Plenoxels and extension to point clouds

Nissim Maruani

March 22, 2022

Figure 1: Our model at epoch 0, 2, 6, 14, 22. Total computation time: 35 minutes on a single GPU.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. TODO [YFKT+21] [MST+20] [KS00] [SSC21] [MESK22]

1 Introduction

The most straightforward way to capture 3D objects from the real world is to use special sensors such as the Kinect or more expensive scanning devices. Another approach relies solely on RGB images: by combining photographs taken at several known viewpoints, one is able to render novel views of a scene. A year ago, a method known as *NeRF* (for Neural Radiance Field [MST+20]) tackled this challenge by using a deep neural network to simultaneously shapes and shadings. More recently, a new approach known as *Plenoxels* [YFKT+21] showed that directly optimizing a voxel grid allowed better results with faster computations. This project is focused on this paper. We re-implemented parts of it and TODO. [1]

2 Our implementation

2.1 Given code

The given code associated to the plenoxels paper requires the compilation of a custom CUDA extension which parallelizes operations. Unfortunately, it requires a CuDNN developer account and quite a bit of C++ experience. In order to avoid losing time on hazardous compilations, we decided to ignore the given code and re-implement everything **from scratch**, except for the utilities functions *get_data* and *get_rays_np* (30 lines of code) from the original plenoxels repository [2] and the Python file *ply.py*, given in the TP of this course.

[1] https://github.com/nissmar/Radiance-Fields
[2] https://github.com/sarafridov/plenoxels

2.2 Our implementation

Since our implementation is "homemade", it doesn't contain every extension described in the paper. It is slower (because it doesn't use a custom CUDA kernel) but still usable in finite time: a simple 128x128x128 grid can be trained in 30 minutes.

The Plenoxel paper introduces remarkable concepts and the given implementation is no less great: the custom CUDA kernel parrelize across rays, which allow a different number of active voxel for each ray and much faster computations. Reproducing the results was a challenge, and improving the given method (in terms of speed or quality of results) was impossible in the scope of this project. TODO dernière approche.

2.3 Datasets

Figure 2: Test images from the *nerf_synthetic* datasets

We used the *nerf_synthetic* dataset, composed of 8 virtual objects.

3 Related works

3.1 Classical methods

Before the rise of neural networks, some methods already existed to render novel views from images: one of them is called *Space Carving* [KS00]. Starting from a dense grid, the voxels from the surface are removed one by one if they aren't consistent with every photograph. This (quite slow) method successfully captures the 3D shape and texture of the photographed scenes, but is unable to capture the transparent/specular properties of the objects. However, we noticed the rigor of the authors who provide theoretical guarantees, absent from nowadays' optimization methods.

3.2 NeRF rendering

The *NeRF* [MST+20] approach consists of training a fully-connected network which takes a 5D input (x, y, z, θ, ϕ) composed of a 3D position and a viewing direction, and outputs a 4D tensor (r, g, b, σ) composed of an RGB color and an opacity σ. For each pixel of the rendered image, N points are sampled along a camera ray and their colors and intensities are integrated to obtain a final RGB value. This method achieved state-of-the-art when it was published, as it allowed to reproduce shading and specular effects efficiently. The training time, however, was quite slow (12 hours for a single scene).

3.3 Concurrent papers and posterior approach

DirectVoxGo [SSC21] is similar to Plenoxels: the main differences are that it uses a neural network (and not a custom CUDA implementation). It produces NeRF-comparable results in less than 15 minutes. More recently, a method using multiresolution Hash encoding [MESK22] was able to reproduce the same results in less than 1 second!

4 The method

In this section, we'll describe the method of the original paper [YFKT+21], and list the different between their implementation (see Figure 3) and ours.

