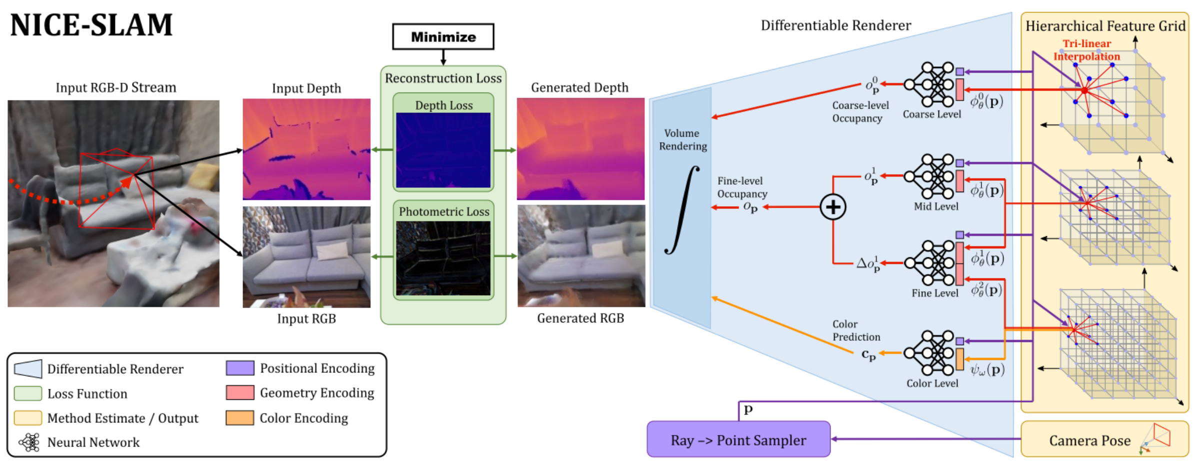
内容：

- 系统架构：NICE-SLAM是如何运行的
- 网络模型：网络是如何具体实现的
- 渲染：网络输出是怎么渲染得到图像和深度图的

代码：`NICE-SLAM.py`、`decoder.py`、`Renderer.py`、`common.py`

源码阅读笔记（1）

系统架构



演示

```
# 展示训练的运行效果（Demo数据集，服务器）
python -W ignore run.py configs/Demo/demo.yaml
# 展示可视化的运行效果（Apartment数据集，本机）
python visualizer.py configs/Apartment/apartment.yaml --output
output/vis/Apartment
```

函数`def grid_init(self, cfg)`

论文对应部分：附录A.2. Hierarchical Feature Grid Initialization

- **Coarse-level Feature Grid**：随机初始化；
- **Mid-level Feature Grid**：也是随机初始化的，经验表明随机初始化具有更好的收敛性；

- **Fine-level Feature Grid**：需要初始化到fine-level的decoder输出为0，毕竟这里给出来的是残差，初始化的残差肯定要是0。这样才能够保证从coarse-to-fine的优化过程中，能量是平稳过渡的。论文也提到，在对fine-level对应的decoder进行预训练的时候，增加了额外的正则化损失，保证如果fine-level的特征为0，那么无论mid-level的特征如何，输出的残差都应该为0。这样能够允许NICE-SLAM在运行的时候对fine-level的特征网格进行零初始化。

