的定义，看起来复杂一些。其实就是比刚刚的描述比多了一步：在用Y轴与Z轴叉乘得到X轴后，再次用Z轴与X轴叉乘得到新的Y轴。为什么这么做呢？这是因为传入的up(Y)轴是通过一些计算得到的，不一定和Z轴垂直，所以多这么一步。

```
# load_llff.py
def viewmatrix(z, up, pos):
    vec2 = normalize(z)
    vec1_avg = up
    vec0 = normalize(np.cross(vec1_avg, vec2))
    vec1 = normalize(np.cross(vec2, vec0))
    m = np.stack([vec0, vec1, vec2, pos], 1)
    return m
```

把点集pts变换到相机坐标系（**这个函数并没有用到**）：

```
SirinL *
def ptstocam(pts, c2w): # 将给定的点集 pts 变换到世界坐标系
    tt = np.matmul(c2w[:3,:3].T, (pts-c2w[:3,3])[...,np.newaxis])[...,0]
    return tt
```

(pts-c2w[:3,3])　点平移

旋转矩阵的逆变换等于转置。

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

load_llff.py文件

poses_avg()

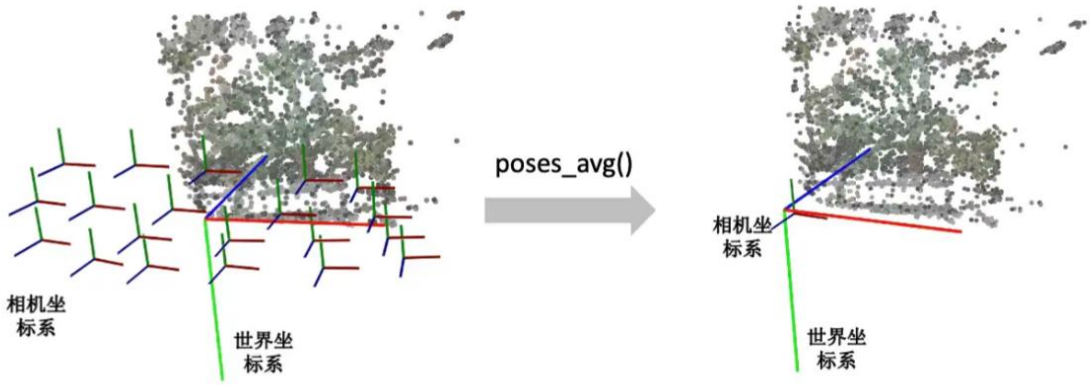
这个函数其实很简单，顾名思义就是多个相机的平均位姿（包括位置和朝向）。输入是多个相机的位姿。

- 第一步对多个相机的中心进行求均值得到**center**。
- 第二步对所有相机的Z轴求平均得到**vec2**向量（方向向量相加其实等效于平均方向向量）。
- 第三步对所有的相机的Y轴求平均得到**up**向量。
- 最后将vec2, up, 和center输入到刚刚介绍的viewmatrix()函数就可以得到平均的相机位姿了。

```
def poses_avg(poses):

    hwf = poses[0, :3, -1:]

    center = poses[:, :3, 3].mean(0)
    vec2 = normalize(poses[:, :3, 2].sum(0))
    up = poses[:, :3, 1].sum(0)
    c2w = np.concatenate([viewmatrix(vec2, up, center), hwf], 1)

    return c2w
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

下图展示了一个poses_avg()函数的例子。左边是多个输入相机的位姿，右边是返回的平均相机姿态。可以看出平均相机位姿的位置和朝向是之前所有相机的均值。



中间大的坐标系是世界坐标系，每一个小的坐标系对应一个相机的局部坐标系。红绿蓝(RGB)轴分别代表XYZ轴

load_llff.py文件

recenter_poses()

recenter_poses()函数的名字听起来是中心化相机位姿（同样包括位置和朝向）的意思。输入N个相机位姿，会返回N个相机位姿。

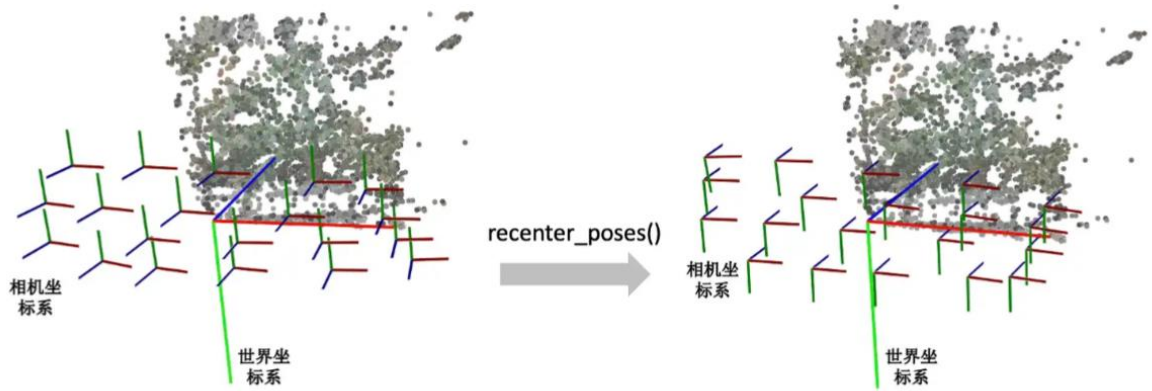
具体的操作了解起来可能有点跳跃。第一步先用刚刚介绍的poses_avg(poses)得到多个输入相机的平均位姿c2w，接着用这个平均位姿c2w的逆左乘到输入的相机位姿上就完成了归一化。

```
def recenter_poses(poses):

    poses_ = poses+0
    bottom = np.reshape([0,0,0,1.], [1,4])
    c2w = poses_avg(poses)
    c2w = np.concatenate([c2w[:3,:4], bottom], -2)
    bottom = np.tile(np.reshape(bottom, [1,1,4]), [poses.shape[0],1,1])
    poses = np.concatenate([poses[:,:3,:4], bottom], -2)

    poses = np.linalg.inv(c2w) @ poses
    poses_[:,:3,:4] = poses[:,:3,:4]
    poses = poses_
    return poses
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

首先我们要知道利用同一个旋转平移变换矩阵左乘所有的相机位姿是对所有的相机位姿做一个**全局的旋转平移变换**，那下一个问题就是这些相机会被变到什么样的一个位置？我们可以用平均相机位姿作为支点理解，如果把平均位姿的逆c2w^-1左乘平均相机位姿c2w，返回的相机位姿中旋转矩阵为单位矩阵，平移量为零向量。也就是变换后的平均相机位姿的位置处在世界坐标系的原点，XYZ轴朝向和世界坐标系的向一致。

下图我们用一个例子帮助理解。左边和右边分别是输入和输出的相机位姿示意图。我们可以看到变换后的多个相机的平均位姿处在世界坐标系的原点，并且相机坐标系的XYZ轴与世界坐标系保持一致了。



