Pixel-NeRF Project

Computer Vision Report

Moreno D'Incà
moreno.dinca@studenti.unitn.it

Thomas Reolon
thomas.reolon@studenti.unitn.it

***Abstract* — We present a system capable of learning 3D scenes using sparse views. Specifically we modified the pixel-nerf (which is based on Neural Radiance Fields) open source code to automate the learning of a scene. Given n images corresponding to the views of the scene, the system extracts the object involved in the scene (chair), the resulting images are then used to generate the 3D model of the scene. We test the system on a fixed and controlled environment, the views are 8 taken from 8 equal cameras (same intrinsic parameters), using a chair as target. The results are highly dependent on the estimated poses of the cameras and on the number of epochs the system is trained with. The final 3D learned model shows good results considering the difficulty of the task (a few real grayscale images).**
The source code can be found here: repository.

I. Introduction

Synthesis of novel views is a widely treated argument in the literature, unfortunately these approaches are usually complex, for example Wiles et al. [1] propose to use neural network to extract features from the sparse views, reconstruct a point cloud representation of the scene and then use a rendering method to query novel views (they also use a GAN to improve the realisticity of the output images).

A novel method, called neural radiance fields (NeRF), was proposed in early 2020 with great success. Since then, many variations have been proposed, for example Barron et al. propose to substitute rays with cones for the rendering [2], Srinivasan et al. propose to render views in different lighting conditions by predicting new attributes, eg. the material [3], recently Lin et al. were working on a variation of NeRF (BARF, Bundle-Adjusting Neural Radiance Fields) which aims at creating good models even with incorrectly estimated camera poses [6].

Our work is based on pixelNeRF [5] which aims at embedding some prior knowledge into the scene by using a CNN encoder to extract some features from the view images.

The report proceeds by summarizing differences between NeRF and pixelNeRF, then we explain the structure of the repository and our work. Finally, we show the obtained results.

II. PixelNeRF

We briefly summarize NeRF [4] by saying that it can learn a scene by overfitting a MLP model taking a point in a 3D space and a direction as inputs and outputting a depth and a color. We can obtain the 3D coordinates by sampling some points along a ray; the ray is computed using the intrinsic and extrinsic parameters of the camera that took a specific picture (there is a ray for each pixel of each image and the predicted color is the rgb value of that pixel).

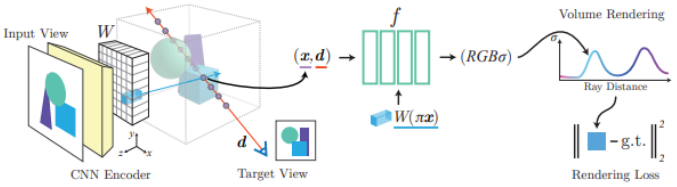

Fig. 1: PixelNeRF in a nutshell (image from the original paper)

PixelNeRF differs from the original NeRF because it takes as input another parameter: the global features of the scene extracted with a CNN (which is resnet34), these features do not change between different views.

An advantage of this model is that we can train a single MLP for different scenes (eg. different types of chairs) which is capable of distinguishing which scene is being asked by looking at the CNN embeddings, moreover this method allows the model to acquire general knowledge of that type of scene (eg. chairs), in this way, when we have only a few views, we can obtain far better results wrt. the original NeRF method.

The original codebase of pixelnerf is mainly thought to reproduce their results, but there is a script called eval_real, which takes as input one image of a car, extracts the features with the CNN encoder, load a pretrained model (trained on the srn_car dataset), feeds the features to the model and queries some new views to produce a video. Even if the results seem decent, we want to exploit the information contained in all our 8 views, moreover, (in that script) they never train the model directly on the views.

For this reason, we made a script (run.py), which improves these aspects and works as follows (when we process the scene of a chair):

- we load a pretrained model on the srn_chair dataset
- we train the model on novel views
- we generate a video in the same way as eval_real

III. Project Structure

We now explain how the project is structured and the role of the most important files. The main folder of the project is src, here the run.py script is the main script of the project. Since the datasets given are videos, we assume the input images to be extracted from the n videos (8 - one for each camera). The videos can be uploaded on a new folder <project>/input/, the correct frames will be automatically extracted by the script. The extracted frames will be processed with detectron2 to remove the background, then, a pretrained model (if the user does not say otherwise) will be fine tuned to the new scene, in the end the 3D video is generated. The folders nerf and scripts contain the main scripts from pixel-nerf code, some of these scripts have been modified to our scope.

We now present how pixelNeRF is used to train the NN for a scene. The process starts by loading the dataset, which contains the poses, the images and some attributes like the focal length and z-far z-near hyperparameters. We then use the poses (which are the camera transforms from the world space to the camera space) to compute the ray origin (4th column) and the ray direction (top-left 3x3 matrix).
Before training the NN we have to extract the features with the CNN, to do so, we pass all the images to the *net.encode* method, which produces the final features by averaging the results of the various views.

