
Contents

1 3DGS 原理与实现	1
1.1 高斯椭球表示、优化与代码实现	1
1.1.1 椭球定义	1
1.1.2 场景创建与 I/O	2
1.1.3 训练与优化	4
1.2 Splatting 渲染原理与实现	5
1.2.1 前向传播渲染	5
1.2.2 反向传播优化	7

3DGS 原理与实现

代码仓库与版本: [54c035f7834b564019656c3e3fcc3646292f727d](#).

1.1 高斯椭球表示、优化与代码实现

本节代码位于 `scene/gaussian_model.py` 文件中.

1.1.1 椭球定义

一个高斯球由以下几个核心属性定义:

1. 位置: `_xyz`. 表示每个高斯球在三维空间中的中心点位置.
2. 协方差: 描述高斯球的形状和方向. 为了保证矩阵在优化过程中始终为正定, 假设协方差矩阵 Σ 可以由一个缩放矩阵 S 和一个旋转矩阵 R 分解得到, 即 $\Sigma = RSS^T R^T$. 在代码中, 为了存储方便, 其存储的是一个三维向量与一个四元数.
 - 缩放: `_scaling`. 一个三维向量, 即缩放矩阵 S 的对角线元素.
 - 旋转: `_rotation`. 一个四元数, 即旋转矩阵 R 的四元数表示.
3. 颜色: 表示高斯球的颜色属性. 使用 Y_{lm} (l 阶 m 次球谐函数) 来表示颜色, 这样高斯球的颜色可以随着观察角度的变化而变化, 从而模拟更复杂的光照和材质效果. 简单来说, 假设视角是 (θ, φ) , 那么颜色可以表示为球面 \mathbb{S}^2 上的函数 $C(\theta, \varphi)$, 其可以展开为球谐函数的线性组合:

$$C(\theta, \varphi) = \sum_{l=0}^{\infty} \sum_{m=-l}^l c_{lm} Y_{lm}(\theta, \varphi).$$

在实际应用中, 通常只考虑低阶的球谐函数 (例如 $l = 0, 1, 2$), 以减少计算复杂度. 在 3DGS 代码中取的是最高 3 阶. 对于 $0, 1, \dots, l$ 阶的球谐函数, 共有 $1 + 3 + \dots + (2l + 1) = (l + 1)^2$ 个球谐系数 c_{lm} 需要优化. 在代码中, 这些系数被分开存储在两个属性中.

- 漫反射系数: `_feature_dc`. 也即球谐函数的零阶系数 c_{00} , 这代表了高斯球的基础颜色, 不随视角变化. 这是因为 $Y_{00} = 1/\sqrt{4\pi}$ 是常数.
- 高阶系数: `_feature_rest`. 也即剩下的从 1 阶到 l 阶的球谐系数.

所以, 对于一个颜色通道, 若最高阶数为 l , 则总共有 $(l + 1)^2$ 个球谐系数. 对于 RGB 三个颜色通道, 总共有 $3(l + 1)^2$ 个系数需要存储.

4. 不透明度: `_opacity`. 表示高斯球的透明度属性.

总的来说, 对于一个高斯球, 需要存储 $3 + 3 + 4 + 3(l + 1)^2 + 1 = 11 + 3(l + 1)^2$ 个参数. 若 $l = 3$, 也即最高计算 3 阶球谐系数, 则总共有 $11 + 3 \times 16 = 59$ 个参数.

由于上述参数有些需要一定的数学约束, 例如四元数的四个参数必须满足单位长度约束, 缩放参数必须为正等等, 因此在代码实现中, 3DGS 使用无约束的参数进行优化, 然后通过一些变换将其映射到有约束的参数空间中. 例如, 四元数参数通过归一化来确保其单位长度, 缩放参数通过指数映射来确保其为正值等等. 具体来说, 代码中定义了以下参数变换函数:

```
def setup_functions(self):
    def build_covariance_from_scaling_rotation(scaling, scaling_modifier, rotation):
        L = build_scaling_rotation(scaling_modifier * scaling, rotation)
        actual_covariance = L @ L.transpose(1, 2)
        symm = strip_symmetric(actual_covariance)
        return symm

    self.scaling_activation = torch.exp
    self.scaling_inverse_activation = torch.log
    self.covariance_activation = build_covariance_from_scaling_rotation
    self.opacity_activation = torch.sigmoid
    self.inverse_opacity_activation = inverse_sigmoid
    self.rotation_activation = torch.nn.functional.normalize
```

