

NeRF 原始论文复现

刘铮

摘要

本文对经典的 NeRF 原始论文进行了复现。首先克隆了 github 上 yenchelin^[1]的 pytorch 版 NeRF 代码进行学习，并训练了一次乐高铲车数据集；然后制作了自己的数据集，使用源代码再次训练了 NeRF 模型，生成结果和视频；最后是改进部分，由于能力有限，能看懂论文的原始代码已不容易，所以改进部分只进行了对 loss 函数的优化，经过测试，对比原来的数据提升了 0.75% 的峰值信噪比，略有提升但差距不大。之后如果有能力可以进一步优化。

关键词：NeRF；pytorch；损失函数优化

1 引言

本文复现的论文是《NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis》^[2]。这篇文章是三维重建领域中 NeRF 系列的开山之作，在 2020 年 NeRF 获得 ECCV best paper 之后受到了广泛的关注，如今各大 CV&CG 会议中此类工作数不胜数。作为一个没有接触过科研的研一学生，通过复现这篇原始 NeRF 文章，可以了解当今走在计算机科学前沿的研究人员的研究方向和成果，同时体会科研的整个过程，为将来自己的科研旅程打好基础。在复现论文之前，本人先学习了一些预备知识，比如 python、pytorch 和神经网络，只有知道了这些才能看懂具体的代码，才能了解论文的背景、要解决的问题以及对问题的解决方法。然后本人搜索了开源代码，发现原始 NeRF 给出的代码并非基于 pytorch，所以复现过程中，首先找到了一篇由 yenchelin 复现的、基于 pytorch 复现的 NeRF 代码进行学习。这就是我的主要学习过程，以下论文是对复现过程中学习的具体内容的总结。

2 相关工作

由于本文复现的文章是三维重建领域中 NeRF 方向的开山之作，可以进行比较的论文、实现结果并不多，所以本文除了列举传统的三维重建方法，还主要列举了其他对原始 NeRF 的优化的论文。。

2.1 三维重建简介

广义上，三维重建是指对某些三维物体或者三维的场景的一种恢复和重构，重建出来的模型，方便计算机表示和处理。在实际重建过程中，三维重建是对三维空间中的物体、场景、人体等图像描述的一种逆过程，由二维的图像还原出三维的立体物体、场景和动态人体。因此三维重建技术是在计算机中建立表达客观世界的虚拟现实的关键技术。

基于图像的三维重建是从若干幅图片计算提取出场景和物体的三维深度信息，根据获取的三维深度信息，重构出具备很强真实感的物体或者场景的三维模型的方法。该方法是涉及到多个热门领域，比如涉及到计算机图像处理、计算机图形学、计算机视觉和计算机辅助设计等很多的领域。目前，基于图像的三维重建技术已经成为一个极具潜力的热门领域，在诸多方面有着很重要的应用，比如数字城市、航天飞行、遥感测绘、数字文博等领域。

传统的建模方式多采用建模软件（例如 3DMax、AutoCAD 等）进行正向设计和建模。同时，对于已有物体、场景则可使用三维扫描仪通过逆向扫描重建后得到模型。基于计算机图形学的图像三维重建方法，其成本低廉、真实感强、自动化程度高，在诸多场景中得到应用。另外，随着计算机视觉领域深度学习技术在图像处理方面取得的成就，学术界中出现了一些基于深度神经网络的三维物体、场景重建研究。

2.2 三维重建研究现状

当前三维重建技术主要分成两大技术方向：一是基于视觉几何的传统三维重建，二是基于深度学习的三维重建。第一种三维重建方法研究时间较久远，技术相对成熟。其主要通过多视角图像对采集数据的相机位姿进行估计，再通过图像提取特征后进行比对拼接完成二维图像到三维模型的转换。第二种基于深度学习的三维重建就是本文所要复现的文章的研究方向。这种方法主要使用了深度神经网络超级强大的学习和拟合能力，可以对 RGB 或 RGBD 等图像进行三维重建。这种方法多为监督学习方法，对数据集依赖程度很高。由于数据集的收集和标注问题，目前多在体积较小的物体重建方向上研究较多。

