浙江大學

本科生课程报告

学年、学期：2021 — 2022 学年 春夏 学期

课程名称：人工智能

任课教师：龚小谨

学生姓名：黄嘉欣、喻治滔、钟函志

目录

Project：Simple Scene Editing Based on NeRF

黄嘉欣、喻治滔、钟函志

1、Introduction to this Project

NeRF, aka neural radiance field, is an emerging method for reconstructing 3D scenes. Different from the explicit methods such as mesh, point clouds and voxel, NeRF uses a neural field to represent scenes implicitly. That is, the information of the scenes will be stored in a neural network until it is queried. Due to the extraordinary performance in reconstruction quality using NeRF, there has been more and more researchers focusing on its further applications and improvements. Take the opportunity of this project, we will attempt to replicate a simplified method proposed by Object-NeRF, which managed to extract objects from scenes and thus realizing a work of scene editing.

2、Methods of NeRF

① Basic Idea

As is shown in Figure 2.1 (a) and (b), NeRF uses a neural field to map a 5D input, that is, the 3D position $\boldsymbol{x}$ of a point in the space and the viewing direction $\boldsymbol{d}$ of the ray, where the point is sampled, to the color $\boldsymbol{c}$ and volume density σ at the spatial location, which consist of a 4-dimentional vector together.

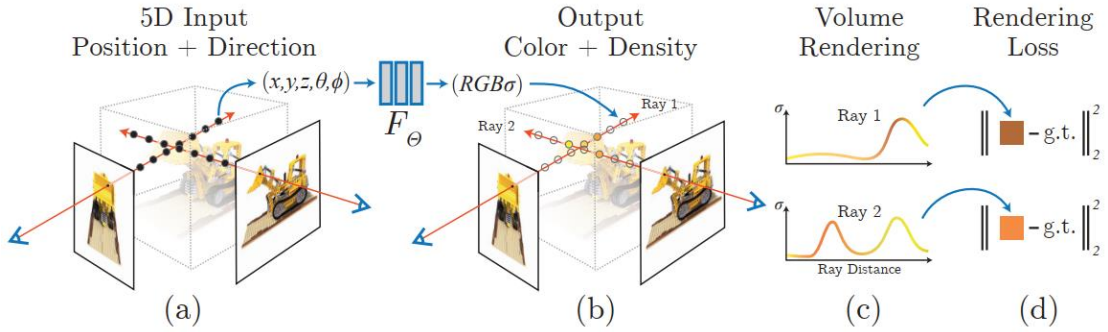

Figure 2.1 basic idea of NeRF[1]

When training on a set of images, NeRF will first use the known poses to compute each image's viewing rays, which correspond to the pixels of images, and then sample on the rays to get discrete points. After querying the deep network, colors and volume

densities of the points will be returned, and a volume rendering technique will be applied to synthesize color of the rays, which can be mathematically represented as follows:

$$\hat{\boldsymbol{C}}(\boldsymbol{r}) = \sum_{i=1}^{N} T_i \left(1 - \exp(-\sigma_i \delta_i)\right)\mathbf{c}_i, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)$$

As we can see, σ_i can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location $\boldsymbol{x}_i$ and T_i is the cumulative transmittance at the ith point of the ray. In this case, the larger σ_i is, the greater ith point's weight of color will be, and the final synthesized color of the pixel will be affected accordingly. Because the process mentioned above is differentiable, it is allowed to use gradient descent to optimize the model by minimizing the mean square error between the synthesized color and the ground truth, which is indicated in Figure 2.1 (c) and (d).

On the other hand, in order to obtain a better performance at representing high-frequency variation in color and geometry, NeRF also adopts an encoding method named positional encoding[2], which is shown as follows:

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

Note that if p is a vector with a dimension of n, then this function will encode every dimension component, resulting in a vector whose dimension is $n \times 2L$. In our project, we will set $L = 10$ for $\gamma(\boldsymbol{x})$ and $L = 4$ for $\gamma(\boldsymbol{d})$, so the dimensions of $\gamma(\boldsymbol{x})$ and $\gamma(\boldsymbol{d})$ are 60 and 24, respectively.

