Plenoxels and extension to point clouds

Nissim Maruani

March 13, 2022

Abstract

[YFKT+21] [MST+20] [KS00]
[SSC21] [MESK22]

1 Introduction

The most straightforward way to capture 3D objects from the real world requires special sensors such as the Kinect or more expensive scanning devices. Another approach only relies on RGB images: by intersecting the information 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 [YFKT+21] showed that directly optimizing a voxel grid (known as *Plenoxels*) allowed better results with faster computations. This project is focused on this paper. We re-implemented parts of it and showed that its method can be extended to generate point clouds from photographs.

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 across rays, opacities and spherical harmonics coefficients. Unfortunately, it requires a CuDNN developer account and quite a bit of C++ experience (another alternative relies on JAX, but is much slower to train). In order to avoid losing time on hazardous compilations, we decided to code everything **from scratch**, except for the following elements:

- The utilities functions *get_data* and *get_rays_np* (30 lines of code) from the original plenoxels repository [1]
- The Python file *ply.py*, given in the TP of this course.

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 it also much easier to run (a simple TODOx128x128 grid can be trained in TODO time). Our code is organized in the following manner:

- The file *VoxelGrid.py* contains the different voxel grids (RGB colors, spherical harmonics and tri-linear interpolation)
- The files *main.py* and *main_spherical.py* are used to optimize our grids
- The different jupyter notebooks allow for a more interactive training and computation of PSNR
- Some useful functions (including *get_data* and *get_rays_np*) are located inside *utilities.py*.

[1] https://github.com/sxyu/svox2

The Plenoxel paper introduces remarkable concepts and the given implementation is no less great: 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. However, we found a way to extend this approach by integrating a point cloud export to the code, which links this project to the NPM3D course.

2.3 Datasets

TODO

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]. This method successfully captures the 3D shape and texture of the photographed scenes, but is unable to render the shading and capture the specular properties of the objects. However, we noticed the rigor of the authors who provide theoretical guarantees, absent from *NeRF* or *Plenoxels*.

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 and ours (see Figure 1).

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 $[0, 1]$ (in the original paper they use 9, but we used 4 to optimize computations). 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 of our implementation.

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

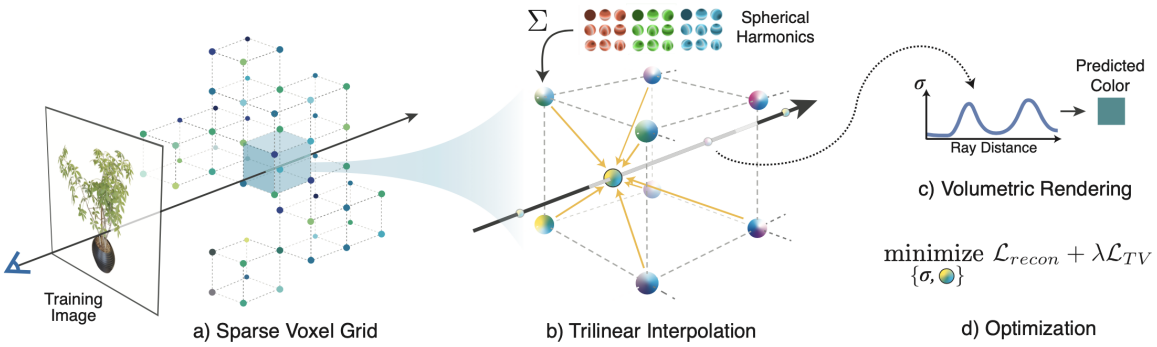


Figure 1: [image source: [YFKT+21]] We implemented a simpler version of the the full plenoxel pipeline. First we extract camera rays from training images and sample them on a dense (not sparse) voxel grid (a). Then we compute the spherical harmonics (or simply the RGB colors if the desired output is a pointcloud) for each sample through nearest neighbor interpolation (and not trilinear interpolation) (b). Finally we sum the resulting colors and opacities to obtain a single pixel value (c), and optimize it with a total variation regularizer (d).

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.

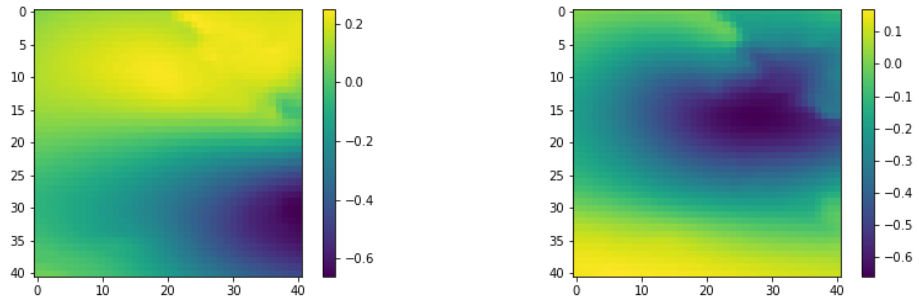
4.3 Optimization

Then it suffices to use back-propagate the mean squared error loss with total variation regularization: $L = \sum_{p_x} \|p_x - \hat{c}(r(p_x))\|^2 + \lambda_{TV} L_{TV}$. 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 Using plenoxels to generate point clouds

5.1 Point cloud generation

5.2 CloudCompare pipeline



(a) Corrélations pour la ville 1

(b) Corrélations pour la ville 2

Figure 2: Corrélations entre la surcote et la pression

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.