NeRF

Neural radiance fields for representing scenes (Neural Rendering)

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

Authors: PhD Comp Sci students & professors @ Berkeley (Google interns)

▼ **Terminology**

Scene: the ground truth 3D construction of a space containing objects

Ray: a ray of light originating from the camera and going through the scene it's directed at (a line)

Ray marching: a ray is a line, we step along it in intervals and sample the colour & density at each point

---

1. Standard Vs NeRF Approach to Training

Standard

Features x	Label y
Multiple angles of images for 🏡	True 3D reconstruction of 🏡
Multiple angles of images for 🏝️	True 3D reconstruction of 🏝️
…	…

- Train NN on multiple camera angle images of multiple scenes & output 3D reconstruction (of some form) which we can take cross-sections of to grab 2D images.

NeRF

Features x
🏡 Angle 1 Image
🏡 Angle 2 Image

Features x
…

- Train NN on multiple camera angle images for one scene: each new scene has a new neural network. ‘Overfit the scene.’
  - ‘the scene is stored in the NN weights’

---

2. NeRF Neural Network Structure

$\text{Input: 5D vector } (x, y, z, \theta, \phi) \text{ where:}$

- $\text{Ray marching position : } \mathbf{x} = (x, y, z)$
- $\text{Ray angle vector : } \mathbf{d} = (\theta, \phi)$

$\text{Output: } (d, \mathbf{c}) \text{ where:}$

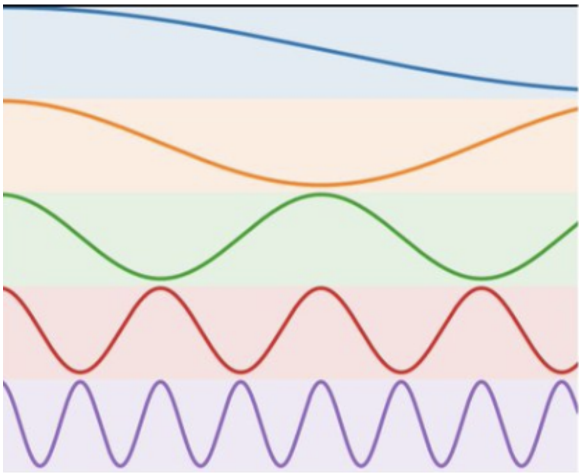
- $\text{Density (transparency) : } d$
- $\text{Colour : } \mathbf{c} = (r, g, b)$

for each sampled point $\mathbf{x}$ on the ray.

Fully connected network with no convolutional layers (MLP)

1. Positional encoding
   - $\gamma(p) = (sin(2^0 \pi p), cos(2^0 \pi p), ..., sin(2^{L-1} \pi p), cos(2^{L-1} \pi p))$
   - We use this positional encoding (gamma function) to transform each coordinate & viewing direction input parameter, which are first normalised to lie in [-1,1], separately to transform each number into a higher dimensional list of values. Set L=10 for gamma(x) and L=4 for gamma(d). These transformed values are then the entries passed into the model.
   - Helps network focus on fine details since small changes in p can have big changes in $\gamma(p)$
     - This is a similar mechanism to positional encoding used in transformers, but transformers use it for a different reason: ‘providing the discrete positions of

tokens in a sequence as input to an architecture that does not contain any notion of order'



2. Process only $\mathbf{x}$ (point coords) through first 8 layers
    - ReLu activation & 256 neurones per layer
    - outputs a 256-dimension feature vector
3. Concatenate this feature vector with the viewing direction
4. Pass this concatenation to a penultimate dense layer
    - ReLu activation & 128 neurones
5. Output layer
    - ReLu activation & 4 neurones (r, g, b, d)

▼ **Note**: within the model, it's specified that density only depends on ray marching position $\mathbf{x}$ not ray angle vector $\mathbf{d}$ whereas colour depends on both. This is an assumption made since very few materials are more transparent from different viewing directions

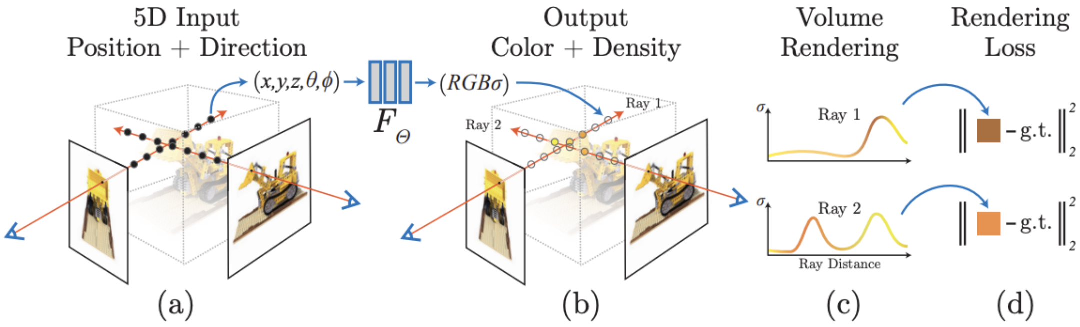
- Enforced by: the network only uses x ….