② Network Architecture

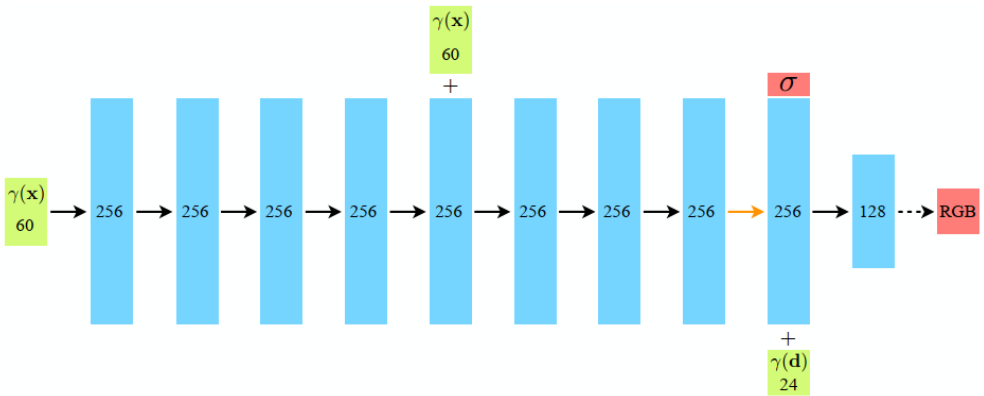

Figure 2.2 network architecture of NeRF

As is shown in Figure 2.2, NeRF utilizes a fully-connected architecture, which is concise and fascinating. In this figure, the input vectors are shown in green, intermediate hidden layers are shown in blue, output vectors are shown in red, and the number inside each block signifies the vector's dimension. All layers are standard fully-connected layers, black arrows indicate layers with ReLU activations, orange arrows indicate layers with no activation, dashed black arrows indicate layers with sigmoid activation, and "+" denotes vector concatenation[1]. Because the volume density σ is a function of only the location $\boldsymbol{x}$ while color $\boldsymbol{c}$ should be related to both location and viewing direction $\boldsymbol{d}$, the input 3D coordinate $\boldsymbol{x}$ will be processed by 8 fully-connected layers first to obtain σ and a 256-dimensional feature vector, which will be concatenated with $\gamma(\boldsymbol{d})$ to further produce the color $\boldsymbol{c}$ of our queried spatial point.

3、Methods of Object-NeRF

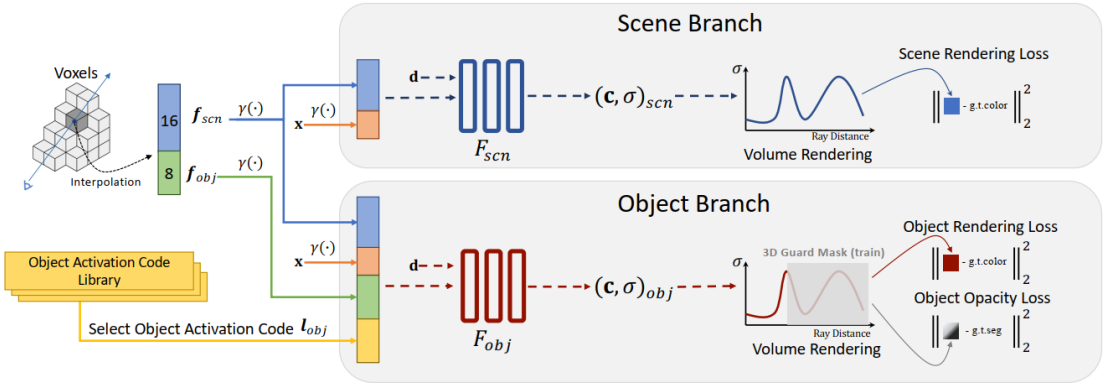

Figure 3.1 basic idea of Object-NeRF[3]

Based on model and network proposed by NeRF, Object-NeRF raises a two-pathways architecture to represent the scenes and objects respectively, which is illustrated in Figure 3.1. For brevity, the scene branch aims to learn the entire scene geometry and appearance, while the object branch only focuses on the single objects in this scene. Regardless of this somehow more complicated framework, scene branch and object branch are both modified on the basis of the original NeRF. To be more specific, an additional voxel encoding[4] is applied as a part of input to the network besides $\gamma(\boldsymbol{x})$ and $\gamma(\boldsymbol{d})$ so as to obtain a better reconstruction quality. Furthermore,

a latent code named Object Activation Code is also used in object branch to specify the object to be rendered.

