DLCV HW4 Report

R11943004 黃子青

Problem: 3D Novel View Synthesis

1. (15%) Please explain:

a. Try to explain 3D Gaussian Splatting in your own words

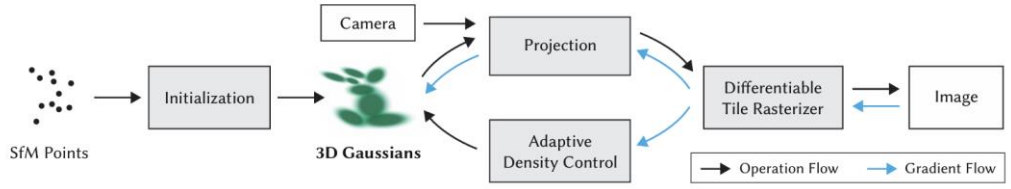
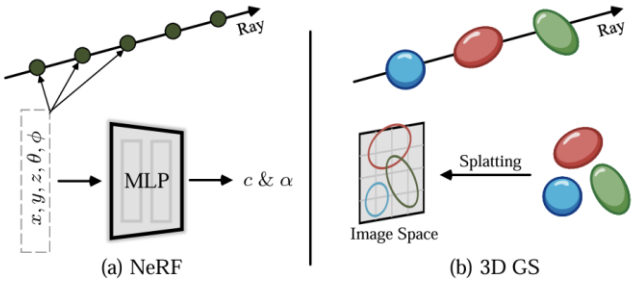


Fig. 2. Optimization starts with the sparse SfM point cloud and creates a set of 3D Gaussians. We then optimize and adaptively control the density of this set of Gaussians. During optimization we use our fast tile-based renderer, allowing competitive training times compared to SOTA fast radiance field methods. Once trained, our renderer allows real-time navigation for a wide variety of scenes.

3D Gaussian Splatting (3D GS), as described in the paper "3D Gaussian Splatting for Real-Time Radiance Field Rendering" (Fig. 2), is a novel approach to synthesizing views of 3D scenes. Starting from a sparse set of 3D points obtained via Structure-from-Motion (SfM), the method initializes a collection of 3D Gaussians to represent the spatial and color properties of the scene. These Gaussians are optimized iteratively by projecting them into the 2D image plane using camera parameters and rendering via a Differentiable Tile Rasterizer.

An important part of the pipeline is Adaptive Density Control, which dynamically adjusts the number and density of Gaussians based on the optimization process to balance computational cost and visual fidelity. Unlike ray-based backward mapping in NeRF, 3D GS performs forward mapping by projecting all Gaussians directly onto the image space, enabling real-time rendering.

b. Compare 3D Gaussian Splatting with NeRF (pros & cons)



(a) NeRF (b) 3D GS

Fig. 3. NeRFs *vs.* 3D GS. (a) NeRF samples along the ray and then queries the MLP to obtain corresponding colors and opacities, which can be seen as a *backward* mapping (ray tracing). (b) In contrast, 3D GS projects all 3D Gaussians into the image space (*i.e.*, splatting) and then performs parallel rendering, which can be viewed as a *forward* mapping (splatting and rasterization). Best viewed in color.

Advantages of 3D Gaussian Splatting (Fig. 2, Fig. 3b):

1.Efficiency:
As shown in "3D Gaussian Splatting for Real-Time Radiance Field Rendering" (Fig. 2), 3D GS avoids NeRF's ray-based sampling and instead projects all Gaussians simultaneously onto the image plane. This direct forward mapping (Fig. 3b) significantly reduces computational complexity.

2.Real-Time Rendering: The combination of Differentiable Tile Rasterization and efficient projection allows 3D GS to support real-time view synthesis, unlike NeRF.

Disadvantages of 3D Gaussian Splatting:

1.Dependence on SfM:
As shown in Fig. 2, 3D GS requires high-quality SfM points as input, which may limit its applicability to scenes where such points are difficult to obtain.

2.Representation Limitations: While the Gaussian splatting approach is efficient, it may not capture fine-grained details or complex lighting interactions as well as NeRF's neural network-based representation.

3.Initialization Sensitivity:
Errors in the initial SfM point cloud can propagate through the optimization process, affecting rendering quality.

Advantages of NeRF (Fig. 3a):

1.Comprehensive Scene Modeling:
NeRF samples points along rays and uses an MLP to predict densities and colors. This process (Fig. 3a) allows it to model intricate scene details and complex light interactions.

