[论文解析]Scaffold-GS：视图自适应渲染的重建

Scaffold-GS：视图自适应渲染的重建

一种隐式和显式表达的结合方法，三个主要贡献点：1. 三维高斯表示场景，锚点和voxel结合去掉冗余高斯，更注重场景的结构表示；2. 用MLP来训练高斯
的增删策略替换了原版的高斯增删策略；

论文地址：[2312.00109] Scaffold-GS: Structured 3D Gaussians for View-Adaptive Rendering (arxiv.org)

源码地址：GitHub - city-super/Scaffold-GS: [CVPR 2024 Highlight] Scaffold-GS: Structured 3D Gaussians for View-Adaptive Rendering

项目地址：Scaffold-GS: Structured 3D Gaussians for View-Adaptive Rendering (city-super.github.io)

Scaffold-GS

论文解读（来自ChatGpt ）

研究背景与动机

在计算机视觉 领域，尤其是3D场景渲染方面，实现高质量的视觉效果与实时渲染 性能 的平衡一直是一个挑战。尽管最近的方法如3D Gaussian Spla
渲染质量和速度上取得了显著进展，但它们在处理大规模场景和复杂光照条件下的冗余性和鲁棒性不足。

研究贡献

本研究的核心贡献在于提出了Scaffold-GS，这是一种基于锚点的分层3D高斯表示方法，能够动态适应不同的观察角度和距离，显著提升了渲染的鲁棒性
包括：

1. **结构化3D场景表示**：通过从结构从运动（SfM）初始化的稀疏锚点网格，引导局部3D高斯的分布，形成层次化和区域感知的场景表示。
    1. 从论文的图6中可以看到点云的结构特征比较明显，聚类算法可以将物体点云聚集为同一类；
2. **视图自适应属性预测**：在视锥体内，基于锚点特征和视图依赖信息实时预测神经高斯的属性，如不透明度和颜色，以适应多样化的视图方向和距离。
3. **锚点生长与修剪策略**：开发了一种基于神经高斯重要性的锚点生长和修剪机制，以提高场景覆盖的可靠性。
    1. 这一点其实是适配他们自己提出的anchor特性，**感觉可以在destiny和prune的条件部分做些优化和修改(可惜暂时没有好的想法)**

方法

Scaffold-GS采用了一种新颖的方法，包括：

- **锚点初始化**：利用SfM点云构建稀疏锚点网格，为场景提供一个粗略的几何框架。原版是用SfM的点作为高斯初始化点，这里将点云体素化，每一个v
anchor，并规定了voxel尺寸，为每个anchor构造了相应的feature bank；
- **神经高斯衍生**：从每个锚点生成一组神经高斯，其属性通过小型多层感知器（MLP）基于视图方向和距离动态预测。
- **属性预过滤策略**：为了提高光栅化效率，引入了基于不透明度阈值的预过滤步骤，以减少计算负载。

实验设计与评估

研究者在多个公共数据集上对Scaffold-GS进行了全面评估，包括：

- **数据集**：涵盖了从Mip-NeRF360到Tanks&Temples，再到DeepBlending和Blender合成数据集的多样化场景。
- **评估指标**：采用了峰值信噪比（PSNR）、结构相似性（SSIM）、感知损失（LPIPS）等指标，以及存储大小和帧率（FPS）来衡量模型的性能和效率
- **细节设置**：设置每个voxel为10个3D高斯，MLP设置为2层，RELU激活函数，隐含层为32维度，SSIM和vol的损失权重为0.2和0.001；
- **结果分析**：Scaffold-GS在保持与原始3D-GS相似的渲染速度的同时，显著减少了存储需求，并在具有挑战性的场景中展现出更好的视觉质量和鲁棒性

结论与未来工作

Scaffold-GS通过其结构化的3D高斯和视图自适应属性预测，为3D场景渲染领域提供了一种高效的解决方案。论文还讨论了该方法的局限性，并对未来可了展望，包括在更大规模场景中的应用和对无纹理区域的处理策略。