可以看到, `scaling_activation` 定义为指数函数, 用于确保缩放参数属性后为正值; `covariance_activation` 定义为通过缩放和旋转构建协方差矩阵的函数 (即 $\Sigma = RSS^T R^T$); `opacity_activation` 定义为 Sigmoid 函数, 确保不透明度在 $[0, 1]$ 范围内; `rotation_activation` 定义为归一化函数, 确保四元数属性变换后为单位四元数. 在返回实际属性时, 3DGS 会调用这些激活函数将无约束参数映射到有约束的属性空间中, 例如:

```
@property
def get_scaling(self):
    return self.scaling_activation(self._scaling)
```

1.1.2 场景创建与 I/O

首先从一个初始点云创建高斯模型, 这是训练开始的第一步, 具体代码实现如下:

```
def create_from_pcd(self, pcd : BasicPointCloud, cam_infos : int, spatial_lr_scale :
→ float):
    self.spatial_lr_scale = spatial_lr_scale
    fused_point_cloud = torch.tensor(np.asarray(pcd.points)).float().cuda()
```

```

fused_color = RGB2SH(torch.tensor(np.asarray(pcd.colors)).float().cuda())
features = torch.zeros((fused_color.shape[0], 3, (self.max_sh_degree + 1) ** 2)).float().cuda()
features[:, :3, 0] = fused_color
features[:, 3:, 1:] = 0.0

print("Number of points at initialisation : ", fused_point_cloud.shape[0])

dist2 =
→ torch.clamp_min(distCUDA2(torch.from_numpy(np.asarray(pcd.points)).float().cuda()),
→ 0.0000001)
scales = torch.log(torch.sqrt(dist2))[...,None].repeat(1, 3)
rots = torch.zeros((fused_point_cloud.shape[0], 4), device="cuda")
rots[:, 0] = 1

opacities = self.inverse_opacity_activation(0.1 *
→ torch.ones((fused_point_cloud.shape[0], 1), dtype=torch.float,
→ device="cuda"))

self._xyz = nn.Parameter(fused_point_cloud.requires_grad_(True))
self._features_dc = nn.Parameter(features[:, :, 0:1].transpose(1, 2).contiguous().requires_grad_(True))
self._features_rest = nn.Parameter(features[:, :, 1:].transpose(1, 2).contiguous().requires_grad_(True))
self._scaling = nn.Parameter(scales.requires_grad_(True))
self._rotation = nn.Parameter(rots.requires_grad_(True))
self._opacity = nn.Parameter(opacities.requires_grad_(True))
self._max_radii2D = torch.zeros((self.get_xyz.shape[0]), device="cuda")
self.exposure_mapping = {cam_info.image_name: idx for idx, cam_info in enumerate(cam_infos)}
self.pretrained_exposures = None
exposure = torch.eye(3, 4, device="cuda")[None].repeat(len(cam_infos), 1, 1)
self._exposure = nn.Parameter(exposure.requires_grad_(True))

```

首先将点云的坐标和颜色转换成 `Tensor` 类型, 初始的零阶球谐系数就是对应的 RGB 颜色, 高阶球谐系数全为零. 然后通过计算每个点到其最近邻点的距离 (`distCUDA2()` 函数) 来估算一个初始大小, 避免高斯球过大或过小. 初始化旋转为单位四元数. 初始化不透明度为一个较小的值 (0.1), 以确保初始模型较为透明. 最后将这些初始参数封装为 `nn.Parameter` 对象, 以便在训练过程中进行优化.

高斯模型可以输出为 `.ply` 格式的点云文件, 里面包含高斯球的所有属性, 代码如下:

```

def save_ply(self, path):
    mkdir_p(os.path.dirname(path))

```

```

xyz = self._xyz.detach().cpu().numpy()
normals = np.zeros_like(xyz)
f_dc = self._features_dc.detach().transpose(1,
    ↪ 2).flatten(start_dim=1).contiguous().cpu().numpy()
f_rest = self._features_rest.detach().transpose(1,
    ↪ 2).flatten(start_dim=1).contiguous().cpu().numpy()
opacities = self._opacity.detach().cpu().numpy()
scale = self._scaling.detach().cpu().numpy()
rotation = self._rotation.detach().cpu().numpy()

dtype_full = [(attribute, 'f4') for attribute in
    ↪ self.construct_list_of_attributes()]

elements = np.empty(xyz.shape[0], dtype=dtype_full)
attributes = np.concatenate((xyz, normals, f_dc, f_rest, opacities, scale,
    ↪ rotation), axis=1)
elements[:] = list(map(tuple, attributes))
el = PlyElement.describe(elements, 'vertex')
PlyData([el]).write(path)