When it comes to training, the scene branch only takes the ray's synthesized color into consideration, which is the same as NeRF. However, in order to extract objects from the scene, we should ignore the background as possible as we can, that is, an object radiance field should only be opaque at the area occupied by the object and transparent elsewhere. To accomplish this goal, a significant modification has been proposed to the design of object branch's loss function, which is shown as follows:

$$L_{obj} = \sum_{\boldsymbol{r}\in N_r} \sum_{k\in[1,\dots,K]} \lambda_1 M(\boldsymbol{r})^k ||\widehat{\boldsymbol{C}}(\boldsymbol{r})^k_{obj} - \boldsymbol{C}(\boldsymbol{r})||_2^2 + \lambda_2 ||\hat{O}(\boldsymbol{r})^k_{obj} - M(\boldsymbol{r})^k||_2^2,$$

where $O(\boldsymbol{r})$ is the opacity of the ray and can be computed as:

$$\hat{O}(\boldsymbol{r}) = \sum_{i=1}^{N} T_i \left(1 - \exp(-\sigma_i \delta_i)\right)$$

λ_1 and λ_2 are the weight of these two loss components, and $M(\boldsymbol{r})^k$ is constructed by setting 1 or 0 with respect to the instance label at the corresponding pixel belongs to the object k or not. In this case, if a pixel is not part of object k, we will not supervise its color, instead, we need to minimize its opacity to make it transparent. When a pixel belongs to the object, we will try to restore its original color and make it opaque. As is shown in the loss function, object branch is capable of learning K objects simultaneously by computing their own color and opacity loss along each ray and then summing them up. However, for simplicity, we will only consider one object in our implementation, which will be further discussed in the rest of this report.

4、Our Implementation

① Framework Design

As has been proposed by Object-NeRF, we will now design a system to realize the method of simple compositional rendering and scene editing, which can be summarized in the following aspects.

a. Object-NeRF

In this project, we will only extract one object from the scene and then edit its location. Therefore, we are allowed to simplify the Object-NeRF's network and its loss function as follows.

Firstly, when only one object in the scene is considered, there will be no need to specify objects to be rendered, so Object Activation Code could be removed from the inputs of object branch. On the other hand, according to references [4], voxel encoding serves as a method to improve overall quality of large scene, which may not be so necessary in our project. Hence, we will only feed the neural fields $\gamma(\boldsymbol{x})$ and $\gamma(\boldsymbol{d})$, which is the same as the original NeRF.

Secondly, when considering single object extraction, the number of objects will be reduced to 1 naturally, so the loss function introduced in part 3 can be simplified as:

$$L_{obj} = \sum_{\boldsymbol{r}\in N_r} \lambda_1 M(\boldsymbol{r}) ||\widehat{\boldsymbol{C}}(\boldsymbol{r})_{obj} - \boldsymbol{C}(\boldsymbol{r})||_2^2 + \lambda_2 ||\hat{O}(\boldsymbol{r})_{obj} - M(\boldsymbol{r})||_2^2$$

What we have to do is specify an instance in all the images of the training set and apply color loss and opacity loss constraint to each ray according to its affiliation. If the pixel corresponding to a ray belongs to our specified object, we will set $M(\boldsymbol{r}) = 1$ to obtain a realistic color of the ray, otherwise, $M(\boldsymbol{r})$ will equal to 0 and the color in those non-object area will be black, according to the rendering equation.

b. Joint Optimization

Although it works to raise a two-pathways architecture to render the scenes and objects respectively, which will be shown in part 4.2, we find the depth accumulation results fail to meet our expectations. To be brief, the depths of our wanted object are different in scene branch and object branch, that means the scene scale learnt by these two branches are inconsistent. However, if we attempt to realize scene editing using ray bending — which will be further introduced in part c — based on the results proposed by this architecture, we are supposed to maintain the relative positional relationship between foreground and background, which is equivalent to making the depth results of the two branches the same. To accomplish this goal, scene branch and

object branch must be coupled and learn together. Therefore, we modified the feed scheme of the two branches as well as jointly optimizing their losses.

