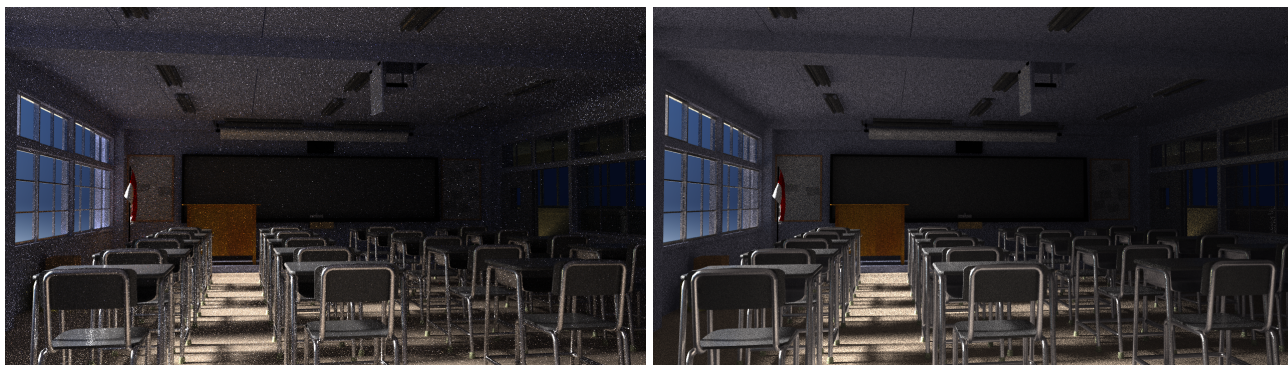


图形绘制技术 2026 Final Project

Neural Radiance Cache

姓名: 阅同学 邮箱: [REDACTED]@smail.nju.edu.cn

速览



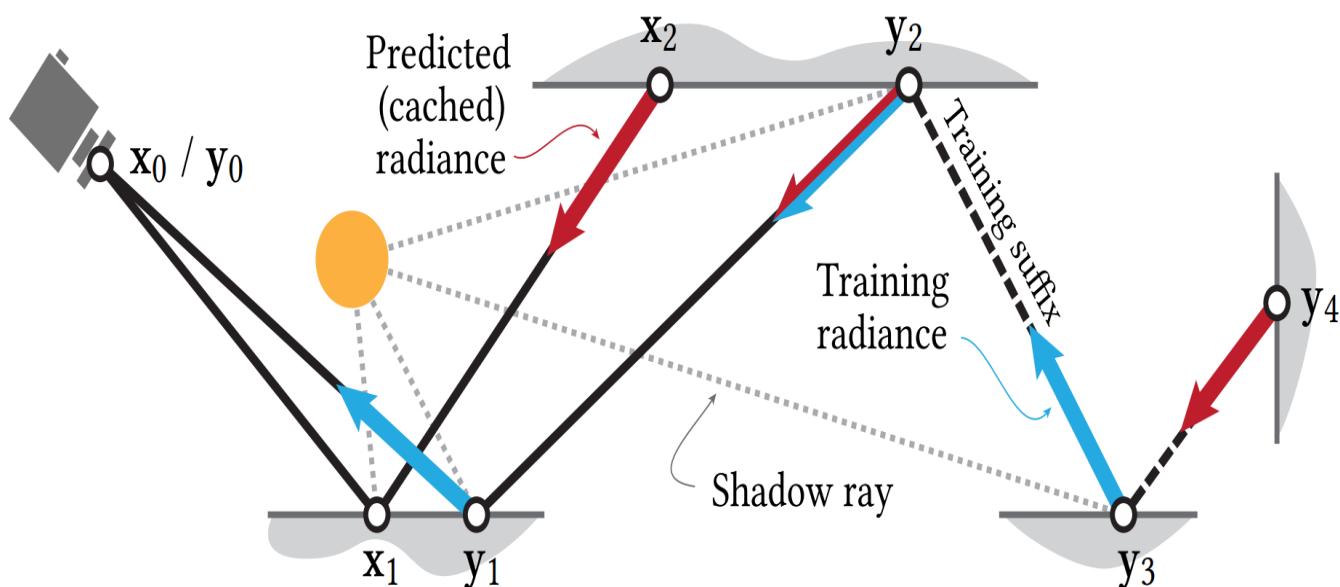
左图: PathTracing, spp=128

右图: NRC, spp=32, max_bounce=1

一. 选题理解

第一遍浏览选题清单以后,最吸引我的就是 Neural Radiance Cache.一方面,将神经网络引入 PathTracing 的构思让人感到十分新颖;另一方面,本次课程结束之后,自己却没有手动实现一次 PathTracing,多少有些遗憾.因此,我毫不犹豫地选择了这个题目.