```

这个函数将高斯球的各个属性提取出来，转换为 NumPy 数组，然后按照 PLY 文件格式的要求进行组织和存储。每个点的属性标签由 `construct_list_of_attributes()` 函数生成，首先是六个位置属性 `'x'`, `'y'`, `'z'`, `'nx'`, `'ny'`, `'nz'`，然后是颜色属性 `'f_dc_i'`, `'f_rest_i'`，最后是透明度和形状属性 `'opacity'`, `'scale_i'`, `'rot_i'`。

当然，也可以从上述格式的 PLY 文件中加载高斯模型，代码位于 `load_ply()` 函数中，这里不再赘述。

1.1.3 训练与优化

接下来我们介绍高斯模型的训练与优化过程，代码位于 `train.py` 文件中。代码的主要流程都在 `training()` 函数中，我们逐步进行解读。首先初始化各项设置：

```

1 first_iter = 0
2 tb_writer = prepare_output_and_logger(dataset)
3 gaussians = GaussianModel(dataset.sh_degree, opt.optimizer_type)
4 scene = Scene(dataset, gaussians)
5 gaussians.training_setup(opt)

```

其中 `scene` 对象维护着相机和高斯模型，并提供渲染接口。

1.2 Splatting 渲染原理与实现

1.2.1 前向传播渲染

3DGS 的渲染使用了 EWA Splatting 技术 [3], 大致流程是: 首先将每个高斯球投影到图像平面上, 形成一个椭圆形的 splat, 然后根据每个 splat 的颜色和不透明度属性, 对图像进行逐像素的颜色累积和混合.

对于一个高斯球, 设其参数为: 均值 $\mu \in \mathbb{R}^3$, 协方差 $\Sigma \in \mathbb{R}^{3 \times 3}$, 颜色 $c \in \mathbb{R}^3$ 和不透明度 $o \in \mathbb{R}$. 假设相机外参是

$$T_{\text{cw}} = \begin{bmatrix} R_{\text{cw}} & t_{\text{cw}} \\ 0 & 1 \end{bmatrix}.$$

首先将世界坐标系中的 μ 转换到相机坐标系:

$$t = T_{\text{cw}} \begin{bmatrix} \mu \\ 1 \end{bmatrix}.$$

得到相机坐标系的中心点 $t = (t_x, t_y, t_z, 1)^T$. 下一步是计算像素坐标, 但是并不是用内参矩阵直接投影到像素坐标系, 因为我们需要遵循图形学 (OpenGL/DirectX) 的标准管线, 以便利用 GPU 的硬件加速和排序算法. 因此, 我们需要把 t 转换到 NDC 坐标系 (归一化设备坐标系), 这是三维图形渲染管线中的中间坐标系统, 用于将不同分辨率的屏幕坐标统一到一个空间中, xyz 的范围都是 $[-1, 1]$. 假设内参矩阵

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

图像宽高分别为 w, h . 根据传统的方法, 利用内参得到的像素坐标应该为

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} t_x/t_z \\ t_y/t_z \\ 1 \end{bmatrix} = \begin{bmatrix} f_x t_x/t_z + c_x \\ f_y t_y/t_z + c_y \\ 1 \end{bmatrix}.$$

然后我们需要构建一个线性映射把 u, v 的值变到 $[-1, 1]$ 内, 最简单的方法为:

$$t'_x = 2 \frac{u}{w} - 1, \quad t'_y = 2 \frac{v}{h} - 1. \quad (1.1)$$

显然 $u = 0$ 时 $t'_x = -1$, $u = w$ 时 $t'_x = 1$, 在这个过程中, t'_x, t'_y 在 $[-1, 1]$ 外的点会被扔掉. 于是

$$t'_x = 2 \frac{f_x t_x/t_z + c_x}{w} - 1, \quad t'_y = 2 \frac{f_y t_y/t_z + c_y}{h} - 1.$$

于是投影矩阵 P 应该满足

$$\begin{bmatrix} t'_x \\ t'_y \\ t'_z \\ t'_w \end{bmatrix} = \begin{bmatrix} 2f_x/w & 0 & 2c_x/w - 1 & 0 \\ 0 & 2f_y/h & 2c_y/h - 1 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} t_x/t_z \\ t_y/t_z \\ 1 \\ 1/t_z \end{bmatrix}, \quad (1.2)$$

接下来处理深度 t_z , 将其映射到 $[-1, 1]$ 范围内, 假设近平面和远平面分别为 n, f , 那么 $t_z = n$ 的时候应该映射到 $t'_z = -1$, $t_z = f$ 的时候映射到 $t'_z = 1$. 而 $t'_z = A + B/t_z$, 所以

$$\begin{cases} A + B/n = -1 \\ A + B/f = 1 \end{cases} \implies \begin{cases} A = (f + n)/(f - n) \\ B = -2fn/(f - n) \end{cases}.$$

