ELEC 6910A Assignment 4

Novel View Synthesis with Neural Radiance Fields

Professor: TAN, Ping

Due Dec. 15th, 2024

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.

A key challenge for NeRF is its demanding computational requirements, with both training and rendering processes being notably time-consuming for practical use. To address this, recent research [2, 3] has introduced innovative solutions, including the use of volume grids or hash-based methods like Instant Neural Graphics Primitives (Instant-NGP) [2]. By leveraging explicit data structures, these methods aim to accelerate the neural network's query process, thereby significantly boosting the efficiency of NeRF.

In this project, you will develop the computational infrastructure for Neural Radiance Fields (NeRF) and conduct a simplified demonstration of 3D scene reconstruction and rendering. NeRF is an advanced machine learning technique that models 3D scenes as continuous volumetric functions, enabling the generation of high-fidelity images from new viewpoints. NeRF's pioneering approach to scene representation using neural networks allows for the generation of photorealistic images from sparse and indirect observations, offering a level of detail and realism that eclipses conventional 3D reconstruction methods.

In this project, we provide a skeleton code for NeRF training and you should complete the rest by yourself. Every script you write in this section should be included in the *python/* directory. The file "run.sh" will be run as the program entry to get the result and you also need to upload the screenshot of the output.

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, and its Python implementation, and attempt to verify its effectiveness in a 2D toy demo;
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. Use Instant-NGP to further accelerate the training process of NeRF.

2.2 Fit a single image with positional encoding (25pts)

Previous solutions for novel view synthesis or scene reconstruction often employed MLPs to represent the entire scene. However, standard coordinate-based MLPs cannot natively represent high-frequency functions on low-dimensional domains. To better preserve high-frequency details in the data, we can encode each of the scalar input coordinates with a sequence of sinusoids with exponentially increasing frequencies before passing them into the network. This allows the network to capture finer details, which is an essential component of NeRF's success.

Now, let's try to overfit a 2D image with a multilayer perceptron (MLP). To do this, we can store a 2D image with a coordinate-based MLP (as shown in the figure 1). The input to this MLP is 2D pixel coordinate as a pair of floating point numbers, and the output is RGB color of the corresponding pixel. This is a simple supervised learning problem, and we can just use simple gradient descent to train the network weights and see what happens. Here's a simplified explanation of how you can proceed:

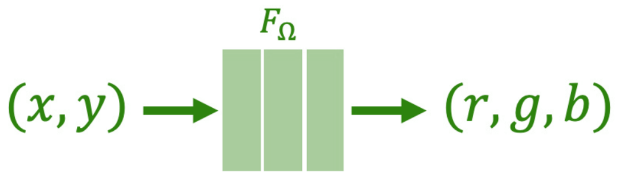


Figure 1: Network structure of fitting a single image

Network Architecture Definition: You'll start by defining the architecture of your MLP for the 2D fitting task. This includes deciding on the number of layers, the size of the hidden dimensions, and whether to include positional embedding in your model.

Naïve Solution: You should experiment with different model structures to determine the most effective configuration by using just a simple MLP.

Positional Encoding: To enhance the performance of your 2D image fitting task, you'll need to incorporate positional encoding techniques into your network architecture. 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:

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

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.

Experimentation: You can try more different settings. This includes testing with and without positional embedding, varying the number of frequency bands, adjusting the number of layers, and changing the size of the hidden dimensions.

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.