粗浅地阅读原始论文后得知,NRC 诞生的目的其实是支持实时光追,要求在渲染的同时进行在线训练和权重更新.而本次选题的只需要在 Moer Renderer 上实现一个预训练版本的 NRC 组件,支持对离线光追渲染加速.



如图所示,在给定几何、材质、光照信息后,物体表面的每个点上,沿某个方向的 radiance 就可以确定了.在原始论文的实现中,为了拟合精确的光影效果,将该点发现、粗糙度、漫反射和镜面反射都视为了自变量,但实际上 radiance 是通过位置和方向五个维度确定的函数 F .PathTracing 中,需要不断地求解不同位置和方向上,方程左侧代表的射出 radiance.这个过程理论上是无限递归的,只有反射多次时才能近似得到真实值,因此 PathTracing 非常消耗时间.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_r(x, \omega_i \rightarrow \omega_o) L_i(x, \omega_i) (n \cdot \omega_i) d\omega_i$$

NRC 的思路就是,通过一次 PathTracing,收集过程中所有的**位置-方向-radiance** 8元组作为数据集,训练一个神经网络,而这个网络拟合的五元函数 F ,就是 Neural Radiance Cache.训练完毕后,PathTracing 可以在反弹次数很小时截断,使用 Cache 预测的射出 Radiance 并返回,和无限次反弹的收敛结果等价.

二.方案设计和字段规范

由上一节可知,预训练版 NRC 分为三个阶段:收集 radiance 数据,训练神经网络,在 PathTracing 启用 NRC.由于时间限制,这三个阶段尚未集成为一体化的功能,需要分步执行

场景.json文件字段已拓展

```
"renderer": {
  "overwrite_output_files": false,
  "adaptive_sampling": true,
  "enable_resume_render": false,
  "stratified_sampler": true,
  "scene_bvh": true,
  "spp": 128,
  "spp_step": 16,
  "checkpoint_interval": "0",
  "timeout": "0",
  # hdr文件命名
  "output_file": "classroom_128spp_nrc_view_2",
  "resume_render_file": "TungstenRenderState.dat",
  "hdr_output_file": "classroom.exr",
  "nrc": {
    # 同目录下文件,存放训练数据收集的视角(支持多视角)
    "views_file": "nrc_views.json",
    "threads": 12,
    "flush_threshold": 32768,
    # 数据收集保留的最大 bounce 数
    "max_bounce_record": 8,
    # 控制数据整体保留比率
    "global_keep": 0.25,
    # 控制不同 bounce 下数据保留比率
    "bounce_keep_probs": [1.0, 0.5, 0.25, 0.25, 0.1, 0.1, 0.1, 0.1],
    # 同目录下文件,存放 NRC 预测的测试视角(支持多视角)
    "test_views_file": "nrc_test_views.json",
    # 加载权重路径
    "checkpoint_file": "",
```



```
# NRC 预测时的最大反弹次数
"pt_max_bounce": 1,
# NRC 渲染能量补偿(1 表示无补偿)
"energy_gain": 1,
```

核心函数:NRCCollectorIntegrator::Li.

```
# 首碰检测能量补偿倍数(1 表示无补偿)
"firsthit_energy_gain": 1
}
```

用法示例(具体配置方法见仓库链接)

```
# 训练数据会以.bin文件格式保存在 Moer\NRC\runtime\scenes\classroom 下
.\target\bin\Release\Moer-NRC-Collect_r.exe .\scenes\classroom\
# 训练权重和 loss 数据保存在 Moer\NRC\runtime\checkpoints\classroom_5x128 下
python train_nrc.py --dataset-dir ..\runtime\scenes\classroom\ --steps 20000 --
batch-size 262144 --lr 0.001 --config configs\nrc_mlp.json --output-dir
..\runtime\checkpoints\classroom_5x128
# 使用 NRC 预测,渲染 hdr 图保存在当前目录下
.\target\bin\Release\Moer-NRC-Render_r.exe .\scenes\classroom\
```

三.核心实现

1.数据收集:

作用:完整执行一次路径追踪,同时把每个有效 bounce 的训练监督量整理出来,最后写入 bin.

```
// NRC/src/NRCCollectorIntegrator.cpp
Spectrum NRCCollectorIntegrator::Li(const Ray &initialRay,
std::shared_ptr<Scene>scene) {
    const double eps = 1e-4;
    Spectrum L{0.0};
    Spectrum throughput{1.0};
    Ray ray = initialRay;
    int nBounces = 0;
    auto itsOpt = scene->intersect(ray);

    std::vector<BounceInfo> bounceInfos;

    // 开启路径追踪循环
    while (true) {
        // 首次交点在光源上
        if (nBounces == 0) {
```

```
    auto evalLightRecord = evalEmittance(scene, itsOpt, ray);
    L += throughput * evalLightRecord.f;
}
}
```

```
if (!itsOpt.has_value()) {
    break;
}

auto its = itsOpt.value();
nBounces++;

// 跳过 null 材质, 不计入 bounce
if (its.material->getBxDF(its)->isNull()) {
    nBounces--;
    ray = Ray{its.position + ray.direction * eps, ray.direction};
    itsOpt = scene->intersect(ray);
    continue;
}

// 俄罗斯轮盘赌
double pSurvive = russianRoulette(throughput, nBounces);
if (sampler->sample1D() >= pSurvive) {
    break;
}
throughput /= pSurvive;

// 记录当前顶点训练样本状态
bounceInfos.emplace_back();
BounceInfo &info = bounceInfos.back();
info.position = its.position;
info.wo = normalize(-ray.direction); // 训练方向是 wo(指向上一跳/相机)
info.normal = its.geometryNormal;
info.bounce = static_cast<uint16_t>(nBounces);

// 直接光(NEE)加入渲染结果, 同时写入该顶点 localRadiance
for (int i = 0; i < nDirectLightSamples; ++i) {
    auto sampleLightRecord = sampleDirectLighting(scene, its, ray);
    auto evalScatterRecord = evalScatter(its, ray, sampleLightRecord.wi);

    if (!sampleLightRecord.f.isBlack()) {
        double misw = MISWeight(sampleLightRecord.pdf, evalScatterRecord.pdf);
        if (sampleLightRecord.isDelta) {
            misw = 1.0;
        }

        Spectrum localValue = sampleLightRecord.f * evalScatterRecord.f /
            sampleLightRecord.pdf * misw /
            nDirectLightSamples;
        L += throughput * localValue;
        info.localRadiance += localValue / pSurvive;
    }
}
```

```

    }
}

// 采样 BSDF,更新路径 throughput
auto sampleScatterRecord = sampleScatter(its, ray);
if (sampleScatterRecord.f.isBlack() || sampleScatterRecord.pdf == 0) {
    break;
}

```

```

}
info.bsdfWeight = sampleScatterRecord.f / sampleScatterRecord.pdf;
info.hasBsdfWeight = true;
throughput *= info.bsdfWeight;

ray = Ray{its.position + sampleScatterRecord.wi * eps,
sampleScatterRecord.wi};
itsOpt = scene->intersect(ray);

// BSDF 采样命中光源的贡献
auto evalLightRecord = evalEmittance(scene, itsOpt, ray);
if (!evalLightRecord.f.isBlack()) {
    double misw = MISWeight(sampleScatterRecord.pdf, evalLightRecord.pdf);
    if (sampleScatterRecord.isDelta) {
        misw = 1.0;
    }

    Spectrum bsdfHitLightValue = throughput * evalLightRecord.f * misw;
    L += bsdfHitLightValue;

    // 更新 localRadiance
    Spectrum localHitValue = info.bsdfWeight * evalLightRecord.f * misw /
pSurvive;
    info.localRadiance += localHitValue;
}
}

// 从后向前回代,把后续 bounce 的间接贡献累计到前面 bounce
if (!bounceInfos.empty()) {
    for (int i = static_cast<int>(bounceInfos.size()) - 2; i >= 0; --i) {
        if (bounceInfos[i].hasBsdfWeight) {
            bounceInfos[i].localRadiance += bounceInfos[i + 1].localRadiance *
            bounceInfos[i].bsdfWeight;
        }
    }

    // 过滤非法样本,交给 appendSample 写入二进制数据集
    for (const auto &info : bounceInfos) {
        if (!info.localRadiance.isBlack() && !info.localRadiance.hasNaN()) {
            appendSample(info);
        }
    }
}
}

```

```
return L;
}
```

说明:

- 按照json文件的参数控制收集数量, 同时按照mesh计算场景AABB, 舍弃位置在AABB以外的数据点。
- `Li`函数在正常的PT渲染过程中构建监督数据, 不需要额外再追路径。
- `info.localRadiance`的定义是当前点沿`wo`的出射辐射估计经过回代后包含后续bounce间接贡献。
- 最终样本由 `appendSample` 按 `max_bounce_record / global_keep / bounce_keep_probs` 过滤并落盘。

2. 网络训练:

神经网络使用的是 tiny-cuda-nn 的 MLP

读取 .bin 文件, 数据归一化后构建特征

```
# NRC/python/dataset.py
def build_features(records, aabb_min, aabb_max):
    positions = normalize_positions(records["position"], aabb_min, aabb_max) # [0,1]
    outgoing = normalize_vectors(records["outgoing_dir"]) # unit wo
    outgoing_01 = (0.5 * (outgoing + 1.0)).astype(np.float32) # 映射到 [0,1]
    targets = records["target_rgb"].astype(np.float32)
    features = np.concatenate([positions, outgoing_01], axis=1).astype(np.float32)
    # 6D
    return features, targets, records["bounce"].astype(np.int32)
```