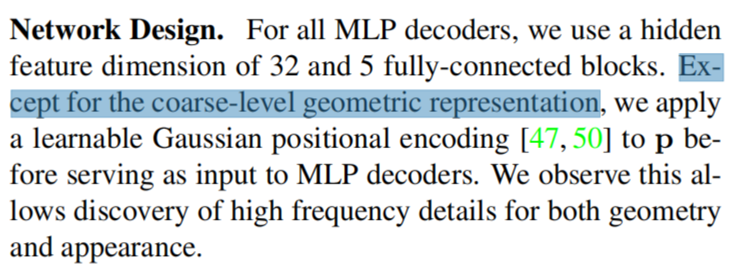
演示

```
import torch
import numpy as np
# 加载bound
bound=torch.from_numpy(np.array([[-5.8,11.3],[-4.0,4.5],[-7.9,4.9]]) * 1.0)
# 分别计算xyz轴的长度
xyz_len = bound[:, 1] - bound[:, 0]
# ============ 初始化一个特征网格 ============
# coarse要特别注意一下，有一个enlarge
# val_shape=list(map(int, (np.array(xyz_len * 2 / 3).tolist())))
# middle、fine、color
val_shape=list(map(int, (np.array([5.6,7.5,10]) / 0.16).tolist()))
# 交换xz轴数据
val_shape[0], val_shape[2] = val_shape[2], val_shape[0]
# 定义特征网格的shape，即BCDHW
grid_val_shape = [1, 8, *val_shape]
# 初始化特征网格
feature_grid=torch.zeros(grid_val_shape).normal_(mean=0, std=0.01)
# 对于fine-level，特征尽量接近0
# feature_grid=torch.zeros(grid_val_shape).normal_(mean=0, std=0.0001)
# ============ 演示特征网格的使用 ============
# 初始化特征网格
feature_grid=torch.zeros([1,8,2,3,4]).normal_(mean=0, std=0.01)
# 提取某个点的特征
feature=feature_grid[0,...,1,2,3]
```

网络模型

类`class NICE(nn.Module)`

论文对应部分： 3.1. Hierarchical Scene Representation



Network Design. For all MLP decoders, we use a hidden feature dimension of 32 and 5 fully-connected blocks. Except for the coarse-level geometric representation, we apply a learnable Gaussian positional encoding [47, 50] to **p** before serving as input to MLP decoders. We observe this allows discovery of high frequency details for both geometry and appearance.

对`def forward(self, p, c_grid, stage='middle', **kwargs)`**的理解：**

这部分主要服务于论文介绍的**3.3. Mapping and Tracking**，每个stage应该是和优化相关的，而不是和论文里提到的middle-leve、fine-level的输出相关。

下面是论文提到的Mapping阶段的三阶段优化：

Staged optimization:
- mid-level
- mid and fine-level
- BA : mid & fine & color-level + camera poses of selected keyframes

演示

```
import torch
raw = torch.zeros(8, 4).float()
raw[..., -1]=0.5
```

类`class MLP(nn.Module)`和类`class MLP_no_xyz(nn.Module)`

> 可以把nerf-pytorch的代码拿过来简单对比一下

相关链接：

- ConvONet论文：https://arxiv.org/pdf/2003.04618.pdf
- ConvONet代码：https://github.com/autonomousvision/convolutional_occupancy_networks/blob/master/src/conv_onet/models/decoder.py
- 类`GaussianFourierFeatureTransform`对应的论文：https://arxiv.org/abs/2006.10739