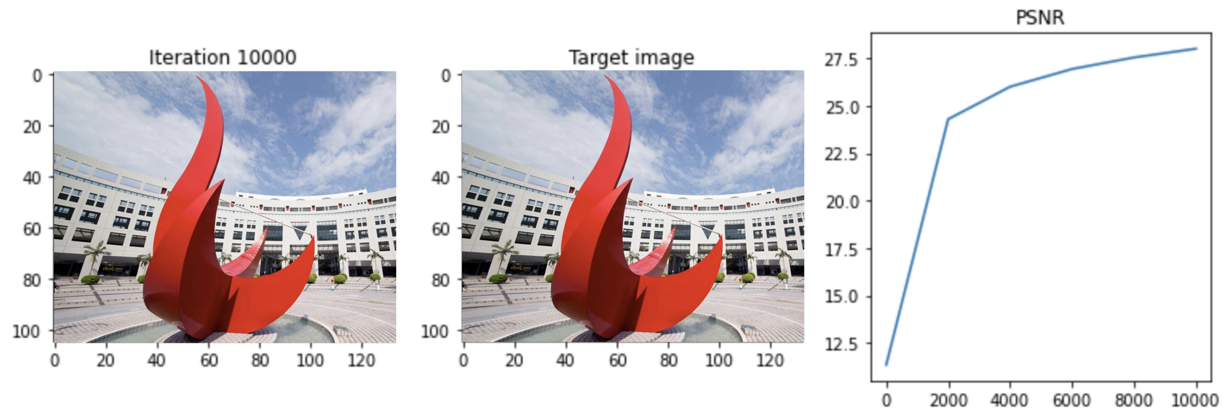
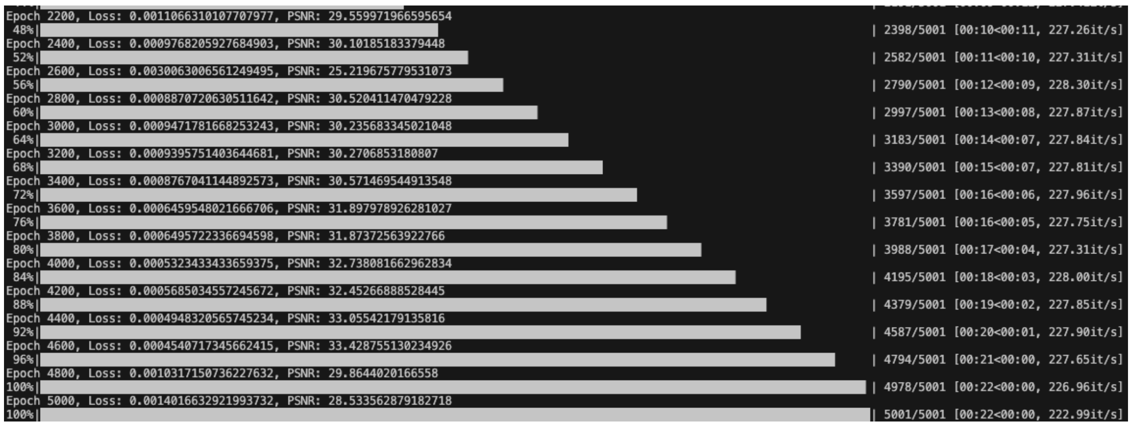