When it comes to the feed scheme, the scene and object branch are both queried by the same points sampled on the same rays, then loss of the branches will be computed. The total loss can be mathematically represented as:

$$L = \omega_1 L_{scn} + \omega_2 L_{obj},$$

where,

$$L_{scn} = \sum_{\boldsymbol{r}\in N_r} ||\hat{\boldsymbol{C}}(\boldsymbol{r})_{scn} - \boldsymbol{C}(\boldsymbol{r})||_2^2$$

$$L_{obj} = \sum_{\boldsymbol{r}\in N_r} \lambda_1 M(\boldsymbol{r})||\hat{\boldsymbol{C}}(\boldsymbol{r})_{obj} - \boldsymbol{C}(\boldsymbol{r})||_2^2 + \lambda_2||\hat{O}(\boldsymbol{r})_{obj} - M(\boldsymbol{r})||_2^2$$

In our implementation, we set $\omega_1 = 0.05,\ \omega_2 = 1,\ \lambda_1 = 1,\ \lambda_2 = 0.1$ to obtain an overall better reconstruction quality.

c. Scene Editing

After joint optimization, the scale of the neural fields will now be more consistent, which make it available to use ray bending to combine the results rendered by scene branch and object branch. The principle of this procedure can be shown as follows:

Figure 4.1 rays bending[5]

Different from letting a ray advance straight, we bend the ray at some point to sample points that originally do not belong to it. In this way, we are able to obtain colors and volume densities at other spatial locations and integrate them together to form a different scene. In our implementation, we are tent to duplicate an object extracted from the scene to an empty area (but not totally blank, there will also be background such as trees and so on), which requires us to bend the rays corresponding

to this area of the images so that they can pass through both the object and the background.

However, because we have maintained an object branch which will only render the object and keep elsewhere being black (at the same time, the depths in this area will be zero as well, which will be shown in part 4.2), the task mentioned above is totally equivalent to only bending the rays for object branch while those for scene branch remain constant. Note that for normal rendering tasks, the two branches share the same ray for each pixel but different sampled points. But for scene editing objective, we will now split the rays fed to the neural fields—although they are still in correspondence to the same pixels — and integrate their returned information correspondingly. By processing like this, the rays which originally correspond to the empty pixels in object branch will now be bended to pass through the object instead of the background, returning points' color and volume density about the stuff. After concatenating with the unbent results returned by scene branch, we are capable of utilizing the rendering equation to synthesis a scene with a newly added object, which can be also defined as object duplication. The results of our experiments will be illustrated later, which meet our expectations.

② Experiments and Results

a. Object-NeRF

We trained and evaluated our implementation on KITTI-360, which is a large-scale dataset containing rich sensory information and full annotations. In order to specify an object, such as a car, we need to use the 2D instance information of the dataset, which saves instance label in single-channel 16-bit PNG format. To accomplish this idea, we first chose some of the rectified images that contain sufficient viewing information of the wanted car as the training set and read their corresponding instance labels. As we have queried, the car's instance id equals to 26323, which means pixels whose id are 26323 all belong to this car. In this case, we are allowed to set each ray's loss for object branch according to whether its corresponding pixel is part of the car or not.

Detailed source code can be found in our-nerf/lib/train/loss.py.

Results of this part are shown as follows:

1) NeRF

Firstly, we replicated NeRF on KITTI-360, which obtained a reconstruction of the scene. Figure 4.2.1 and 4.2.2 have shown the result and corresponding ground truth.

Figure 4.2.1 reconstruction result by NeRF

Figure 4.2.2 ground truth of the scene

As we have computed, the peak Signal-to-Noise Ratio (PSNR) of this model is 17.73, which is slightly low because of the complicated environment. Furthermore, all of the evaluation results of this model can be checked in folder results/object-nerf-split/nerf_psnr17.73.

2) Object-NeRF

On the basis of NeRF's implementation, we modified the data preparation process and framework design as illustrated before to extract a single car from the scene. As we have mentioned, the instance id of our wanted stuff is 26323, which is exactly the white car in Figure 4.2.2.

To better show the role of each part of the loss function in object branch, we carried out an ablation study and the results can be freely check in folder

results/object-nerf-split. Here we will only show some of the synthesized images to make a brief explanation.