代码及论文具体细节

如何进行锚点初始化

1. 从SfM点云进行voxel初始化，生成了 $V = \{\frac{P}{\epsilon}\} \cdot \epsilon$ 个anchor锚点，anchor的生成就是简单的voxel采样

```python
def create_from_pcd(self, pcd : BasicPointCloud, spatial_lr_scale : float):
    self.spatial_lr_scale = spatial_lr_scale
    points = pcd.points[::self.ratio]

    if self.voxel_size <= 0: # 保留了原版的点云初始化
        init_points = torch.tensor(points).float().cuda()
        init_dist = distCUDA2(init_points).float().cuda()
        median_dist, _ = torch.kthvalue(init_dist, int(init_dist.shape[0]*0.5))
        self.voxel_size = median_dist.item()
        del init_dist
        del init_points
        torch.cuda.empty_cache()

    print(f'Initial voxel_size: {self.voxel_size}')


    points = self.voxelize_sample(points, voxel_size=self.voxel_size) # 下采样
    fused_point_cloud = torch.tensor(np.asarray(points)).float().cuda()
    offsets = torch.zeros((fused_point_cloud.shape[0], self.n_offsets, 3)).float().cuda()
    # anchor的feature初始化
    anchors_feat = torch.zeros((fused_point_cloud.shape[0], self.feat_dim)).float().cuda()

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

    dist2 = torch.clamp_min(distCUDA2(fused_point_cloud).float().cuda(), 0.0000001)
    scales = torch.log(torch.sqrt(dist2))[...,None].repeat(1, 6)

    rots = torch.zeros((fused_point_cloud.shape[0], 4), device="cuda")
    rots[:, 0] = 1

    opacities = inverse_sigmoid(0.1 * torch.ones((fused_point_cloud.shape[0], 1), dtype=torch.float, device="cuda"))
        # 以下变量为优化器需要进行优化的参数
    self._anchor = nn.Parameter(fused_point_cloud.requires_grad_(True))
    self._offset = nn.Parameter(offsets.requires_grad_(True))
    self._anchor_feat = nn.Parameter(anchors_feat.requires_grad_(True))
    self._scaling = nn.Parameter(scales.requires_grad_(True))
    self._rotation = nn.Parameter(rots.requires_grad_(False))
    self._opacity = nn.Parameter(opacities.requires_grad_(False))
      self.max_radii2D = torch.zeros((self.get_anchor.shape[0]), device="cuda")
```

1. 需要优化的参数有下面几个：

- `self._anchor` 的xyz是需要进行优化的参数，也就是anchor是需要不断调整的；
- `self._offset` 是每个anchor衍生出的Gaussian的位置，代码中设置为10，每个anchor生成10个Gaussian，这里的offset相当于以anchor为原点的值码中的表达就是方向向量；
- `self._anchor_feat` 就是下面要讲的每个anchor的特征，这个特征量应该是作为MLP的输入层；
- `self._scaling` 初始化是一个6维参数，前3维是offset的缩放系数，后3维表示neural-gs的cov的初值，对应论文公式8中的l_v

```
# render里面的 generate_neural_gaussians函数
# post-process cov
scaling = scaling_repeat[:,3:] * torch.sigmoid(scale_rot[:,:3]) # * (1+torch.sigmoid(repeat_dist))
rot = pc.rotation_activation(scale_rot[:,3:7])

# post-process offsets to get centers for gaussians
offsets = offsets * scaling_repeat[:,:3]
xyz = repeat_anchor + offsets
```

- `self._rotation`， `self._opacity` 初始化，后续在 `adjust_anchor()` 进行更新调用

MLP如何进行初始化

1. 每个锚点配置了4个小型MLP用来计算三维高斯需要优化的属性，也就是用MLP对属性进行预测，而不是原版的优化策略；**MLP的初始化细节和结构 `GaussianModel` 里面的 `setup_functions` 和 `__init__` 函数。**
2. 第一个MLP是feature bank的权重，是每个anchor都有的32维特征(个人感觉32就是为了方便下采样和小型网络)，通过下采样扩展出另外两种特征；这个featbank设计的，如下图
3. 剩余3个MLP是对应了三维高斯的属性，如下图所示；N是anchor个数，由于颜色是通过MLP预测，所以没有SH系数；MLP的设计在论文的第7节。

