ELEC 5630 (L1) Assignment 4

Novel View Synthesis with
Neural Radiance Fields & 3D Gaussian Splatting

Professor: TAN, Ping

Due Nov. 23th, 2025

1 Introduction of Neural Radiance Fields(NeRF)

The Neural Radiance Field (NeRF) [1], introduced in the groundbreaking paper by Mildenhall et al. (2020), has revolutionized the fields of computer graphics and computer vision. It facilitates the creation of highly realistic images of 3D scenes from new viewpoints by employing a neural network to model the scene's geometry and appearance as a continuous, differentiable function. NeRF has attracted significant attention, spawning numerous enhancements and adaptations to boost its performance, precision, and scope.

NeRF's framework harnesses deep learning to capture a probabilistic depiction of a scene's density and radiance. By learning from a collection of images captured from diverse angles, the model is capable of predicting the color and density at any spatial location, facilitating the rendering of the scene from novel views. This capability is particularly valuable in domains such as computer graphics, virtual reality, and robotics, where the accurate and efficient portrayal of 3D environments is essential.

However, despite its success, NeRF's implicit volumetric representation also exposes several limitations, such as slow training and rendering, high memory consumption for large scenes, and challenges in directly integrating with traditional graphics pipelines. To address these issues, recent works have explored explicit and factorized scene representations that retain NeRF's photorealistic quality while enabling real-time rendering and scalable reconstruction. Among them, 3D Gaussian Splatting (GS) [2] has emerged as a particularly compelling alternative: it models a scene as a set of anisotropic 3D Gaussians with learnable positions, opacities, and radiance attributes, which can be rasterized efficiently using standard graphics hardware. This shift from an implicit MLP to an explicit Gaussian primitive representation not only accelerates view synthesis, but also opens up new possibilities for editing, compositing, and downstream 3D vision tasks.

In this project, you will develop the computational infrastructure for both Neural Radiance Fields (NeRF) and 3D Gaussian Splatting (GS), and conduct a simplified demonstration of 3D scene reconstruction and rendering. NeRF models 3D scenes as continuous volumetric functions parameterized by neural networks, enabling the synthesis of high-fidelity images from novel viewpoints. In contrast, GS represents a scene explicitly as a set of learnable 3D Gaussian primitives that can be rendered efficiently using rasterization-style pipelines. By implementing and comparing these two paradigms, the project will showcase how modern neural scene representations can reconstruct and render photorealistic scenes from sparse observations, while highlighting the trade-offs between implicit (NeRF) and explicit (GS) formulations in terms of quality and efficiency.

Post questions to Canvas so everybody can share unless the questions are private. Please look at Canvas first if similar questions have been posted.

2 Method

2.1 Objective

In this project, you will learn and implement:

1. The fundamentals of utilizing the PyTorch deep learning library and some necessary knowledge for compiling CUDA programs;
2. Understand and implement the mathematical principles of positional encoding, a core feature of NeRF.
3. Use input images from multiple viewpoints along with corresponding camera parameters to perform NeRF reconstruction, render multi-view images, and evaluate the model's performance;
4. Perform 3D GS reconstrcution for novel view synthesis.

2.2 Implement NeRF and fit on multi-view images(60pts)

In this part, you will construct a Neural Radiance Fields (NeRF) model using a collection of images captured from various viewpoints, which includes both real-world images and synthetic data.

2.2.1 Dataset

Here are some commonly used NeRF datasets for reference and we use **'lego'** in **Blender Dataset** in this assignment:

Blender Dataset: This dataset includes 8 scenes rendered by Blender software, including Chair, Drums, Ficus, Hotdog, Lego, Materials, Mic, and Ship.

Real Forward-Facing: This dataset includes 8 real-world scenes, including Room, Fern, Leaves, Fortress, Orchids, Flower, Trex, and Horns. These images are taken by moving a smartphone up, down, left, and right in front of the scene.

