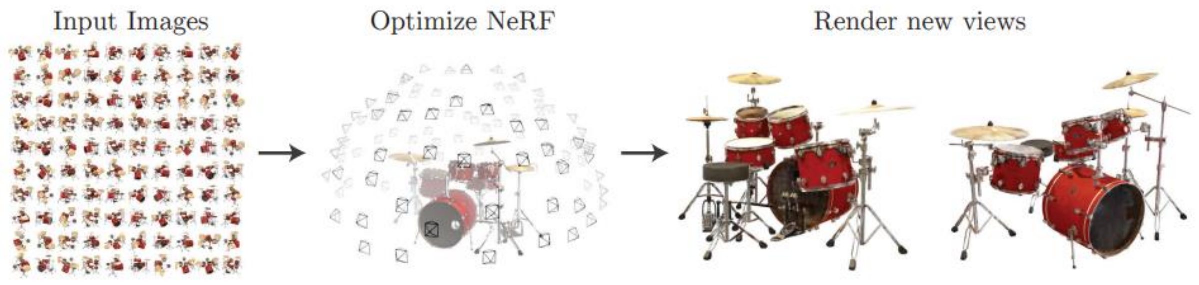
Neural Radiance Field (NeRF)

A neural radiance field (Nerf) is a fully-connected neural network that can generate novel views of complex 3D scenes, based on a partial set of collection of 2D images and accurate poses (e.g. position and rotation). It is trained to use a rendering loss to reproduce input views of a scene. It works by taking input images representing a scene and interpolating between them to render one complete scene. Nerf is a highly effective way to generate images for synthetic data. Also, it explicitly defines the 3D shape and appearance of the scene as a continuous function. Nerf although it learns directly from image data, they use neither convolutional nor transformer layers. A benefit of Nerf is compression the weights of a NeRF model may be smaller than the collection of images used to train them.

A NeRF network is trained to map directly from viewing direction and spatial location (5D input) to opacity and color (4D output), using volume rendering to render new views. NeRF is a computationally-intensive algorithm, and processing of complex scenes can take hours or days. However, new algorithms (such as FastNerf) are available that dramatically improve performance.

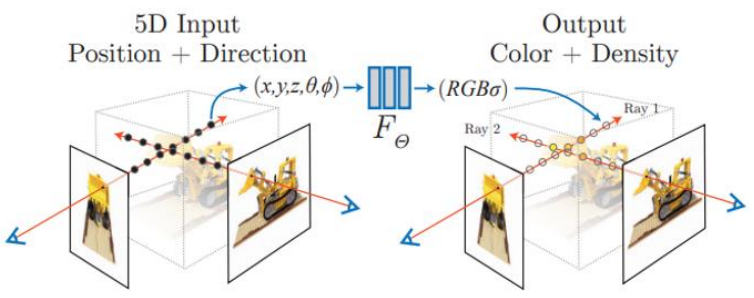


In the above figure we visualize the set of 100 input views of the synthetic Drums scene randomly captured on a surrounding hemisphere, and we show two novel views rendered from our optimized NeRF representation.

