Report for CV Final Project: NeRF

Gaoxiang Ye
Peking University

Xiaole Wang
Peking University

Abstract

Neural rendering, which combines machine learning with geometric reasoning, has emerged as one of the most promising ways to assemble sparse images into new views of a scene. Among these technologies, stands out the Neural radiance fields (NeRF), which trains a deep network to map 5D input coordinates (representing spatial location and viewing direction) into a volume density and view-dependent emitted radiance. Despite applications to static scenes, there also emerges many methods applied to dynamic scenes. In this project, we not only implement NeRF with both static and dynamic scenes, but we also make some improvements and create our own NeRF to render depth map. In general, our model achieved good results, but due to a number of factors, such as equipment and training time, our results were obviously flawed in some aspects. Either way, it was a rewarding experiment.

Our source code (environment file and README contained):

- Task 1:
  https://github.com/TerminaD/CV-NeRF-Task1.git
- Task 2 and 5:
  https://github.com/TerminaD/CV-NeRF-Task2.git
- Task 3 (time):
  https://github.com/TerminaD/CV-NeRF-Task3-SpatialTemporal.git
- Task 3 (deformation):
  https://github.com/TerminaD/CV-NeRF-Task3-Deformation-New.git
- Task 5:
  https://github.com/TerminaD/CV-NeRF-Task4.git

Our video visualizations and checkpoints:
https://drive.google.com/file/d/1oV5ZspyZHKfsPZ8BP-GaSeaGNkUDstKj/view?usp=sharing

1. Introduction

Neural radiance fields (NeRF) is a novel implicit 3D scene representation and a new approach to novel view synthesis, achieving outstanding visual fidelity.

1.1. Problems and overview of our solution

The problems we try to solve and our solution in this work are four-fold:

1. Implement positional encoding and two-dimensional neural rendering: two important building blocks for NeRF.
2. Implement a version of the original NeRF. we use two models to co-work on the basis of original NeRF.
3. Implement a neural radiance field that supports dynamic scenes. We recurred the model of D-NeRF and video-NeRF and made some improvements such as using auxiliary loss function to it.
4. Apply NeRF to a novel use case. We chose to create our own NeRF to render depth map of the scene.

1.2. Importance of NeRF

As the various technologies in NeRF become more mature and more widely applied, the importance of our work is very broad. NERF can learn to reconstruct high-quality 3D scenes from 2D views. Unlike traditional geometric-based approaches, NERF does not need to explicitly represent the geometry of the scene. It learns the depth and color information of the scene directly from the image, thereby avoiding the need for manual modeling or using explicit geometric representations. Our work has helped us understand NeRF more deeply, and we create our own NeRF to render depth map of scenes, which can reduce complex calculations and measurements in actual work. We believe that we can make our NeRF even better with further work!

2. Related work

Neural implicit representation for 3D geometry. The success of deep learning on the 2D domain has spurred a grow-

ing interest in the 3D domain. The NeRF is a good example for this. Published work initially came from NeRF[3]. In the paper, Ben et al. present a method that achieves state-of-the-art results for synthesizing novel views of complex scenes by optimizing an underlying continuous volumetric scene function using a sparse set of input views. When this work was published, it immediately set off a wide range of research on NeRF.

The study of NeRF has also gradually moved from static scenario to dynamic scenario, such as D-NeRF[5] and video-NeRF[6]. In our project, we implement classical NeRF, D-NeRF and video-NeRF. Except to repeat the practices of those who came before, we also did some improvements on it. Such as this, we use two models co-working in the NeRF. On the last task, we design our own NeRF to render depth map of scenes based on existing code. The implementation details will be showed in specific sections.

3. Data

The following is the datasets we've used:

NeRF Dataset (from NeRF[3])

Dynamic NeRF dataset (from D-NeRF[5])

All of the images listed above include camera parameters. For the Dynamic NeRF, we use the form of the original paper to split the video into a frame of pictures. Besides, in order to speed things up, we use super sampling to reduce the size of the images from $(800, 800)$ to $(400, 400)/(200, 200)$.