2.3 NeRF 的训练速度优化论文

NeRF 方法产生图像时，每个像素都需要近 200 次 MLP 深度模型的前向预测。尽管单次计算规模不大，但逐像素计算完成整幅图像渲染的计算量还是很可观的。其次，NeRF 针对每个场景需要进行训练的时间也很慢。针对推理时间慢的问题研究较多，例如 AutoInt^[3]、FastNeRF^[4]等。针对训练时间慢的问题，Depth-supervised NeRF^[5]使用 SfM 的稀疏输出监督 NeRF，能够实现更少的视角输入和更快的训练速度。

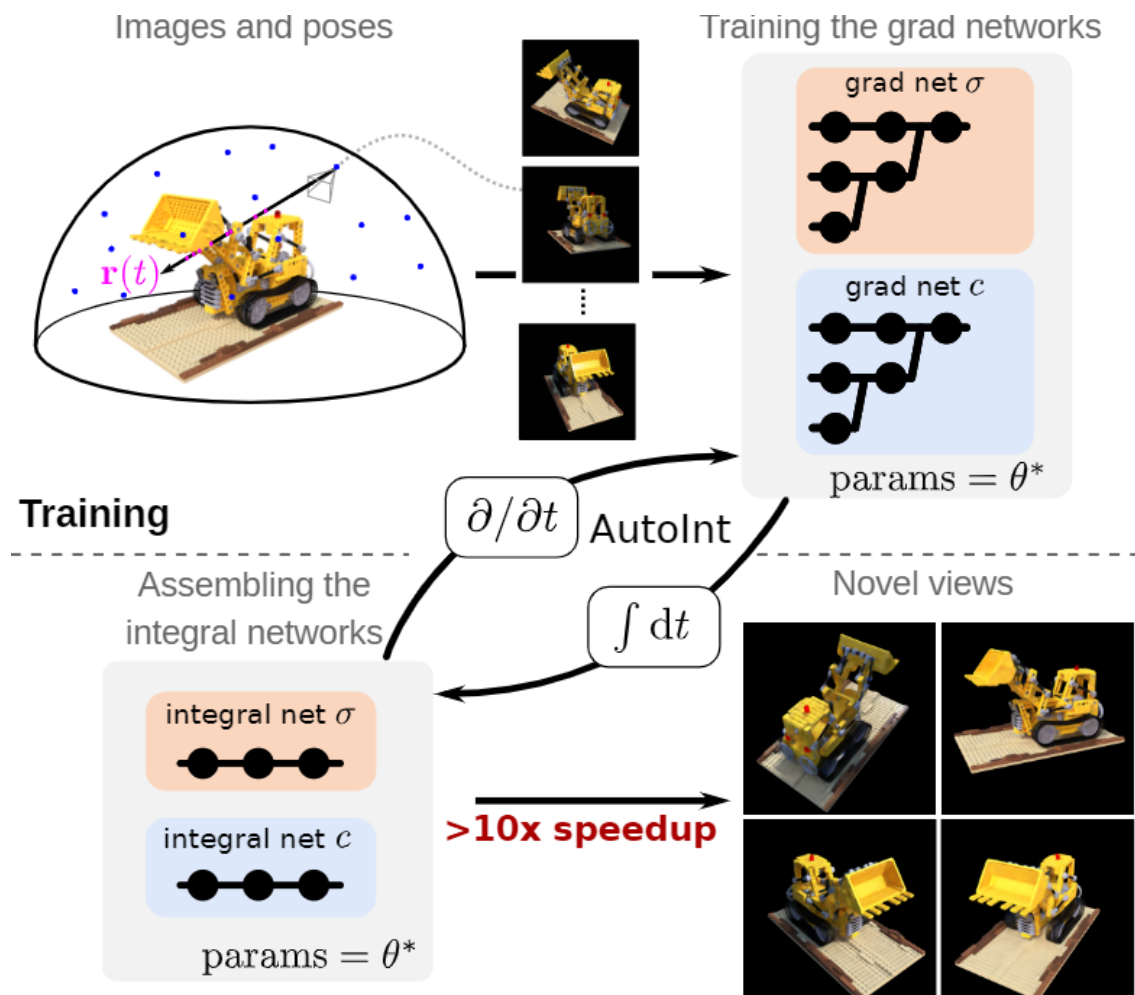


图 1: AutoInt 在测试时直接预测积分值，能够比 NeRF 快 10 倍

3 本文方法

3.1 本文方法概述

NeRF 使用隐式表达，以 2d posed images 为监督完成 novel view synthesis。通俗来说，就是用多张 2d 图片隐式重建三维场景，其展示的生成效果让人十分震撼。Neural Radiance Fields 的缩写就是 NeRF，它实际上是一种隐式的三维场景表示，之所以是隐式的原因在于 NeRF 不能像点云、网格那样以直接的三维模型让人看见。NeRF 将场景以空间中任何点的 volume density（体积密度 σ ）和有向的颜色值 Color (c) 的方式来表示，在此基础上，通过对场景进行渲染，从而得到新视角的模拟图。可以将 NeRF 简要总结为一个隐式的静态三维场景用一个 MLP 神经网络进行学习。想要训练一个 MLP 神经网络需要大量的相机拍摄的图片，通过这些图片训练好 MLP，最后就可以从各个角度渲染了。NeRF 指的是将一个连续场景表示为一个输入为单个连续 5D 向量的函数。Neural Radiance Fields 里面的 Radiance Fields 指的是映射函数，Neural Radiance Fields 则是指用神经网络拟合映射函数。映射函数输入的是 x 和 d ，而输出是 δ 和 c 。NeRF 是一种场表示，其映射函数的输入 x 是三维空间某个顶点的三维坐标，输入 d 是观察角度。而映射函数的输出 δ 是 Volume Density 体积密度作用于确定影响最终颜色确定的程度， c 是 Color 即颜色表示每个 3D 体积具有的 RGB 值。具体的实现中， x 首先被输入 MLP 神经网络中，得到 δ 和中间特征，再将 d 和中间特征输入到另外全连接层中并与测试颜色。由此证明 δ 之和空间位置相关，而颜色则于观察角度和空间位置都有关系。具体基本原理如图 2 所示，工作流程如图 3 所示：