Figure 4.2.3 Object-NeRF with no opacity supervision

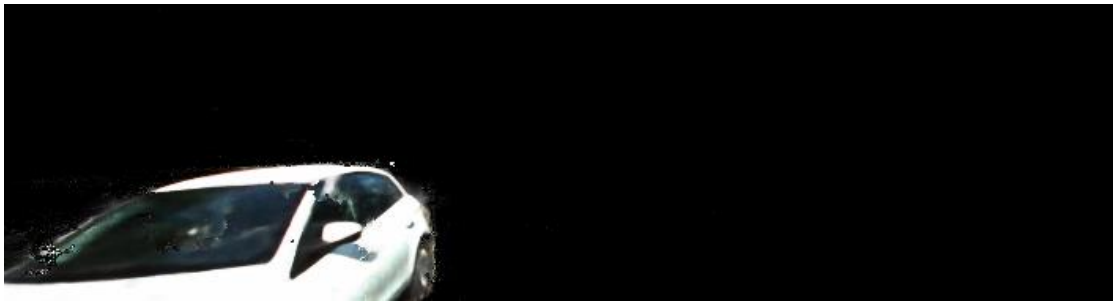

Figure 4.2.4 Object-NeRF with color and opacity supervision

As is illustrated in Figure 4.2.3, we only supervised the color of the pixels belong to the car in the images, while elsewhere are not constrained at all by ignoring the second term in the loss function proposed in part 4.1.a. Therefore, the appearance and geometry of the car are generally reconstructed with a PSNR of 34.06, leaving other areas irregular. When we apply the complete loss function, that is, both color of the specified car and the opacity of the whole scene are supervised, so we can successfully render the single object only and filter out other things with a PSNR of 32.48, which satisfies our requirement.

However, as we have pointed out in part 4.1.b, the scale and depth of the branches are inconsistent, shown in Figure 4.2.5 and Figure 4.2.6

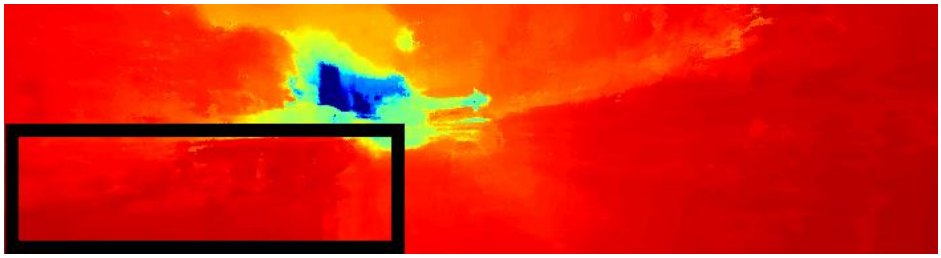

Figure 4.2.5 depth of scene branch

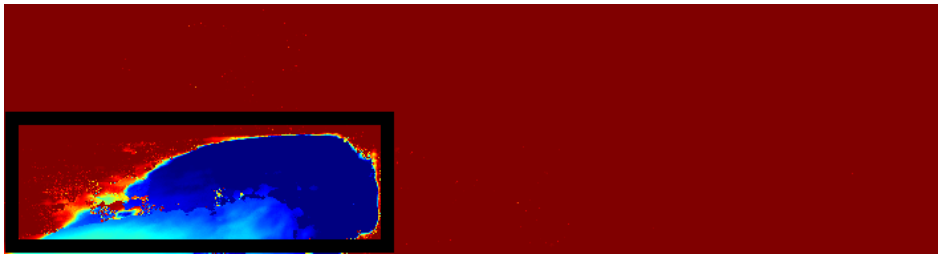

Figure 4.2.6 depth of object branch

Note that the area enclosed by the black rectangle is exactly where our specified car lies. The smaller a pixel's depth is, the redder it will be. In this case, it is obvious that the car obtained by object branch is much further than the scene branch. If we use these models to duplicate the car to the street, using the way we have introduced in part 4.1.c, it is very likely that the street will appear in front of the car, which will fail to accomplish our goal.

b. Joint Optimization

In order to couple the two neural fields, we feed them the same points along the ray and jointly optimized their losses. That is, when an image with known pose is provided, we will compute viewing rays, which have a one-to-one correspondence to the pixels of the image. After that, we will randomly sample some rays and sample on them to get a set of sparse spatial points, which will act as the inputs for both of the branches to produce their corresponding color and volume density, and eventually, the color and opacity of each ray. By jointly minimize the loss, the scale of the neural fields will couple up and a correct foreground and background relationship will be obtained.

The rendered results are shown in Figure 4.2.7 and Figure 4.2.8 and full tested results can be checked in folder results/object-nerf-joint/result.

