

# Instant NSR

- [zhaofuq/Instant-NSR: Pytorch implementation of fast surface reconstructor \(github.com\)](https://github.com/zhaofuq/Instant-NSR).
- 他的验证集怎么就一张图片，还是一张也属于训练集的图片：<https://github.com/zhaofuq/Instant-NSR/blob/c008cee25818c460f19966e7cec996770068e4a0/nerf/provider.py#L176>

## 关于tcnn

- 直接使用自动微分+tcnn，会报错，double backward not implemented
- 使用有限差分+tcnn，不会报错，但是可能是因为采样次数变多了，速度反而更慢
  - 有限差分需要设置 `--cuda_ray`，会导致[https://github.com/zhaofuq/Instant-NSR/blob/c008cee25818c460f19966e7cec996770068e4a0/nerf/renderer\\_sdf.py#L479](https://github.com/zhaofuq/Instant-NSR/blob/c008cee25818c460f19966e7cec996770068e4a0/nerf/renderer_sdf.py#L479)中使用 `run_cuda()` 而不是 `run()`
- batch为2048，epoch为100时
  - vanilla mlp全程速度为8.7min；一个epoch大概2s
  - tcnn fully fused mlp跑不下来，似乎是因为采样次数变多的关系，evaluation总是会显存溢出（估计这就是代码里说的unstable）；一个epoch大概10s

We add an Eikonal loss [Gropp et al. 2020] on the normal of sampled points. As shown in Fig. 3, our PyTorch implementation obtains high-quality surface result within 10 minutes, favorably compared to existing tools [PhotoScan 2019] or neural arts [Wang et al. 2021a] in terms of efficacy and efficiency. For CUDA-based acceleration similar to Instant-NGP, we adopt *finite difference function to approximate the gradient for efficient normal calculation* in Eqn. 6 and 8. Such strategy avoids tedious backpropagation of second-order gradients, which is still not fully-supported yet in existing libraries like tiny-CUDANN [Müller 2021]. To stimulate future work for more faithful CUDA implementation, we will make our version publicly available.

## 有限差分

- [https://github.com/zhaofuq/Instant-NSR/blob/c008cee25818c460f19966e7cec996770068e4a0/nerf/network\\_sdf\\_tcnn.py#L136](https://github.com/zhaofuq/Instant-NSR/blob/c008cee25818c460f19966e7cec996770068e4a0/nerf/network_sdf_tcnn.py#L136)
- 简单的前向后向差分求一阶梯度，需要额外六个点的采样，即  $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$

## 关于材质

- 与IRON不同，IRON的材质mlp有专用的编码器，而这里的材质mlp直接使用sdf输出的feature作为输入的一部分

## 关于TSDF

- [Usage of TSDF · Issue #12 · zhaofuq/Instant-NSR \(github.com\)](https://github.com/zhaofuq/Instant-NSR/issues/12)

## 实现的neus为什么这么快

- mlp很小
  - 把两个mlp都增大到256\*4之后（再大就显存溢出了），一个epoch大概10s，估算下来大概40min
- 迭代次数少
  - 100个epoch就是7,200，而NeuS需要300,000

## Instant NSR pl

- [bennyguo/instant-nsr-pl: Neural Surface reconstruction based on Instant-NGP. Efficient and customizable boilerplate for your research projects. Train NeuS in 10min!\(github.com\)](https://github.com/bennyguo/instant-nsr-pl)
- 相比Instant NSR，加入了pytorch-lightning包 (pl)

## 对比Instant NSR

- sdf的mlp和color net的mlp都还要再少一层，feature也要更短

	Instant NSR	Instant NSR pl
sdf	64*2	64*1
color net	64*3	64*2
feature	15	13

- 同样有为了使用fully fused mlp使用有限差分的做法
- 使用的sdf全为vanilla mlp: [https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/network\\_utils.py#L95](https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/network_utils.py#L95)，部分使用progressive hashgrid: <https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/configs/neus-colmap.yaml#L44>
- 如何实现progressive hashgrid?
  - 增加一个mask，需要增加等级的时候就改变这个mask，hashgrid的编码结果在返回前要经过mask: [https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/network\\_utils.py#L40](https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/network_utils.py#L40)

## 有限差分

- <https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/geometry.py#L175>
- 简单的前向后向差分求一阶梯度，需要额外六个点的采样，即  $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$
- 二阶梯度使用有限差分模拟laplace算子，即  $\frac{f(x+\epsilon)+f(x-\epsilon)-2f(x)}{\epsilon^2}$

## 关于材质

---

- 与IRON不同，IRON的材质mlp有专用的编码器，而这里的材质mlp直接使用sdf输出的feature作为输入的一部分

## 实现的neus为什么这么快

---

- mlp很小
- 体渲染只在nerfacc加速过的ray\_marching进行，而后的sdf/nerf访问和texture访问都近似于表面渲染
- 迭代次数少
  - 最多就20,000，而NeuS需要300,000
  - 学习率更大，为0.01，而NeuS用的是5e-4
  - 采用的是epoch，而不是简单的iteration
  - 用的是adamW而不是NeuS使用的adam；adamw相比adam集成了权重正则化的weight decay
- 使用了nerfacc和pl
  - nerfacc
    - neus使用了nerfacc提供的occupancy field: <https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/neus.py#L63>
    - neus渲染时先使用nerfacc的occupancy field和raymarching得到物体主题在每个光线的哪一段，再去访问网络: <https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/neus.py#L210>
    - 可以将 `t_starts` 视为光线从外部进入物体的位置，而 `t_ends` 视为光线从物体内部退出的位置。二者求中值作为实际的交点，再去访问sdf: <https://github.com/bennyguo/instant-nsr-pl/blob/e5fe3f246cf2d512494a73c727174c3ca1c3c695/models/neus.py#L226>
  - pl
    - 主要是使用了它的框架
- acceleration techniques from [Instant-NGP](#): multiresolution hash encoding and fully fused networks by [tiny-cuda-nn](#), occupancy grid pruning and rendering by [nerfacc](#)
- out-of-the-box multi-GPU and mixed precision training by [PyTorch-Lightning](#)
- hierarchical project layout that is designed to be easily customized and extended, flexible experiment configuration by [OmegaConf](#)

## nerfacc

---

- [加速Nerf训练: nerfacc快速入门 - 知乎 \(zhihu.com\)](#)
- [NerfAcc Documentation — nerfacc 0.3.5 documentation](#)
- 主要就是利用occupancy field做了加速

# pytorch-lightning

---

- [Pytorch Lightning 完全攻略 - 知乎 \(zhihu.com\)](#)
- [Welcome to ⚡ PyTorch Lightning — PyTorch Lightning 2.1.0dev documentation](#)
- 这个对速度应该没有太大影响，就是一个工程框架

## omegaconf

---

- [OmegaConf \(yaml文件\) hxxjxw的博客-CSDN博客](#)
- [omry/omegaconf: Flexible Python configuration system. The last one you will ever need. \(github.com\)](#)
- [OmegaConf — OmegaConf 2.4.0.dev0 documentation](#)
- OmegaConf 是一个基于YAML的分层配置系统，支持合并来自多个源（文件、CLI 参数、环境变量）的配置，无论配置是如何创建的，都能提供一致的 API。OmegaConf还通过结构化配置提供运行时类型安全性。