---

3. Volume Rendering

To build 2D image: query 5D coordinates along camera rays & use classic volume rendering techniques to project the output (colours and densities) into an image

- Classical volume rendering is naturally differentiable so GD methods work & we don't require ground truth (3D occupancy fields), just 2D images with camera parameters



1. We take n points along each ray & parameterise each point by it's distance from ray origin denoted t: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$

   To obtain points, we randomly sample the point from within bins: $t_i \backsim \mathcal{U}[t_n + \frac{i-1}{n}(t_f - t_n), t_n + \frac{i}{n}(t_f - t_n)]$

   - Obtains spaced out but random points

2. Run each point with it's viewing angle through the network to obtain it's colour & density

3. Compute overall ray colour $C(r)$: a function of individual point densities & colours

   Classical volume rendering:

   $C(r) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t)), \mathbf{d})dt$

   where $T(t) = exp(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds)$

   But here we estimate the above using numerical quadrature

---

4. Optimising NeRF Run Speed

'Hierarchical Sampling to allocate the MLP's capacity towards space with visible scene content'

- Volume rendering by sampling n points along a ray is inefficient
- So lets sample points which are expected to have a big impact on the final colour (ie. ignore free space and occluded (blocked) regions)

In each training/calculation step:

1. Choose 64 sample points per ray as specified in ***volume rendering section*** (randomly from bins)
2. Run ‘coarse network’
   - input the 64 points per ray into the MLP
3. Express the overall ray colour as a weighted expression of the 64 points

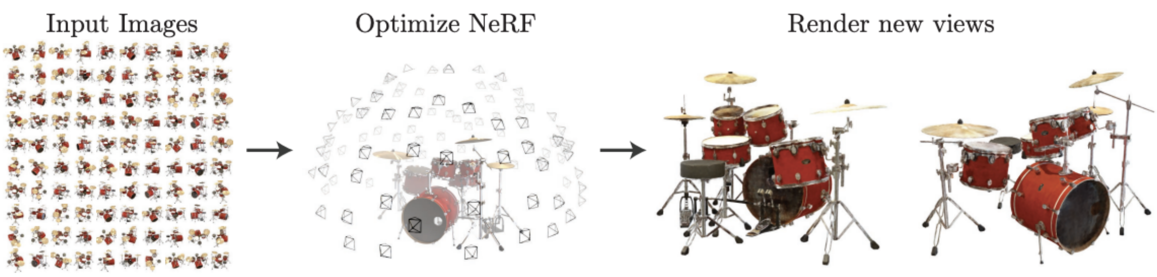
   $\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i, \qquad w_i = T_i(1 - exp(-\sigma_i \delta_i))$
4. Generate 128 new ray sample points but so that new points are close to the high weighted points from before (using a special sampling distribution)
5. Run ‘fine network’
   - input the original 64 + 128 new =192 points per ray into the MLP
6. Compute overall colour per ray
7. Collect lots of rays to create 2D images (overall ray colour is a pixel)

---

5. Compiling & Training NeRF

Training Images

- Collected 20-50 images from around the scene with camera position & angle measurements
  - known problem: hard to know ur exact camera position and direction



Training

- Batch size = 4096 rays

- The optimisation for a single scene typically take around 100–300k iterations to converge on a single NVIDIA V100 GPU (about 1–2 days)

In each iteration of learning:

- sample at random a batch of camera rays
- random sample points on each ray & run coarse followed by fine networks to obtain expected ray colour.

Loss Function

Loss = total squared error between rendered and true pixel colour for both fine and coarse renderings:

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

- Including coarse loss allows the coarse model to get better at predicting relevant points of colour too

Optimiser

- Adam optimiser
    - learning rate decays exponentially from 5×10^{-4} to 5×10^{-5}
    - (other Adam hyper-parameters are left at default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$

---

6. Results

- SOTA on multiple datasets & metrics:
    - PSNR/SSIM (higher is better)
    - LPIPS (lower is better)
- Can artificially change reflections/lighting in images by fixing ray marching positions but changing ray viewing directions (which would be impossible in reality)