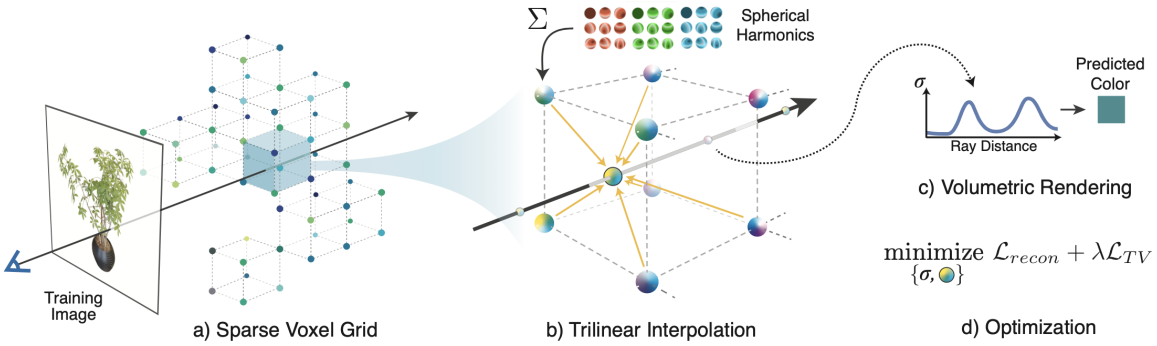

Figure 3: [image source: [YFKT+21]] The plenoxels method is composed of four steps : (a) camera rays are extracted from training images and sampled on a sparse voxel grid ; (b) the spherical harmonics of degree 9 and opacities are computed for each sample through trilinear interpolation ; (c) the resulting colors and opacities are summed to obtain a single pixel value for each ray (d) the mean squared error loss with a total variation regularizer is back-propagated to optimize the grid

4.1 The voxel grid

The clever idea of Plenoxels is to optimize a voxel grid directly. Each voxel stores an opacity $\sigma \in \mathbf{R}^+$ along with spherical harmonics coefficients in $[0, 1]$. These spherical harmonics allow each voxel to have a different color depending on the viewing direction, which renders specular effects and reflections.

The original paper uses a sparse voxel grid optimized with the custom CUDA implementation. Ours is a dense voxel grid which which justifies the lower definition (128x128x128 vs 512x512x512) and the longer computation time. We experimented with spherical harmonics of degree 1 (each voxel has a single color) and 4, but we couldn't get good results with the latter. It may be due to the more complicated convergence.

4.2 Color estimation from camera rays

Once this grid is initialized at random, we optimize its voxels based on images taken from known viewpoints. Each pixel p_x of the training images corresponds to a certain camera ray $r(p_x) \in (\mathbf{R}^3, \mathbf{R}^3)$ composed of an origin and a direction. We sample this ray with $N = 200$ points, compute the points colors and opacities via nearest neighbour interpolation (we implemented trilinear interpolation like in the paper but it was 8 times slower) and sum the sampled points to obtain the estimated RGB color $\hat{c}(r(p_x))$:

$$\hat{c}(r(p_x)) = \sum_{0 \leq i < N} T_i (1 - \exp(-\sigma_i \delta_i)) c_i$$

Where $T_i = \exp(-\sum_{0 \leq j < i} \sigma_j \delta_j)$ represents how much light is transmitted to sample i and δ_i represents the distance between samples.

At first, we were quite puzzled at by this formula, so we considered the following example: an object of constant color c and opacity σ sampled by N points. The distance between points is thus D/N, where D is a constant. The resulting color of a ray through this material is:

$$\sum_{1 \leq i \leq N} \exp(-(i-1)\frac{\sigma D}{N})(1 - \exp(-\frac{\sigma D}{N}))c = c(1 - exp(-\sigma D))$$

In this simple case, the color is indeed independent from the number of samples. Note that it assumes however constant values of color and opacity between samples.

4.3 Optimization

The loss is then computed as $L = \sum_{p_x} \|p_x - \hat{c}(r(p_x))\|^2 + \lambda_{TV} L_{TV}$, and back-propagate through the grid. At each training step, we sample 5000 random rays and compute the total variation over the whole grid. We used an SGD optimizer to do the gradient descent with a learning rate of 1000, and $\lambda_{TV} = 10^{-4}$.

5 Results and experiments

5.1 Visual interpretation

Figure 4: Our models evaluated on novel viewpoints (see our GitHub repository for animated GIFs)

Considering the fact that we couldn't use a custom CUDA kernel, we are quite satisfied with our results. They are visually plausible, enven though at slight blur is sometimes noticeable. As we use spherical harmonics of degree 1, specular effects aren't rendered, and perhaps that is why some surfaces are semi-transparent. Visually, the ship is the least realistic: because of its small details, a grid of size 128x128x128 isn't sufficient to represent it.

5.2 PSNR comparison

The PSNR (for peak signal to noise ratio), defined as $PSNR = 10 \log_{10}(1/MSE)$, is an image metric often used in image restoration. In this context, novel test views are used to compare the rendered model and the real image. In the original paper, a score of 23.74 dB is reached with a 128x128x128 grid and nearest neighbor interpolation. We obtain a score of $TODO$. We are quite satisfied with this result, considering the fact that we don't use spherical harmonics.

Dataset	TV (30 min training)	No TV (30 min training)
Drums	20.12 dB	20.12 dB
Chair	23.69 dB	23.70 dB
Ship	20.65 dB	20.66 dB
Ficus	23.86 dB	23.85 dB
HotDog	23.79 dB	23.75 dB
Materials	20.98 dB	20.99 dB
Lego	21.43 dB	21.42 dB
Mic	22.32 dB	22.29 dB
Mean	22.10 ± 1.4 dB	22.10 ± 1.4 dB

Table 1: PSNR on different training methods.