将色彩先转入 log1 空间, 网络在 log1 空间训练, NRC 预测时重新转回线性空间;

```
# NRC/python/train_nrc.py
def to_log1p_targets(targets: torch.Tensor) -> torch.Tensor:
    return torch.log1p(torch.clamp_min(targets.float(), 0.0))

def logspace_mse_loss(pred_log: torch.Tensor, target_log: torch.Tensor) -> torch.Tensor:
    return torch.mean((pred_log.float() - target_log.float()) ** 2)

train_y = to_log1p_targets(train_set.targets).to(data_device)
val_y = to_log1p_targets(val_set.targets).to(data_device) if len(val_set) > 0 else None

pred = model(features) # 模型输出即 log 空间预测
data_loss = logspace_mse_loss(pred, targets)
loss = data_loss + args.tv_weight * tv_loss
```

说明:

- loss函数使用 MSE + TV_loss, TV_loss 可用于降低高频噪点, 默认不开启

3.NRC 预测

核心函数:NRCPathCutoffIntegrator::traceSampleWithCutoff.

作用:执行"PT + 截断 + NRC接管"策略,并返回单条相机路径的像素贡献.

```
// NRC/src/NRCPathCutoffIntegrator.cpp
Spectrum NRCPathCutoffIntegrator::traceSampleWithCutoff(const Ray &initialRay,
std::shared_ptr<Scene>
scene,
const Point2i &pixel) {
    const double eps = 1e-4;
    Spectrum L{0.0};
    Spectrum throughput{1.0};
    Ray ray = initialRay;
    int nBounces = 0;
    auto itsOpt = scene->intersect(ray);

    while (true) {
        if (nBounces == 0) {
            // 相机首段可见发光
            auto evalLightRecord = evalEmittance(scene, itsOpt, ray);
            L += throughput * evalLightRecord.f;
        }

        if (!itsOpt.has_value()) {
            break;
        }

        auto its = itsOpt.value();
        nBounces++;

        // 跳过 null 材质,不计入 bounce
        if (its.material->getBxDF(its)->isNull()) {
            nBounces--;
            ray = Ray{its.position + ray.direction * eps, ray.direction};
            itsOpt = scene->intersect(ray);
            continue;
        }

        // maxPtBounce 语义:
        // 0 -> 仅保留首个有效交点直接光,不用 NRC
        // k>0 -> 保留前 k 个 bounce 的 PT,k+1 起由 NRC 接管
        if (maxPtBounce_ == 0 && nBounces == 1) {
            for (int i = 0; i < nDirectLightSamples; ++i) {
                auto sampleLightRecord = sampleDirectLighting(scene, its, ray);
                auto evalScatterRecord = evalScatter(its, ray, sampleLightRecord.wi);
                if (!sampleLightRecord.f.isBlack()) {
                    double misw = MISWeight(sampleLightRecord.pdf, evalScatterRecord.pdf);
                    if (sampleLightRecord.isDelta) {
                        misw = 1.0;
                    }
                }
            }
        }
    }
}
```

```

    }
    L += throughput * sampleLightRecord.f * evalScatterRecord.f /
sampleLightRecord.pdf * misw
    / nDirectLightSamples;
    }
    }

```

```

break;
}

// 超过 PT 预算时,记录 NRC 查询状态并终止该路径
if (maxPtBounce_ > 0 && nBounces > maxPtBounce_ &&
predictor_.insideAabb(its.position)) {
    PendingQuery q;
    q.pixel = pixel;
    q.throughput = throughput;
    q.feature = predictor_.buildFeature(its.position, normalize(-ray.direction));
    pendingQueries_.push_back(q);
    break;
}

// 未截断时继续常规 PT
for (int i = 0; i < nDirectLightSamples; ++i) {
    auto sampleLightRecord = sampleDirectLighting(scene, its, ray);
    auto evalScatterRecord = evalScatter(its, ray, sampleLightRecord.wi);
    if (!sampleLightRecord.f.isBlack()) {
        double misw = MISWeight(sampleLightRecord.pdf, evalScatterRecord.pdf);
        if (sampleLightRecord.isDelta) {
            misw = 1.0;
        }
        L += throughput * sampleLightRecord.f * evalScatterRecord.f /
sampleLightRecord.pdf * misw
        / nDirectLightSamples;
    }
}

const double pSurvive = russianRoulette(throughput, nBounces);
if (sampler->sample1D() >= pSurvive) {
    break;
}
throughput /= pSurvive;

auto sampleScatterRecord = sampleScatter(its, ray);
if (sampleScatterRecord.f.isBlack() || sampleScatterRecord.pdf == 0) {
    break;
}
throughput *= sampleScatterRecord.f / sampleScatterRecord.pdf;

ray = Ray{its.position + sampleScatterRecord.wi * eps, sampleScatterRecord.wi};
itsOpt = scene->intersect(ray);