2.2.2 Build NeRF model

NeRF models a continuous scene as a function utilizing a Multi-Layer Perceptron (MLP) network. The network takes a 3D location as input and produces the corresponding RGB color and volume density as output. To complete this part, you need to implement the following key components using provided dataset. We have defined a function for each of the following parts, and you need to implement the corresponding function:

Positional Encoding: For each coordinate $p = (x, y)$ in your 2D space, you'll apply a positional encoding function. This function transforms each coordinate value into a sequence of sinusoids with exponentially increasing frequencies. The encoding can be mathematically represented as a series of sine and cosine functions with different frequencies. The formula for Positional Encoding can be expressed as follows:

$$\gamma(p) = \big(\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)\big) \tag{1}$$

where p represents the input coordinate value, and L is the number of frequency bands used in the encoding. Typically, $L = 10$ is used for spatial coordinates, and $L = 4$ is used for view directions. This encoding method maps each coordinate value into a higher-dimensional space, allowing the model to better capture high-frequency variations in the data, thereby improving the performance of NeRF.

Rendering process: We first need to cast a camera ray from the pixel through the scene. To do this, we need: 1) Extract all pixel coordinates in the image space, formatted as [height, width, 2], where each coordinate (x, y) represents the horizontal and vertical indices, respectively; 2) Transform these image coordinates into camera coordinates using the camera's intrinsic matrix. This involves converting pixel coordinates to homogeneous coordinates and then applying the transformation; 3) Convert the camera coordinates and ray directions to the world coordinate frame using the camera-to-world transformation matrix.

Ray sampling: Along each camera ray, sample points and compute their 3D coordinates. Given the ray's origin and direction, sample 3D points within a range defined by near and

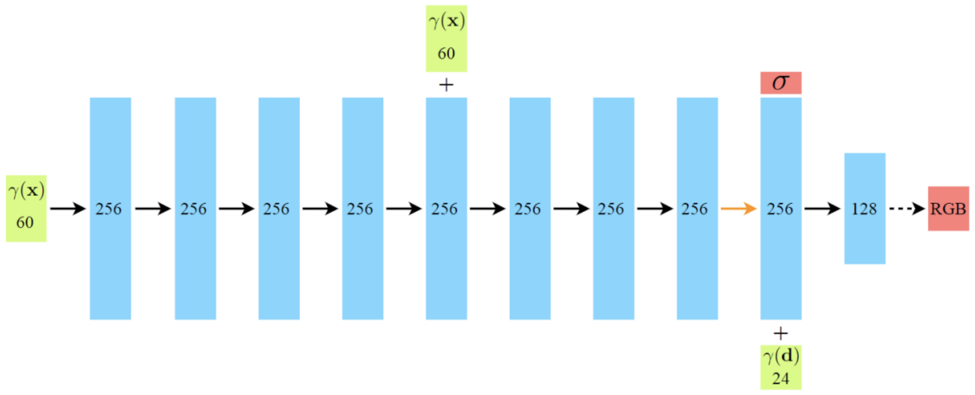

Figure 1: The network structure of NeRF (figure comes from the NeRF paper [1])

far thresholds. Divide the ray into evenly spaced bins and sample one point per bin to ensure thorough coverage during training.

Color and density composition: Combine the colors and densities of the sampled points along a camera ray to determine the final color of the pixel. Calculate the expected of a camera ray using the formula that accounts for the accumulated transmittance along the ray. Approximate this continuous integral using a discrete set of samples with the quadrature rule, considering the distances between adjacent samples.s The color integration formula in NeRF can be expressed as follows:

$$C(r) = \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t), d)\, dt, \tag{2}$$

where $T(t) = \exp\left(-\int_{t_n}^{t} \sigma(r(s))\, ds\right)$. This formula represents that the accumulated color $I(0)$ along the light path from the near plane t_n to the far plane t_f is obtained by integrating the product of the color $c(r(t), d)$ and volume density $\sigma(r(t))$ at each point on the path, multiplied by the transmittance $T(t)$.