5.3 Adaptative grid size for faster optimization

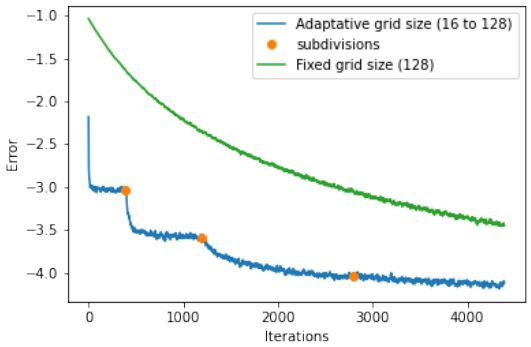

Figure 5: Evolution of the error curve on the microphone model

In the paper, they subdivide the voxel grid via tri-linear interpolation to boost the training phase. We tested and implemented this method which greatly accelerated our optimization (Figures 1 and 5). We use grids of size $16, 32, 64, 128$, on respectively $2, 4, 8, 8$ epochs with a learning rate of $1000, 1000, 500, 500$. We use trilinear interpolation on colors and opacities to subdivide the grid.

6 Using plenoxels to generate point clouds

6.1 CloudCompare pipeline

A point cloud can be seen as a voxel grid with discrete (0 or 1) opacities. We implemented a *.ply* export of our voxel grids, storing the RGB colors and the opacity as a scalar field.

The point cloud is then loaded in CloudCompare. We found that applying the following transformations resulted in a clean, usable point cloud:

- **Edit** > **Scalar fields** > **Filter by value** (Range $2.0; +\infty$) to remove the transparent voxels
- **Tools** > **Clean** > **Noise filter** (Radius 0.01) to clean the point cloud
- **Tools** > **Clean** > **S.O.R. filter** (Radius 0.2) to remove the outliers.

6.2 Discussion

Although these parameters work great, the optimal ones may vary for each model (little less value filter for the lego model, little more noise filter for the hotdog model..). We are pretty satisfied with the results, and a good extension to this project would be to compare the point cloud obtained with this method and a more specialized device (kinect...) on a real scene.

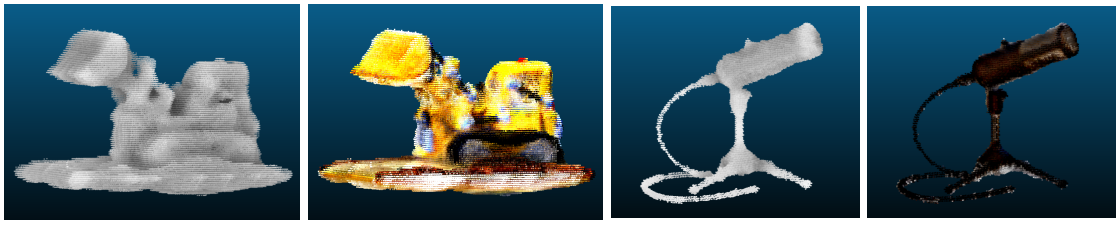

Figure 6: Lego and mic models loaded in cloud compare

7 Our extension: voxel grid optimization on a single pass

We were quite dissatisfied with the obtained point clouds, especially because the opacities aren't binary (fully opaque or fully transparent).

7.1 The idea

Since the *nerf_synthetic* photographs have a transparent background, some of the rays shouldn't hit any voxels. Our first step is removing the voxels on the path of rays corresponding to transparent pixels. Note that this idea was inspired by [KS00]. Once the voxel grid has been carved, we must add the colors. The function to optimize, $L = \sum_{p_x} \|p_x - \hat{c}(r(p_x))\|^2$ is not convex. However we found a way to color each voxel: we average the colors of the corresponding pixels, with weights proportional to the opacity in each visit. The formula is $c_v = \frac{\sum_{r \in R_v} T_{i,r}(1-\exp(-\sigma_i \delta_i))p_r}{\sum_{r \in R_v} T_{i,r}(1-\exp(-\sigma_i \delta_i))}$ where R_v is the set of all rays visiting v.

8 Conclusion

References

[KS00] Kiriakos N Kutulakos and Steven M Seitz. A theory of shape by space carving. *International journal of computer vision*, 38(3):199–218, 2000.

[MESK22] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *arXiv preprint arXiv:2201.05989*, 2022.

[MST+20] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pages 405–421. Springer, 2020.

[SSC21] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *arXiv preprint arXiv:2111.11215*, 2021.

[YFKT+21] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. *arXiv preprint arXiv:2112.05131*, 2021.