中间大的坐标系是世界坐标系，每一个小的坐标系对应一个相机的局部坐标系。红绿蓝(RGB)轴分别代表XYZ轴

load_llff.py文件

render_path_spiral() **生成观察轨迹**

这个函数写的有点复杂，它和模型训练没有关系，主要是用来生成一个相机轨迹用于新视角的合成

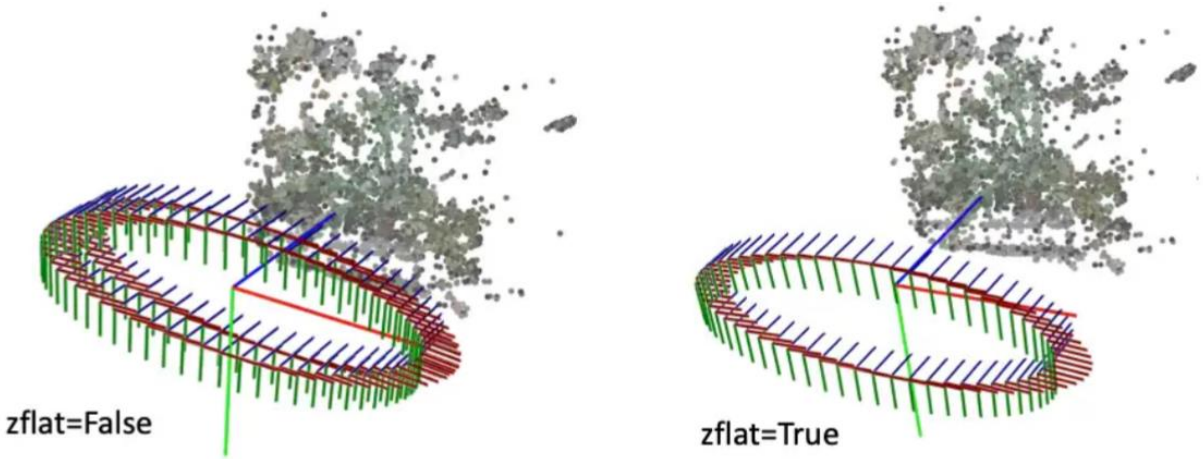
```
def render_path_spiral(c2w, up, rads, focal, zdelta, zrate, rots, N):
    render_poses = []
    rads = np.array(list(rads) + [1.])
    hwf = c2w[:,4:5]

    for theta in np.linspace(0., 2. * np.pi * rots, N+1)[:-1]:
        c = np.dot(c2w[:3,:4], np.array([np.cos(theta), -np.sin(theta), -np.sin(theta*
        z = normalize(c - np.dot(c2w[:3,:4], np.array([0,0,-focal, 1.])))
        render_poses.append(np.concatenate([viewmatrix(z, up, c), hwf], 1))
    return render_poses
```

需要知道这个函数它是想生成一段螺旋式的相机轨迹，相机绕着一个轴旋转，其中相机始终注视着一个焦点，相机的up轴保持不变。简单说一下上面的代码：

首先是一个for循环，每一迭代生成一个新的相机位置。c是当前迭代的相机在世界坐标系的位置，np.dot(c2w[:3,:4], np.array([0,0,-focal, 1.])是焦点在世界坐标系的位置，z是相机z轴在世界坐标系的朝向。接着使用介绍的viewmatrix(z, up, c)构造当前相机的矩阵。

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

下面这个图可视化了 render_path_spiral()生成的轨迹。



中间大的坐标系是世界坐标系，每一个小的坐标系对应一个相机的局部坐标系。红绿蓝(RGB)轴分别代表XYZ轴

load_llff.py文件

spherify_poses()　　生成观察轨迹

刚刚介绍的render_path_spiral()假设所有相机都朝向某一个方向，也就是所谓的faceforward场景。对于相机围绕着一个物体拍摄的360度场景，NeRF代码提供了一个spherify_poses()的函数用于"球面化"相机分布并返回一个环绕的相机轨迹用于新视角合成。这里插一句，在训练360度场景的时候，需要配合"--no_ndc --spherify --lindisp"三个参数以得到好的结果，具体原理这里不展开介绍。

```
if spherify:
    poses, render_poses, bds = spherify_poses(poses, bds)
```

这个函数也比较复杂，前半部分是在将输入的相机参数进行归一化，后半部分是生成一段相机轨迹用于合成新视角。对输入相机参数进行归一化时，思路是：

- 用 pt_mindist = min_line_dist(rays_o, rays_d)找到离所有相机中心射线距离之和最短的点（可以先简单理解成场景的中心位置）

```
rays_d = poses[:,:3,2:3]
rays_o = poses[:,:3,3:4]

def min_line_dist(rays_o, rays_d):
    A_i = np.eye(3) - rays_d * np.transpose(rays_d, [0,2,1])
    b_i = -A_i @ rays_o
    pt_mindist = np.squeeze(-np.linalg.inv((np.transpose(A_i, [0,2,1]) @ A_i).mean
    return pt_mindist

pt_mindist = min_line_dist(rays_o, rays_d)
```

- 将得到的场景中心位置移到世界坐标系的原点，同时将所有相机z轴的平均方向转到和世界坐标系的z轴相同

```
center = pt_mindist
up = (poses[:,:3,3] - center).mean(0)

vec0 = normalize(up)
vec1 = normalize(np.cross([.1,.2,.3], vec0))
vec2 = normalize(np.cross(vec0, vec1))
pos = center
c2w = np.stack([vec1, vec2, vec0, pos], 1)

poses_reset = np.linalg.inv(p34_to_44(c2w[None])) @ p34_to_44(poses[:,:3,:4])
```

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

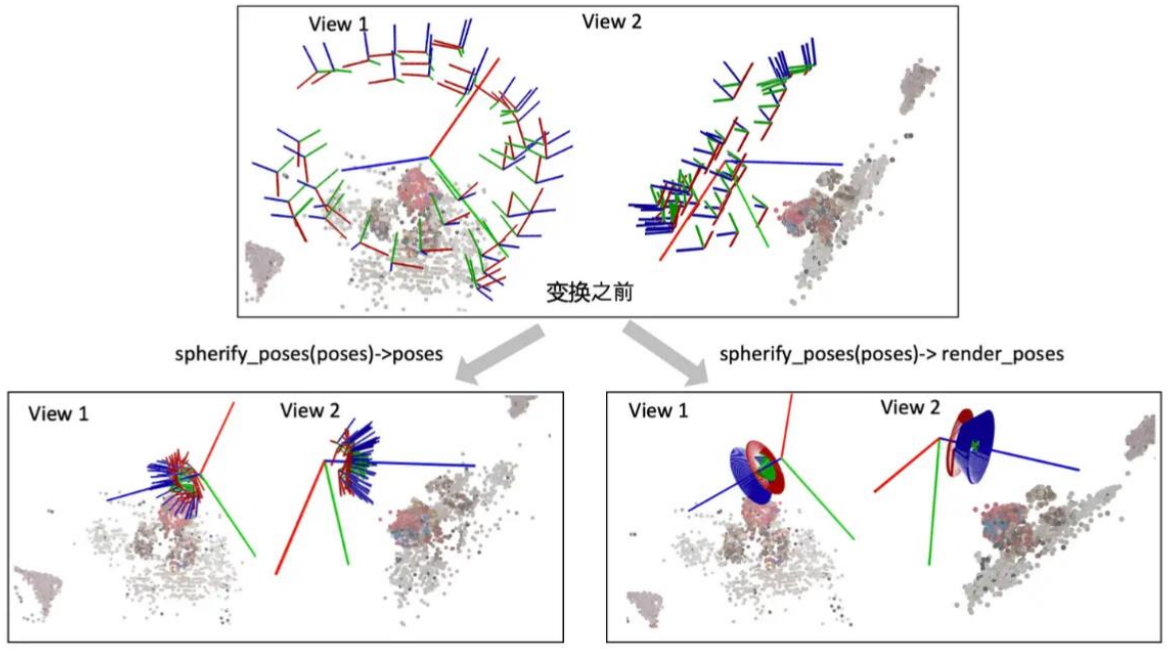
load_llff.py文件

- 最后将相机的位置缩放到单位圆内

生成观察轨迹

```
rad = np.sqrt(np.mean(np.sum(np.square(poses_reset[:,:3,3]), -1)))
sc = 1./rad
poses_reset[:,:3,3] *= sc
```