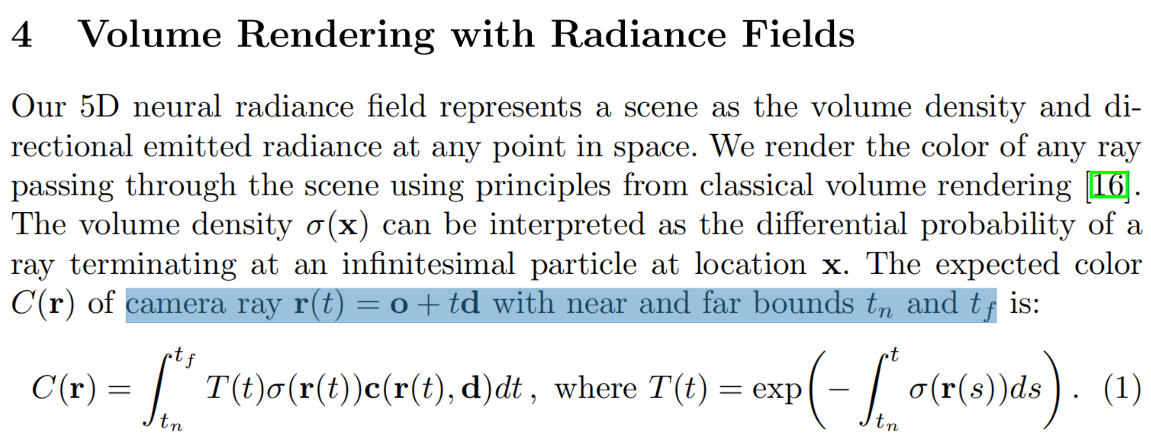
演示“sample_grid_feature”的过程

```
import torch
import numpy as np
import torch.nn.functional as F

p_nor=torch.from_numpy(np.array(
    [[ 0.  ,  0.  ,  0. ],
     [ 0.1,  0.2,  0.3],
     [-0.5, -0.6, -0.7],
     [ 1.  ,  0.  , -1. ]])).unsqueeze(0)
# [1, 4, 3]
# [1,point_num,3]

vgrid = p_nor[:, :, None, None].float()
# [1, 4, 1, 1, 3]
# [1,point_num,1,1,3]

feature_grid=torch.zeros([1,8,2,3,4]).normal_(mean=0, std=0.01)
# [1,8,2,3,4]
# [1,c_dim,D,H,W]

c = F.grid_sample(feature_grid, vgrid, padding_mode='border', align_corners=True,
mode='bilinear').squeeze(-1).squeeze(-1)
# [1,8,4]
# [1,c_dim,point_num]

c = c.transpose(1, 2).squeeze(0)
# [point_num,c_dim]
```

渲染

函数`render_batch_ray`

near和far可以参考NeRF论文这个部分：

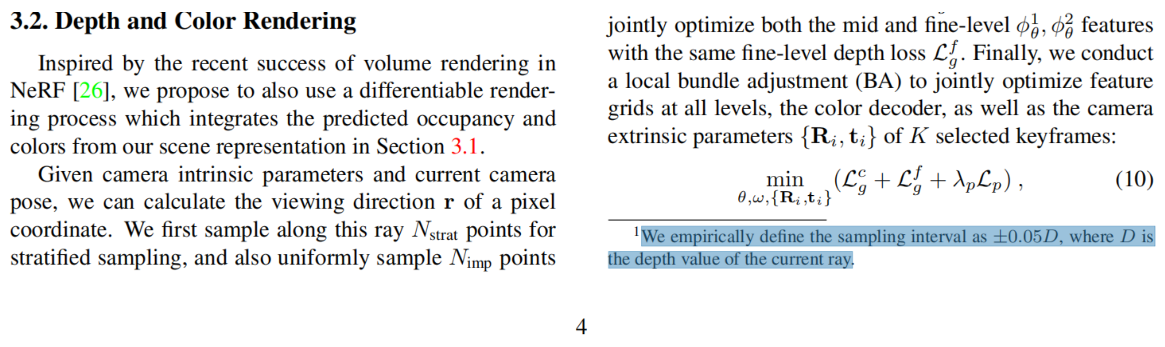


4 Volume Rendering with Radiance Fields

Our 5D neural radiance field represents a scene as the volume density and directional emitted radiance at any point in space. We render the color of any ray passing through the scene using principles from classical volume rendering [16]. The volume density $\sigma(\mathbf{x})$ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location $\mathbf{x}$. The expected color $C(\mathbf{r})$ of camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with near and far bounds t_n and t_f is:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right). \quad (1)$$

z_vals_surface可以参考NICE-SLAM论文**3.2. Depth and Color Rendering**



3.2. Depth and Color Rendering

Inspired by the recent success of volume rendering in NeRF [26], we propose to also use a differentiable rendering process which integrates the predicted occupancy and colors from our scene representation in Section 3.1.