综上所述, 投影矩阵为

$$P = \begin{bmatrix} 2f_x/w & 0 & 2c_x/w - 1 & 0 \\ 0 & 2f_y/h & 2c_y/h - 1 & 0 \\ 0 & 0 & (f + n)/(f - n) & -2fn/(f - n) \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

NDC 坐标为 $t' = Pt \in \mathbb{R}^4$. 最后, 再把 t' 变回像素坐标系. 也即把 (1.1) 反过来, 得到

$$\mu'_u = \frac{w}{2}(t'_x + 1) + \frac{1}{2}, \quad \mu'_v = \frac{h}{2}(t'_y + 1) + \frac{1}{2}. \quad (1.3)$$

注意这里加上了 0.5, 是因为像素坐标系的原点在像素的中心而不是左上角. 通常情况下光心 c_x 和 c_y 位于图像中心, 即 $c_x = w/2$, $c_y = h/2$, 此时 P 可以简化为

$$P = \begin{bmatrix} 2f_x/w & 0 & 0 & 0 \\ 0 & 2f_y/h & 0 & 0 \\ 0 & 0 & (f + n)/(f - n) & -2fn/(f - n) \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

下面我们需要计算投影后的协方差矩阵. 首先, 经过外参变换点 $x \mapsto R_{\text{cw}}x + t_{\text{cw}}$, 这个变换后依然是高斯椭球, 协方差变为 $\Sigma \mapsto R_{\text{cw}}\Sigma R_{\text{cw}}^\top$. 然后是像素坐标转换

$$u = f_x \frac{t_x}{t_z} + c_x, \quad v = f_y \frac{t_y}{t_z} + c_y,$$

这不是一个仿射变换, 所以变换后实际上不再是高斯分布. 但是我们可以在 (t_x, t_y, t_z) 处做微分, 得到近似后的高斯分布. 我们有

$$J = \begin{bmatrix} \frac{\partial u}{\partial t_x} & \frac{\partial u}{\partial t_y} & \frac{\partial u}{\partial t_z} \\ \frac{\partial v}{\partial t_x} & \frac{\partial v}{\partial t_y} & \frac{\partial v}{\partial t_z} \end{bmatrix} = \begin{bmatrix} f_x/t_z & 0 & -f_x t_x/t_z^2 \\ 0 & f_y/t_z & -f_y t_y/t_z^2 \end{bmatrix}.$$

于是 splatting 后的 2D 协方差矩阵为

$$\Sigma' = J(R_{\text{cw}}\Sigma R_{\text{cw}}^\top)J^\top \in \mathbb{R}^{2 \times 2}.$$

为了方便后面的反向传播, 设四元数 $q = (q_x, q_y, q_z, q_w)$, 尺度 $s \in \mathbb{R}^3$, 那么四元数对应的旋转矩阵为

$$R = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_zq_w & 2q_xq_z + 2q_yq_w \\ 2q_xq_y + 2q_zq_w & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_xq_w \\ 2q_xq_z - 2q_yq_w & 2q_yq_z + 2q_xq_w & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}. \quad (1.4)$$

此时 $\Sigma = RSS^\top R^\top$.

在高斯球的混合过程中, [1] 使用以下排序策略. 将屏幕划分为 16×16 的小块, 然后对每个块内的 2D 高斯按照深度排序. 具体来说, 对于每个高斯球, 先计算一个包围盒, 这个包围盒的大小覆盖了 99% 的置信区间(即 3 倍标准差), 如果一个高斯球的包围盒与某个小块相交, 它就会被放入该小块的处理队列中. 然后在每个小块内, 对这些高斯球按照其深度进行排序, 深度较小的高斯球优先渲染.

然后对每个小块内排序好的高斯球进行光栅化. 对于像素 i 的颜色, 设该像素被 N 个高斯球影响, 那么这个像素的颜色为

$$C_i = \sum_{n \leq N} c_n \alpha_n T_n, \quad T_n = \prod_{m < n} (1 - \alpha_m), \quad (1.5)$$

其中 c_n 是第 n 个高斯球的颜色, 由球谐函数算出, α_n 是第 n 个高斯球在该像素处的不透明度, 由以下公式计算:

$$\alpha_n = o_n \exp(-\sigma_n), \quad \sigma_n = \frac{1}{2} \Delta_n^\top \Sigma_n'^{-1} \Delta_n,$$

这里 $\Delta_n \in \mathbb{R}^2$ 表示 2D 高斯球中心 $\mu' \in \mathbb{R}^2$ 和像素中心的偏移, o_n 是高斯球的全局不透明度属性. T_n 是透射率, 表示光线到达当前第 n 个高斯球时, 还剩下多少光没被前面的遮挡. 如果前面的高斯球 $m < n$ 非常不透明 (α_m 接近 1), 那么 $1 - \alpha_m$ 接近 0, 累乘后 T_n 就会变得非常小. 这意味着后面的高斯球会被前面的挡住, 对最终颜色的贡献几乎为 0.