In addition to these, we also used many random images found on the web to test our model in the first task.

4. Methods

4.1. Fitting a 2D Image

4.1.1 Positional Encoding

Following the formulas provided, we implemented positional encoding as a PyTorch module that does not require gradients.

Note that for positional encoding to take effect, the coordinates need to be non-integer, as encoded integers have only two possible values. And for positional encoding to be at its most effective, the coordinates should be normalized to fall within $(-1, 1)$. We will evaluate the effect positional encoding has on the final rendered image in later sections.

4.1.2 Fitting

The dataloader for this task should, at each iteration, return a pixel's RGB value and its pixel coordinates. This is not available in off-the-shelf dataloader implementations, thus, we created a custom dataloader.

As for the architecture of the neural network, we opted for a multi-layer perceptron, as did numerous NeRF-based works. The network is 256 at its deepest, and gradually narrows down to 3. A sigmoid activation function is placed at the end to ensure that the output value will fall within the $(0, 1)$ range.

At training-time, we sample RGB-coordinate pairs in batches from the dataloader, encode the coordinates after normalizing them to $(0, 1)$, input the encoding into NeRF and calculate the mean squared error between the predicted RGB value and the ground-truth value.

4.2. Implementing NeRF

The general pipeline involved in training NeRF is similar to that of fitting a 2D image. At each training loop iteration, the dataloader provides a batch of rays and their ground-truth colors, dots along the rays are sampled, encoded and fed into the NeRF, returning their RGB and sigma values, with which the final color of the ray is rendered. The MSE loss between the rendered color and ground-truth color is calculated and back-propagated.

For better results, we have adopted a dual-NeRF architecture devised in [3], in which a "coarse" NeRF learns the density distribution of the scene and guides a more targeted sampling, on which samples the "fine" NeRF is evaluated. We will go into greater detail about this sampling process in later sub-sections.

We will examine components involved in this process one by one, placing heavy emphasis on the rendering process.

4.2.1 Dataloader

For this part, we chose to use the NeRF-Synthetic dataset. The dataset directory contains three split directories (train, val and test), each containing a collection of images; and the dataset directory also contains three `json` files, each containing the intrinsic parameters of the camera and the extrinsic parameters for each image.

We ported the component for loading such a dataset from an open source project nerf-pl [1]. It returns the ray and ground-truth color for each training-time iteration, and returns all rays and ground-truth colors for an entire image at validation-time or testing-time. It also caches rays and ground-truth colors from the training split into memory at initialization-time for better performance.

4.2.2 Ray Representation

We chose to represent a ray with 4 components: the location of the camera, or the origin of the ray; the normalized direction vector of the ray; the minimum depth and the maximum depth.

The minimum and maximum depth are provided by the dataloader and are consistent and fixed for all images in a given synthetic dataset scene.

We first calculate the ray directions in the camera coordinate space with the reverse of the camera-to-image matrix. We then apply the world-camera matrix to the direction vectors to get the ray directions in the world coordinate space, and normalize them.

The ray origins, being the world-coordinate-space coordinates of the cameras, are simply the last column in the camera-to-world matrix.

4.2.3 Point Sampling

We, as in [3], adopted a two-stage sampling process to deliver comparable image quality at a smaller computation requirement.

The first stage of sampling is the same as described in the requirements. The range of possible depths are evenly divided into bins and a depth is uniformly randomly sampled from each bin. We then calculate the coordinates of sampled points by, in simple yet imprecise terms, starting from the ray origin and travelling in the direction of the direction vector by depth units.