Figure 2: Fit a single image with positional encoding



```
Epoch 2200, Loss: 0.0011066310107707977, PSNR: 29.559971966595654
 48%|████████████████████████                              | 2398/5001 [00:10<00:11, 227.26it/s]
Epoch 2400, Loss: 0.0009768205927684903, PSNR: 30.10185183379448
 52%|██████████████████████████                            | 2582/5001 [00:11<00:10, 227.31it/s]
Epoch 2600, Loss: 0.0030063006561249495, PSNR: 25.219675779531073
 56%|████████████████████████████                          | 2790/5001 [00:12<00:09, 228.30it/s]
Epoch 2800, Loss: 0.0008870720630511642, PSNR: 30.520411470479228
 60%|██████████████████████████████                        | 2997/5001 [00:13<00:08, 227.87it/s]
Epoch 3000, Loss: 0.0009471781668253243, PSNR: 30.235683345021048
 64%|████████████████████████████████                      | 3183/5001 [00:14<00:07, 227.84it/s]
Epoch 3200, Loss: 0.0009395751403644681, PSNR: 30.2706853180807
 68%|██████████████████████████████████                    | 3390/5001 [00:15<00:07, 227.81it/s]
Epoch 3400, Loss: 0.0008767041144892573, PSNR: 30.571469544913548
 72%|████████████████████████████████████                  | 3597/5001 [00:16<00:06, 227.96it/s]
Epoch 3600, Loss: 0.0006459548021666706, PSNR: 31.897978926281027
 76%|██████████████████████████████████████                | 3781/5001 [00:16<00:05, 227.75it/s]
Epoch 3800, Loss: 0.0006495722336694598, PSNR: 31.87372563922766
 80%|████████████████████████████████████████              | 3988/5001 [00:17<00:04, 227.31it/s]
Epoch 4000, Loss: 0.0005323433433659375, PSNR: 32.738081662962834
 84%|██████████████████████████████████████████            | 4195/5001 [00:18<00:03, 228.00it/s]
Epoch 4200, Loss: 0.0005685034557245672, PSNR: 32.45266888528445
 88%|████████████████████████████████████████████          | 4379/5001 [00:19<00:02, 227.85it/s]
Epoch 4400, Loss: 0.0004948320565745234, PSNR: 33.05542179135816
 92%|██████████████████████████████████████████████        | 4587/5001 [00:20<00:01, 227.90it/s]
Epoch 4600, Loss: 0.0004540717345662415, PSNR: 33.428755130234926
 96%|████████████████████████████████████████████████      | 4794/5001 [00:21<00:00, 227.65it/s]
Epoch 4800, Loss: 0.0010317150736227632, PSNR: 29.8644020166558
100%|█████████████████████████████████████████████████████▉| 4978/5001 [00:22<00:00, 226.96it/s]
Epoch 5000, Loss: 0.0014016632921993732, PSNR: 28.533562879182718
100%|██████████████████████████████████████████████████████| 5001/5001 [00:22<00:00, 222.99it/s]
```

Figure 3: It takes about 20 seconds to overfit an image using RTX4080 GPU.

Please note that, the objective here is to construct the model architecture and evaluate the impact of positional encoding on the model's performance. You are **NOT** need to focus on the actual PSNR values for the assessment; instead, the emphasis is on understanding and demonstrating how positional encoding can potentially enhance the model's ability to fit a 2D image. You can also refer to this link for more details about field encoders.

2.3 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

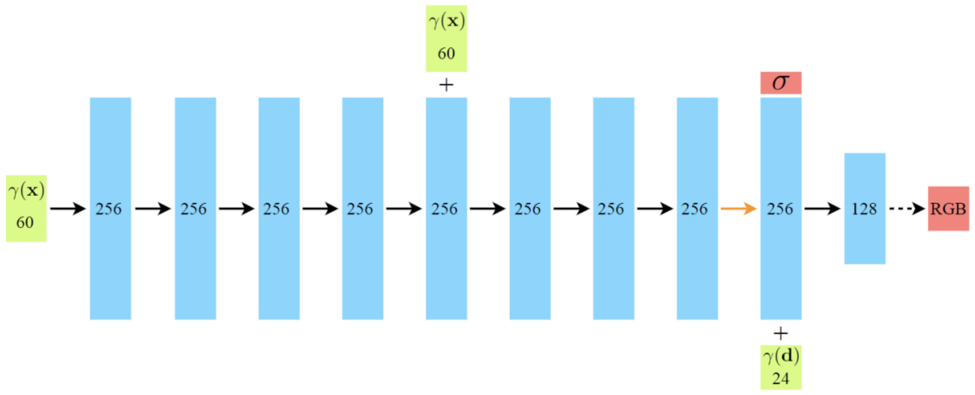


Figure 4: The network structure of NeRF (figure comes from the NeRF paper [1])

synthetic data.

2.3.1 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.

2.3.2 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:

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 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.

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.