2.Input Flexibility:
NeRF can be trained purely from multi-view 2D images, eliminating the need for external 3D initialization.

3.Robust to Initialization:
Unlike 3D GS, NeRF's reliance on optimization makes it less sensitive to external pre-computed input.

Disadvantages of NeRF:

1.Inefficiency:
NeRF's backward mapping (ray sampling and querying an MLP) is computationally expensive, as illustrated in Fig. 3a. Each ray requires multiple point samples and neural network evaluations, leading to long training and rendering times.

2.Lack of Real-Time Support:
NeRF's rendering pipeline is too slow for real-time applications.

3.Complex Optimization:
Training NeRF can be challenging and computationally costly, especially for dynamic or large-scale scenes.

c. Which part of 3D Gaussian Splatting is the most important you think? Why?

I think that the most critical component of 3D Gaussian Splatting is Adaptive Density Control, as highlighted in "3D Gaussian Splatting for Real-Time Radiance Field Rendering" (Fig. 2). This mechanism dynamically refines the number and spatial density of 3D Gaussians during optimization, which is vital for several reasons:

Efficiency Optimization: Adaptive Density Control ensures that computational resources are concentrated on regions of the scene that contribute the most to rendering quality. This avoids excessive Gaussian redundancy, saving both memory and processing power.

Convergence and Quality: By dynamically adjusting Gaussian densities, this component enables faster convergence to high-quality renderings. Without it, areas requiring high detail might be under-represented, or the method could become computationally wasteful.

Real-Time Applications: The ability to adapt the density of Gaussians ensures that the method remains computationally efficient while maintaining visual fidelity, making real-time rendering feasible.

In contrast to NeRF, where all operations are tied to the MLP and ray sampling (Fig. 3a), Adaptive Density Control in 3D GS leverages spatial locality and optimizes the distribution of 3D Gaussians directly in the image space (Fig. 3b). This feature not only sets 3D GS apart but also defines its practicality and scalability.

2. (15%) Describe the implementation details of your 3D Gaussian Spaltting for the given dataset. You need to explain your ideas completely.

I used the GitHub links(https://github.com/graphdeco-inria/gaussian-splatting)provided by TA.
The implementation details of this project involves initializing the point cloud by providing the spatial positions (x, y, z) and colors (r, g, b) of the points. Each point is treated as a three-dimensional Gaussian distribution, and its covariance matrix is constructed using its size and rotation matrix.
These Gaussian points are then projected into the camera's perspective to form 2D representations on the image plane. The process involves multiple steps such as optimization, densification, and pruning to iteratively improve the quality of scene rendering.
During each training iteration:

Randomly select a camera perspective and render the corresponding image.
Compute the loss and update the parameters.
Periodically perform densification to add details and pruning to reduce redundant points.
This iterative process enables the model to progressively learn and enhance the quality of the rendered scene.

After tracing the entire project's code, I found that the key point of training lies in densification. Densification is a crucial step in refining the scene and adding more details.
The training quality is directly related to the densification strategy, and the following parameters need attention:

1. Densification Threshold (percent_dense): Setting this value too high will limit the number of new points added, while setting it too low will introduce excessive redundant points, affecting training efficiency.

```
def densify_and_split(self, grads, grad_threshold, scene_extent, N=2):
    n_init_points = self.get_xyz.shape[0]
    # Extract points that satisfy the gradient condition
    padded_grad = torch.zeros((n_init_points), device="cuda")
    padded_grad[:grads.shape[0]] = grads.squeeze()
    selected_pts_mask = torch.where(padded_grad >= grad_threshold, True, False)
    selected_pts_mask = torch.logical_and(selected_pts_mask,
                                          torch.max(self.get_scaling, dim=1).values > self.percent_dense*scene_extent)
```

(parameter) scene_extent: Any

2. Densification Interval: If the interval is too large, the densification updates will be slow, making it harder to capture details. Conversely, if the interval is too small, it will increase unnecessary computational costs.

```python
if iteration > opt.densify_from_iter and iteration % opt.densification_interval == 0:
    size_threshold = 20 if iteration > opt.opacity_reset_interval else None
    gaussians.densify_and_prune(opt.densify_grad_threshold, 0.005, scene.cameras_extent, size_threshold, radii)
```

Additionally, since the baseline of the assignment requires passing SSIM, I believe the training process should focus on reducing the SSIM loss. This helps the model capture image details and structural features.

```python
loss = (1.0 - opt.lambda_dssim) * Ll1 + opt.lambda_dssim * (1.0 - ssim_value)
```

Therefore, I adjusted parameters like percent_dense and lambda_dssim in the class OptimizationParams(ParamGroup) to improve the training process.