In the ray rendering process, the probability that light does not hit a particle from one sampled point to the nearer neighboring sampled point is given as $\exp(-\sigma_i\delta_i)$. Thus, the cumulative transmittance, or the probability light emitted from a sampled point travels to the camera without hitting a particle is $T_i = \prod_{i=1}^{N} \exp(-\sigma_i\delta_i)$. For a non-light-emitting particle to appear bright, it need to both reflect light itself and not have the reflected light reflected by nearer particles. Thus the weight, or the combined brightness of a point is $T_i(1 - \exp(-\sigma_i\delta_i)$. This weight serves as a useful approximation for how density is distributed.

In our second phase of sampling, we sample around regions more likely to be non-empty, i.e. those with high weights. We sample around points sampled in the first round of sampling, with probability derived from the weights calculated from the NeRF outputs on the coarsely sampled points, using inverse transform sampling. We then disturb the points to make the sampled locations continuous.

Before sending the coordinates and directions of the sampled points into NeRF, they are normalized to fit within $(-1, 1)$ by being divided by 3. This value is derived empirically, based on the observation that all axes of the coordinates of all sampled points fit cleanly in $(-3, 3)$. The coordinates and directions are then positionally encoded and fed into NeRF.

The parameters of the coarse NeRF and the fine NeRF are optimized jointly.

4.2.4 Ray Rendering and Image Rendering

As illustrated in the previous section, the weights represent the "density" of matter in the scene. Thus, the rendered ray color is a weighted sum of the sampled points' RGB values.

It is also worth mentioning that in the ray rendering process, we made extensive use of PyTorch tensor operations to speed up calculation and enable scaling at larger batch sizes.

Our image rendering function builds upon this batched ray rendering function by dividing all rays in the image to batches and passing them through the batched ray rendered, after which they are concatenated and reshaped to the same shape of the ground-truth image.

4.2.5 Model Architecture

Following the model proposed in [3], we implemented the NeRF neural network as a MLP. At first, the network only takes the encoded positions as input. After a few layers, it outputs the sigmas, and concatenates the encoded direction to the intermediate value, and after another few layers, the model returns the color.

This architecture ensures that sigma is not a function of the viewing direction, leveraging basic physical properties to restrict the degree of freedom for the model.

The neural network also leverages a skip connection for better performance.

4.2.6 Loss

To better serve our dual-NeRF architecture, we implemented a custom loss that combines the MSE loss between the finely rendered image and the ground-truth image and the MSE loss between the coarsely rendered image and the ground-truth image. This allows us to optimize the image rendering quality of the fine model and the density estimation of the coarse model in unison.

4.3. Dynamic NeRF with Time as NeRF Input

In this subsection, we attempt to create a dynamic NeRF by extending the input of vanilla NeRF to include time.

For dynamic NeRFs, we adopted the dataset in the [5]. Its structure is very similar to that of NeRF-Synthetic, with an extra time entry for each image in the `json` files.

We extended the dataloader to read the time for each frame and assign a time to each ray.

In the rendering process, the time for each ray is encoded and fed into the NeRF along with the encoded position and direction.

4.4. Dynamic NeRF with Deformation Field

Another approach, devised in [5], is to introduce another neural network that, given the encoded position and

encoded time, returns the position offset. This is then added to the original position, encoded, concatenated with the encoded directions, and fed into a NeRF.

The deformation field models the movement of particles in the given timeframe.

While the overall implementation follows the paragraph above, there are a few places to pay notice to. The output of the deformation field is passed through a `tanh` activation function and is timed by 2 during the forward process to limit the amount of offset to $(-2, 2)$. The offset position is clamped to $(-1, 1)$. The parameters for the coarse NeRF, the fine NeRF, and the deformation field are optimized jointly.

The neural network approximating the deformation field is a trivial MLP with a depth of 256 and the second-to-last layer narrowing down to 64 elements. This architecture follows that proposed in [5].

4.5. Dynamic NeRF with Deformation Field and Depth Loss

Fitting a dynamic three-dimensional scene is a under-constrained problem, often leading to slow convergence or convergence at local minima. Thus, we propose a method in which the corresponding depth map of an image is used to provide additional regularization.