1.2.2 反向传播优化

我们现在计算一个标量 loss 相对于高斯参数的梯度. 给定输出图像相对于每个像素的一个标量 loss \mathcal{L} , 我们利用链式法则反向计算到每个输入参数的梯度. 本节主要参考了 [2].

我们先介绍一些矩阵求导的符号和基本法则. 对于两个矩阵 X, Y , 定义内积

$$\langle X, Y \rangle = \text{Tr}(X^\top Y).$$

这实际上就是把 X 和 Y 展开成向量后做点积. 容易得出这个内积的一些基本性质:

$$\begin{aligned} \langle X, Y \rangle &= \langle Y, X \rangle, \\ \langle X^\top, Y^\top \rangle &= \langle X, Y \rangle, \\ \langle X, YZ \rangle &= \langle Y^\top X, Z \rangle = \langle XZ^\top, Y \rangle, \\ \langle X, Y + Z \rangle &= \langle X, Y \rangle + \langle X, Z \rangle. \end{aligned}$$

现在假设有一个矩阵函数 $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$, 以及一个矩阵变量 $X : \mathbb{R} \rightarrow \mathbb{R}^{m \times n}$, 那么复合函数 $f \circ X : \mathbb{R} \rightarrow \mathbb{R}$ 的导数为

$$(f \circ X)'(x) = \sum_{i,j} \frac{\partial f}{\partial X_{ij}}(X_{ij}(x)) X'_{ij}(x) = \left\langle \frac{\partial f}{\partial X}(X(x)), X'(x) \right\rangle,$$

我们简记为

$$\frac{\partial f}{\partial x} = \left\langle \frac{\partial f}{\partial X}, \frac{\partial X}{\partial x} \right\rangle \in \mathbb{R}.$$

这里 $\frac{\partial f}{\partial X}, \frac{\partial X}{\partial x} \in \mathbb{R}^m \times n$ 就是各个元素的偏导数组成的矩阵。在这个记号下，链式法则仍然是有效的。例如，如果 $X = AY$ ，其中 $A : \mathbb{R} \rightarrow \mathbb{R}^{m \times p}$, $Y : \mathbb{R} \rightarrow \mathbb{R}^{p \times n}$ ，那么

$$\begin{aligned} \frac{\partial f}{\partial x} &= \left\langle \frac{\partial f}{\partial X}, \frac{\partial(AY)}{\partial x} \right\rangle = \left\langle \frac{\partial f}{\partial X}, \frac{\partial A}{\partial x} Y + A \frac{\partial Y}{\partial x} \right\rangle \\ &= \left\langle \frac{\partial f}{\partial X} Y^\top, \frac{\partial A}{\partial x} \right\rangle + \left\langle A^\top \frac{\partial f}{\partial X}, \frac{\partial Y}{\partial x} \right\rangle. \end{aligned}$$

如果把 x 替换为 A_{ij} ，可以得到

$$\frac{\partial f}{\partial A_{ij}} = \left\langle \frac{\partial f}{\partial X} Y^\top, \frac{\partial A}{\partial A_{ij}} \right\rangle = \left(\frac{\partial f}{\partial X} Y^\top \right)_{ij},$$

所以

$$\frac{\partial f}{\partial A} = \frac{\partial f}{\partial X} Y^\top \in \mathbb{R}^{m \times p}.$$

同理可得

$$\frac{\partial f}{\partial Y} = A^\top \frac{\partial f}{\partial X} \in \mathbb{R}^{p \times n}.$$

于是，我们有

$$\frac{\partial f}{\partial x} = \left\langle \frac{\partial f}{\partial A}, \frac{\partial A}{\partial x} \right\rangle + \left\langle \frac{\partial f}{\partial Y}, \frac{\partial Y}{\partial x} \right\rangle.$$