In addition, I adjusted the number of iterations because I noticed that using the default iterations for training caused the process to stop before the results had fully converged.

Here is the final hyperparameter configuration I used:

```python
#best
class OptimizationParams(ParamGroup):
    def __init__(self, parser):
        self.iterations = 30_0000           #
        self.position_lr_init = 0.00016
        self.position_lr_final = 0.00000016 #
        self.position_lr_delay_mult = 0.01
        self.position_lr_max_steps = 30_000
        self.feature_lr = 0.0025
        self.opacity_lr = 0.025
        self.scaling_lr = 0.005
        self.rotation_lr = 0.001
        self.exposure_lr_init = 0.01
        self.exposure_lr_final = 0.001
        self.exposure_lr_delay_steps = 0
        self.exposure_lr_delay_mult = 0.0
        self.percent_dense = 0.1            #
        self.lambda_dssim = 0.3             #
        self.densification_interval = 100
        self.opacity_reset_interval = 3000
        self.densify_from_iter = 300
        self.densify_until_iter = 18_000
        self.densify_grad_threshold = 0.0002
        self.depth_l1_weight_init = 1.0
        self.depth_l1_weight_final = 0.01
        self.random_background = False
        self.optimizer_type = "default"
        super().__init__(parser, "Optimization Parameters")
```

The following image shows the results:

```
number of 3D gaussians: 343387 [26/11 15:15:09]
Testing psnr 38.43681678771973 (avg)
Testing ssim 0.9805382730378348 (avg)
Testing lpips (vgg) 0.07734241843223572 (avg)
Testing lpips (alex) 0.025768080260604 62 (avg)
```

3. (15%) Given novel view camera pose, your 3D gaussians should be able to render novel view images. Please evaluate your generated images and ground truth images with the following three metrics (mentioned in the 3DGS paper). Try to use at least three different hyperparameter settings and discuss/analyze the results.

Setting	PSNR	SSIM	LPIPS (vgg)	Number of 3D gaussians
Setting 1 self.position_lr_final = 0.0000016 self.percent_dense = 0.01 self.lambda_dssim = 0.2	35.622	0.969	0.108	390878
Setting 2 self.position_lr_final = 0.00000016 self.percent_dense = 0.02 self.lambda_dssim = 0.2	36.950	0.976	0.087	363911
Setting 3 self.position_lr_final = 0.00000016 self.percent_dense = 0.1 self.lambda_dssim = 0.3	37.768	0.98	0.08	339649

Explain the meaning of these metrics:

1. PSNR (Peak Signal-to-Noise Ratio)

   Meaning:

   PSNR measures the ratio between the maximum possible value of a signal and the power of the noise affecting its representation. It is calculated in decibels (dB).

   It is commonly used to measure the quality of reconstructed images, where higher PSNR indicates less distortion or noise.

   PSNR is based on the Mean Squared Error (MSE) between the predicted and ground-truth images.

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right)$$

   MAX is the maximum possible pixel value (e.g., 1.0 for normalized images).

   MSE is the mean squared error between the ground-truth and predicted images.

2. SSIM (Structural Similarity Index Measure)
   Meaning:
   SSIM assesses the similarity between two images by comparing luminance, contrast, and structure. It provides a value between 0 (no similarity) and 1 (perfect similarity).
   Unlike PSNR, SSIM takes into account the structural information in an image, making it more perceptually relevant for human vision.

3. LPIPS (Learned Perceptual Image Patch Similarity)
   Meaning:
   LPIPS is a deep-learning-based perceptual similarity metric. It uses pre-trained neural networks (e.g., VGG or AlexNet) to extract feature representations of images and computes the distance between these features.
   It correlates well with human perceptual judgments compared to PSNR and SSIM.The smaller LPIPS value indicate that two images are more similar.

Discuss/analyze the results:

Building on the previous explanation, the main parameter I adjusted was percent_dense. It can be observed that as percent_dense increases, the Number of 3D Gaussians decreases. This is because a point is only considered for densification when its spatial scale exceeds self.percent_dense * scene_extent.