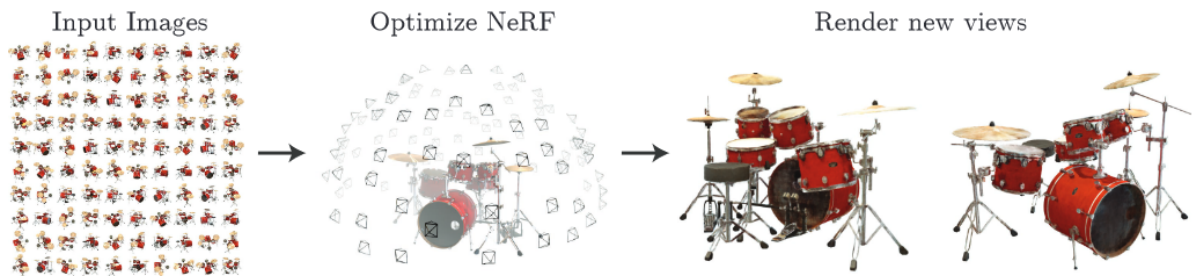


图 2: nerf 的基本原理示意

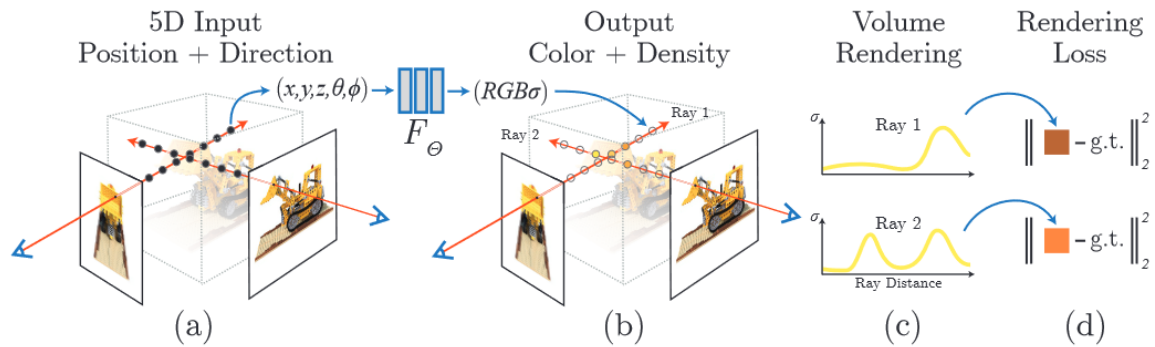


图 3: nerf 的具体过程

3.2 代码学习总结

本人对 nerf 的代码实现进行详细的学习和总结。以下图 3 是对代码整体结构的总结。

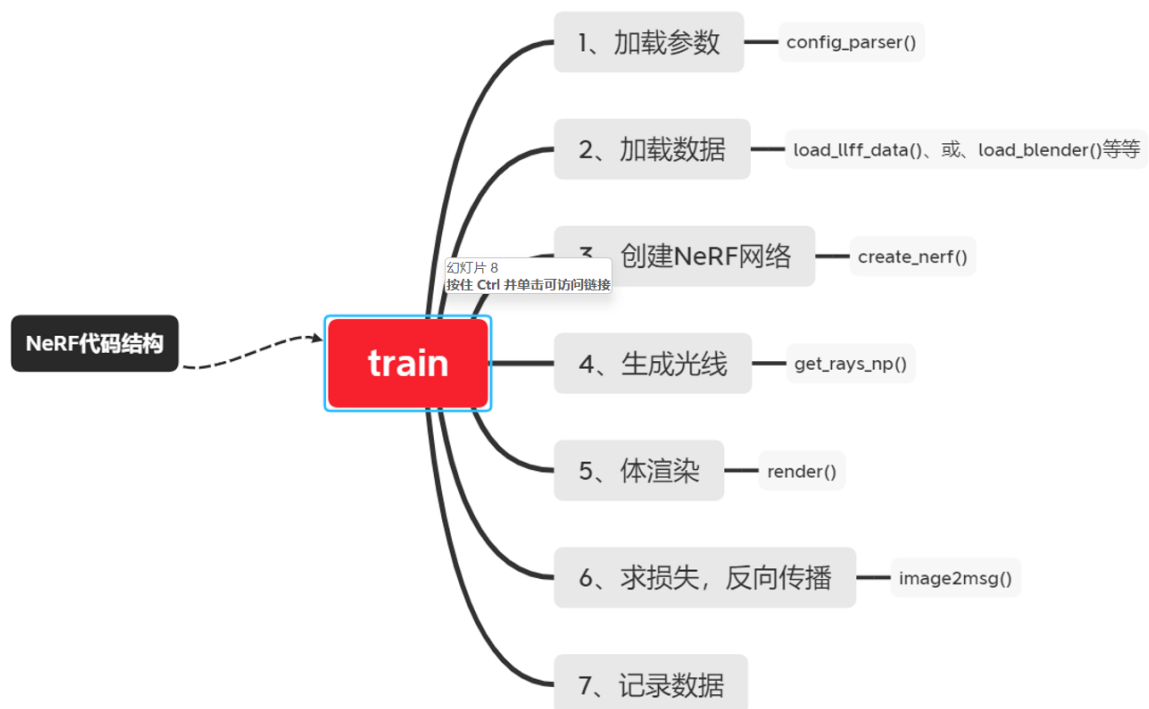


图 4: 代码整体结构

除了训练 NeRF 的整个过程，文章还给出了两个要点的解释：通过上述所描述的内容，现在已经详细的描述了将场景建模为神经辐射场和从该表示中渲染新视图的核心部分。但是，考虑到主要用这部分来实现成果的实际效率并不突出。因此在这部分中选择加入两个改进来提高当前的效率，第一种是帮助 MLP 表示高频函数的输入坐标的位置编码，第二种是分层采样过程，这可以更有效地采样高频表示。位置编码：通过实验发现直接将位置和角度输入 MLP 网络操作会导致渲染效果很差，无法

表示颜色和几何的高频变化。在此基础上进一步研究发现将输入传递到网络之前，使用高频函数将其映射到高维空间可以实现更好地拟合包含高频变化的数据。因此，将这一发现应用到 NeRF 的表示中，选用位置编码的方式将连续输入坐标映射到更高维空间，使 MLP 网络更容易逼近更高频率的函数，这大大提高了清晰度。分层采样：由于该方法的渲染策略是在每个摄像机光线的 N 个查询点上对神经辐射场网络进行密集评估，这样渲染的工作量太大导致效率很低。针对此问题提出了一种分层表示法，通过将样本按比例分配给最终渲染的预期效果，提高了渲染效率。这一部分的优化，作者选用了一种“coarse to fine”的方法，即同时优化两个网络：一个“粗略”和一个“精细”。这种方法减小采样计算开销，加快训练速度。