下面我们分两个部分计算梯度。

深度混合梯度 我们首先把像素 i 的梯度传播到该像素影响的每个高斯球上。具体来说，对于像素 i 影响的第 n 个高斯球，我们需要计算相对于颜色的梯度 $\partial \mathcal{L}/\partial c_n \in \mathbb{R}^3$ ，不透明度的梯度 $\partial \mathcal{L}/\partial o_n \in \mathbb{R}$ ，2D 均值的梯度 $\partial \mathcal{L}/\partial \mu'_n \in \mathbb{R}^2$ ，以及 2D 协方差的梯度 $\partial \mathcal{L}/\partial \Sigma'_n \in \mathbb{R}^{2 \times 2}$ 。

对于颜色，根据 (1.5) 式，每个颜色通道 k 的梯度为

$$\frac{\partial C_i(k)}{\partial c_n(k)} = \alpha_n T_n. \quad (1.6)$$

为了加速 T_n 的计算，我们在前向传播步骤会保存 T_N 的值，然后依次向前计算 $T_{N-1}, T_{N-2}, \dots, T_1$ ，只需要利用递推公式 $T_{n-1} = T_n / (1 - \alpha_{n-1})$ 。

对于 α_n 的梯度，对于每个颜色通道 k ，有

$$\frac{\partial C_i(k)}{\partial \alpha_n} = c_n(k) T_n - \frac{S_n(k)}{1 - \alpha_n}, \quad S_n = \sum_{m>n} c_m \alpha_m T_m. \quad (1.7)$$

同样，我们也可以从后往前计算 S_n 。只需要注意到

$$S_N(k) = 0, \quad S_{n-1}(k) = c_n(k) \alpha_n T_n + S_n(k).$$

对于不透明度 α_n 和 σ_n , 有

$$\frac{\partial \alpha_n}{\partial o_n} = \exp(-\sigma_n), \quad \frac{\partial \alpha_n}{\partial \sigma_n} = -o_n \exp(-\sigma_n) = -\alpha_n. \quad (1.8)$$

对于 2D 均值 μ'_n , 有

$$\frac{\partial \sigma_n}{\partial \mu'_n} = \frac{\partial \sigma_n}{\partial \Delta_n} = \Sigma_n'^{-1} \Delta_n. \quad (1.9)$$

对于 2D 协方差 Σ'_n , 回顾 $\sigma_n = \frac{1}{2} \Delta_n^\top \Sigma_n'^{-1} \Delta_n$, 我们先计算

$$\begin{aligned} \frac{\partial \sigma_n}{\partial x} &= \left\langle \frac{\partial \sigma_n}{\partial (\Sigma_n'^{-1})}, \frac{\partial (\Sigma_n'^{-1})}{\partial x} \right\rangle = \left\langle \frac{1}{2} \Delta_n \Delta_n^\top, -\Sigma_n'^{-1} \frac{\partial \Sigma_n'}{\partial x} \Sigma_n'^{-1} \right\rangle \\ &= \left\langle -\frac{1}{2} \Sigma_n'^{-1} \Delta_n \Delta_n^\top \Sigma_n'^{-1}, \frac{\partial \Sigma_n'}{\partial x} \right\rangle, \end{aligned}$$

再取 $x = (\Sigma'_n)_{ij}$, 便得到

$$\frac{\partial \sigma_n}{\partial \Sigma'_n} = -\frac{1}{2} \Sigma_n'^{-1} \Delta_n \Delta_n^\top \Sigma_n'^{-1}. \quad (1.10)$$

投影梯度 接下来我们处理投影变换导致的梯度传播. 给定 \mathcal{L} 相对于 2D 均值 μ'_n 和 2D 协方差 Σ'_n 的梯度, 我们需要传播到相对于 3D 均值 μ 和 3D 协方差 Σ 的梯度. 这个步骤只会涉及到单个高斯球, 所以我们可以扔掉下标 n , 目标是计算 $\partial \mathcal{L} / \partial \mu \in \mathbb{R}^3$ 以及 $\partial \mathcal{L} / \partial \Sigma \in \mathbb{R}^{2 \times 2}$.

首先我们注意到 loss \mathcal{L} 可以视为 μ' 和 Σ' 的函数, 而 μ' 和 Σ' 又是相机系坐标 $t \in \mathbb{R}^3$ (这里我们去掉最后一个常数 1) 的函数, 所以 \mathcal{L} 相对于相机系坐标 $t \in \mathbb{R}^3$ 的梯度为

$$\frac{\partial \mathcal{L}}{\partial t_i} = \left\langle \frac{\partial \mathcal{L}}{\partial \mu'}, \frac{\partial \mu'}{\partial t_i} \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma'}, \frac{\partial \Sigma'}{\partial t_i} \right\rangle. \quad (1.11)$$

其中 $\partial \mathcal{L} / \partial \mu' \in \mathbb{R}^2$ 和 $\partial \mathcal{L} / \partial \Sigma' \in \mathbb{R}^{2 \times 2}$ 是前一步计算得到的梯度. 所以接下来我们需要计算 $\partial \mu' / \partial t_i$ 和 $\partial \Sigma' / \partial t_i$.