```

```

auto evalLightRecord = evalEmittance(scene, itsOpt, ray);
if (!evalLightRecord.f.isBlack()) {
    double misw = MISWeight(sampleScatterRecord.pdf, evalLightRecord.pdf);
    if (sampleScatterRecord.isDelta) {
        misw = 1.0;
    }
    L += throughput * evalLightRecord.f * misw;
}

```

```

}
}
return L;
}

```

说明:

- 该函数先尽量按 PT 计算,只有超过 `maxPtBounce` 时才把余下求值交给 NRC.
- 延迟缓存机制,只打包待查询量,网络推理由 `flushNrcQueries()` 批量调用 Python 完成.
- 最终每个查询的像素增量是 `throughput * pred_rgb * energy_gain`,用于与前段 PT 结果拼接.

```

# NRC/python/infer_nrc_batch.py
target_transform = str(ckpt.get("target_transform", "linear_clamp"))
y_raw = model(x_batch).float()

if target_transform == "log1p":
    y = torch.clamp_min(torch.exp1(y_raw), 0.0) # 逆变换到线性 radiance
elif target_transform == "linear_softplus":
    y = torch.nn.functional.softplus(y_raw)
else:
    y = torch.clamp_min(y_raw, 0.0)

```

四.成果对比

本地渲染平均统计时间:

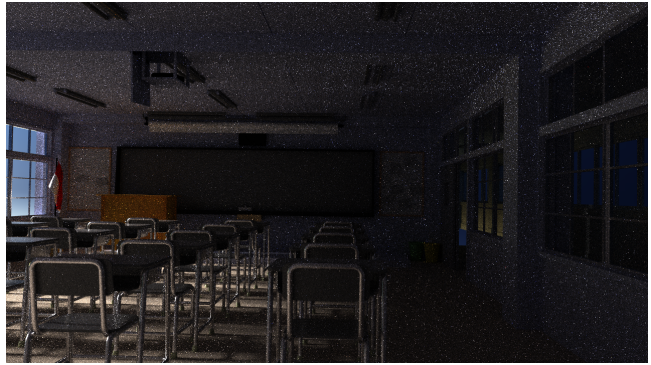
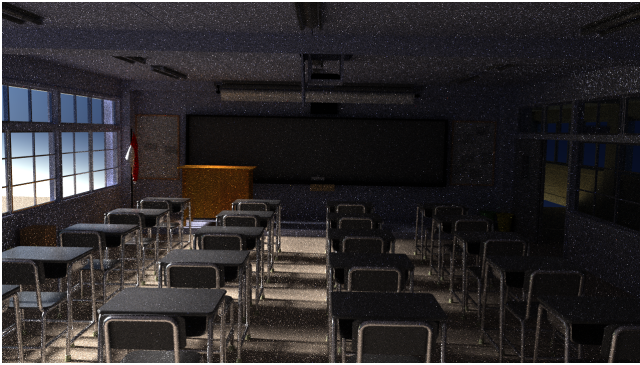
1280x720,PathTracing,spp = 128,大约 20min;

1280x720,NRC,5x128 MLP,max_bounce = 1,spp = 32,大约 3min;spp = 4,大约 30s.