Nerf Scene Representation

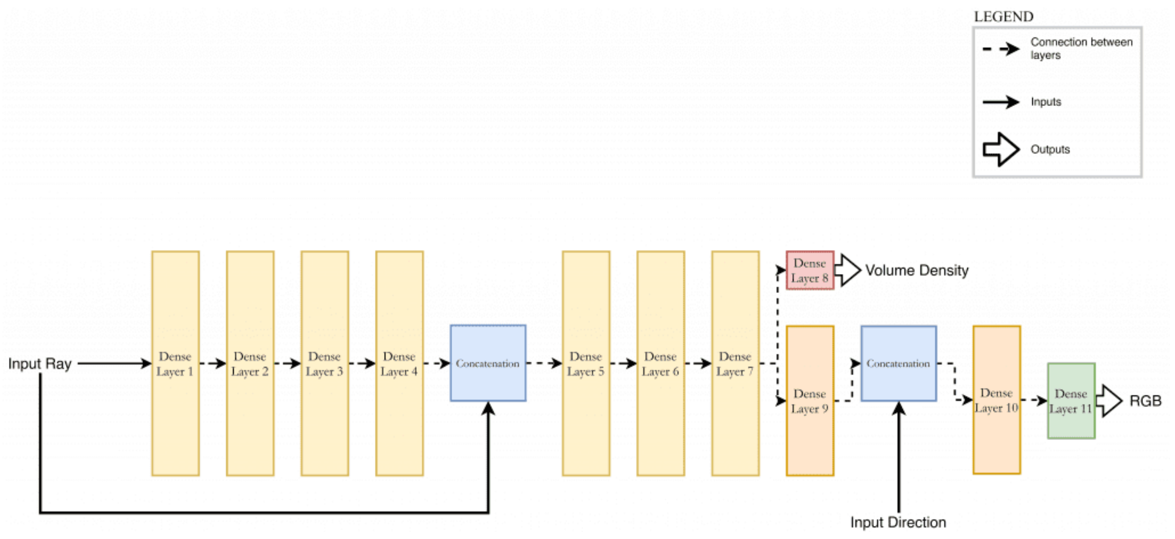
Nerfs by contrast rely on an old concept called light fields, or radiance fields. A light field is a function that describes how light transport occurs throughout a 3D volume. It describes the direction of light rays moving through every x=(x, y, z) coordinate in space and in every direction d, described either as θ and φ angles or a unit vector. Collectively they form a 5D feature space that describes light transport in a 3D scene. The Nerf, inspired by this representation, attempts to approximate a function that maps from this space into a 4D space consisting of color c=(R,G,B) and a density σ, which you can think of as the likelihood that the light ray at this 5D coordinate space is terminated (e.g. by occlusion).

We approximate this continuous 5D scene representation with an MLP network $F\Theta : (x, d) \rightarrow (c, \sigma)$ and optimize its weights Θ to map from each input 5D coordinate to its corresponding volume density and directional emitted color. The network is acting as the "volume" so we can use volume rendering to differentiably render new views. In the below figure we can see a nerf scene representation in which we first synthesize images by sampling 5D coordinates (location and viewing direction) along camera rays then we feed those locations into an MLP to produce a color and volume density.



Nerf MLP Network

Nerf optimizes a deep fully-connected neural network without any convolutional layers (often referred to as a multilayer perceptron or MLP) to represent this function by regressing from a single 5D coordinate (x, y, z, θ, ϕ) to a single volume density and view-dependent RGB color. The architecture is shown below:



We encourage the representation to be multi view consistent by restricting the network to predict the volume density σ as a function of only the location x, while allowing the RGB color c to be predicted as a function of both location and viewing direction.

To accomplish this, the MLP FΘ first processes the input 3D coordinate x with 8 fully-connected layers in which at fifth layer we concatenated the view direction with output feature of fourth layer. We used ReLU activations and 256 channels per layer, and outputs σ and a 256-dimensional feature vector. For getting σ we passed the the output of seventh layer to another MLP layer to output a single channel σ.

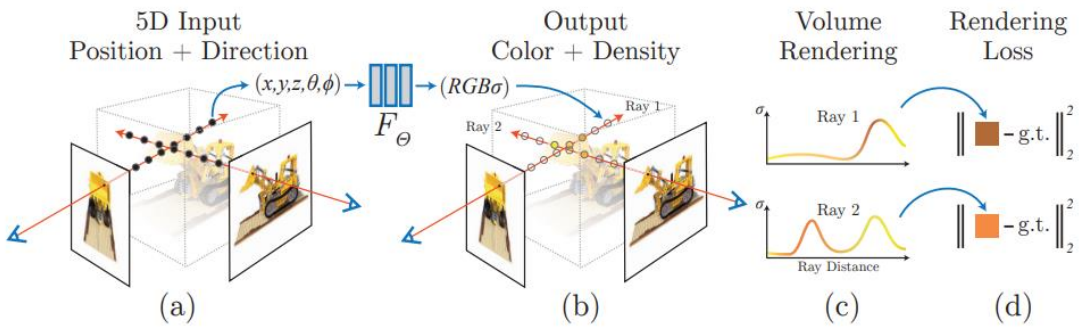
This feature vector is then concatenated with the camera ray's viewing direction and passed to two additional using a ReLU activation. The first layer takes concatenated input and outputs a 128 dimensional vector which is then passed to another layer to output the view-dependent RGB color.