下面这个图可视化了spherify_poses()返回的结果。



中间大的坐标系是世界坐标系，每一个小的坐标系对应一个相机的局部坐标系。红绿蓝(RGB)轴分别代表XYZ轴

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

load_llff_data函数还有最后一段代码需要讲解一下：

```
dists = np.sum(np.square(c2w[:3,3] - poses[:,:3,3]), -1)
i_test = np.argmin(dists)
print('HOLDOUT view is', i_test)
```

i_test 是一个通过计算得到的索引，代表了从一组姿态（poses）中找出与平均姿态（c2w，通常是相机位置的平均值）最接近的一个姿态。这通常用于留出数据集中的一个视图作为验证或测试，确保这个视图在空间上与训练数据相似，但不完全相同，用以评估模型的泛化能力。

load_blender.py文件

加载blender数据的过程解读可见：

【三维重建】NeRF原理+代码讲解_深度学习_杀生丸学AI-GitCode 开源社区 (csdn.net)

run_nerf_helpers.py文件

```
1 usage   SirinL
def get_rays_np(H, W, K, c2w):...
```

输入参数：图像的长、宽和内参矩阵K

3D空间射线怎么构造

最后我们看一下这个射线是怎么构造的。**给定一张图像的一个像素点，我们的目标是构造以相机中心为起始点，经过相机中心和像素点的射线。**

首先，明确两件事：

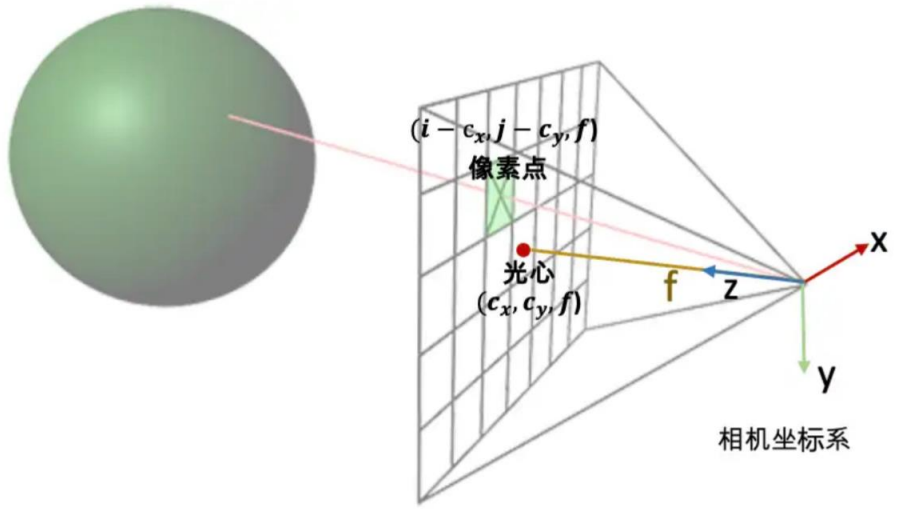
1. 一条射线包括一个起始点和一个方向，起点的话就是相机中心。对于射线方向，我们都知道两点确定一条直线，所以除了相机中心我们还需另一个点，而这个点就是成像平面的像素点。
2. NeRF代码是在相机坐标系下构建射线，然后再通过camera-to-world (c2w)矩阵将射线变换到世界坐标系。

通过上述的讨论，我们第一步是要先写出相机中心和像素点在相机坐标系的3D坐标。下面我们以OpenCV/Colmap的相机坐标系为例介绍。相机中心的坐标很明显就是[0,0,0]了。像素点的坐标可能复杂一点：首先3D像素点的x和y坐标是2D的图像坐标 (i, j)减去光心坐标 (cx,cy)，然后z坐标其实就是焦距f (因为图像平面距离相机中心的距离就是焦距f)。

所以我们就可以得到射线的方向向量是 $(i-c_x, j-c_y, f)-(0,0,0)=(i-c_x, j-c_y, f)$ 。因为是向量，我们可以把整个向量除以焦距f归一化z坐标，得到 $(\frac{i-c_x}{f}, \frac{j-c_y}{f}, 1)$ 。

解读来自：
https://zhuanlan.zhihu.com/p/593204605/

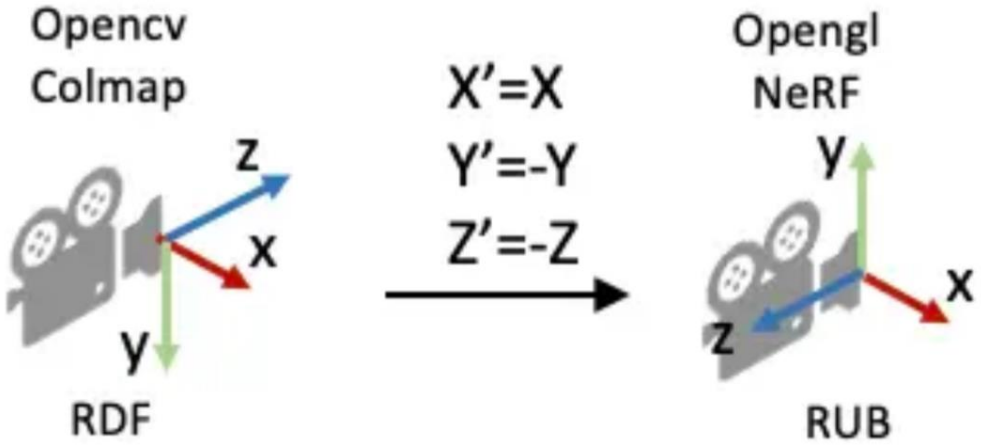
接着只需要用c2w矩阵把相机坐标系下的相机中心和射线方向变换到世界坐标系就搞定了。



OpenCV/Colmap相机坐标系下射线的构造示意图

函数的第二行中dirs的y和z的方向值需要乘以负号

这是因为OpenCV/Colmap的相机坐标系里相机的Up/Y朝下，相机光心朝向+Z轴，而NeRF/OpenGL相机坐标系里相机的Up/朝上，相机光心朝向-Z轴，所以这里代码在方向向量dir的第二和第三项乘了个负号。



run_nerf_helpers.py文件

```
1 usage   SirinL
def get_rays_np(H, W, K, c2w):
    i, j = np.meshgrid(np.arange(W, dtype=np.float32), np.arange(H, dtype=np.float32), indexing='xy')
    dirs = np.stack([(i-K[0][2])/K[0][0], -(j-K[1][2])/K[1][1], -np.ones_like(i)], -1)
    # Rotate ray directions from camera frame to the world frame
    rays_d = np.sum(dirs[..., np.newaxis, :] * c2w[:3,:3], -1)  # dot product, equals to: [c2w.dot(dir) for dir in dirs]
    # Translate camera frame's origin to the world frame. It is the origin of all rays.
    rays_o = np.broadcast_to(c2w[:3,-1], np.shape(rays_d))
    return rays_o, rays_d
```

```
if K is None:
    K = np.array([
        [focal, 0, 0.5*W],
        [0, focal, 0.5*H],
        [0, 0, 1]
    ])
```

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

get_rays是其对应的pytorch版本：

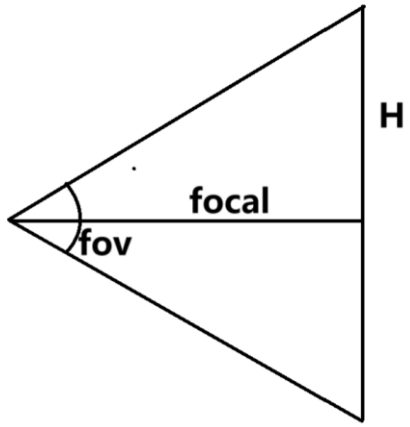
```
# Ray helpers
3 usages   SirinL
def get_rays(H, W, K, c2w):...
```

生成射线原点时，只需要把变换矩阵的最后一列的前三个坐标取出即可：

$$\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

run_nerf_helpers.py文件

注意现在生成的方向向量存在一个可能的问题，即如果图像的长、宽都是几百乘以几百的分辨率，focal也应该是几百这个数量级，否则fov会无限接近于180度。



我们需要根据场景的边界范围来确定采样距离。

数据读取后有个bounds参数（bds），根据其限定的场景最近点和最远点，把射线变为从近平面为原点发射，并在最远点终止采样。

这是run_nerf.py文件中加载llff数据后的定义，根据是否使用ndc坐标来重新定义远点和近点。注意近点乘以了0.9，远点乘以了1.0，这是为了稍微扩大一点近景边界。

```
print('DEFINING BOUNDS')
if args.no_ndc:
    near = np.ndarray.min(bds) * .9
    far = np.ndarray.max(bds) * 1.

else:
    near = 0.
    far = 1.
print('NEAR FAR', near, far)
```