Given camera intrinsic parameters and current camera pose, we can calculate the viewing direction $\mathbf{r}$ of a pixel coordinate. We first sample along this ray N_{strat} points for stratified sampling, and also uniformly sample N_{imp} points

jointly optimize both the mid and fine-level $\phi_\theta^1, \phi_\theta^2$ features with the same fine-level depth loss $\mathcal{L}_g^f$. Finally, we conduct a local bundle adjustment (BA) to jointly optimize feature grids at all levels, the color decoder, as well as the camera extrinsic parameters $\{\mathbf{R}_i, \mathbf{t}_i\}$ of K selected keyframes:

$$\min_{\theta,\omega,\{\mathbf{R}_i,\mathbf{t}_i\}} (\mathcal{L}_g^c + \mathcal{L}_g^f + \lambda_p \mathcal{L}_p)\,, \quad (10)$$

[1] We empirically define the sampling interval as $\pm 0.05D$, where D is the depth value of the current ray.

4

演示near的获取

```python
import torch

# 假设只有三条射线，然后采样的数量N_samples=10
gt_depth=torch.tensor([[1],[3],[9]])

gt_depth = gt_depth.reshape(-1, 1)  # 展开成向量
gt_depth_samples = gt_depth.repeat(1, 10)
near = gt_depth_samples * 0.01  # 每条射线的near根据这条射线对应的深度确定
```

演示far_bb的获取

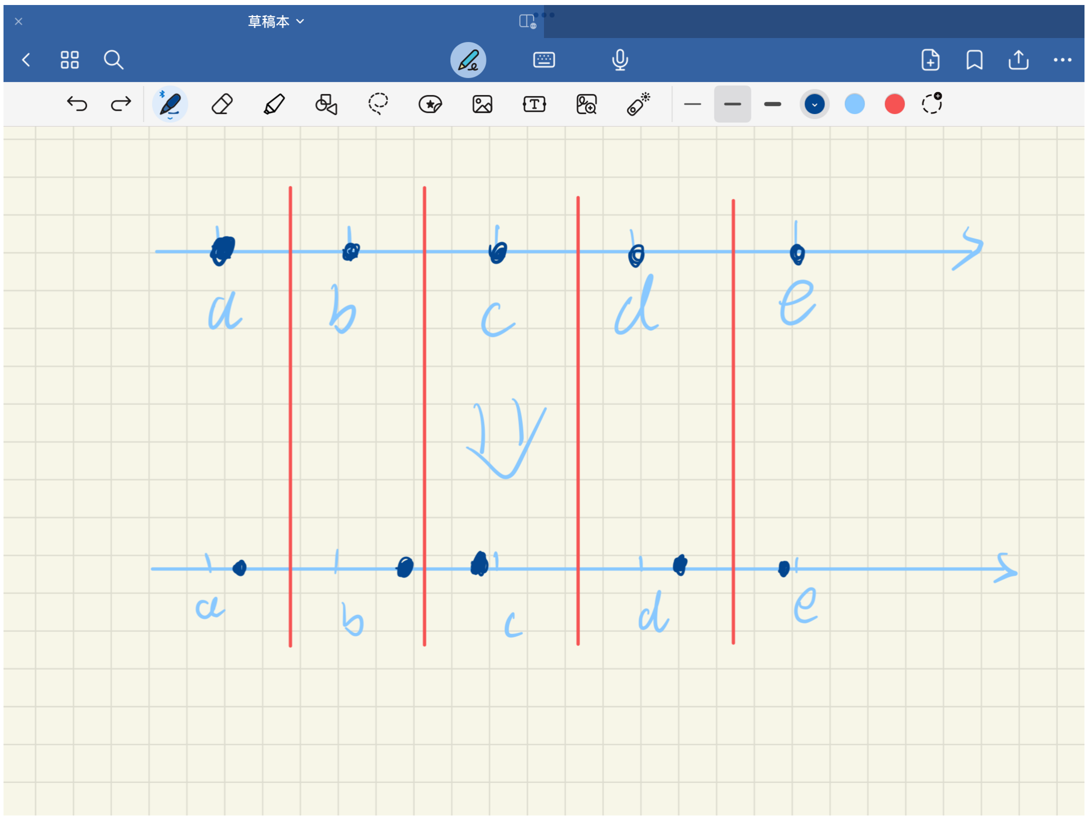
```python
import torch

# bound
bound=torch.tensor([[-10,10],[-20,30],[-40,100]])

# 假设只有两条射线
# det_rays_o
det_rays_o=torch.tensor([[[-5],[-6],[7]],[[9],[-8],[-10]]])

# det_rays_d
det_rays_d=torch.tensor([[[1],[2],[3]],[[0.1],[0.5],[5]]])
```

```
# t的计算
t = (bound - det_rays_o)
t = t / det_rays_d

# far_bb的计算
torch.max(t, dim=2)[0]
torch.min(torch.max(t, dim=2)[0], dim=1)[0]
```