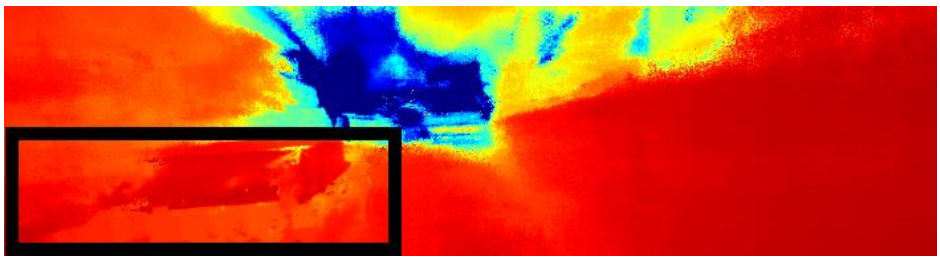

Figure 4.2.7 depth of scene branch after joint optimization

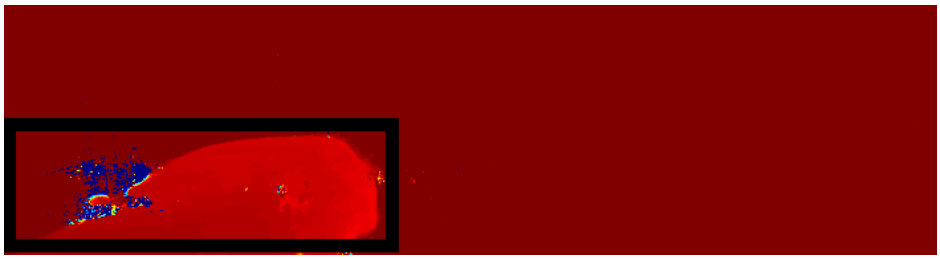

Figure 4.2.8 depth of object branch after joint optimization

It is apparent that results after joint optimization are back to normal, and PSNRs of scene and object branch are 17.64 and 37.95 respectively, which will lay the foundation for our subsequent implementation of scene editing.

c. Scene Editing

As is illustrated in Figure 4.2.9, we used ray bending introduced in part 4.1.c to combine the information obtained in both scene branch and object branch, which duplicate the car to the street successfully (enclosed by red rectangle).

Figure 4.2.9 scene editing

In our implementation, we remain the rays in scene branch unbent while adding an offset of 2.5 meters to the rays in y-axis for object branch to pan the car horizontally. By integrating the points' color and density, a newly added car will be eventually synthesized in the image. Full tested images can be checked freely in folder results/object-nerf-joint/result/edited.

5、Conclusion and Future Works

To sum up, in this project, we replicate a simplified method proposed by Object-NeRF to decompose the scene and extract a single object from it. Through the process of our implementation, we carried out solutions to various problems that arose, and

finally realized the goal of simple scene editing. However, there are still some difficulties that we have not solved yet, which can be addressed in our future work. Firstly, our rendered object may be affected greatly by noises so the reconstruction quality may be not so good. To mitigate this influence, we can sample the batched rays regularly instead of randomly. Because in each frame, our wanted object will occupy a different ratio of area, in this case, we should make sure that sufficient sampled rays should belong to the object so as to obtain more comprehensive information of the stuff, especially when the object is only in one corner of the image. Secondly, we can propose more diverse scene editing method such as rotation and so on, which requires us to learn more knowledge in computer vision.

References

[1] Mildenhall B, Srinivasan P P, Tancik M, et al. Nerf: Representing scenes as neural radiance fields for view synthesis[C]//European conference on computer vision. Springer, Cham, 2020: 405-421.

[2] Rahaman N, Baratin A, Arpit D, et al. On the spectral bias of neural networks[C]//International Conference on Machine Learning. PMLR, 2019: 5301-5310.

[3] Yang B, Zhang Y, Xu Y, et al. Learning object-compositional neural radiance field for editable scene rendering[C]//Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021: 13779-13788.

[4] Liu L, Gu J, Zaw Lin K, et al. Neural sparse voxel fields[J]. Advances in Neural Information Processing Systems, 2020, 33: 15651-15663.

[5] Pumarola A, Corona E, Pons-Moll G, et al. D-nerf: Neural radiance fields for dynamic scenes[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2021: 10318-10327.