Image rendering: Render an image using the trained NeRF model by integrating the colors of all samples along a ray, which is a weighted sum based on the compositing weight. Repeat this for all pixels to generate the final RGB image.

PSNR Metric: After training, you'll report the Peak Signal-to-Noise Ratio (PSNR) metric, which measures the similarity between the original image and the image produced by your network. A higher PSNR value indicates a better match between the two images.

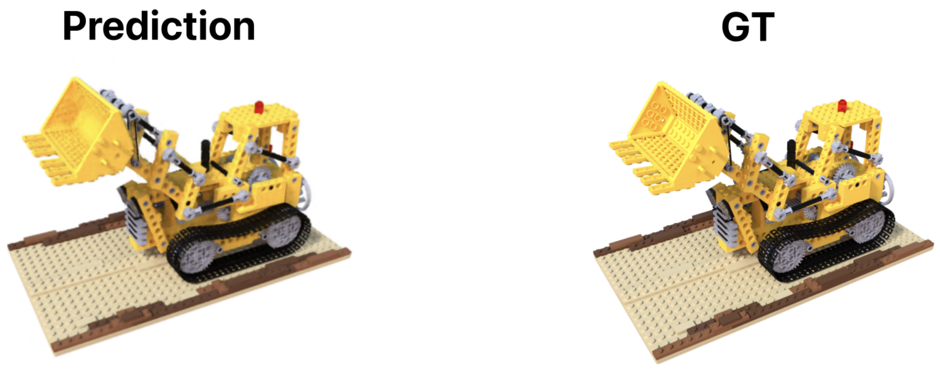

Figure 2: NeRF results on ‘LEGO‘ dataset.

We have provided the skeleton code for the aforementioned functionalities. You need to complete the corresponding parts and train the model to obtain the desired output results. Please note that the final PSNR results will **NOT** be used as a criterion for grading.

2.3 Implement GS and fit on multi-view images(40pts)

3D Gaussian Splatting (GS) represents a continuous scene not as an MLP-based function, but as a set of learnable 3D Gaussian primitives distributed in space. Each Gaussian is parameterized by its 3D position, covariance (shape and orientation), opacity, and color (e.g., via RGB or spherical harmonics), and contributes to the final image through differentiable splatting and alpha compositing in screen space. To complete this part, you need to implement the following key components of the GS pipeline using the provided dataset. We have defined a function for each component, and you are required to implement the corresponding function bodies.

3D Covariance Build: For each Gaussian primitive, construct an anisotropic 3D covariance matrix from its learnable scale and orientation parameters to model its spatial extent. Concretely, the learnable quaternion representing the Gaussian’s orientation are converted into a rotation matrix R and and then, together with the axis-aligned scales S, the covariance of the Gaussian is built as:

$$\Sigma = RSS^T R^T. \tag{3}$$

2D Covariance Build: After obtaining the 3D covariance of each Gaussian in the world coordinate system, we first transform it into camera space. Concretely, Given a viewing transformation W (the transformation that project points from world coordinate into camera coordinate), the covariance matrix in camera coordinates is given as follows:

$$\Sigma' = JW\Sigma W^T J^T. \tag{4}$$

where J is the Jacobian of the affine approximation of the projective transformation:

$$\mathbf{J} = \begin{pmatrix} \frac{f_x}{z} & 0 & -f_x \frac{x}{z^2} \\ 0 & \frac{f_y}{z} & -f_y \frac{y}{z^2} \\ \frac{x}{l'} & \frac{y}{l'} & \frac{z}{l'} \end{pmatrix}, \tag{5}$$

where $\mathbf{t} = [x, y, z]$ is the 3D Gaussian position in camera coodinate system and f_x, f_y are focal length [3]. Noting that the last line of J could be set to 0.