如果要求用ndc坐标系，那么就意味着整个可见场景被压缩到近平面为0，远平面为1的范围内（把截锥体压缩到立方体内）。注意NDC只比较适合那种llff数据，并且是主要方向超前的视角类型，因此如果不是这种类型会自动修改为不使用NDC (create_nerf函数):

```
# NDC only good for LLFF-style forward facing data
if args.dataset_type != 'llff' or args.no_ndc:
    print('Not ndc!')
    render_kwargs_train['ndc'] = False
    render_kwargs_train['lindisp'] = args.lindisp
```

run_nerf_helpers.py文件

ndc_rays函数主要功能是将给定的射线起点和方向 rays_o 和 rays_d 转换到归一化设备坐标（NDC）空间中。

```
1 usage   SirinL
def ndc_rays(H, W, focal, near, rays_o, rays_d):
```

将射线的起点 rays_o 沿射线方向 rays_d 上移，使其与近裁剪平面对齐：

```
# Shift ray origins to near plane
t = -(near + rays_o[...,2]) / rays_d[...,2]
rays_o = rays_o + t[...,None] * rays_d
```

o0, o1, o2 是归一化设备坐标空间中射线的起点。
d0, d1, d2 是归一化设备坐标空间中射线的方向。

```
# Projection
o0 = -1./(W/(2.*focal)) * rays_o[...,0] / rays_o[...,2]
o1 = -1./(H/(2.*focal)) * rays_o[...,1] / rays_o[...,2]
o2 = 1. + 2. * near / rays_o[...,2]

d0 = -1./(W/(2.*focal)) * (rays_d[...,0]/rays_d[...,2] - rays_o[...,0]/rays_o[...,2])
d1 = -1./(H/(2.*focal)) * (rays_d[...,1]/rays_d[...,2] - rays_o[...,1]/rays_o[...,2])
d2 = -2. * near / rays_o[...,2]
```

注意这里的NDC坐标其实就是把全部视角可见的场景都限定到一个(0,0,0)-(1,1,1)这个深度范围的空间内。因此，**对于所有的视角，其NDC表示都是一样的，而不是传统意义上光栅化和光线追踪中的对每个相机视角都定义的相机空间NDC，这点要清楚！**

然后我们计算之前发射的射线与这个空间的交点，并进行修正。之后我们计算和更新的NeRF都是针对这个NDC空间的神经表示进行更新的。

run_nerf_helpers.py文件

```
# Misc
img2mse = lambda x, y : torch.mean((x - y) ** 2)
mse2psnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.]))
to8b = lambda x : (255*np.clip(x,0,1)).astype(np.uint8)
```

img2mse 这个 lambda 函数用于计算两个张量 x 和 y 之间的均方误差（Mean Squared Error, MSE）

mse2psnr 这个 lambda 函数用于将均方误差（MSE）转换为峰值信噪比（Peak Signal-to-Noise Ratio, PSNR）

to8b = lambda x : (255*np.clip(x,0,1)).astype(np.uint8) 这个 lambda 函数用于将一个浮点数张量 x（范围在 [0, 1] 内）转换为 8 位无符号整数张量（np.uint8 类型）

run_nerf_helpers.py文件

位置编码解读：

论文中对3D位置坐标和两个视角坐标的编码方式不同：对于3D位置坐标，论文中使用频率为10的PE，对于2D坐标，论文中使用频率为4的PE。

```
parser.add_argument("--multires", type=int, default=10,
                    help='log2 of max freq for positional encoding (3D location)')
parser.add_argument("--multires_views", type=int, default=4,
                    help='log2 of max freq for positional encoding (2D direction)')
```

$$\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \cdots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right).$$

In our experiments, we set $L = 10$ for $\gamma(\mathbf{x})$ and $L = 4$ for $\gamma(\mathbf{d})$.

class Embedder:

下面的函数被get_embedder调用以生成位置编码：

```
def create_embedding_fn(self):
```

可以将输入数据（位置或方向）也作为位置编码的一部分：

```
# 如果需要包含输入向量，则添加一个恒等函数到嵌入函数列表中
if self.kwargs['include_input']:
    embed_fns.append(lambda x : x)
    out_dim += d # 输出维度增加d
```

编码是否是对数采样：

```
# 根据是否使用对数采样来生成频率波段
if self.kwargs['log_sampling']:
    freq_bands = 2.**torch.linspace(0., max_freq, steps=N_freqs)
else:
    freq_bands = torch.linspace(2.**0., 2.**max_freq, steps=N_freqs)
```

如果是对数采样，就生成2^0,2^1,⋯,2^(n_freqs-1)
如果不是对数采样，就生成2^0到2^(n_freqs-1)的均匀间隔值。

调用periodic_fns的函数（sin和cos）生成编码：

```
# 对于每个频率波段和每个周期函数，生成对应的嵌入函数，并计算输出维度
for freq in freq_bands:
    for p_fn in self.kwargs['periodic_fns']:
        embed_fns.append(lambda x, p_fn=p_fn, freq=freq : p_fn(x * freq))
        out_dim += d
```

run_nerf_helpers.py文件

位置编码解读:

get_embedder函数在生成位置编码时的参数如下:

```python
def get_embedder(multires, i=0):
    if i == -1:
        return nn.Identity(), 3

    embed_kwargs = {
                'include_input' : True,
                'input_dims' : 3,
                'max_freq_log2' : multires-1,
                'num_freqs' : multires,
                'log_sampling' : True,
                'periodic_fns' : [torch.sin, torch.cos],
    }

    embedder_obj = Embedder(**embed_kwargs)
    embed = lambda x, eo=embedder_obj : eo.embed(x)
    return embed, embedder_obj.out_dim
```

embed = lambda x, eo=embedder_obj : eo.embed(x)
定义了一个 lambda 函数 embed，它接受一个参数 x，并将其传递给 embedder_obj 的 embed 方法处理。

可以看到其实对于2D方向和3D位置，都设置输入维度为2。

get_embedder函数获得的只是一个编码器以及编码后的输出维度，而不是编码的结果。
编码输出维度在创建MLP时使用。

调用get_embedder函数的参数是用户自己定义的:

```python
1 usage  SirinL
def create_nerf(args):
    """Instantiate NeRF's MLP model.
    """
    embed_fn, input_ch = get_embedder(args.multires, args.i_embed)

    input_ch_views = 0
    embeddirs_fn = None
    if args.use_viewdirs:
        embeddirs_fn, input_ch_views = get_embedder(args.multires_views, args.i_embed)
```

注意，虽然方向是2D信息，但表示和用于编码时还是表示为三维向量。

run_nerf_helpers.py文件

NeRF类

下面是NeRF类的初始化函数：

```
# Model
3 usages  SirinL
class NeRF(nn.Module):
    SirinL
    def __init__(self, D=8, W=256, input_ch=3, input_ch_views=3, output_ch=4, skips=[4], use_viewdirs=False):
```

在介绍NeRF之前，我们先介绍create_nerf函数中调用参数，见create_nerf函数。

```
embed_fn, input_ch = get_embedder(args.multires, args.i_embed)

input_ch_views = 0
embeddirs_fn = None
if args.use_viewdirs:
    embeddirs_fn, input_ch_views = get_embedder(args.multires_views, args.i_embed)
output_ch = 5 if args.N_importance > 0 else 4
skips = [4]
model = NeRF(D=args.netdepth, W=args.netwidth,
             input_ch=input_ch, output_ch=output_ch, skips=skips,
             input_ch_views=input_ch_views, use_viewdirs=args.use_viewdirs).to(device)
grad_vars = list(model.parameters())

model_fine = None
if args.N_importance > 0:
    model_fine = NeRF(D=args.netdepth_fine, W=args.netwidth_fine,
                      input_ch=input_ch, output_ch=output_ch, skips=skips,
                      input_ch_views=input_ch_views, use_viewdirs=args.use_viewdirs).to(device)
    grad_vars += list(model_fine.parameters())
```