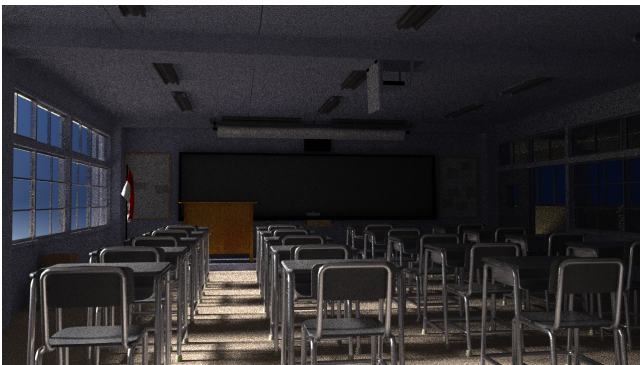
Scene1:classroom

使用 5x128 的 MLP 训练了4个视角下的 radiance 数据

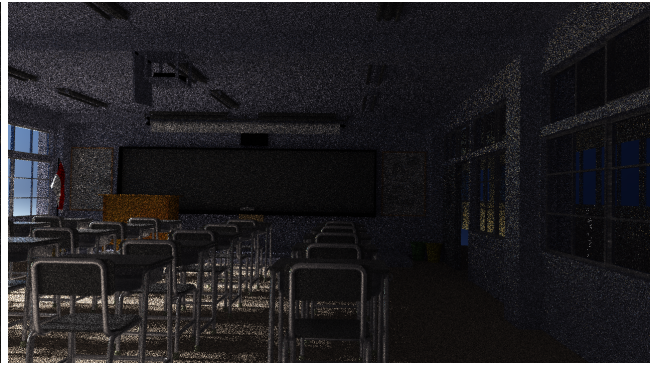
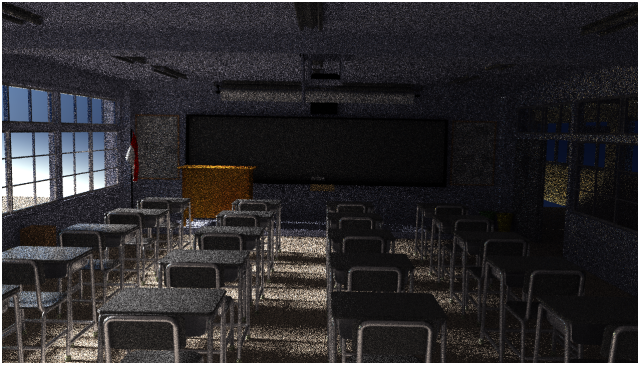
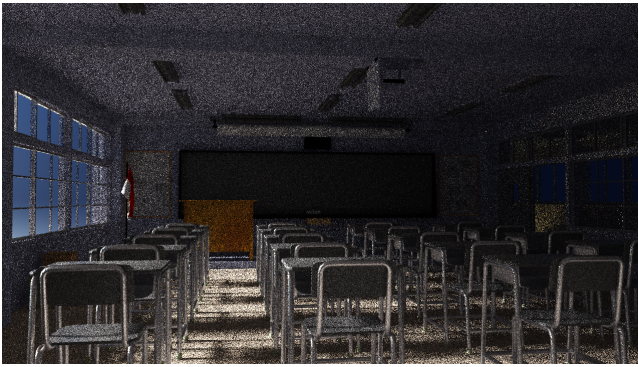
PathTracing, spp = 128



NRC, spp = 32, max_bounce = 1



NRC, spp = 4, max_bounce = 1



同时测试了两个新视角,和训练视角有良好的一致性

NRC,spp = 32,max_bounce = 1



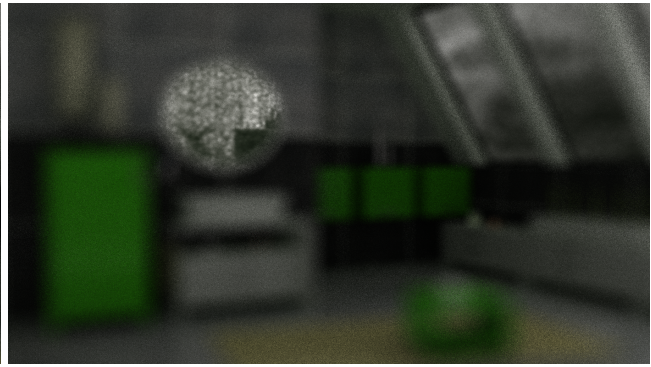
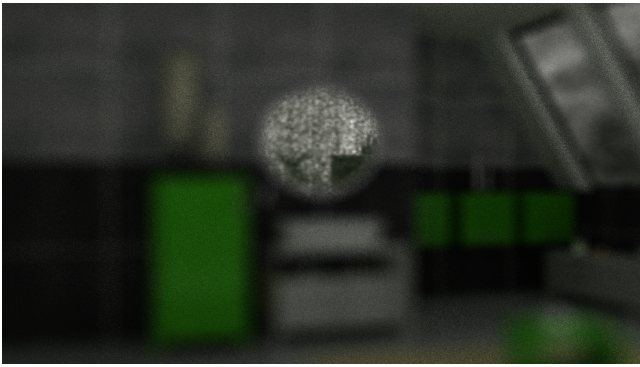
Scene2:green-bathroom