In the training process, the depth of a ray can be estimated in two ways, either by taking the weighted average of all sampled depths, or taking the depth at which transmittance falls below a certain threshold. Both of these methods are intuitive. We also experimented with using the coarse NeRF or the fine NeRF, and opted to use the weighted average of all sampled depths evaluated on the coarse NeRF. Experimental results leading to this decision are in the experiments section.

The depth map of a image is obtained by running a pre-trained monocular depth estimation (MDE) model. We used Hugging Face's `transformer` package [2], which provides a clean and user-friendly experience. However, most off-the-shelf MDE models are trained on real-life data and does not perform well on synthetic scenes of a singular object on white background. Thus, we used the RGB image to select occupied regions where the alpha value is non-zero, and apply this mask to the estimated depth map, setting the background's depth to the maximum value.

We then calculate the MSE between the MDE-estimated depth map and the NeRF-estimated depth map, and add this loss to the total loss.

4.6. NeRF for Depth

During our work, we discovered that NeRF's ray rendering process makes it easy to derive ray depth from the training process. Additionally, using NeRF to estimate depth eliminates the need to evaluate the last few layers of the neural network and perform ray rendering, lifting the computational workload. Experiments also show that the coarse NeRF alone returns a depth map of comparable quality to that generated by the fine NeRF, eliminating the need to evaluate multiple networks. These combined make estimating depth with NeRF viable and more computationally efficient.

Our implementation does not require re-training a NeRF model. Instead, during test-time, each ray is evaluated and its depth calculated with the method above. This makes our implementation flexible and plug-and-play.

The ray depths are collected and reshaped to form a depth map, which is then resealed to match the value range of the depth maps provided in the testing split.

5. Experiments

The following experiments are conducted on a NVIDIA RTX4090 GPU with 24GB of VRAM.

This section provides a thorough evaluation of our system. In every subsection, we will give a detailed description of our operation for the tasks of project.

5.1. Fitting a 2D Image

In this section, we implement positional encoding and two-dimensional neural rendering: two important building blocks for NeRF. In the flowing paragraph, we will make our description briefly. More details can be found in the Milestone report.

5.1.1 Dissecting the Model

We evaluate the ability of the network by reappearing images of various scene. To test the quality of our results, we calculate the PSNR(Peak Signal-to-Noise Ratio) and keep track of its changes. For the two example images Milestone mentioned, we end up with PSNR of 23.37 and 31.08 in 50 iterations. Compared to the typical example given by project, the results of our 50 iterations have achieved the results of the example thousands of iterations.

5.1.2 Ablation Study

To further optimize the training process of the model, we also make several ablation experiments. In the firstset of experiments, we test the difference between the results with and without positional encoding are apparent. The no-positional-encoding version struggles to capture fine details in the image, only rendering rough geometric shapes. We speculate this is due to the difference in neighboring pixel coordinates sufficiently emphasized in later elements in the positional encoding and the lack thereof in the no-positional-encoding version. Note the lack in representation power manifests as "blocky" artifacts. In addition, we

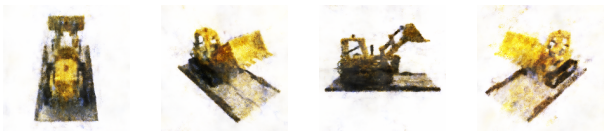

Figure 1. NeRF Rendering Results on Novel Views

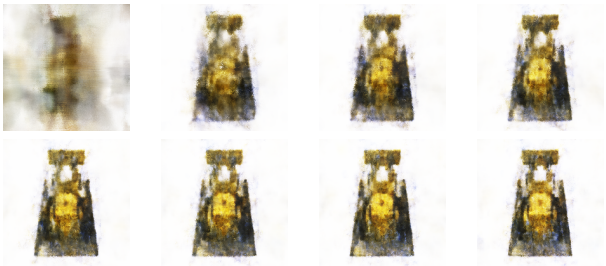

Figure 2. Novel Views Rendered by NeRF after 1, 6, 11, 16, 26, 41, 51, 96 Epochs

test different frequencies in Positional Encoding, different depth and width of net. And we find that the visual quality gets better with increase of frequency, depth and width.