1. Increasing self.percent_dense and its Effect on PSNR、SSIM

Setting 1: With self.percent_dense = 0.01, the model achieves a PSNR of 35.622 and SSIM of 0.969, but has a relatively higher number of 3D Gaussians (390,878).
Setting 2: Doubling self.percent_dense to 0.02 improves both PSNR (36.950) and SSIM (0.976), with a slight reduction in the number of 3D Gaussians (363,911). This indicates that increasing density improves model precision while reducing redundancy in the representation.
Setting 3: A further increase in self.percent_dense to 0.1 yields the best performance in terms of PSNR (37.768) and SSIM (0.98), but the number of Gaussians decreases significantly to 339,649. This indicates that better-distributed Gaussians can enhance representation quality and reduce redundant points.

2. Impact on LPIPS (vgg)

LPIPS measures perceptual similarity; lower values indicate better visual fidelity. Setting 1 achieves an LPIPS of 0.108, which is the worst among the three settings.

As self.percent_dense increases, LPIPS improves significantly, with the best value (0.080) seen in Setting 3. This reinforces that better-distributed and higher-density 3D Gaussians lead to more perceptually accurate results.

3. Balancing Number of 3D Gaussians

The number of 3D Gaussians decreases as self.percent_dense increases. This counterintuitive result arises because the densification step creates more precise Gaussians by pruning redundant or less-influential points while refining high-gradient regions.

Setting 3, with the fewest 3D Gaussians, still achieves the best performance metrics, showing the effectiveness of pruning and densification.

4. Influence of self.lambda_dssim

Setting 3 uses a slightly higher value for self.lambda_dssim (0.3) compared to the others (0.2). This higher weighting for structural similarity ensures the model prioritizes preserving image structure, contributing to the improved SSIM and LPIPS scores.

4. (15%) Instead of initializing from SFM points [dataset/sparse/points3D.ply], please try to train your 3D gaussians with random initializing points.

Describe how you initialize 3D gaussians:

First, I added a print in def fetchPly(path) function to check the values of xyz and rgb. I observed that the rgb values were normalized by dividing by 255,so it should be in range(0~1) ,and the range of xyz appeared to be in the range (-1.3~1.3):

```python
def fetchPly(path):
    # Read the .ply file
    plydata = PlyData.read(path)
    vertices = plydata['vertex']

    # Extract x, y, z positions
    positions = np.vstack([vertices['x'], vertices['y'], vertices['z']]).T

    # Extract r, g, b colors (normalize them to [0, 1])
    colors = np.vstack([vertices['red'], vertices['green'], vertices['blue']]).T / 255.0

    # Extract normals
    normals = np.vstack([vertices['nx'], vertices['ny'], vertices['nz']]).T
# Calculate and print the range for x, y, z
    x_range = (np.min(positions[:, 0]), np.max(positions[:, 0]))
    y_range = (np.min(positions[:, 1]), np.max(positions[:, 1]))
    z_range = (np.min(positions[:, 2]), np.max(positions[:, 2]))

    print("Range of x (min, max):", x_range)
    print("Range of y (min, max):", y_range)
    print("Range of z (min, max):", z_range)

    R_range = (np.min(colors[:, 0]), np.max(colors[:, 0]))
    G_range = (np.min(colors[:, 1]), np.max(colors[:, 1]))
    B_range = (np.min(colors[:, 2]), np.max(colors[:, 2]))

    print("Range of R (min, max):", R_range)
    print("Range of G (min, max):", G_range)
    print("Range of B (min, max):", B_range)
    return BasicPointCloud(points=positions, colors=colors, normals=normals)
```

```
images [29/11 11:03:26]
Reading camera 59/59 [29/11 11:03:26]
Range of x (min, max): (-1.2999785, 1.2999992) [29/11 11:03:26]
Range of y (min, max): (-1.2999902, 1.2999974) [29/11 11:03:26]
Range of z (min, max): (-1.2999982, 1.2999688) [29/11 11:03:26]
Range of R (min, max): (0.4980392156862745, 0.7803921568627451) [29/11 11:03:26]
Range of G (min, max): (0.4980392156862745, 0.7803921568627451) [29/11 11:03:26]
Range of B (min, max): (0.4980392156862745, 0.7803921568627451) [29/11 11:03:26]
```

Based on this, I followed a similar approach for initialization and set num_pts = 300,000. The detailed implementation is shown in the image below.

```python
try:
    num_pts = 300_000
    print(f"Generating random point cloud ({num_pts})...")

    # We create random points inside the bounds of the synthetic Blender scenes
    xyz = np.random.random((num_pts, 3)) * 2.6 - 1.3
    shs = np.random.random((num_pts, 3))
    pcd = BasicPointCloud(points=xyz, colors=SH2RGB(shs), normals=np.zeros((num_pts, 3)))

    storePly(ply_path, xyz, SH2RGB(shs) * 255)
    #pcd = fetchPly(ply_path)
```

