# Implementing a Neural Radiance Field from Scratch for Novel View Synthesis

Aston Ki Chan
Zongze Li

# 1. Introduction

Introduced by Mildenhall et. al in 2020, neural radiance fields (NeRFs) represent a major step forward in view synthesis. NeRFs enable the photorealistic rendering of complex three-dimensional scenes from a limited set of two-dimensional images.

A NeRF, specifically, models a continuous five-dimensional function:

$$F_\theta : (\mathbf{x}, \mathbf{d}) \mapsto (\sigma, \mathbf{c}),$$

where $\mathbf{x} \in \mathbb{R}^3$ represents a three-dimensional location, $\mathbf{d} \in \mathbb{S}^2$ represents a viewing direction, $\sigma \in \mathbb{R}$ represents the volume density at $\mathbf{x}$ oriented in the direction of $\mathbf{d}$, and $\mathbf{c} \in \mathbb{R}^3$ represents the RGB color at $\mathbf{x}$ oriented in the direction of $\mathbf{d}$.

By querying the NeRF and performing volume rendering, we can synthesize novel views consistent with our limited set of training images.

In our project, we implement a NeRF model from scratch in PyTorch, following the formulation from the original NeRF paper. We perform training on the official **nerf_synthetic** Lego dataset provided by the authors of the original NeRF paper. Our main objective is to build the full end-to-end NeRF pipeline (ray generation, ray sampling, positional encoding, NeRF, and volume rendering).

# 2. Methodology (Tasks A-G)

To implement the full end-to-end NeRF pipeline from scratch, we performed the following:

## 2.1. Dataset Preparation (Task A)

### 2.1.1. Dataset Description

We employed the **nerf_synthetic** Blender-rendered dataset released by the NeRF authors. Specifically, we employed the Lego dataset. For each scene, the **nerf_synthetic** dataset provides:

- 100 – 200 RGB images rendered from a virtual camera
- A 4 × 4 camera-to-world transformation matrix for each image
- The horizontal field-of-view, **camera_angle_x**.

### 2.1.2. Data Loading Pipeline

We implemented a custom PyTorch Dataset subclass that:

- Loads all images from disk, normalizes pixel values to [0, 1], and removes the alpha channel if present
- Stacks the images into a tensor of shape $(N, H, W, 3)$.
- Stacks the camera-to-world matrices into a tensor of shape $(N, 4, 4)$.
- Stores the horizontal FOV as a scalar shared by all images.
- Returns, for index $i$, a dictionary containing:
    - The image $\mathbf{I}_i$,
    - Image height $H$ and width $W$,
    - Camera-to-world matrix $T_{c\to w}^{(i)}$,
    - Horizontal FOV.

This design ensures that all per-image geometry and appearance information is available in a single data structure for downstream ray generation and training.

### 2.1.3. Image Preprocessing

Each image is loaded using PIL, converted to an RGB NumPy array, and normalized. The result is then converted to a PyTorch Tensor of shape $(H, W, 3)$. If an alpha channel exists, it is dropped. This normalization step is essential for faster model convergence.

## 2.2. Ray Generation (Task B)

Given the camera intrinsics and extrinsics, our goal is to generate, for each pixel, a ray origin and ray direction in world coordinates.

- The ray origin **o** is the camera center in world coordinates, obtained from the translation part of the camera-to-world matrix.
- For each pixel $(u, v)$ in image coordinates, we compute its direction in the camera coordinate system as

$$\mathbf{d}_{cam}(u, v) = \left(\frac{u - W/2}{f}, \; -\frac{v - H/2}{f}, \; -1\right),$$

where *f* is the focal length derived from the horizontal FOV. Our coordinate system follows the OpenGL convention (in alignment with the official NeRF implementation).

We then rotate this direction into world coordinates:

$$\mathbf{d}_{world} = R\mathbf{d}_{cam},$$

Where $R$ is the 3×3 rotation from the camera-to-world matrix. Finally, we normalize $\mathbf{d}_{world}$ to unit length. The resulting outputs are:

- $\mathbf{o}_i(u, v) \in \mathbb{R}^3$: ray origin,
- $\mathbf{d}_i(u, v) \in \mathbb{R}^3$: ray direction.