该场景介质更加复杂,效果不如 classroom,但没有时间进行进一步分析和优化

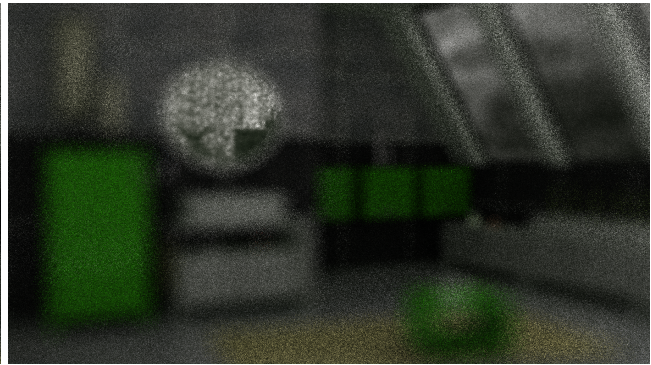
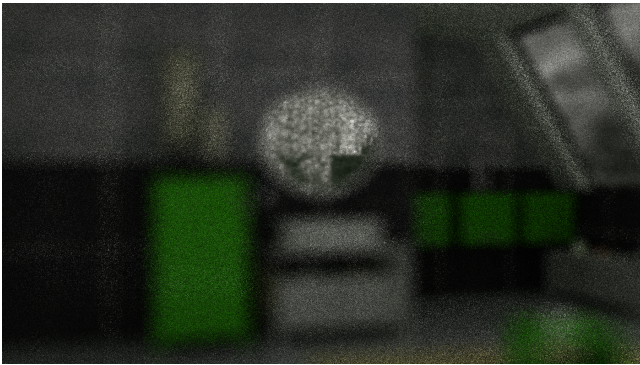
PathTracing, spp = 128



NRC, spp = 32, max_bounce = 1



NRC, spp = 4, max_bounce = 1



五.实验过程(踩坑实录)

虽然一、三阶段的主干都是 PathTracing,二阶段的网络直接调用 TCNN,但是本次作业依然极其坎坷.前后用时将近1周,由于严重缺乏训练和调参经验,我在报告写完的前一天半夜才第一次跑出正确的结果.

初始设置:

位置、方向、法线的编码分别是HashGrid、OneBlob、OneBlob.

```
scene = classroom
data_size = 5e6
step = 20000, batch_size = 262144, lr = 1e-3
```

1.train 和 val loss几乎不是 nan 就是 inf,即使我在 loss 计算中将 target 和 pred 都先转到了 log1 空间:

排查了很久才意识到可能是浮点数精度不够,从 FP16 改成了 FP32,解决了数值爆炸的问题.

2.train loss 几乎不降低,test loss 持续增大:

于是在 AI 的蛊惑下先后尝试了去掉normal,方向换成球谐,给 adam 加上 weight_decay,切换各种 loss 函数等等.最终在只保留位置和方向,使用HashGrid和球谐编码,增加 weight_decay,以及使用**相对 MSE**的配置下,成功让 test loss 在 step = 8000 之前保持降低.

按照数据量估算,8000 step 已经走过了 400 epoch,发生过拟合似乎也是正常的事情,于是增加了早停机制.重训之后在 val loss 上升时停止

3.将权重接入网络后,max_bounce = 1 下输出的渲染图非常暗,几乎没有光:

左图为 $spp = 128$ 的 PT 图;右图为 $spp = 32, max_bounce = 1$, 对网络预测结果扩大 60 倍在不同 max_bounce 下测试 NRC 发现, 几乎退化为了纯 PathTracing, max_bounce 越大亮度越接近 GT. 对 NRC 进行首撞测试, 在光线首次打到物体上时直接输出 Cache 值, 发现结果几乎为全黑, 意味着网络预测值极低. 于是从两条线排查, 一是网络训练设置, 二是收集数据和 NRC 预测时的路径追踪逻辑

沿第一条路排查, 尝试了将 clamp 换位 softplus, 基本没用;

沿第二条路排查, 在 NRC 预测阶段发现 PathTracing 在每次 bounce 处少计算了一次直接光照, 修复之后意识到只是起到了将 $max_bounce + 1$ 的作用, 没有本质区别; 在数据收集阶段, 发现 localRadiance 的累计逻辑有问题, 导致训练数据偏小. 重新收集数据后统计发现, 新数据的 hdr 分布确实更大, 但是对训练没有帮助.

至此陷入僵局, 在我的奇思妙想和侥幸心理下, 尝试了两种大力出奇迹的解决方法

(1) 对 NRC 预测值进行能量补偿:



可以看到补偿之后的亮度相当, 且物体明暗边缘和 GT 保持一致. 但是由于不同像素处需要的补偿倍数不同, 补偿结果必然与 GT 存在色差.