4 复现细节

4.1 与已有开源代码对比

与原有代码差别并不大，原因是本人之前并没有学过 pytorch，所以读懂代码已经是很不容易了。复现内容中，在学习过程中并没有多少注释可以理解代码。为了理解代码，我在原有的代码中添加了大量的注释以帮助自己看懂代码，同时也是记录了自己的学习过程与学习工作量证明。至于修改改进现有代码，只能从刚学习的神经网络知识部分入手，修改了训练过程中的 loss 函数。

```
# 为采样点加入扰动
if perturb > 0.:
    # get intervals between samples 获取样本之间的间隔
    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 这些区间的分层样本
    t_rand = torch.rand(z_vals.shape) # 生成均匀分布的数据

    # Pytest, overwrite u with numpy's fixed random numbers
    if pytest:
        np.random.seed(0)
        t_rand = np.random.rand(*list(z_vals.shape))
        t_rand = torch.Tensor(t_rand)

    z_vals = lower + (upper - lower) * t_rand

pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None] # [N_rays, N_samples, 3]

# raw = run_network(pts)这里进行了预测
raw = network_query_fn(pts, viewdirs, network_fn) # 数据送入NeRF网络
rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd,
                                                             pytest=pytest) # raw2outputs则是实现了文中的公式3，得到
```

图 5: 代码的部分注释

4.2 个人数据集的创建

个人数据集的创建主要是在学校图书馆门口的水泥空地上，放了深圳大学书包，对其拍摄了不同角度下的 21 张图片，然后使用 colmap 软件提取出了这 21 张图片的位姿信息。基本的数据集创建好了，然后是设置代码 config 部分，使得训练数据集从 lego 变为训练自己的数据集。