So far we've explored NeRF's design and applications initially, which has laid the foundation for further exploration.

5.2. Implementing NeRF

The following experiment results are obtained with a coarse-fine NeRF architecture, batch size of 4096, position encoding frequency of 10, direction encoding frequency of 4, 64 coarse samples per ray, 128 additional fine samples per ray, a learning rate of 5e-4 that exponentially decays to 5e-5 in the coarse of 100 epochs, after training for 160 epochs. These hyperparameters match closely those of [3].

5.2.1 Visualizations

Here we present visualization results for our implementation of NeRF 1. For video visualizations, see the Google Drive link provided.

We also visualize the convergence process during training. See 2.

5.2.2 Qualitative Results

From some viewing directions, NeRF produces images in which the center object have reasonably well-defined contours and features, and its colors are close to the ground-truth image. However, it's worth pointing out that in our experiments, the quality of reconstructions depend heavily on the viewing direction, as can be seen in 3.

Although extensive testing and debugging have been done, we still cannot determine the cause of this behavior, potential reasons may be that the frequency of direction encoding is not large enough, or that our model has not sufficiently converged.

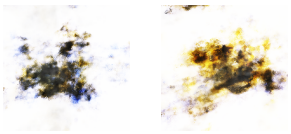

Figure 3. Unsatisfactory NeRF Rendering Results on Novel Views

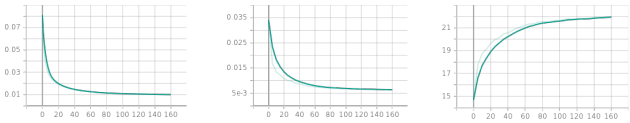

Figure 4. Training NeRF loss, testing MSE loss, and testing PSNR curves during NeRF training

Figure 5. Novel View rendered by NeRF with deformation field (not blank). Other dynamic models exhibit similar results.

5.2.3 Quantitative Results

Over all tested novel views, our NeRF report an average loss of 0.027 and a PSNR of 16.19, with a max per-view PSNR of 22.22. See the loss and PSNR curve during training in 4.

5.3. Dynamic NeRF: Time as NeRF Input, Deformation Field, and Deformation Field with Depth Loss

While we have extensively debugged our implementations for these NeRF variations (abbreviated to TimeNeRF, DeformNeRF and DDNeRF), we did not get satisfactory results from them. The training loss per epoch show minimal convergence (in one case, 0.05417, 0.054195, 0.05418, 0.05418 for the first 4 epochs), and the rendered image in testing is almost completely white, except for slight noise. 5

Quantitatively, TimeNeRF's final test yields an average PSNR of 16.40, DeformNeRF yields 16.19, and DDNeRF yields 17.02. While these values do not seem outlandishly low, it is the result of large areas of white in the ground-truth images.

Potential reasons for this unsatisfactory performance might be the added complexity of the time dimension requires more epochs to train, which is unfeasible due to the prohibitively long training time at 10 minutes per epoch.