训练过程

1. 训练初始化：模型初始化，学习率设置

```
l = [
# 高斯参数
    {'params': [self._anchor], 'lr': training_args.position_lr_init * self.spatial_lr_scale, "name": "anchor"},
    {'params': [self._offset], 'lr': training_args.offset_lr_init * self.spatial_lr_scale, "name": "offset"},
    {'params': [self._anchor_feat], 'lr': training_args.feature_lr, "name": "anchor_feat"},
    {'params': [self._opacity], 'lr': training_args.opacity_lr, "name": "opacity"},
    {'params': [self._scaling], 'lr': training_args.scaling_lr, "name": "scaling"},
    {'params': [self._rotation], 'lr': training_args.rotation_lr, "name": "rotation"},
# MLP参数
    {'params': self.mlp_opacity.parameters(), 'lr': training_args.mlp_opacity_lr_init, "name": "mlp_opacity"},
    {'params': self.mlp_feature_bank.parameters(), 'lr': training_args.mlp_featurebank_lr_init, "name": "mlp_featurebank'
    {'params': self.mlp_cov.parameters(), 'lr': training_args.mlp_cov_lr_init, "name": "mlp_cov"},
    {'params': self.mlp_color.parameters(), 'lr': training_args.mlp_color_lr_init, "name": "mlp_color"},
    {'params': self.embedding_appearance.parameters(), 'lr': training_args.appearance_lr_init, "name": "embedding_appear
]
# 优化器设置
self.optimizer = torch.optim.Adam(l, lr=0.0, eps=1e-15)
**# 后面为每个参数设置了学习率更新方法，和原版一致**
```

2. 每次训练随机选择一个相机进行训练，避免过拟合

```
# Pick a random Camera
if not viewpoint_stack:
    viewpoint_stack = scene.getTrainCameras().copy()
viewpoint_cam = viewpoint_stack.pop(randint(0, len(viewpoint_stack)-1))
```

3. 过滤视椎体之外的anchor，并生成neural-gs

```
# train函数
voxel_visible_mask = prefilter_voxel(viewpoint_cam, gaussians, pipe,background)
# 生成Gaussian，这段代码在render函数里面
if is_training:
    xyz, color, opacity, scaling, rot, neural_opacity, mask = generate_neural_gaussians(viewpoint_camera, pc, visible_ma
else:
    xyz, color, opacity, scaling, rot = generate_neural_gaussians(viewpoint_camera, pc, visible_mask, is_training=is_tra
```

4. 计算anchor到相机中心的距离和方向，对应论文公式5

```
## get view properties for anchor
ob_view = anchor - viewpoint_camera.camera_center
# dist
ob_dist = ob_view.norm(dim=1, keepdim=True)
# view
ob_view = ob_view / ob_dist
```

5. 高斯参数的预测，**MLP的调用**

```
cat_local_view = torch.cat([feat, ob_view, ob_dist], dim=1) # [N, c+3+1]
cat_local_view_wodist = torch.cat([feat, ob_view], dim=1) # [N, c+3]
if pc.appearance_dim > 0:
    camera_indicies = torch.ones_like(cat_local_view[:,0], dtype=torch.long, device=ob_dist.device) * viewpoint_camera.u
    # camera_indicies = torch.ones_like(cat_local_view[:,0], dtype=torch.long, device=ob_dist.device) * 10
    appearance = pc.get_appearance(camera_indicies)

# get offset's opacity
if pc.add_opacity_dist:# 该flag为false
    neural_opacity = pc.get_opacity_mlp(cat_local_view) # [N, k]
else: # 调用此分支，用anchor的feat和方向预测高斯的opacity，输出维度[N, k]
  neural_opacity = pc.get_opacity_mlp(cat_local_view_wodist)

# get offset's color， color的预测
if pc.appearance_dim > 0:
    if pc.add_color_dist:
        color = pc.get_color_mlp(torch.cat([cat_local_view, appearance], dim=1))
    else:
        color = pc.get_color_mlp(torch.cat([cat_local_view_wodist, appearance], dim=1))
else:
    if pc.add_color_dist:
        color = pc.get_color_mlp(cat_local_view)
    else:
        color = pc.get_color_mlp(cat_local_view_wodist)
# 颜色矩阵改为[N*k, 3]，对应到每个三维高斯
color = color.reshape([anchor.shape[0]*pc.n_offsets, 3])# [mask]

# get offset's cov协方差预测
if pc.add_cov_dist:
    scale_rot = pc.get_cov_mlp(cat_local_view)
else: # 使用该分支，输入35维参数，输出为70维
    scale_rot = pc.get_cov_mlp(cat_local_view_wodist)
scale_rot = scale_rot.reshape([anchor.shape[0]*pc.n_offsets, 7]) # [mask]