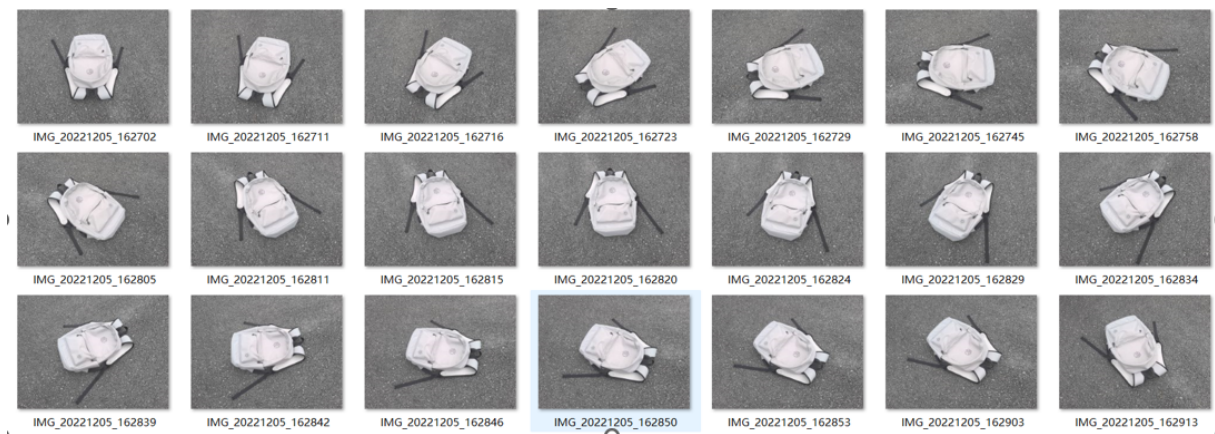


图 6: 个人创建的数据集

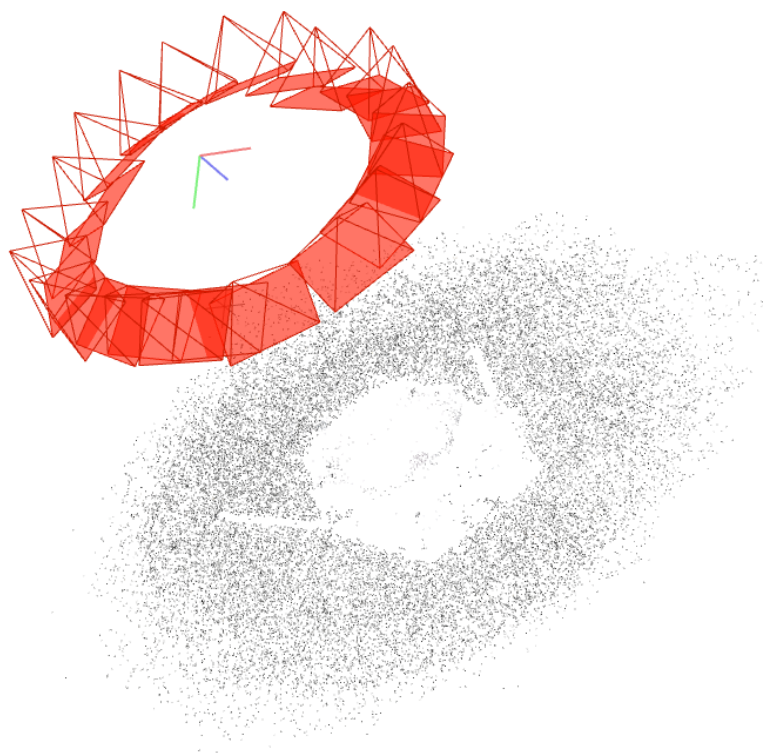


图 7: 个人创建的数据集在 colmap 中生成位姿信息

4.3 创新点

本文的复现过程的主要创新点是修改了损失函数 loss 。源代码中的损失函数是均方误差 MSE ，是回归损失函数中最常用的误差，它是预测值 $f(x)$ 与目标值 y 之间差值平方和的均值，其公式如下所示：

$$MSE = \frac{\sum_{i=1}^n (f(x) - y)^2}{n}$$

图 8: 均方损失的公式

MSE 的优点是：MSE 的函数曲线光滑、连续，处处可导，便于使用梯度下降算法，是一种常用的损失函数。而且，随着误差的减小，梯度也在减小，这有利于收敛，即使使用固定的学习速率，也能较快的收敛到最小值；MSE 的缺点是：当真实值 y 和预测值 $f(x)$ 的差值大于 1 时，会放大误差；而当差值小于 1 时，则会缩小误差，这是平方运算决定的。MSE 对于较大的误差 (>1) 给予较大的惩罚，较小的误差 (<1) 给予较小的惩罚。也就是说，对离群点比较敏感，受其影响较大。如果样本中存在离群点，MSE 会给离群点更高的权重，这就会牺牲其他正常点数据的预测效果，最终降低整体的模型性能。本文改进了 MSE，不适用均方损失，而是使用 Smooth L1 损失。以下是对 Smooth L1 损失的介绍：首先引入 L1 范数损失函数。L1 范数损失函数也被称为最小绝对值偏差 (LAD)，最小绝对值误差 (LAE)。总的说来，它是把目标值 y 与估计值 $f(x_i)$ 的绝对差值的总和 S 最小化，它和 MSE 损失的差距不大，所以优缺点是互通的。然后是 smooth L1 损失。在 Faster R-CNN 以及 SSD 中对边框的回归使用的损失函数都是 Smooth L1 作为损失函数。其实顾名思义，smooth L1 说的是光滑之后的 L1，前面说过了 L1 损失的缺点就是有折点，不光滑，那如何让其变得光滑呢？smooth L1 损失函数为：