对于 2D 均值 μ' 相对于 t 的梯度, 这就是矩阵 J , 即

$$\frac{\partial \mu'}{\partial t} = J = \begin{bmatrix} f_x/t_z & 0 & -f_x t_x/t_z^2 \\ 0 & f_y/t_z & -f_y t_y/t_z^2 \end{bmatrix} = \begin{bmatrix} \frac{\partial \mu'}{\partial t_x} & \frac{\partial \mu'}{\partial t_y} & \frac{\partial \mu'}{\partial t_z} \end{bmatrix}. \quad (1.12)$$

对于 2D 协方差 Σ' , 回顾 $\Sigma' = J(R_{\text{cw}} \Sigma R_{\text{cw}}^\top) J^\top$, 其中只有 J 是 t 的函数. 我们有

$$\frac{\partial \Sigma'}{\partial t_i} = \frac{\partial (JR_{\text{cw}})}{\partial t_i} \Sigma (JR_{\text{cw}})^\top + (JR_{\text{cw}}) \Sigma \frac{\partial (JR_{\text{cw}})^\top}{\partial t_i}, \quad (1.13)$$

对于 J , 有

$$\begin{aligned} \frac{\partial J}{\partial t_x} &= \begin{bmatrix} 0 & 0 - f_x/t_z^2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \frac{\partial J}{\partial t_y} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -f_y/t_z^2 \end{bmatrix}, \\ \frac{\partial J}{\partial t_z} &= \begin{bmatrix} -f_x/t_z^2 & 0 & 2f_x t_x/t_z^3 \\ 0 & -f_y/t_z^2 & 2f_y t_y/t_z^3 \end{bmatrix}. \end{aligned}$$

把 (1.12) 和 (1.13) 带入 (1.11) 中, 就可以得到

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial t_i} &= \left\langle \frac{\partial \mathcal{L}}{\partial \mu}, \frac{\partial \mu'}{\partial t_i} \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma}, \frac{\partial J}{\partial t_i} R_{\text{cw}} \Sigma R_{\text{cw}}^T J^T + J R_{\text{cw}} \Sigma R_{\text{cw}}^T \frac{\partial J^T}{\partial t_i} \right\rangle \\ &= \left\langle \frac{\partial \mathcal{L}}{\partial \mu'}, \frac{\partial \mu'}{\partial t_i} \right\rangle + \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma} J R_{\text{cw}} \Sigma^T R_{\text{cw}}^T + \left(\frac{\partial \mathcal{L}}{\partial \Sigma} \right)^T J R_{\text{cw}} \Sigma R_{\text{cw}}^T, \frac{\partial J}{\partial t_i} \right\rangle, \\ &= \left\langle \frac{\partial \mathcal{L}}{\partial \mu'}, \frac{\partial \mu'}{\partial t_i} \right\rangle + \left\langle \left(\frac{\partial \mathcal{L}}{\partial \Sigma'} + \left(\frac{\partial \mathcal{L}}{\partial \Sigma'} \right)^T \right) J R_{\text{cw}} \Sigma R_{\text{cw}}^T, \frac{\partial J}{\partial t_i} \right\rangle,\end{aligned}\quad (1.14)$$

然后只需要计算 t_i 对 μ 的梯度即可完成对 μ 的反向传播. 由于 $t = R_{\text{cw}}\mu + t_{\text{cw}}$, 所以

$$\frac{\partial \mathcal{L}}{\partial \mu} = R_{\text{cw}}^T \frac{\partial \mathcal{L}}{\partial t}. \quad (1.15)$$

下面我们转向对 Σ 的梯度传播. 由于 μ' 不依赖于 Σ , 所以只有 Σ' 会传递这部分的梯度, 于是

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \Sigma_{ij}} &= \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma'}, \frac{\partial \Sigma'}{\partial \Sigma} \right\rangle = \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma'}, J R_{\text{cw}} \frac{\partial \Sigma}{\partial \Sigma_{ij}} R_{\text{cw}}^T J^T \right\rangle \\ &= \left\langle R_{\text{cw}}^T J^T \frac{\partial \mathcal{L}}{\partial \Sigma'} J R_{\text{cw}}, \frac{\partial \Sigma}{\partial \Sigma_{ij}} \right\rangle.\end{aligned}$$

