Cornell University

CS 5670: Introduction to Computer Vision

Fall 2025. Instructor: Andrew Owens

Problem Set 6: Neural Radiance Fields

Posted: Thursday, November 20, 2025 **Due:** Monday, December 8, 2025

We have created two Gradescope assignments for PS6. This is because problem 6.1 are theory questions, and problems 6.2 are programming questions. For the theory questions, please prepare a PDF with your solutions and upload it to the corresponding assignment on Gradescope. For the programming questions, you should finish the Colab Notebook, generate a PDF and then upload it to the corresponding assignment on Gradescope.

Colab Notebook: `https://drive.google.com/file/d/1rLEU6d0A2oX7mB_SlT85vwaIrUoXyGWK`

Problem 6.1 *The Mechanics of Neural Radiance Fields (NeRF)*

In this problem, you will perform (by hand, a programming language, or a calculator) some of the steps that go into rendering a view from a NeRF. Like you've done with other theory problems, you will submit your answer to this question as a separate PDF on Gradescope.

NeRF notation reference:

- Ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Volume Density: $\sigma(t)$
- Color: $\mathbf{c}(t)$
- Interval: $\delta_i = t_{i+1} - t_i$

(a) **Positional Encoding (2 points).** NeRF uses positional encoding to map input coordinates to a higher-dimensional space to capture high-frequency details. The encoding function $\gamma(p)$ is defined as:

$$\gamma(p) = \big(\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)\big) \tag{1}$$

Assume that we are encoding a single scalar coordinate $x = 0.25$. Calculate the resulting positional encoding vector $\gamma(x)$ for $L = 2$.

(b) **Discrete Volume Rendering (5 points).** The core of NeRF is the discretized volume rendering equation used to calculate the color of a pixel.

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i \tag{2}$$

where the transmittance T_i is:

$$T_i = \exp\Big(-\sum_{j=1}^{i-1} \sigma_j \delta_j\Big) \tag{3}$$

Consider a single ray passing through a scene. We sample 3 points along this ray ($N = 3$). The distance between samples is constant $\delta_i = 1.0$ for all i. The network predicts the following outputs (Density σ and Grayscale color $\mathbf{c}$) for these points:

- Sample 1: $\sigma_1 = 0.2, c_1 = 1.0$
- Sample 2: $\sigma_2 = 2.0, c_2 = 0.5$
- Sample 3: $\sigma_3 = 0.1, c_3 = 0.0$

(i) Calculate the **Alpha** (α_i) for each sample, where $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$.

(ii) Calculate the **Transmittance** (T_i) for each sample.

(iii) Calculate the **Weights** (w_i) for each sample, where $w_i = T_i \alpha_i$.

(iv) Calculate the **Final Pixel Color** $\hat{C}$.

(c) **The Solid Wall Intuition (4 points).** Understanding transmittance is crucial for understanding how NeRF handles occlusion. Suppose there is a ray passes through 3 samples ($i = 1, 2, 3$) with constant distance δ.

- Sample 1 empty air ($\sigma_1 = 0$).
- Sample 2 is a solid wall, meaning its density $\sigma_2 \to \infty$.
- Sample 3 is behind the wall.

Show that the weight contribution of the point behind the wall (sample 3) is exactly 0 by evaluating the limit of w_3 as $\sigma_2 \to \infty$

(d) **Stratified Sampling (4 points).** In practice, NeRF does not sample points at fixed intervals during training; it uses stratified sampling to avoid aliasing.

$$t_i \sim \mathcal{U}[t_{near} + \frac{i-1}{N}(t_{far} - t_{near}), t_{near} + \frac{i}{N}(t_{far} - t_{near})] \tag{4}$$

Let $t_{near} = 2.0$ and $t_{far} = 6.0$, and number of samples $N = 4$.

(i) Define the boundaries of the 4 "bins" from which we will draw samples.

(ii) Explain in one sentence why we randomize t during training but usually fix t during inference (e.g., when rendering a "fly through" video that moves through different locations of the captured scene).

Problem 6.2 *Neural Radiance Field Implementation*

In this project, you will learn:

- Basic usage of the PyTorch deep learning library.
- How to understand and build neural network models in PyTorch.
- How to build a Neural Radiance Field (NeRF) from a set of images.
- How to synthesize novel views from a NeRF.

In class, we learned that standard coordinate-based MLPs cannot natively represent high-frequency functions on low-dimensional domains. To better preserve high frequency variation in the data, we can encode each of the scalar input coordinates with a sequence of sinusoids with exponentially increasing frequencies before passing them into the network.

1. **Fitting a single image with positional encoding.**

   (a) **Positional Encoding (4 points).** Implement a function `positional_encoding`() that can encode each of the scalar input coordinates with a sequence of sinusoids with exponentially increasing frequencies before passing them into the network.

   (b) **2D Image Fitting (4 points).** Now, let's try to fit a 2D image with a multilayer perceptron (MLP). We provide an example of network architecture called **Model2d**. You can run all the way to the last cell to execute the training process. Without any modification, you should get PSNR ~ 27 after training for 10,000 iterations. Now, your task is to modify **Model2d**, such that after training for 10,000 iterations with `num_encoding_functions` = 6, PSNR is greater than or equal to 30.

   (c) **Positional Encoding Effectiveness (4 points).** Try runnng the same training procedure `train_2d_model`, but without positional encoding (i.e., `num_frequencies` = 0), or with different number of frequencies and see what happens. What's the effect of positional encoding and the effect of different numbers of frequencies?

A neural radiance field represents a scene as the volume density σ and the RGB color $\mathbf{c}$ at any point in space. We can render an image from this neural radiance field by estimating the color at each pixel in the image by shooting a camera ray from through that pixel through the scene, and accumulating color and density along the way. Rendering an image from neural radiance fields consists of three steps:

2. **Compute the origin and direction of rays through all pixels of an image (4 points).** The goal is to implement a function `get_rays`() to compute the origin and the direction of camera ray through each pixel of the image in the world coordinate space.

3. **Sample the 3D points on the camera rays (4 points).** Given the origin $\mathbf{o}$ and the direction $\mathbf{d}$ of a ray, implement a function `sample_points_from_rays`() to sample 3D points $\mathbf{x}$ on this ray.

4. **Compute compositing weight of samples on camera ray (4 points).** Implement the function `compute_compositing_weights()` to compute the weights $w_i = T_i(1 - \exp(-\sigma_i \delta_i))$ for each sample point on camera rays.

5. **Render one image with NeRF (4 points).** Combine the previous (2), (3), and (4) implementations, and implement a function `render_image_nerf()` so that we can render an image with NeRF.

6. **Train NeRF on 360 scene (4 points).** If you passed all tests, you can start training NeRF! Expect to reach PSNR greater than or equal to 20 after training for 1,000 iterations with `num_encoding_functions` = `6`.

Acknowledgements. The starter code and data were adapted from Noah Snavely's CS 5670 class. The theory problems were created by Yen-Yu Chang.