Nerf Architecture

With this function alone we can't generate novel images. Overall, given a trained NeRF model and a camera with known pose and image dimensions, we render this neural radiance field (NeRF) by the following process:

1. For each pixel, march camera rays through scene to gather a set of samples at (x, d) locations.
2. Use (x, d) points and viewing directions at each sample as input to produce output (c,σ) values (essentially rgbσ).
3. Use classical volume rendering techniques to accumulate those colors and densities into a 2D image.

Because this process is naturally differentiable, we can use gradient descent to optimize this model by minimizing the error between each observed image and the corresponding views rendered from our representation. Minimizing this error across multiple views encourages the network to predict a coherent model of the scene by assigning high volume densities and accurate colors to the locations that contain the true underlying scene content. The below figure visualizes this overall pipeline.



We first synthesize images by sampling 5D coordinates (location and viewing direction) along camera rays then we feed those locations into an MLP to produce a color and volume density. We then used volume rendering techniques to composite these values into an image . This rendering function is differentiable, so we can optimize our scene representation by minimizing the residual between synthesized and ground truth observed images.

Rays Generate

A ray in computer graphics can be parameterized as:

$$\vec{r}(t) = \vec{o} + t\vec{d}$$

Where:

- $\vec{r}(t)$ is the ray
- $\vec{o}$ is the origin of the ray
- $\vec{d}$ is the unit vector for the direction of the ray

- t is the parameter (e.g., time)

To build the ray equation, we need the origin and the direction. In the context of NeRF, we generate rays by taking the origin of the ray as the pixel position of the image plane and the direction as the straight line joining the pixel and the camera aperture.

We can easily devise the pixel positions of the 2D image with respect to the camera coordinate frame using the following equations.

$$u = f\frac{x_c}{z_c} + o_x \Rightarrow \boxed{x_c = z_c\frac{u - o_x}{f}}$$

$$v = f\frac{y_c}{z_c} + o_y \Rightarrow \boxed{y_c = z_c\frac{v - o_y}{f}}$$

It is easy to locate the origin of the pixel points but a little challenging to get the direction of the rays. We know that this is the matrix for getting world coordinate from camera coordinate

$$\tilde{X}_c = C_{ex} \times \tilde{X}_w \Rightarrow \tilde{X}_w = C_{ex}^{-1} \times \tilde{X}_c$$

The camera-to-world matrix from the dataset is the C_{ex}^{-1} that we need.

$$C_{ex}^{-1} = \begin{bmatrix} r'_{11} & r'_{12} & r'_{13} & t'_x \\ r'_{21} & r'_{22} & r'_{23} & t'_y \\ r'_{31} & r'_{32} & r'_{33} & t'_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To define the direction vector, we do not need the entire camera-to-world matrix; instead, we use the 3×3 upper matrix that defines the camera's orientation.

$$R'_{ex} = \begin{bmatrix} r'_{11} & r'_{12} & r'_{13} \\ r'_{21} & r'_{22} & r'_{23} \\ r'_{31} & r'_{32} & r'_{33} \end{bmatrix}$$

With the rotation matrix, we can get the unit direction vector by the following equation.

$$\vec{d} = \frac{R'_{ex} \times X_c}{|R'_{ex} \times X_c|}$$

The rays' origin will be the translation vector of the camera-to-world matrix.