Tile-based rendering: After computing each Gaussian's 2D covariance and screen-space footprint, we assign Gaussians to image tiles and perform per-tile, front-to-back alpha compositing. Concretely, for each pixel, we composite the contributions of all overlapping Gaussians $\mathcal{N}$, sorted from front to back, using standard alpha compositing:

$$C(\mathbf{p}) = \sum_{i \in \mathcal{N}} c_i \, \alpha_i(\mathbf{p}) \prod_{j=1}^{i-1} \big(1 - \alpha_j(\mathbf{p})\big), \tag{6}$$

where c_i is the color of the i-th Gaussian and $\alpha_i(\mathbf{p})$ is its opacity at pixel position $\mathbf{p}$.

We define $\alpha_i(\mathbf{p})$ as a learned per-Gaussian opacity τ_i multiplied by a 2D Gaussian weight:

$$\alpha_i(\mathbf{p}) = \tau_i \cdot \texttt{gauss\_weight}_i(\mathbf{p}), \tag{7}$$

where `gauss_weight` measures how close the pixel $\mathbf{p}$ is to the projected 2D center $\boldsymbol{\mu}_i$ of the Gaussian:

$$\texttt{gauss\_weight}_i(\mathbf{p}) = \exp\left(-\frac{1}{2} (\mathbf{p} - \boldsymbol{\mu}_i)^\top {\Sigma'}_i (\mathbf{p} - \boldsymbol{\mu}_i) \right), \tag{8}$$

with Σ'_i the 2D image-space covariance of the i-th projected Gaussian. Thus, `gauss_weight` is exactly the value of a 2D Gaussian evaluated at the distance between the pixel center and the Gaussian's projected 2D position.

Requirement: All the code you need to fill in is located in *codes\gaussian_splatting\gauss_render.py*. After completing these parts, run *codes\train_gs.py*, which is the training framework we have provided. Note that the number of epochs in this file is set quite high; you should monitor the test-set results to check for overfitting and are allowed to **stop training early**. Please report the PSNR values and the rendered images at **five** different epochs during this process.

Compared with NeRF training, 3D Gaussian Splatting can leverage more prior information, such as point clouds. These point clouds can come from the SfM reconstruction in Project 3. In the previous task, we randomly generated a set of point clouds; here we provide a point cloud corresponding to the Lego dataset. You can uncomment the *fetchPly* around line 50 in *train_gs.py* to load the Lego point cloud and train with it. In terms of convergence speed, what differences can you observe? Again, you may stop training early. Please report the PSNR values and the rendered images at **five** different epochs for this setting as well.

Reference: Most of our implementations are adapted from `https://github.com/graphdeco-inria/gaussian-splatting`. You can find all the corresponding implementations in this Repo. This assignment is implemented in CUDA-based PyTorch and depends on an additional library called *simple-knn*. To build and install this library, navigate into its folder and run *pip install .* from the command line. You may also consult the environment setup instructions in the official repository for further details on configuring your environment.

3 Submission

You only need to upload **ONE** zip file containing the code, the README file, the results, and the screenshot of your program. **Please do not upload the full dataset. You can prepare a empty folder called *"/data"* as a placeholder.** The code should be in Python format. Implementing another language is also allowed in this assignment. You are allowed to use the high-level function provided in Python.

You should write the necessary information in the README file, including the cmd, the output, and maybe some analysis, etc. You should make the code in the submission self-contained. The script output should be matched with the file in the results folder.

Grading will only consider the completeness of the implementation, not the

complexity of the implementation or the performance of the model. Cite the paper, GitHub repo, or code URL if you use or reference code online. Please keep academic integrity; plagiarism is not tolerated in this course. In addition, explicitly state if you used any AI-assisted programming tools—AI may only be used as an aid, and any work suspected of being fully AI-generated will receive a grade of 0.

4 Tips

You can know more about the best practices of Python via **Zen of Python**.

References

[1] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*. 2020.

[2] Bernhard Kerbl et al. "3D Gaussian splatting for real-time radiance field rendering." In: *ACM Trans. Graph.* 42.4 (2023), pp. 139–1.

[3] Matthias Zwicker et al. "EWA splatting". In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (2002), pp. 223–238.