Homework 1: NeRF Assignment

Name: Yixun Hu

NetID: yh4742

---

1. Implementation Details

1.1 Positional Encoding (`encoder.py`)

The `PositionalEncoder` class maps low-dimensional input coordinates to a higher-dimensional space using sinusoidal functions, enabling the MLP to learn high-frequency scene details.

Key Implementation:

```
if log_sampling:
    # Logarithmic sampling: 2^0, 2^1, ..., 2^(max_freq)
    freq_bands = 2. ** torch.linspace(0., max_freq, N_freqs)
else:
    # Linear sampling: evenly spaced between 2^0 and 2^max_freq
    freq_bands = torch.linspace(2. ** 0., 2. ** max_freq, N_freqs)

for freq in freq_bands:
    for p_fn in periodic_fns:
        embed_fn = lambda x, p_fn=p_fn, freq=freq: p_fn(x * freq)
        embed_fns.append(embed_fn)
        out_dim += d
```

The encoding transforms input `x` into:

$$\gamma(x) = [x, \sin(2^0 x), \cos(2^0 x), \sin(2^1 x), \cos(2^1 x), \ldots, \sin(2^{L-1} x), \cos(2^{L-1} x)]$$

This follows Section 5.1 of the NeRF paper, allowing the network to represent high-frequency variations in color and geometry. For code level implementation, I followed the hint to use `lambda` function to apply the periodic functions to the input.

1.2 NeRF MLP Architecture (`nerf.py`)

The `NeRF` class implements the MLP architecture from Figure 7 of the paper.

Key Implementation:

```
# 8 FC layers with skip connection at layer 5
self.pts_linears = nn.ModuleList(
    [nn.Linear(input_ch, W)] +
    [nn.Linear(W, W) if i not in self.skips else nn.Linear(W + input_ch,
W)
     for i in range(D - 1)]
)

# View-dependent branch
self.feature_linear = nn.Linear(W, W)      # Feature vector
self.alpha_linear = nn.Linear(W, 1)        # Volume density (σ)
self.rgb_linear = nn.Linear(W // 2, 3)     # RGB color
```

Architecture Details:

- **Position branch**: 8 fully-connected layers (256 channels each) with a skip connection at layer 5 that concatenates the input features
- **Density output**: View-independent

  $$\sigma$$

  predicted directly from position features. The alpha linear layer is a one-dimensional linear layer that outputs the volume density.
- **Color output**: View-dependent RGB predicted after concatenating position features with view direction encoding. The rgb linear layer is a three-dimensional linear layer that outputs the RGB color.

1.3 Volume Rendering (`renderer.py`)

The renderer implements classical volume rendering with three key components:

1.3.1 3D Point Sampling

Given a ray origin **o** and direction **d**, I evaluate points at sampled depths

$$t$$

using the parametric ray equation. Broadcasting allows this to be computed efficiently for all rays and all samples simultaneously, producing a tensor of shape `[N_rays, N_samples, 3]`.

```
# Parametric ray equation: p(t) = o + t*d
pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None]
```

1.3.2 Alpha Computation

The alpha value represents the opacity of each sample segment along the ray. As light travels through a medium with volume density

$$\sigma$$

over a distance

$$\delta$$

, the fraction of light absorbed is

$$1 - e^{-\sigma\delta}$$

. ReLU is applied to the raw density to ensure

$$\sigma \geq 0$$

, which is physically meaningful (negative density has no real-world interpretation).

```
# α = 1 - exp(-σ * δ)
alpha = 1. - torch.exp(-act_fn(raw) * dists)
```

1.3.3 Volume Rendering Integration

The final pixel color is computed via numerical quadrature of the volume rendering integral. Each sample's contribution depends on two factors: (1) the transmittance

$$T_i$$

, i.e., how much light has **not** been blocked by samples closer to the camera, and (2) the local opacity

$$\alpha_i$$

. The transmittance is computed as an exclusive cumulative product of

$$(1 - \alpha)$$

. The resulting weights

$$w_i = T_i \cdot \alpha_i$$

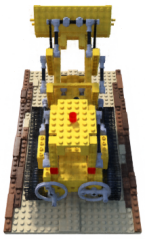
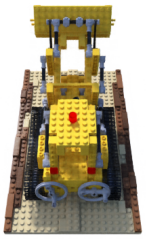
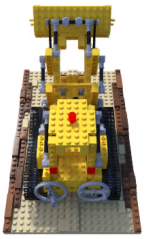
naturally sum to at most 1, and are used to aggregate color and depth via weighted sums.

```
# Transmittance: T_i = Π_{j<i}(1 - α_j)
T = torch.cumprod(
    torch.cat([torch.ones((alpha.shape[0], 1), device=device), 1. - alpha
+ 1e-10], -1), -1
)[:, :-1]

# Weights: w_i = T_i * α_i
weights = alpha * T

# Final color: C = Σ w_i * c_i
rgb_map = torch.sum(weights[..., None] * rgb, -2)

# Expected depth: d = Σ w_i * z_i
depth_map = torch.sum(weights * z_vals, -1)
```

---

2. Training Results and Analysis

2.1 Lego Scene (Synthetic)

50K Iterations	100K Iterations	150K Iterations	200K Iterations
			

Quality Progression:

- **50K iterations**: Basic structure emerges, but details are blurry and colors are washed out. The overall shape of the lego bulldozer is recognizable but edges are soft.
- **100K iterations**: Significant improvement in sharpness and color saturation. Fine details like the treads and mechanical parts become more defined.
- **150K iterations**: High-frequency details are well-captured. Specular highlights and shadows appear more realistic.
- **200K iterations**: Convergence achieved with crisp edges, accurate colors, and proper view-dependent effects. The model successfully captures the metallic appearance and geometric complexity.

2.2 Fern Scene (Real-World LLFF)

50K Iterations	100K Iterations	150K Iterations	200K Iterations
			

Quality Progression: - **50K iterations**: The fern's overall structure is captured but with noticeable blur, especially in the intricate leaf patterns.

- **100K iterations**: Leaf edges become sharper, and the depth layering of overlapping fronds improves.
- **150K iterations**: Fine leaf textures and veins start appearing. The complex occlusion patterns are better resolved.
- **200K iterations**: The model achieves photorealistic quality with detailed leaf structures, proper depth-of-field effects, and accurate color reproduction of the natural scene.

---

3. Novel View Synthesis Videos

I changed the camera path from spiral to a sweeping path around the scene.

```
# render_poses = torch.stack([pose_spherical(angle, -30.0, 4.0) for angle
in np.linspace(-180,180,40+1)[:-1]], 0)
render_poses = torch.stack([
    pose_spherical(0, phi, 4.0)
    for phi in np.linspace(-30, -90, 60)
], 0)
```

The videos are shown in custom_rgb.mp4. This demonstrate the model's ability to synthesize consistent, high-quality views from novel camera positions not seen during training.