We vectorize this computation using torch.meshgrid so that a full grid of rays for an image can be produced in a single cell.

## 2.3. Positional Encoding (Task C)

A vanilla MLP operating directly on $(\mathbf{x}, \mathbf{d})$ tends to be biased toward low-frequency functions. To enable the network to represent fine details, NeRF applies a positional encoding $\gamma(\cdot)$ that maps each scalar input to a set of sine and cosine functions at different frequencies.

For a scalar x, the encoding is:

$$\gamma(x) = (sin(2^0\pi x),\ cos(2^0\pi x), sin(2^1\pi x),..., sin(2^{L-1}\pi x), cos(2^{L-1}\pi x)).$$

For a 3D position $\mathbf{x} = (x, y, z)$, we apply this mapping to each component and concatenate the results with the original coordinates:

$$\gamma(\mathbf{x}) = [x, y, z, \gamma(x), \gamma(y), \gamma(z)].$$

In our implementation, we follow the original NeRF hyperparameters and use:

- $L_x = 10$ frequency bands for 3D positions,
- $L_d = 4$ frequency bands for view directions.

These encodings are computed once per batch of samples and then fed into the NeRF MLP described in the next subsection.

## 2.4. NeRF Architecture (Task D)

Our NeRF network follows the architecture of the original paper with minor simplifications:

- An 8-layer MLP with width 256.
- A skip connection at layer 4 that concatenates the original positional-encoded input with intermediate features.
- Two input streams:
    - Position branch:
      $\gamma(x) \rightarrow$ several fully-connected layers $\rightarrow$ density σ and a feature vector **h**.
    - View-dependent color branch:
      $[\mathbf{h}, \gamma(d)] \rightarrow$ additional layers $\rightarrow$ RGB color **c**.

The network outputs $(\sigma, \mathbf{c})$ for each queried 3D location and viewing direction. We train the model using an MSE loss between the rendered color $\hat{\mathbf{C}}$ along each ray and the ground-truth pixel color **C**:

$$L = ||\hat{\mathbf{C}} - \mathbf{C}||_2^2.$$

## 2.5. Sampling Along Rays (Task E)

For each camera ray $r(t) = \mathbf{o} + t\mathbf{d}$, we sample a fixed number of points between a near and far bound:

- Near bound: $t_{near} = 0$,
- Far bound: $t_{far} = 1$,
- Number of samples per ray: $N = 64$.

We use stratified sampling, which divides the interval $[t_{near}, t_{far}]$ into N bins and samples one depth from each bin:

$$t_i = t_{near} + \frac{i}{N}(t_{far} - t_{near}) + \epsilon_i,$$

where $\epsilon_i$ is a small random offset within the bin.

The sampled 3D points are then:

$$\mathbf{x}_i = \mathbf{o} + t_i\mathbf{d}, \qquad \text{where } i = 1,\ldots,N.$$

These locations (together with the ray direction) are fed into the NeRF MLP to obtain densities $\sigma_i$ and colors $\mathbf{c}_i$. In this project we implement only the coarse stratified sampling stage and do not include the hierarchical importance sampling used in the original paper.

## 2.6. Volume Rendering (Task F)

We approximate classical volume rendering by discretizing the integral along each ray. Given per-sample densities $\sigma_i$, colors $\mathbf{c}_i$, and depth intervals $\delta_i = t_{i+1} - t_i$, we compute:

- The probability of stopping at sample $i$:
  $\alpha_i = 1 - exp(-\sigma_i\delta_i)$,
- The accumulated transmittance up to, but not including, sample $i$:
  $T_i = exp(-\sum_{j=1}^{i-1} \sigma_j\delta_j)$.

The final color along the ray is:

$$\hat{C}(r) = \sum_{i=1}^{N} T_i\alpha_i\mathbf{c}_i.$$

This formulation is fully differentiable; gradients flow back through both densities and colors, enabling end-to-end training of the NeRF MLP.