该代码根据是否使用视角参数，以及是否进行重要性采样来初始化NeRF。

在NeRF论文里会进行两次采样，一次是粗采样，一次细采样。只有当参数N_importance（细采样的样本数）大于0时，说明会进行细采样，此时才会初始化细NeRF。

```
parser.add_argument("--N_samples", type=int, default=64,
                    help='number of coarse samples per ray')
parser.add_argument("--N_importance", type=int, default=64,
                    help='number of additional fine samples per ray')
```

D是网络深度，默认是8层。

run_nerf_helpers.py文件

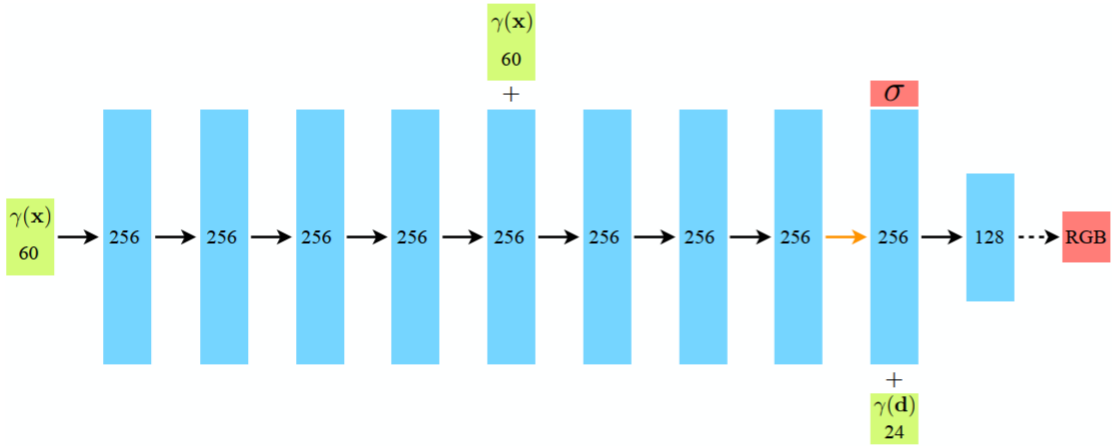
NeRF类

定义线性层

```
self.pts_linears = nn.ModuleList(
    [nn.Linear(input_ch, W)] + [nn.Linear(W, W) if i not in
self.skips else nn.Linear(W + input_ch, W) for i in
range(D-1)])
```

self.pts_linears 是处理三维位置的线性层表 nn.ModuleList，包含多个线性层。第一个元素是将输入特征映射到 W 维度的线性层，其余的层根据 skips 参数决定是否连接输入点的特征。

skips 是一个列表，表示在特定层中连接输入点特征的位置。默认是4，与下图是对应的：




```
self.views_linears =
nn.ModuleList([nn.Linear(input_ch_views + W, W//2)])
```

self.views_linears 是另一个 nn.ModuleList，包含一个线性层，用于处理视角信息。可以看到，这里和图也是一一对应的。

```
if use_viewdirs:
    self.feature_linear = nn.Linear(W, W)
    self.alpha_linear = nn.Linear(W, 1)
    self.rgb_linear = nn.Linear(W//2, 3)
else:
    self.output_linear = nn.Linear(W, output_ch)
```

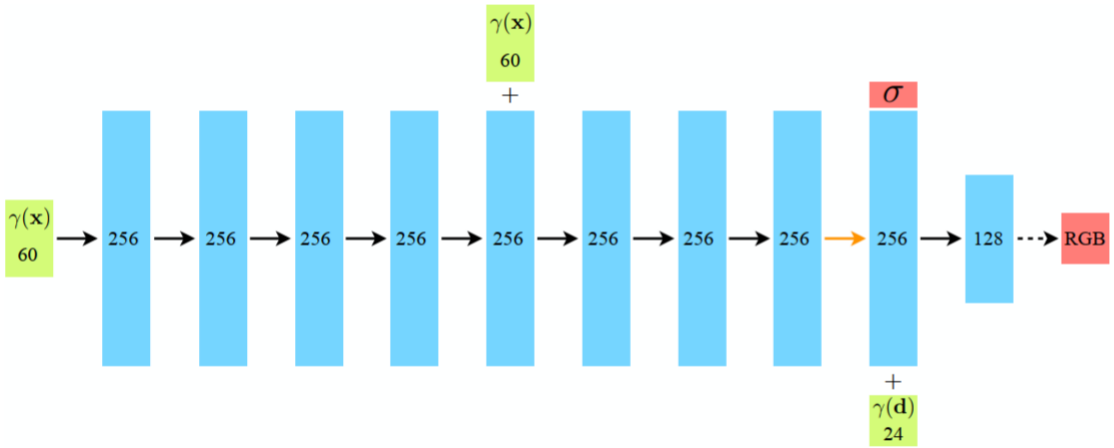
如果 use_viewdirs 为 True，则初始化用于视角信息的线性层 feature_linear、alpha_linear 和 rgb_linear。否则，初始化一个线性层 output_linear 用于直接生成输出。

这里的维度也和图中是一一对应的。

run_nerf_helpers.py文件

NeRF类

def forward(self, x) 函数的网络结构与图中是一致的：



sample_pdf 函数我们后面再介绍。

run_nerf.py文件

train函数

```
if args.dataset_type == 'llff':
    images, poses, bds, render_poses, i_test = load_llff_data(args.datadir, args.factor,
                                                              recenter=True, bd_factor=.75,
                                                              spherify=args.spherify)

    hwf = poses[0,:3,-1]
    poses = poses[:,:3,:4]
    print('Loaded llff', images.shape, render_poses.shape, hwf, args.datadir)
    if not isinstance(i_test, list):
        i_test = [i_test]

    if args.llffhold > 0:
        print('Auto LLFF holdout,', args.llffhold)
        i_test = np.arange(images.shape[0])[::args.llffhold]

    i_val = i_test
    i_train = np.array([i for i in np.arange(int(images.shape[0])) if
                    (i not in i_test and i not in i_val)])
```

默认抽取1/8的图像为测试集：

```
parser.add_argument("--llffhold", type=int, default=8,
                    help='will take every 1/N images as LLFF test set, paper uses 8')
```

np.arange(images.shape[0]) 生成一个从0到图像总数的连续整数数组，[::args.llffhold] 则是从这个数组中以 args.llffhold 为步长取值，生成测试集的索引。

```
# Create log dir and copy the config file
basedir = args.basedir
expname = args.expname
os.makedirs(os.path.join(basedir, expname), exist_ok=True)
f = os.path.join(basedir, expname, 'args.txt')
with open(f, 'w') as file:
    for arg in sorted(vars(args)):
        attr = getattr(args, arg)
        file.write('{} = {}\n'.format(arg, attr))
if args.config is not None:
    f = os.path.join(basedir, expname, 'config.txt')
    with open(f, 'w') as file:
        file.write(open(args.config, 'r').read())
```