Now we can do the effective training as follows: we extract some sample points along the ray, between *z-near* and *z-far* and we encode each 3D point with sin&cos (as in transformers). Then, for each point, we predict volume density (air has volume density of ca. 0) and rgb value using the MLP model (the input is the encoded 3D point plus the direction of viewing). Finally the predicted value for a given ray that is given by the integral below:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right)$$

which is computed as:

sum over every sampled point along the ray of:
*prob_ray_reached_that_point * density_of_point * rgb_of_point*

Once we have our final prediction for the color of a ray, we can compare it with the ground truth (the color of the pixel in the image) and compute the MSE loss of our prediction. Given that this type of rendering is differentiable, we can backpropagate the loss and update the parameters of our neural network. After having trained our model for thousands of epochs, we should end up with a model that, other than remembering the training views, is capable of synthesizing novel views.

IV. poses

One of the main requirements of pixelNeRF is to have the poses for each camera [Fig2]. The poses encode the coordinates from world space to camera space in a 4x4 matrix. The matrix contains the rotation and translation of each camera wrt. the object of the scene. The poses are a fundamental aspect of pixelNeRF, the better the estimate the better the final result. In our case we first estimate the poses taking them from images with similar positions from other datasets. This technique is obviously not the most accurate one, but it gives us the opportunity to try the model in order to understand its capabilities (even with not accurate poses).

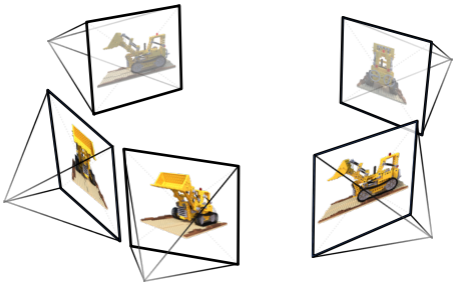

Fig. 2: Visualization of camera poses (image from BARF)

In order to improve the poses we try different settings, we try the imgs2poses script from [7], this script leverages Colmap to extract useful features from the images then it matches the features for each pair of images, finally it computes the poses. This script is obviously highly dependent with the quality of the images, in our case it did not work. We then try manually Colmap with no success. After trying a lot of other scripts from the web without success we decided to code our own pose_estimation script. The script leverages openCv to extract good features from the images, then it computes the matches of the points between each pair of images; at this point the fundamental matrix and essential matrix are computed. Finally we use the essential matrix to extract the estimated rotation matrix (3x3) and the estimated translation vector (1x3). These two are then combined to create the extrinsic matrix (4x4) with the last row [0,0,0,1] to have an affine transformation.

The poses used during training are the one referred to the view of the camera 0 with respect to the other views. After some testing, we understand that even these poses are not good enough (due to the fact that they are relative and not global), so we went back to the first approach.

V. RESULTS

We test the project using the provided data. We have 8 different views of an object (chair). The images are in grayscale capture in a fixed and controlled environment (indoor) [Fig. 3].

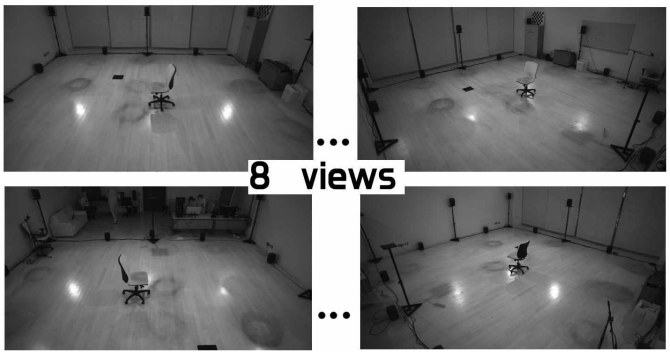

Fig. 3: some of the 8 input views

An important aspect is the number of epochs for the training of the model, in our case we first try the system in a range from 3000 and 5000 epochs. The final model has been trained with 20000 epochs. Since the scope of NeRF is to overfit the scene, the more epochs the better the results. As mentioned above the final result is highly dependent on the quality of the estimated poses, in our case the results can be improved by having the correct poses. Another important factor is the segmentation of the object. We use detectron2 to segment the images, this method is also used from the original pixel-NeRF and it works fine, however we notice some segmentation errors mostly on the chair legs, so we manually clean the images and train the model, noting overall improvements.
The final model trained with 20000 epochs shows good results considering the low number of views at our disposal [Fig. 4].

Fig. 4: synthesized views of the chair from the trained model. gif animation available here

Conclusions: this system is capable of learning 3d scenes from a few sparse views with a possible followup of generating 3d models. An advantage of pixelnerf is that it can embed prior knowledge from other scenes while a drawback is that it is very dependent on the quality of the poses (which seems to be solved in BARF [6]).

REFERENCES

[1] SynSin: End-to-end View Synthesis from a Single Image. SynSin
[2] Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. Mip-NeRF
[3] NeRV: Neural Reflectance and Visibility Fields for Relighting and View Synthesis. NeRV
[4] NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. NeRF
[5] pixelNeRF: Neural Radiance Fields from One or Few Images
[6] BARF : Bundle-Adjusting Neural Radiance Fields. BARF
[7] LLFF: Local Light Field Fusion LLFF