## 2.7. Training Loop (Task G)

We train NeRF using the following settings:

| Hyperparameter | Value |
|---|---|
| Resolution | 400 × 400 |
| Learning rate | 5e-4 |
| Optimizer | Adam |
| Batch size | 4096 rays |
| Epochs | 20k steps |

# 3. Discussion

## 3.1. Implementation Lessons

Implementing NeRF from scratch forced us to carefully understand each stage of the pipeline and how different components interact:

- Ray generation and camera conventions.
  Small mistakes in the camera coordinate system (e.g., using the wrong sign convention or mixing up row-major vs column-major transforms) can lead to completely black or distorted renderings. Debugging this required checking simple test scenes and visualizing ray directions.

- Sampling and near/far bounds.
  Choosing appropriate near and far clipping planes is crucial. If the bounds are too tight, parts of the object are truncated, slowing down training and reducing gradient signal.

- Training stability.
  NeRF is computationally expensive, and naïvely rendering full-resolution images at every iteration is infeasible. We found that randomly sampling rays and rendering low-resolution validation views made it possible to iterate on the implementation within limited compute.

- Positional encoding.
  Although we did not perform an explicit ablation, our experiments confirm that including multi-frequency positional encoding is necessary to

obtain sharp reconstructions. Removing it degrades quality significantly in informal tests.

## 3.2. Limitations

Our implementation omits several components of the original NeRF that are known to improve quality and efficiency:

- We do not implement hierarchical importance sampling, so we sample all points uniformly along the ray.
- We train on a single scene and do not explore generalization across multiple scenes.
- We do not attempt to match the exact training schedule or hyperparameters (number of samples, network width, learning rate schedule) used by the original authors. As a result, our model converges more slowly compared to state-of-the-art NeRF variants.

**Challenges Encountered**

- Training without positional encoding failed to reconstruct meaningful geometry.
  The MLP performed poorly without high-frequency features, producing blurry results, confirming that positional encoding is essential.

- Training was computationally expensive.
  A full-resolution NeRF is slow to train on a single GPU. Rendering validation views also required substantial time, limiting iteration speed.

- Ray generation bugs were frequent early in development.
  Errors such as incorrect coordinate frames, transposed extrinsics, or incorrect near/far spanish initially led to black or diverging renders.

- Full-resolution rendering during training was too slow.
  We adopted low-resolution previews for faster debugging and only produced full-resolution images intermittently.

- Lack of hierarchical sampling limited fine-detail reconstruction.
  Our model produced smooth results but occasionally struggled with sharp edges and specular surfaces.

# 4. Conclusion

In this project, we implemented a Neural Radiance Field (NeRF) model from scratch and trained it on the synthetic Lego scene. Our implementation includes all core components of NeRF: ray generation from camera parameters, sinusoidal positional encoding, a multi-layer perceptron that maps positions and view directions to color and density, and differentiable volume rendering.

Our results show that a carefully implemented NeRF can reconstruct a 3D scene from 2D images and synthesize novel views with realistic geometry and appearance. Although we did not perform extensive ablation studies, the project gave us hands-on experience with coordinate-based neural representations and revealed practical challenges in training such models, from camera calibration issues to memory and compute constraints.

In future work, we would like to extend our implementation with hierarchical sampling, experiment with real captured scenes, and explore newer encoding schemes and acceleration techniques inspired by more recent NeRF variants.

# Works Cited


1. Bao, C., et al.: Sine: semantic-driven image-based nerf editing with prior-guided editing field. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 20919–20929 (2023)

2. Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Mip-nerf 360: unbounded anti-aliased neural radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 5470–5479 (2022)

3. Fridovich-Keil, S., Meanti, G., Warburg, F.R., Recht, B., Kanazawa, A.: K-planes: explicit radiance fields in space, time, and appearance. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12479–12488 (2023)

4. https://arxiv.org/abs/2003.08934