首先检查 args.config 是否为非空，这是一个额外的配置文件路径。如果配置文件存在，则构造一个新的文件路径 config.txt，打开配置文件，读取内容，并将内容复制到新的文件中。这样可以确保配置文件也被保存在实验目录下。

run_nerf.py文件

train函数

剩下的内容的大致过程为：

- 创建NeRF模型（加载之前训练好的参数）。
- 如果只需要渲染，就渲染（调用render_path函数来对render_poses所包含的位置进行渲染）并输出结果
- 如果需要把射线随机batch，就生成raybatch
- 启动渲染循环

run_nerf.py文件

train函数

生成raybatch

```
use_batching = not args.no_batching
if use_batching:
    # For random ray batching
    print('get rays')
    rays = np.stack([get_rays_np(H, W, K, p) for p in poses[:,:3,:4]], 0) # [N, ro+rd, H, W, 3]
    print('rays:', rays.shape)
    print('done, concats')
    rays_rgb = np.concatenate([rays, images[:,None]], 1) # [N, ro+rd+rgb, H, W, 3]
    rays_rgb = np.transpose(rays_rgb, [0,2,3,1,4]) # [N, H, W, ro+rd+rgb, 3]
    rays_rgb = np.stack([rays_rgb[i] for i in i_train], 0) # train images only
    rays_rgb = np.reshape(rays_rgb, [-1,3,3]) # [(N-1)*H*W, ro+rd+rgb, 3]
    rays_rgb = rays_rgb.astype(np.float32)
    print('shuffle rays')
    np.random.shuffle(rays_rgb)
```

注意get_rays_np返回的维度是[2, H, W, 3]，这里的2表示射线的起点和方向，3表示起点和方向都是3维表示的。因此

rays = np.stack([get_rays_np(H, W, K, p) for p in poses[:,:3,:4]], 0)

的总维度就是[N, 2, H, W, 3]。

由于每个像素还可以包含三通道的rgb值，因此

rays_rgb = np.concatenate([rays, images[:,None]], 1)

把图像像素值（[H, W, 3]）也融合进来。

之后再剔除测试数据，只要训练数据。

最后再把rays_rgb进行变形

rays_rgb = np.reshape(rays_rgb, [-1,3,3])

得到变形后的最终结果：

[训练集图像数*H*W, 3, 3]

这里的3*3中，表示3通道ray起点，3通道ray方向，3通道计算得到的rgb值

run_nerf.py文件

train函数中的训练大循环：

```
  start = start + 1
  for i in trange(start, N_iters):
```

首先制作当前轮训练所用的数据，并进行训练：

```
# Sample random ray batch
if use_batching:...

else:...

#####  Core optimization loop  #####
rgb, disp, acc, extras = render(H, W, K, chunk=args.chunk, rays=batch_rays,
                                verbose=i < 10, retraw=True,
                                **render_kwargs_train)
```

如果使用batching，就把全部图像的射线进行打乱。否则就只随机抽取一张图像的数据来训练。

然后计算反向传播。

之后再更新学习率：

```
# NOTE: IMPORTANT!
###   update learning rate   ###
decay_rate = 0.1
decay_steps = args.lrate_decay * 1000
new_lrate = args.lrate * (decay_rate ** (global_step / decay_steps))
for param_group in optimizer.param_groups:
    param_group['lr'] = new_lrate
```

然后根据当前帧数来选择是否保存渲染中的参数或者输出视频、执行测试等。

整个train函数中，最重要的两个函数分别是create_nerf和render。我们分别介绍这两个函数。

注意render_path函数是用于渲染多个位置和角度的函数（用于测试或输出视频），也会调用render函数。

run_nerf.py文件

create_nerf函数主要是为了生成一些训练中的参数:

```
render_kwargs_train = {
    'network_query_fn' : network_query_fn,
    'perturb' : args.perturb,
    'N_importance' : args.N_importance,
    'network_fine' : model_fine,
    'N_samples' : args.N_samples,
    'network_fn' : model,
    'use_viewdirs' : args.use_viewdirs,
    'white_bkgd' : args.white_bkgd,
    'raw_noise_std' : args.raw_noise_std,
}
```

字典推导式遍历 render_kwargs_train 中的所有键 k，并将每个键及其对应的值复制到新字典 render_kwargs_test 中:

```
render_kwargs_test = {k : render_kwargs_train[k] for k in render_kwargs_train}
render_kwargs_test['perturb'] = False
render_kwargs_test['raw_noise_std'] = 0.
```

network_query_fn 是一个lambda函数，用于其输入参数是:

```
network_query_fn = lambda inputs, viewdirs, network_fn : run_network(inputs, viewdirs, network_fn,
                                                                     embed_fn=embed_fn,
                                                                     embeddirs_fn=embeddirs_fn,
                                                                     netchunk=args.netchunk)
```

其中，network_fn 可以是model或者mdel_fine，即粗NeRF和细NeRF。
embed_fn和embeddirs_fn分别是用于编码位置和方向的编码器。

run_nerf.py文件

render函数调用batchify_rays函数， batchify_rays函数调用render_rays函数， render_rays函数是渲染的核心代码。渲染完以后，再调用raw2outputs函数将原始渲染数据（每个采样点的数据值）转换为rgb图像像素数据。

render函数的调用参数

```
#####  Core optimization loop  #####
rgb, disp, acc, extras = render(H, W, K, chunk=args.chunk, rays=batch_rays,
                                verbose=i < 10, retraw=True,
                                **render_kwargs_train)
```

render函数的参数中，chunk表示并行处理的射线数，

batch_rays的形状是[2, 4096, 3]，4096表示每次随机梯度下降都会使用4096个样本，2和3表示有两个三维向量（位置和方向）
verbose表示是否打印调试信息
retraw表示是否返回原渲染数据（不色调映射转换为渲染结果）

render函数的返回结果中， rgb是每个射线积累的颜色值，acc是每个射线累积的不透明度值。disp是视差图，即1/depth值。rgb, acc和disp都是细NeRF最终返回的结果。rgb的形状是[N_rand, 3]，acc和disp的形状都是[N_rand]。

extras表示由render_rays函数的其他所有返回值的集合，里面的内容分别为： raw rgb0 disp0 acc0 z_std。
注意名称末尾带0的（rgb0 disp0 acc0）表示粗NeRF返回的结果。
z_std表示射线距离的标准差，后面再讲解。

run_nerf.py文件

render函数一开始先生成射线的起点和方向，生成方法之前都已经介绍过了，不再赘述。
生成完以后确保其形状都为[N_rand, 3]，也就是最后一个维度都要是3，然后计算射线的近和远距离（射线的范围），维度都是[N_rand, 1]。再次声明一下，N_rand是我们设定的每个batch所使用的射线数。

```
# Create ray batch
rays_o = torch.reshape(rays_o, [-1,3]).float()
rays_d = torch.reshape(rays_d, [-1,3]).float()

near, far = near * torch.ones_like(rays_d[...,:1]), far * torch.ones_like(rays_d[...,:1])
```

还有一点需要注意的是，rays_d不一定是归一化的向量，只是代表方向。而之前生成的viewdirs则是用于生成位置编码的方向向量（其实就是rays_d的归一化值）要在需要使用方向编码时导入进去：

```
rays = torch.cat([rays_o, rays_d, near, far], -1)
if use_viewdirs:
    rays = torch.cat([rays, viewdirs], -1)
```

然后对batchify_rays函数的输出进行形状调整：

```
# Render and reshape
all_ret = batchify_rays(rays, chunk, **kwargs)
for k in all_ret:
    k_sh = list(sh[:-1]) + list(all_ret[k].shape[1:])
    all_ret[k] = torch.reshape(all_ret[k], k_sh)
```

sh 是原始形状（为[N_rand, 3]），sh[:-1] 获取 sh 的所有元素，但去掉最后一个维度（得到[N_rand]）。
list(all_ret[k].shape[1:]) 获取 all_ret[k] 张量的形状，去掉第一个维度，转换为列表。
k_sh 通过将 sh 的前部分维度和 all_ret[k] 的维度连接在一起，形成一个新的形状。（这一步其实执行和不执行没有什么区别，因为维度本身就是[N_rand, ...] 这种形式）。

run_nerf.py文件

batchify_rays 函数:
该函数的主要作用是将处理光线的任务分成较小的批次，以避免因处理数据量过大而导致的内存溢出。

```
all_ret = {}
for i in range(0, rays_flat.shape[0], chunk):
    ret = render_rays(rays_flat[i:i+chunk], **kwargs)
    for k in ret:
        if k not in all_ret:
            all_ret[k] = []
        all_ret[k].append(ret[k])

all_ret = {k : torch.cat(all_ret[k], 0) for k in all_ret}
```