(2) 对训练数据进行预处理, 强行过滤低 HDR 的数据:

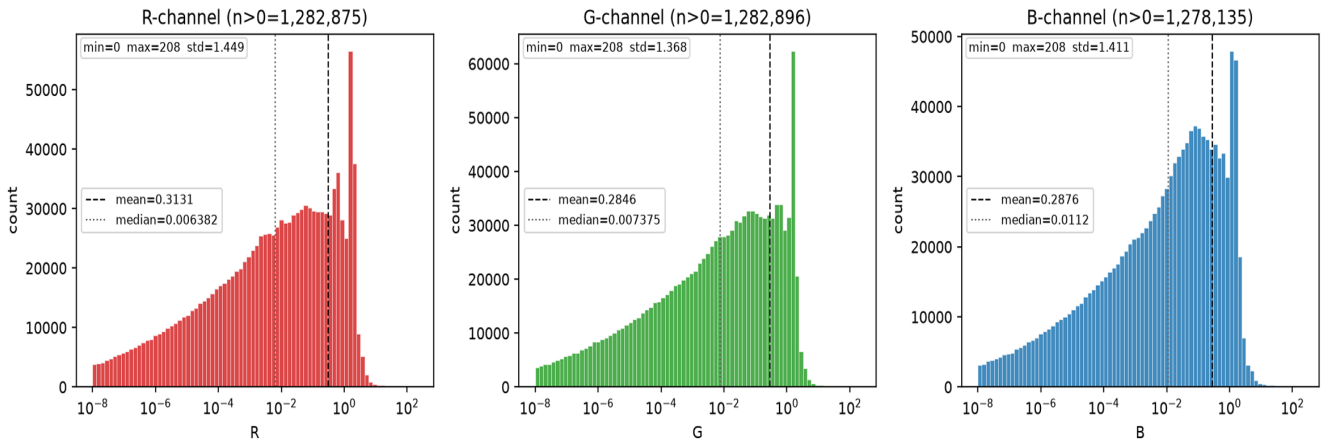
毫无疑问会丧失对阴影的预测能力, 效果极差.

4. 最后一试

至此我(和 AI)几乎想不出有任何问题. 此时我也在反复修改和控制变量后大致意识到, 10^6 次方级别数据量远超 5×128 MLP 过拟合需要的数据量, 因此修改 loss 函数、编码方式等等细节, 不可能解决网络拟合值**几乎全 0**这种结构性错误.

于是我尝试进行了最后一次分析: 结构性错误只可能出现在训练数据和训练方法上, 数据收集我自认为 PathTracing 的实现没有大的瑕疵; 训练方法就是用 MLP 拟合多元映射关系, 也没有什么技巧可言; 那剩下最大的可能性就是, 映射关系本身难以拟合. 我尝试计算了训练数据的 HDR 分布, 如下图:

targetRGB distribution — _view_0.bin (n=1,282,914)



可以看到,训练数据 HDR 三通道上的数值分散得都非常分散,因此之前在计算 loss 时将 target 和 pred 先转为 $\log_1$ 空间以避免梯度爆炸和梯度消失.但是数据横跨多个数量级时,拟合本身会存在困难.在线性空间预测HDR 难以在不同亮度的 radiance 之间下降到最优,为了降低 loss 网络会选择"摆烂",直接输出接近全 0 的值,失去区分高光 and 阴影的能力.

意识到这一点之后,我先将训练数据全部转为 $\log_1$ 空间,然后让网络在 $\log_1$ 空间内拟合;启用 NRC 时,再将预测值逆变换回线性空间,写入渲染图.终于,这次得到了正确的结果.

六.最后

1.实验缺陷:

(1)由于缺乏训练和调参的经验,本次实验从开始到跑出像样的结果,花费了过长的时间.解决预测值**几乎全 0**问题后时间所剩无几,对网络、训练细节的探索,如编码方式对比、loss 函数对比、低 spp 时降噪等,还远不够清晰,后面有时间会进一步研究;

(2)没有时间复现原始论文中对粗糙度、漫反射、镜面反射的编码,对 NRC 拟合不同光影效果的特点、性能差异来不及做细致的思考;

(3)工程上,**缓存待测数据点+批量预测/回写**只能算一个权宜之计,实际上没有真正体现出 NRC 的性能优势.后面有时间会考虑将预测网络完全集成进 C++ 侧,并优化 tensor 打包/读写方式,实现更高效的 NRC 预测功能.

2.课程总结:

本课程从辐射度量学和渲染方程出发,先后讲解了加速结构、阴影、相机、景深和运动模糊、材质、信号处理相关的内容.本课程最大的特色是将路径追踪-求解渲染方程这一主线任务贯穿始终,每一节课的内容都是带着解决这一任务某一步骤的目标而铺陈的.当学完这门课时,会惊喜地意识到自己已经有了造一个完整的轮子的能力.

3.课程建议:

课程将路径追踪-求解渲染方程的全过程讲解地深入浅出,但是的确只有亲手跑通 PathTracing 才能掌握这门课的精髓,可惜这门课的 Lab 和大作业并没有对此作强制要求.(不过两次 Lab 和大作业的选题也都很有意思)

仓库链接:

<https://github.com/mustard501/Moer-NRC/>