2.4 Speedup using Instant Neural Graphics Primitives(15pts)

The heart of Instant-NGP lies in its innovative multiresolution hash encoding method, which efficiently maps 3D spatial coordinates into a high-dimensional feature space. This encoding not only accelerates the training process but also significantly enhances the quality of recon-

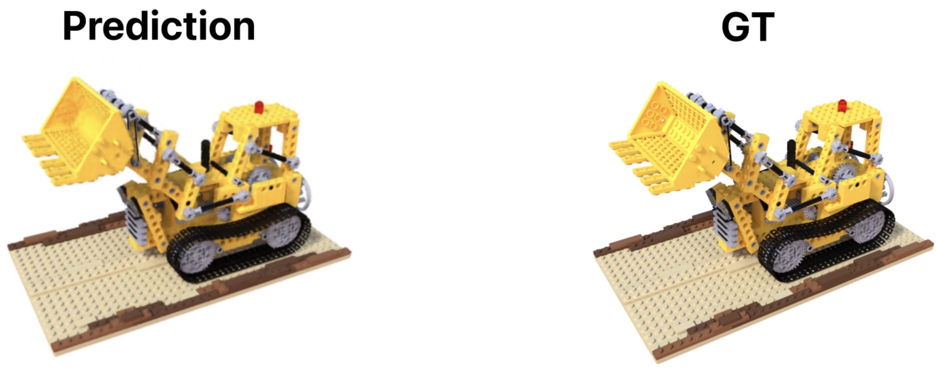


Figure 5: NeRF results on ‘LEGO‘ dataset.

struction, especially in areas with rich details. In this task, you are required to demonstrate the speed and efficiency improvements gained by multiresolution hash encoding compared to the original positional encoding.

You need to transform the original NeRF model expressed with MLP and positional encoding from the previous task, in order to achieve faster training speeds. Please report the training time of different methods.

Reference: In the official implementation of Instant-NGP, the **tiny-cuda-nn library** is a Python library that implements the multiresolution hash encoding method. When working with the tiny-cuda-nn library, you may encounter issues during the installation process, such as CUDA version incompatibility or other errors. It's advisable to check the GitHub issues for tiny-nn-cuda for solutions to common problems.

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

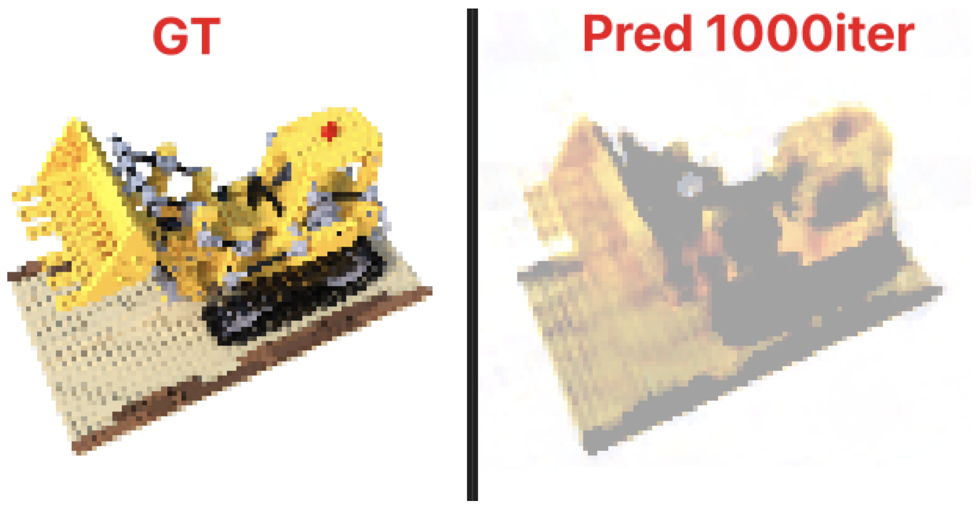


Figure 6: Expect results after 10 mins training on a single 4080

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 the code online. Please keep academic integrity; plagiarism is not tolerated in this course.

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] Thomas Müller et al. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Trans. Graph.* 41.4 (July 2022), 102:1–102:15. DOI: 10.1145/3528223.3530127. URL: https://doi.org/10.1145/3528223.3530127.

[3] Lingjie Liu et al. "Neural Sparse Voxel Fields". In: *NeurIPS* (2020).