最后一行代码中，将 all_ret 中的所有列表合并成一个张量。torch.cat(all_ret[k], 0) 将每个键 k 对应的所有批次的张量沿第一个维度拼接起来，形成最终的结果张量。

目前我们还有最重要的四个函数需要讲解:
render_rays 函数
run_network 函数
raw2outputs 函数
以及**run_nerf_helpers.py**文件中的sample_pdf函数

其中，run_network 函数、sample_pdf函数和raw2outputs函数都是会在render_rays 函数中调用的。

run_nerf.py文件

render_rays 函数

render_rays 函数的输入参数中， network_fn是查询模型（粗NeRF ）， network_fine是细NeRF。由network_query_fn函数调用network_fn或者network_fine来决定是查询粗NeRF还是细NeRF 。 network_query_fn函数本质上就是调用run_network函数。
lindisp表示是采样深度还是采样1/深度。
white_bkgd表示是不是要假设背景是白色。
perturb表示是否把采样位置点在其所在段内进行扰动（类似于分层采样）。

```
def render_rays(ray_batch,
                network_fn,
                network_query_fn,
                N_samples,
                retraw=False,
                lindisp=False,
                perturb=0.,
                N_importance=0,
                network_fine=None,
                white_bkgd=False,
                raw_noise_std=0.,
                verbose=False,
                pytest=False):
```

render_rays 函数首先先把用到的数据都取出来：

```
N_rays = ray_batch.shape[0]
rays_o, rays_d = ray_batch[:,0:3], ray_batch[:,3:6] # [N_rays, 3] each
viewdirs = ray_batch[:,-3:] if ray_batch.shape[-1] > 8 else None
bounds = torch.reshape(ray_batch[...,6:8], [-1,1,2])
near, far = bounds[...,0], bounds[...,1] # [-1,1]
```

函数生成每个采样点的z值（采样点=o+z*d）：

```
t_vals = torch.linspace(0., 1., steps=N_samples)
if not lindisp:
    z_vals = near * (1.-t_vals) + far * (t_vals)
else:
    z_vals = 1./(1./near * (1.-t_vals) + 1./far * (t_vals))

z_vals = z_vals.expand([N_rays, N_samples])
```

注意这里之所以要用expand，是为了避免一个batch不是chunk整数倍时，要让项数统一。

如果需要抖动采样位置，就进行抖动（下一页再讲）。

之后用z值合成采样点位置，pts为[N_rays, N_samples, 3]：

```
pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None]
```

run_nerf.py文件

render_rays 函数

抖动的代码如下：

```
if perturb > 0.:
    # get intervals between samples
    mids = .5 * (z_vals[...,1:] + z_vals[...,:-1])
    upper = torch.cat([mids, z_vals[...,-1:]], -1)
    lower = torch.cat([z_vals[...,:1], mids], -1)
    # stratified samples in those intervals
    t_rand = torch.rand(z_vals.shape)

    # Pytest, overwrite u with numpy's fixed random numbers
    if pytest:...

    z_vals = lower + (upper - lower) * t_rand
```

抖动的原理是先找到相邻两个z的中间值，该值作为低z的上界和高z的下界，之后用(0,1)范围的随机数来取样。

然后将样本输入到网络中运行并得到原始数据：

```
raw = network_query_fn(pts, viewdirs, network_fn)
rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd, pytest=pytest)

if N_importance > 0:...
```

回顾一下network_query_fn函数：

```
network_query_fn = lambda inputs, viewdirs, network_fn : run_network(inputs, viewdirs, network_fn,
                                                                     embed_fn=embed_fn,
                                                                     embeddirs_fn=embeddirs_fn,
                                                                     netchunk=args.netchunk)
```

该函数调用run_network函数。我们详解一下run_network函数。首先先把输入维度展平，将[N_rays, N_samples, 3]展平为[N_rays * N_samples, 3]。之后进行编码。

```
inputs_flat = torch.reshape(inputs, [-1, inputs.shape[-1]])
embedded = embed_fn(inputs_flat)
```

embedded的维度是[N_rays * N_samples, 63]，这个63是这么来的：对于每个位置的xyz分量，会得到10个不同频的sin编码和10个不同频的cos编码（L默认为10），同时还要把这三个分量也作为编码的一部分，因此总维度就是3*(10+10)+3 = 63

然后处理输入向量，方法相同：

```
if viewdirs is not None:
    input_dirs = viewdirs[:,None].expand(inputs.shape)
    input_dirs_flat = torch.reshape(input_dirs, [-1, input_dirs.shape[-1]])
    embedded_dirs = embeddirs_fn(input_dirs_flat)
    embedded = torch.cat([embedded, embedded_dirs], -1)
```

embedded的最终维度是[N_rays * N_samples, 90]，这个90中，除了之前的63维，还有3*(4+4)+3，编码方向的L为4.

run_nerf.py文件

run_network函数

之后，根据chunk决定是一次执行完还是分多次执行。执行完以后变形为之前的形状（除了最后一维保持输出维度）：

```
outputs_flat = batchify(fn, netchunk)(embedded)
outputs = torch.reshape(outputs_flat, list(inputs.shape[:-1]) + [outputs_flat.shape[-1]])
return outputs
```

outputs最后的维度是[N_rays, N_samples, 4]，这个4表示rgb和不透明度。也就是调用network_query_fn得到的raw的值。

重新回到render_rays函数。下面的代码是调用raw2outputs函数得到渲染结果。我们先详细讲解该函数，然后再介绍后面的重要性采样部分。

run_nerf.py文件

raw2outputs函数

```
def raw2outputs(raw, z_vals, rays_d, raw_noise_std=0, white_bkgd=False, pytest=False):
```

计算 z_vals 张量中相邻元素之间的距离，并在结果中添加一个特定的值（1e10），以进行后续处理：

```
dists = z_vals[...,1:] - z_vals[...,:-1]
dists = torch.cat([dists, torch.Tensor([1e10]).expand(dists[...,:1].shape)], -1)
```

torch.Tensor([1e10]).expand(dists[...,:1].shape)得到的是[N_rays, 1]的全为10^10的数组，之后跟形状为[N_rays, N_samples-1]的数组合并，得到[N_rays, N_samples]形状的间距值。注意这里默认最后一个间距是10^10,这样一个超级大的值。

由于这里的rays_d并不是都进行过单位化的，因此需要再分别乘以rays_d的模来得到真实有效的长度值：

```
dists = dists * torch.norm(rays_d[...,None,:], dim=-1)
```

```
rgb = torch.sigmoid(raw[...,:3])
```

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

该函数将输入值压缩到 $[0, 1]$ 范围内。对于 RGB 颜色值，通常需要在此范围内。

然后根据预设选择是否给数据增加一个noise：

```
noise = 0.
if raw_noise_std > 0.:
    noise = torch.randn(raw[...,3].shape) * raw_noise_std

    # Overwrite randomly sampled data if pytest
    if pytest:
        np.random.seed(0)
        noise = np.random.rand(*list(raw[...,3].shape)) * raw_noise_std
        noise = torch.Tensor(noise)

alpha = raw2alpha(raw[...,3] + noise, dists)  # [N_rays, N_samples]
```

上面最后一步是生成alpha不透明度，调用如下函数：

```
raw2alpha = lambda raw, dists, act_fn=F.relu: 1.-torch.exp(-act_fn(raw)*dists)
```

对应原论文中的如下公式：

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples. This function for calculating $\hat{C}(\mathbf{r})$ from the set of $(\mathbf{c}_i, \sigma_i)$ values is trivially differentiable and reduces to traditional alpha compositing with alpha values $\alpha_i = 1 - \exp(-\sigma_i\delta_i)$.