```

```
# post-process cov，协方差分解为旋转和scaling
scaling = scaling_repeat[:,3:] * torch.sigmoid(scale_rot[:,:3]) # * (1+torch.sigmoid(repeat_dist))
rot = pc.rotation_activation(scale_rot[:,3:7])

# post-process offsets to get centers for gaussians
# 将scaling作用到生成的offset赋给每一个高斯生成xyz
offsets = offsets * scaling_repeat[:,:3]
xyz = repeat_anchor + offsets
```

6. 学习率更新，主要的更新函数为原版代码中的 `get_expon_lr_func` ，**具体公式参考之前的3DGS文章解析；**

anchor的调整

统计模型信息 `training_statis` 函数

- 更新了anchor调整中用到的不透明度累计值 `self.opacity_accum` ，锚点观测数量 `self.anchor_demon` ，offset的梯度累积值 `self.offset_gradie` 观测次数 `self.offset_denom` 四个变量

锚点调整函数 `adjust_anchor()`

1. `self.offset_gradient_accum` 是所有视角下可见高斯的梯度累积，在这里对所有视角下的高斯统一再次进行归一化；根据每个offset数量过滤生成 offsetmask，计数超过40的考虑增加anchor；

```
# adding anchors
grads = self.offset_gradient_accum / self.offset_denom # [N*k, 1]
grads[grads.isnan()] = 0.0
grads_norm = torch.norm(grads, dim=-1)
# threshold = 40
offset_mask = (self.offset_denom > check_interval*success_threshold*0.5).squeeze(dim=1)

self.anchor_growing(grads_norm, grad_threshold, offset_mask)
```

1. **`anchor_growing()`** ，也就是锚点的分割；
    1. 通过阶数$i \in (0, 1, 2)$来控制**梯度阈值（i对应论文公式中的m），random pick的比例，voxel的尺寸**，0阶不进行anchor的增加；（**这里控制 件，可以考虑优化**）
    2. 根据voxel（几倍的voxelsize）确定Gaussian和anchor的grid坐标，并对高斯的grid坐标去重；$(\text{voxel\_size} \cdot (\frac{\text{update\_init\_factor}}{\text{update\_hierachy\_factor}^i}))$
    3. 从高斯的grid坐标中去掉anchor坐标（**代码采用分块并行，否则可能会显存爆炸**），生成新的anchor；
    4. 新anchor的初始化和模型初始化的设置一致，**参考 `create_from_pcd()`**

    ```
    # update threshold,i是当前阶数，update_hierachy_factor=0.0002
    cur_threshold = threshold*((self.update_hierachy_factor//2)**i)
    # mask from grad threshold
    candidate_mask = (grads >= cur_threshold)
    candidate_mask = torch.logical_and(candidate_mask, offset_mask)

    # random pick
    rand_mask = torch.rand_like(candidate_mask.float())>(0.5**(i+1))
    rand_mask = rand_mask.cuda()
    candidate_mask = torch.logical_and(candidate_mask, rand_mask)
    ```

2. 更新新增anchor下面的高斯梯度，全部赋值为0
3. anchor删减：**a)透明度小于阈值；b)anchor个数大于阈值；两者取交集，即为anchor的** `prune_mask`
4. 根据 `prune_mask` 再次更新anchor扩增的offset梯度及个数；
5. 根据 `prune_mask` 重新计算 `opacity_accum` 和 `anchor_demon` ；

参考文献

1. 论文解读：https://zhuanlan.zhihu.com/p/682414775
2. 代码链接：https://github.com/city-super/Scaffold-GS
3. 源码解读：https://blog.csdn.net/qq_41623632/article/details/137602801