$$t'_{ex} = \begin{bmatrix} t'_x \\ t'_y \\ t'_z \end{bmatrix}$$

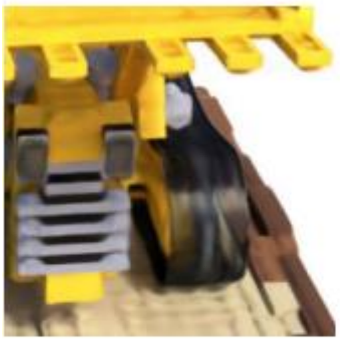
Positional Encoding

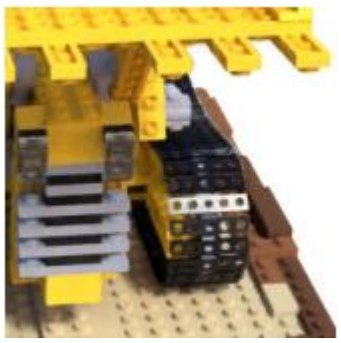
We find that the basic implementation of optimizing a neural radiance field representation for a complex scene does not converge to a sufficiently high-resolution representation and is inefficient in the required number of samples per camera ray. We address these issues by transforming input 5D coordinates with a positional encoding that maps its continuous input to a higher-dimensional space using high-frequency functions to aid the model in learning high frequency variations in the data, which leads to sharper models. This, approach prevents the bias that neural networks have towards lower frequency functions, allowing NeRF to represent sharper details.

The encoding function we use is:

$$\gamma(p) = \left(\sin\left(2^0 \pi p\right), \cos\left(2^0 \pi p\right), \cdots, \sin\left(2^{L-1} \pi p\right), \cos\left(2^{L-1} \pi p\right) \right)$$

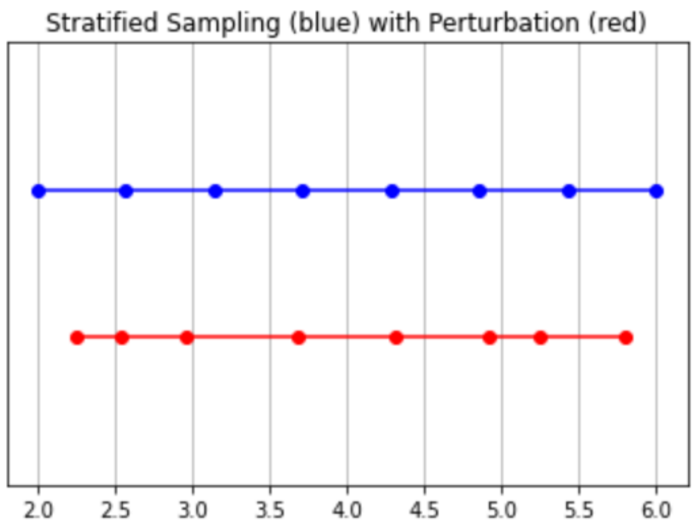
Positional encoders in the NeRF implementation has alternating sine and cosine expressions. The sine and cosine functions make the encoding continuous, and the 2^i term makes it similar to the binary system. This function γ(·) is applied separately to each of the three coordinate values in x (which are normalized to lie in [−1, 1]) and to the three components of Cartesian viewing direction unit vector d (which by construction lie in [−1, 1]). For our Nerf implementation, we set L = 10 for γ(x) and L = 4 for γ(d). Below diagram shows the output for without positional encoding and with positional encoding.