下面这段代码生成的是权重和rgb图：

```
weights = alpha * torch.cumprod(torch.cat([torch.ones((alpha.shape[0], 1)), 1.-alpha + 1e-10], -1), -1)[:, :-1]
rgb_map = torch.sum(weights[...,None] * rgb, -2)  # [N_rays, 3]
```

torch.cat([torch.ones((alpha.shape[0], 1)), 1.-alpha + 1e-10], -1)这段代码将一个[N_rays, 1]的全1数组和[N_rays, N_samples]的数组合并起来。加上1e-10是为了避免出现0引起数值计算错误。 torch.cumprod: 沿着最后一个维度计算累积乘积。累积乘积的结果只要前面的[N_rays, N_samples]个值。

run_nerf.py文件

raw2outputs函数

torch.sum(weights[...,None] * rgb, -2)将组合的结果在倒数第二个维度上相加，得到rgb值。

然后再计算一些可能用到的量，其中acc_map是累加的不透明度值。

```
depth_map = torch.sum(weights * z_vals, -1)
disp_map = 1./torch.max(1e-10 * torch.ones_like(depth_map), depth_map / torch.sum(weights, -1))
acc_map = torch.sum(weights, -1)
```

至此，我们还剩最后一个部分，即当需要进行重要性采样时，根据前面计算得到的权重weight来进行重要性采样。

先回到render_rays函数来看一下流程：

```
if N_importance > 0:

    rgb_map_0, disp_map_0, acc_map_0 = rgb_map, disp_map, acc_map

    z_vals_mid = .5 * (z_vals[...,1:] + z_vals[...,:-1])
    z_samples = sample_pdf(z_vals_mid, weights[...,1:-1], N_importance, det=(perturb==0.), pytest=pytest)
    z_samples = z_samples.detach()

    z_vals, _ = torch.sort(torch.cat([z_vals, z_samples], -1), -1)
    pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None] # [N_rays, N_samples + N_importance, 3]

    run_fn = network_fn if network_fine is None else network_fine
      raw = run_network(pts, fn=run_fn)
    raw = network_query_fn(pts, viewdirs, run_fn)
```

这段代码比较简单，重要性采样的位置要在边界以内：

```
z_vals_mid = .5 * (z_vals[...,1:] + z_vals[...,:-1])
z_samples = sample_pdf(z_vals_mid, weights[...,1:-1], N_importance, det=(perturb==0.), pytest=pytest)
```

z_vals_mid有N_samples-1个点，构成N_samples-2个区间。weights[...,1,-1]有N_samples-2个权重，用于对这N_samples-2个区间进行采样。

需要注意的是，重要性采样到的新位置要和之前的位置合并，然后排序，也就是：

```
z_vals, _ = torch.sort(torch.cat([z_vals, z_samples], -1), -1)
```

下面主要介绍sample_pdf函数。

run_nerf.py文件

sample_pdf函数

在介绍该函数之前，需要说明一下，N_samples在参数中表示对粗网络采样的样本数，N_importance在参数中表示对细网络的采样数。但是sample_pdf函数的参数中，N_importance被写为了N_samples。为了更好地区分，我们将粗网络采样的样本数写为N_batch（默认为64）。

首先计算概率密度函数pdf和累积概率密度函数cdf（[N_rays, N_batch -1]）

```
def sample_pdf(bins, weights, N_samples, det=False, pytest=False):
    # Get pdf
    weights = weights + 1e-5 # prevent nans
    pdf = weights / torch.sum(weights, -1, keepdim=True)
    cdf = torch.cumsum(pdf, -1)
    cdf = torch.cat([torch.zeros_like(cdf[...,:1]), cdf], -1)  # (batch, len(bins))
```

然后根据是否是均匀采样来设置(0,1)随机数分布：

```
# Take uniform samples
if det:
    u = torch.linspace(0., 1., steps=N_samples)
    u = u.expand(list(cdf.shape[:-1]) + [N_samples])
else:
    u = torch.rand(list(cdf.shape[:-1]) + [N_samples])
```

如果 pytest 为 True，使用固定的随机数生成以保证测试的一致性。

```
if pytest:
    np.random.seed(0)
    new_shape = list(cdf.shape[:-1]) + [N_samples]
    if det:
        u = np.linspace(0., 1., N_samples)
        u = np.broadcast_to(u, new_shape)
    else:
        u = np.random.rand(*new_shape)
    u = torch.Tensor(u)
```

run_nerf.py文件

sample_pdf函数

采样（本质上是采样概率密度的逆函数法）：

```
# Invert CDF
u = u.contiguous() # 确保 u 张量在内存中是连续的，以便后续操作的高效计算。
inds = torch.searchsorted(cdf, u, right=True) # 查找 u 在 cdf 中的位置，以确定其对应的 bin 区间
below = torch.max(torch.zeros_like(inds-1), inds-1)  # 获取 inds 的下标。
above = torch.min((cdf.shape[-1]-1) * torch.ones_like(inds), inds) # 获取 inds 的上标。
# 堆叠 below 和 above，形成 (N_rays, N_samples, 2) 形状的张量，表示每个采样点的区间。
inds_g = torch.stack([below, above], -1)  # (batch, N_samples, 2)
```

Inds的维度是(N_rays, N_samples)，表示每个采样样本在cdf上的区间索引。
below计算时，要保证值不能小于0。然后同时需要保证above不能超过cdf的最后一个索引，即不大于N_batch-1。

现在我们需要使用这个坐标来提取对应的cdf值区间（维度为[N_rays, N_batch-1] ）和bin数据对应的值区间（bin就是z_vals_mid，维度为[N_rays, N_batch-1]）。提取出的区间再经过线性插值来得到最终采样值。

这里的提取方式是用维度扩增来实现的：

```
matched_shape = [inds_g.shape[0], inds_g.shape[1], cdf.shape[-1]]
cdf_g = torch.gather(cdf.unsqueeze(1).expand(matched_shape), 2, inds_g)
bins_g = torch.gather(bins.unsqueeze(1).expand(matched_shape), 2, inds_g)
```

matched_shape是(N_rays, N_samples, N_batch-1)

cdf.unsqueeze(1) 在 cdf 张量的第1维插入了一个新的维度，使其形状变为 [..., 1, ...]。这里 ... 表示 cdf 张量的其他维度。
cdf.unsqueeze(1).expand(matched_shape) 将 cdf 张量沿着新插入的维度进行扩展，使其形状匹配 matched_shape。这里 matched_shape 的第二个维度与 cdf.unsqueeze(1) 的第二个维度对齐，以便后续操作能够广播。
torch.gather(..., 2, inds_g) 对扩展后的 cdf 张量沿着第 2 维（即 cdf 的原始最后一维）根据 inds_g 张量的索引进行收集。
inds_g 是包含索引的张量，torch.gather 会根据这些索引从扩展后的 cdf 张量中提取对应的值。

最终，cdf_g 是从 cdf 张量中按照 inds_g 指定的索引位置提取出的值，形状与 inds_g 相同。

bins_g 也是同理的，不再赘述。

run_nerf.py文件

sample_pdf函数

cdf_g 和 bins_g: 从 cdf 和 bins 中提取对应的值，使用 inds_g 索引：

```
denom = (cdf_g[...,1]-cdf_g[...,0])
denom = torch.where(denom<1e-5, torch.ones_like(denom), denom)
t = (u-cdf_g[...,0])/denom
samples = bins_g[...,0] + t * (bins_g[...,1]-bins_g[...,0])
```

denom = torch.where(denom<1e-5, torch.ones_like(denom), denom)
确保 denom 中的所有值都不小于 1e-5。对于接近零的值，它们会被替换为 1，从而避免在后续计算中出现除以零的情况，确保数值稳定性。

根据随机数所在的区间位置插值得到采样的射线位置。

完结撒花！