演示z_vals_surface的获取

```
import torch

# ========== 有深度区域 ==========
# 假设只有五条射线，其中两条条射线没有深度，表面附近采样的数量N_surface=10
gt_depth=torch.tensor([[1],[0],[3],[0],[9]])

# 拿到有深度的部分
gt_none_zero_mask = gt_depth > 0
gt_none_zero = gt_depth[gt_none_zero_mask]
gt_none_zero = gt_none_zero.unsqueeze(-1)

# 每个采样点所属射线对应的深度
gt_depth_surface = gt_none_zero.repeat(1, 10)

# 间隔
t_vals_surface = torch.linspace(0., 1., steps=10).double()

# 在Depth+-0.05Depth的区域进行采样
z_vals_surface_depth_none_zero = (0.95 * gt_depth_surface * (1. - t_vals_surface)
+ 1.05 * gt_depth_surface * (t_vals_surface))

# 保存结果到最终的z_vals_surface中
z_vals_surface = torch.zeros(gt_depth.shape[0], 10).double()
gt_none_zero_mask = gt_none_zero_mask.squeeze(-1)
z_vals_surface[gt_none_zero_mask, :] = z_vals_surface_depth_none_zero

# ========== 无深度区域 ==========
near_surface = 0.001
far_surface = torch.max(gt_depth)
z_vals_surface_depth_zero = (near_surface * (1. - t_vals_surface) + far_surface *
(t_vals_surface))
z_vals_surface_depth_zero.unsqueeze(0).repeat((~gt_none_zero_mask).sum(), 1)

# 保存结果到最终的z_vals_surface中
z_vals_surface[~gt_none_zero_mask, :] = z_vals_surface_depth_zero
```

演示z_vals的获取



```
import torch

near=0.01
far=3

# 假设采样5个点
t_vals = torch.linspace(0., 1., steps=5)

# 根据深度采样（均匀的）
z_vals = near * (1. - t_vals) + far * (t_vals)

# 每个间隔内随机采样（随机的）
mids = .5 * (z_vals[..., 1:] + z_vals[..., :-1])
upper = torch.cat([mids, z_vals[..., -1:]], -1)
lower = torch.cat([z_vals[..., :1], mids], -1)
t_rand = torch.rand(z_vals.shape)
z_vals = lower + (upper - lower) * t_rand
```

演示采样点的计算

```
import torch

# 假设只有两条射线
# det_rays_o
rays_o=torch.tensor([[[-5],[-6],[7]],[[9],[-8],[-10]]])

# det_rays_d
rays_d=torch.tensor([[[1],[2],[3]],[[0.1],[0.5],[5]]])

# 假设采样10+5个点，N_samples=5，N_surface=3
z_vals=torch.tensor([0.0100, 0.3837, 1.1313, 1.8788, 2.6263])
z_vals_surface=torch.tensor([1.000, 1.1086, 1.2122])
```

```python
# 合并N_samples和N_surface，再从小到大排序
z_vals, _ = torch.sort(torch.cat([z_vals, z_vals_surface.double()], -1), -1)

# 得到最后的采样点
pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None]
pointsf = pts.reshape(-1, 3)
```