Without Positional Encoding


With Positional Encoding

Stratified Sampling

In this model, the RGB value that the camera ultimately picks up is the accumulation of light samples along the ray passing through that pixel. The classical volume rendering approach is to accumulate and then integrate points along this ray, estimating at each point the probability that the ray travels without hitting any particle. Each pixel therefore requires a sampling of points along the ray passing through it. To best approximate the integral, their stratified sampling approach is to divide the space evenly into N bins and draw a sample uniformly from each. Rather than simply drawing samples at regular spacing, the stratified sampling approach allows the model to sample a continuous space, therefore conditioning the network to learn over a continuous space. The perturb setting determines whether to sample points uniformly from each bin or to simply use the bin center as the point. In most cases, we want to keep perturb = True as it will encourage the network to learn over a continuously sampled space. It may be useful to disable for debugging.



Volume Rendering

We render the color of any ray passing through the scene using principles from classical volume rendering. The volume density σ(x) can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location x. The expected color C(r) of camera ray r(t) = o + td with near and far bounds tn and tf is:

$$C(r) = \int_{tn}^{tf} T(t)\sigma(r(t))c(r(t), d)dt$$

Let us break this equation down into simple parts.

- The term $C(r)$ is the color of the point of the object.
- $r(t) = o + td$ is the ray that is fed into the network where the variable stands for the following:
    - o as the origin of the ray point
    - d is the direction of the ray
    - t is the samples between the near and far points used for the integral
- $\sigma(r(t))$ is the volume density which can also be interpreted as the differential probability of the ray terminating at the point t.
- $c(r(t))$ is the color of the ray at the point t

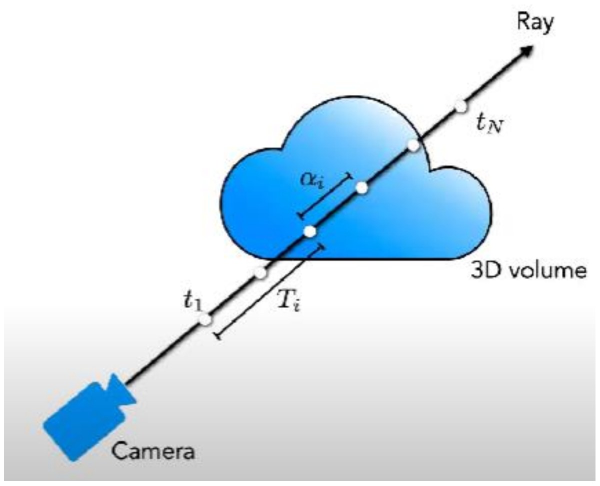
- Another term

$$T(t) = \exp\left(-\int_{tn}^{t} \sigma(r(s))ds\right)$$

This represents the transmittance along the ray from near point tn to the current point t. Think of this as a measure of how much the ray can penetrate the 3D space to a certain point.

Rendering a view from our continuous neural radiance field requires estimating this integral from t_n to t_f C(r) for a camera ray traced through each pixel of the virtual camera image.

To estimate this integral we use a discrete set of samples, stratified sampling enables us to represent a continuous scene representation because it results in the MLP being evaluated at continuous positions over the course of optimization. We use these samples to estimate C(r). This function below is for calculating C(r) from the set of (ci , σi). Each RGB sample is weighted by its alpha value. Higher alpha values indicate higher likelihood that the sampled area is opaque, therefore points further along the ray are likelier to be occluded. The cumulative product ensures that those further points are dampened.



- Rendering model ray for $r(t) = o + td$:

$$C \approx \sum_{i=1}^{N} \underbrace{T_i}_{\text{weights}} \alpha_i \underbrace{c_i}_{\text{colors}}$$

- How much light is blocked earlier along ray:

$$T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$$

- How much light is contributed by ray segment i:

$$\alpha_i = 1 - e^{-\sigma_i \delta t_i}$$

where δi = ti+1 − ti is the distance between adjacent samples

Hierarchical Sampling

There is one problem with the original structure that is the stratified sampling method would sample N points along each camera ray. This lead to free space and occluded regions that do not contribute to the rendered image are still sampled repeatedly. That ultimately leads to an inefficient rendering. This means we don't have any prior understanding of where it should sample.