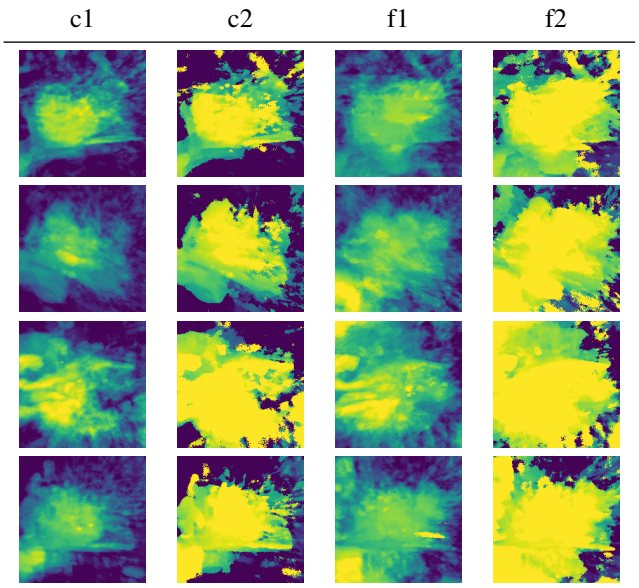

Figure 6. Qualitative comparison between different depth calculation methods

c1	c2	f1	f2
10.44	4.27	6.91	1.95

Figure 7. PSNR comparison between different depth calculation methods

5.4. NeRF for Depth

5.4.1 Comparing Different Methods

As previously stated, we have proposed 4 methods for estimating the depth of a ray (coarse NeRF fine NeRF, weighted depth average transmittance cut-off). We name these 4 candidate methods c1, c2, f1, f2.

We compare the qualitative results of these methods in 6. The transmittance cut-off method, intuitively, results in crisper edges. While this sometimes lead to more well-defined object edges, it also makes floating artifacts have a greater effect on the final depth map. The fine model introduced extra density in empty regions. Overall, we choose c1 as the optimal method.

This is also reflected in quantitative evaluations. In 7, the c1 method achieves the highest PSNR value.

5.4.2 Analysis

Quantitatively, the depth maps do show the contours of the object, but a large amount of artifacts are still present, leading to sub-optimal estimation. Qualitatively, their PSNR with regard to the ground-truth depth maps are not satisfactory either.

This is mostly due to poorly defined edges and floating artifacts. The former can be solved with more training epochs. The latter, a larger problem, calls for changes in the training pipeline. A possible solution is to periodically lower all sigmas in hope that the object will "re-solidify" with later training and the artifacts will fade away. This can be done by feeding the NeRF with images in which the main object appears "transparent".

6. Conclusion

Through this project, we have implemented a system that fits neural networks to a 2D image, a neural radiance field, multiple varients of dynamic radiance field, and one that outputs depth maps. We have investigated the effect of different components on the end result and the training process itself.

We have significantly deepened our understanding of the neural radiance field method: not only the general pipeline, but also the details. Scattered throughout the implementation are numerous details that seem trivial at first sight, but could have detrimental results on the end result.

We have also learned how to effectively manage a sizable project between a team. To keep our codebases in sync between our two computers, a rented remote machine and a Kaggle runtime, we used git and git-lfs to synchronize not only the codebase but also the assets. To keep the codebase organized, we also adopted a hierarchal and compartmentalized project layout.

In future works, if not limited by compute resources, we would investigate the training process over thousands of epochs. We would also explore large-scale modifications to the NeRF system architecture, such as the density-reduction step mentioned above, or more radical variants such as Instant-NGP [4].

References

[1] Q.-A. Chen(kwea123). nerf-pl. https://kwea123.github.io/nerf_pl/, 2020. NeRF (Neural Radiance Fields) and NeRF in the Wild using pytorch-lightning. 2

[2] H. Face. Monocular depth estimation. https://huggingface.co/docs/transformers/v4.28.0/tasks/monocular_depth_estimation. v4.28.0. 4

[3] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *CoRR*, abs/2003.08934, 2020. 2, 3, 5

[4] T. Müller, A. Evans, C. Schied, and A. Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. 6

[5] A. Pumarola, E. Corona, G. Pons-Moll, and F. Moreno-Noguer. D-nerf: Neural radiance fields for dynamic scenes. *CoRR*, abs/2011.13961, 2020. 2, 3, 4

[6] W. Xian, J. Huang, J. Kopf, and C. Kim. Space-time neural irradiance fields for free-viewpoint video. *CoRR*, abs/2011.12950, 2020. 2