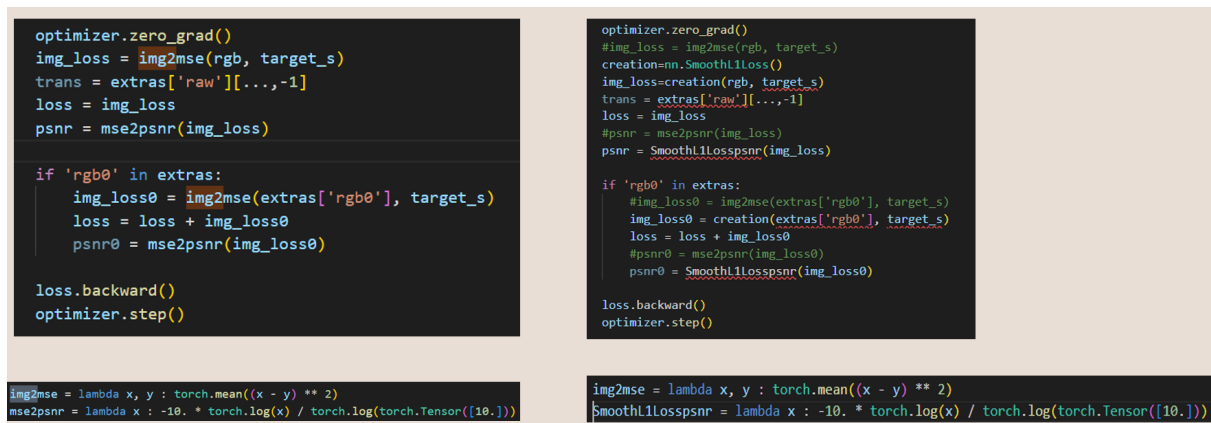
$$loss(x, y) = \frac{1}{n} \sum_i z_i$$

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

图 9: smooth L1 的公式

Smooth L1 能从两个方面限制梯度：当预测框与 ground truth 差别过大时，梯度值不至于过大；当预测框与 ground truth 差别很小时，梯度值足够小。该函数实际上就是一个分段函数，在 $[-1, 1]$ 之间实

际上就是 L2 损失，这样解决了 L1 的不光滑问题，在 $[-1,1]$ 区间外，实际上就是 L1 损失，这样就解决了离群点梯度爆炸的问题。Smooth L1 和 L1 Loss 函数的区别在于，L1 Loss 在 0 点处导数不唯一，可能影响收敛。Smooth L1 的解决办法是在 0 点附近使用平方函数使得它更加平滑。具体代码改进对比图如下：



```
optimizer.zero_grad()
img_loss = img2mse(rgb, target_s)
trans = extras['raw'][..., -1]
loss = img_loss
psnr = mse2psnr(img_loss)

if 'rgb0' in extras:
    img_loss0 = img2mse(extras['rgb0'], target_s)
    loss = loss + img_loss0
    psnr0 = mse2psnr(img_loss0)

loss.backward()
optimizer.step()

img2mse = lambda x, y : torch.mean((x - y) ** 2)
mse2psnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.]])

optimizer.zero_grad()
#img_loss = img2mse(rgb, target_s)
creation=nn.SmoothL1Loss()
img_loss=creation(rgb, target_s)
trans = extras['raw'][..., -1]
loss = img_loss
#psnr = mse2psnr(img_loss)
psnr = SmoothL1Losspsnr(img_loss)

if 'rgb0' in extras:
    #img_loss0 = img2mse(extras['rgb0'], target_s)
    img_loss0 = creation(extras['rgb0'], target_s)
    loss = loss + img_loss0
    #psnr0 = mse2psnr(img_loss0)
    psnr0 = SmoothL1Losspsnr(img_loss0)

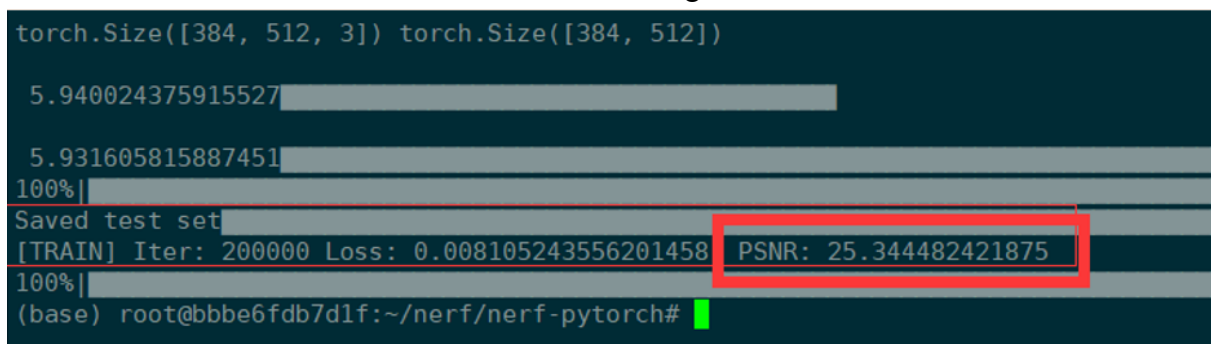
loss.backward()
optimizer.step()

img2mse = lambda x, y : torch.mean((x - y) ** 2)
SmoothL1Losspsnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.]])
```