The solution to the above problem that we are using in nerf:

- Build two identical NeRF MLP models, the coarse and fine network. Coarse network encodes the broad structural properties of the scene whereas the fine network enables thin and complex structures like meshes and branches to be realized.
- Sample a set of Nc points along the camera ray using the stratified sampling strategy. These points will be used to evaluate the coarse network at these locations.
- The output of the coarse network is used to produce a more informed sampling of points along each ray. The new samples are where there is more relevant parts of the 3D scene along the ray.

Hierarchical sampling procedure:

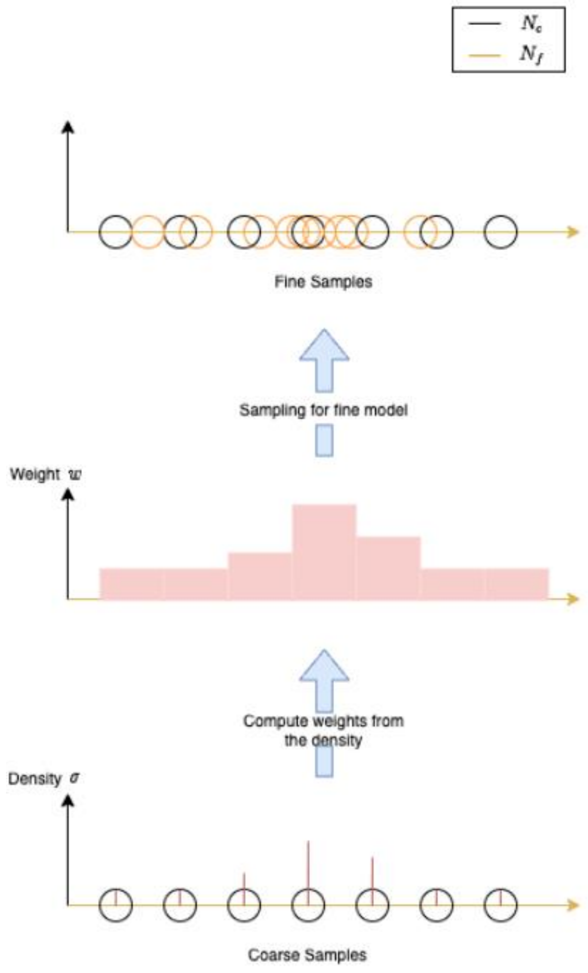
- To implement this we first rewrite the alpha composited color from the coarse network as a weighted sum of all sampled colors ci along the ray

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i\,, \qquad w_i = T_i(1 - \exp(-\sigma_i \delta_i))$$

- The weights, when normalized, produce a piecewise-constant probability density function.

$$\widehat{w}_i = \frac{w_i}{\sum_{j=1}^{N_c} w_j}$$

- We sample a second set of Nf locations from this distribution using inverse transform sampling, evaluate our "fine" network at the union of the first and second set of samples, and compute the final rendered color of the ray Cˆ$_f$ (r) using volume rendering procedure but using all Nc+Nf samples.



Loss Function

Loss function that we used is the total squared error between the rendered and true pixel colors for both the coarse and fine renderings. We use the volume rendering procedure described earlier to render the color of each ray from both sets of samples. Below is the formula of loss function:

$$\mathcal{L} = \sum_{\mathbf{r}\in\mathcal{R}} \left[\left\| \hat{C}_c(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 + \left\| \hat{C}_f(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 \right]$$

where R is the set of rays in each batch, and C(r), Cˆ$_r$ (r), and Cˆ$_f$ (r) are the ground truth, coarse volume predicted, and fine volume predicted RGB colors for ray r respectively. Note that even though the final rendering comes from Cˆ$_f$ (r), we also minimize the loss of Cˆ$_c$ (r) so that the weight distribution from the coarse network can be used to allocate samples in the fine network. This function, when applied to the entire pipeline, is still fully differentiable. This allows us to train the model parameters using backpropagation.