于是

$$\frac{\partial \mathcal{L}}{\partial \Sigma} = R_{\text{cw}}^T J^T \frac{\partial \mathcal{L}}{\partial \Sigma'} J R_{\text{cw}}. \quad (1.16)$$

尺度和旋转梯度 最后, 我们只需要把 Σ 的梯度传播到尺度 s 和旋转四元数 q 上. 回顾 $\Sigma = RSS^T R^T$, 于是

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial S_{ij}} &= \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma}, \frac{\partial (RSS^T R^T)}{\partial S_{ij}} \right\rangle = \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma}, R \frac{\partial S}{\partial S_{ij}} S^T R^T + RS \frac{\partial S^T}{\partial S_{ij}} R^T \right\rangle \\ &= \left\langle R^T \frac{\partial \mathcal{L}}{\partial \Sigma} R, \frac{\partial S}{\partial S_{ij}} S^T + S \frac{\partial S^T}{\partial S_{ij}} \right\rangle = \left\langle R^T \left(\frac{\partial \mathcal{L}}{\partial \Sigma} + \left(\frac{\partial \mathcal{L}}{\partial \Sigma} \right)^T \right) RS, \frac{\partial S}{\partial S_{ij}} \right\rangle,\end{aligned}$$

所以

$$\frac{\partial \mathcal{L}}{\partial S} = R^T \left(\frac{\partial \mathcal{L}}{\partial \Sigma} + \left(\frac{\partial \mathcal{L}}{\partial \Sigma} \right)^T \right) RS. \quad (1.17)$$

最后, 注意到 S 是对角矩阵, 且 $S_{ii} = s_i$, 所以

$$\frac{\partial S}{\partial s_i} = E_{ii}.$$

这里 E_{ii} 是只有第 (i, i) 元素为 1, 其余元素为 0 的矩阵.

对于四元数, 首先有

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial R_{ij}} &= \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma}, \frac{\partial (RSS^T R^T)}{\partial R_{ij}} \right\rangle = \left\langle \frac{\partial \mathcal{L}}{\partial \Sigma}, \frac{\partial R}{\partial R_{ij}} SS^T R^T + RSS^T \frac{\partial R^T}{\partial R_{ij}} \right\rangle \\ &= \left\langle \left(\frac{\partial \mathcal{L}}{\partial \Sigma} + \left(\frac{\partial \mathcal{L}}{\partial \Sigma} \right)^T \right) RSS^T, \frac{\partial R}{\partial R_{ij}} \right\rangle.\end{aligned}$$

也即

$$\frac{\partial \mathcal{L}}{\partial R} = \left(\frac{\partial \mathcal{L}}{\partial \Sigma} + \left(\frac{\partial \mathcal{L}}{\partial \Sigma} \right)^\top \right) R S S^\top. \quad (1.18)$$

最后, 根据旋转矩阵和四元数的关系 (1.4), 可以计算

$$\begin{aligned} \frac{\partial R}{\partial q_x} &= 2 \begin{bmatrix} 0 & q_y & q_z \\ q_y & -2q_x & q_w \\ q_z & q_w & -2q_x \end{bmatrix}, & \frac{\partial R}{\partial q_y} &= 2 \begin{bmatrix} -2q_y & q_x & -q_w \\ q_x & 0 & q_z \\ q_w & q_z & -2q_y \end{bmatrix}, \\ \frac{\partial R}{\partial q_z} &= 2 \begin{bmatrix} -2q_z & q_w & q_x \\ -q_w & -2q_z & q_y \\ q_x & q_y & 0 \end{bmatrix}, & \frac{\partial R}{\partial q_w} &= 2 \begin{bmatrix} 0 & -q_z & q_y \\ q_z & 0 & -q_x \\ -q_y & q_x & 0 \end{bmatrix}. \end{aligned}$$

Bibliography

- [1] Bernhard Kerbl et al. “3D Gaussian Splatting for Real-Time Radiance Field Rendering”. In: **ACM Transactions on Graphics** 42.4 (July 2023). URL: <https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/>.
- [2] Vickie Ye and Angjoo Kanazawa. **Mathematical Supplement for the gsplat Library**. 2023. arXiv: [2312.02121 \[cs.MS\]](https://arxiv.org/abs/2312.02121). URL: <https://arxiv.org/abs/2312.02121>.
- [3] M. Zwicker et al. “EWA splatting”. In: **IEEE Transactions on Visualization and Computer Graphics** 8.3 (2002), pp. 223–238. doi: [10.1109/TVCG.2002.1021576](https://doi.org/10.1109/TVCG.2002.1021576).