图 10: 代码改进对比

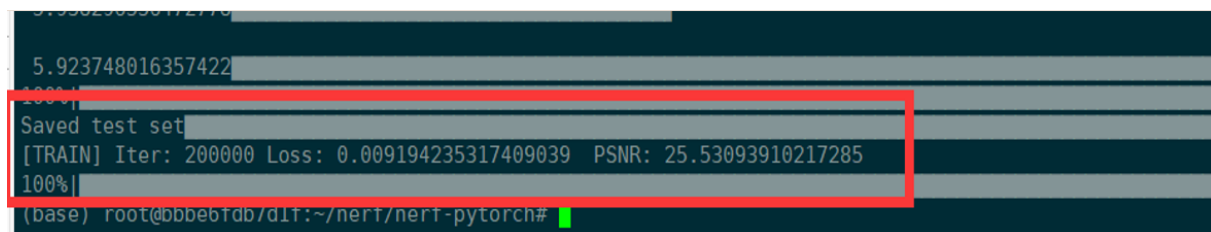
5 实验结果分析

本次复现先实验训练了原数据集 lego 模型，然后使用源代码但换了自己制作的数据集 bags 进行训练。最后再修改了代码的损失函数部分，然后使用 bags 数据集进行训练。训练结果如下图所示：



```
torch.Size([384, 512, 3]) torch.Size([384, 512])
5.940024375915527
5.931605815887451
100%|
Saved test set
[TRAIN] Iter: 200000 Loss: 0.008105243556201458 PSNR: 25.344482421875
100%|
(base) root@bbbe6fdb7d1f:~/nerf/nerf-pytorch#
```

图 11: 改进前训练结果



```
5.93023030172770
5.923748016357422
100%|
Saved test set
[TRAIN] Iter: 200000 Loss: 0.009194235317409039 PSNR: 25.53093910217285
100%|
(base) root@bbbe6fdb7d1f:~/nerf/nerf-pytorch#
```

图 12: 改进后训练结果

由图片可以看出，psnr 提升了 0.75%，实际上具体的提升并没有多大，可见简单的改进损失函数并不能对 NeRF 产生多大的帮助，所以还是得在其它方面进行改进。

6 总结与展望

本文对经典的 NeRF 原始论文进行了复现。首先克隆了 github 上 yenchinlin 的 pytorch 版 NeRF 代码进行学习，并训练了一次乐高铲车数据集；然后制作了自己的数据集，使用源代码再次训练了 NeRF 模型，生成结果和视频；最后是改进部分，由于能力有限，能看懂论文的原始代码已不容易，所以改进部分只进行了对 loss 函数的优化，经过测试，对比原来的数据提升了 0.75% 的峰值信噪比，略

有提升但差距不大。之后如果有能力可以进一步优化。

参考文献

- [1] Yenchenlin. nerf-pytorch[J]. GitHub repository, 2020.
- [2] MILDENHALL B, SRINIVASAN P P, TANCIK M, et al. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis[J/OL]. CoRR, 2020, abs/2003.08934. arXiv: 2003.08934. <https://arxiv.org/abs/2003.08934>.
- [3] LINDELL D B, MARTEL J N P, WETZSTEIN G. AutoInt: Automatic Integration for Fast Neural Volume Rendering[J/OL]. CoRR, 2020, abs/2012.01714. arXiv: 2012.01714. <https://arxiv.org/abs/2012.01714>.
- [4] GARBIN S J, KOWALSKI M, JOHNSON M, et al. FastNeRF: High-Fidelity Neural Rendering at 200FPS [J/OL]. CoRR, 2021, abs/2103.10380. arXiv: 2103.10380. <https://arxiv.org/abs/2103.10380>.
- [5] DENG K, LIU A, ZHU J, et al. Depth-supervised NeRF: Fewer Views and Faster Training for Free[J/OL]. CoRR, 2021, abs/2107.02791. arXiv: 2107.02791. <https://arxiv.org/abs/2107.02791>.