The training process (hyperparameter same as best setting) is shown in the image below. It can be observed that the initial results are not very good, and the model struggles to escape the local minima.

```
[ITER 7000] Evaluating train: L1:0.07892398163676262 PSNR:18.119678497314453 SSIM:0.6997470617294312[74/1981]
0:56:59]
Training progress:  10%| | 29990/300000 [22:33<3:49:54, 19.57it/s, Loss=0.0688605, Number of points=:{148366$
[ITER 30000] Evaluating train: L1:0.035255231708288 2 PSNR:23.81504936218262 SSIM:0.8063977360725403 [29/11 0$
:17:25]
Training progress:  20%|| | 59990/300000 [47:40<3:15:15, 20.49it/s, Loss=0.0689386, Number of points=:{148366$
[ITER 60000] Evaluating train: L1:0.03187385499477387 PSNR:24.69858093261719 SSIM:0.8241252183914185 [29/11 $
1:42:32]
Training progress:  30%|| | 89990/300000 [1:11:31<2:41:16, 21.70it/s, Loss=0.0724534, Number of points=:{1483$
[ITER 90000] Evaluating train: L1:0.029398987442255022 PSNR:25.108468246459964 SSIM:0.8336883425712586 [29/1$
 02:06:23]
Training progress:  40%|| | 119990/300000 [1:35:08<2:24:37, 20.74it/s, Loss=0.0644833, Number of points=:{148$
[ITER 120000] Evaluating train: L1:0.028616609796881676 PSNR:25.328335952758792 SSIM:0.8394883394241334 [29/$
1 02:30:00]
Training progress:  50%|| | 149990/300000 [1:58:42<2:02:22, 20.43it/s, Loss=0.0466744, Number of points=:{148$
[ITER 150000] Evaluating train: L1:0.02735128402709961 PSNR:25.57396011352539 SSIM:0.846147620677948 [29/11 $
2:53:34]
Training progress:  60%|| | 179990/300000 [2:22:17<1:35:44, 20.89it/s, Loss=0.0524491, Number of points=:{148$
[ITER 180000] Evaluating train: L1:0.02961445450782776 PSNR:25.074953079223633 SSIM:0.8413417696952821 [29/1$
 03:17:08]
Training progress:  70%|| | 209990/300000 [2:45:52<1:08:47, 21.81it/s, Loss=0.0620995, Number of points=:{148$
[ITER 210000] Evaluating train: L1:0.02733830660581589 PSNR:25.683930587768558 SSIM:0.8510032773017884 [29/1$
 03:40:44]
Training progress:  80%|| | 239990/300000 [3:09:26<48:18, 20.70it/s, Loss=0.0582643, Number of points=:{14836$
[ITER 240000] Evaluating train: L1:0.026667318120598794 PSNR:25.83929023742676 SSIM:0.8515807867050171 [29/1$
 04:04:18]
Training progress:  80%|| | 240000/300000 [3:09:26<50:57, 19.62it/s, Loss=0.0559954, Number of points=:{14836$
```

Compare the performance with that in previous question.

Setting	PSNR	SSIM	LPIPS (vgg)	Number of 3D gaussians
random initializing points	18.639	0.648	0.537	1483668
initializing from SFM points	38.437	0.981	0.077	343387

Initialization Quality Matters:

The random initialization fails to provide meaningful starting points, leading to inefficient optimization and poor scene representation. In contrast, SfM initialization leverages pre-aligned points that capture the scene's geometry, providing a strong foundation for Gaussian refinement.

Efficiency vs. Quality:

Random initialization requires significantly more 3D Gaussians to approximate the scene, yet it still delivers inferior results compared to SfM initialization. This highlights the efficiency of SfM in reducing the computational burden while maintaining or improving quality.

Training Convergence:

Random initialization likely struggles with convergence due to the absence of spatial structure, requiring more iterations to reach an acceptable solution (if at all). SfM initialization accelerates convergence by starting closer to the optimal configuration.

Using SfM initialization drastically outperforms random initialization in all metrics, delivering higher-quality reconstructions (PSNR, SSIM, LPIPS) with far fewer 3D Gaussians. This demonstrates the critical importance of leveraging structured initialization methods for 3D Gaussian Splatting, especially for complex scenes requiring efficient and high-quality rendering.