Saatvik Billa CS180 Final Project: Neural Radiance Fields!

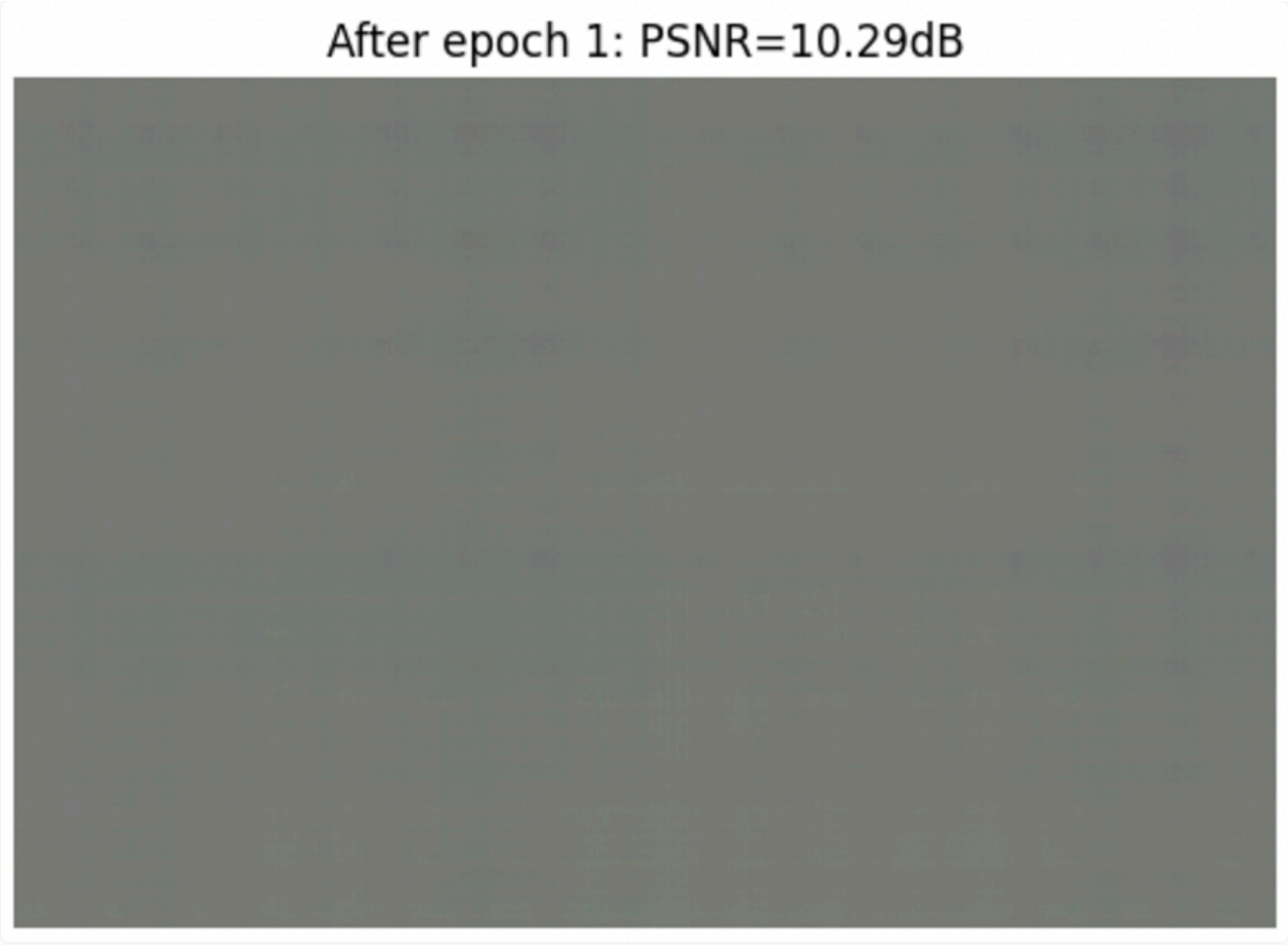
Part 1: Fit a Neural Field to a 2D Image

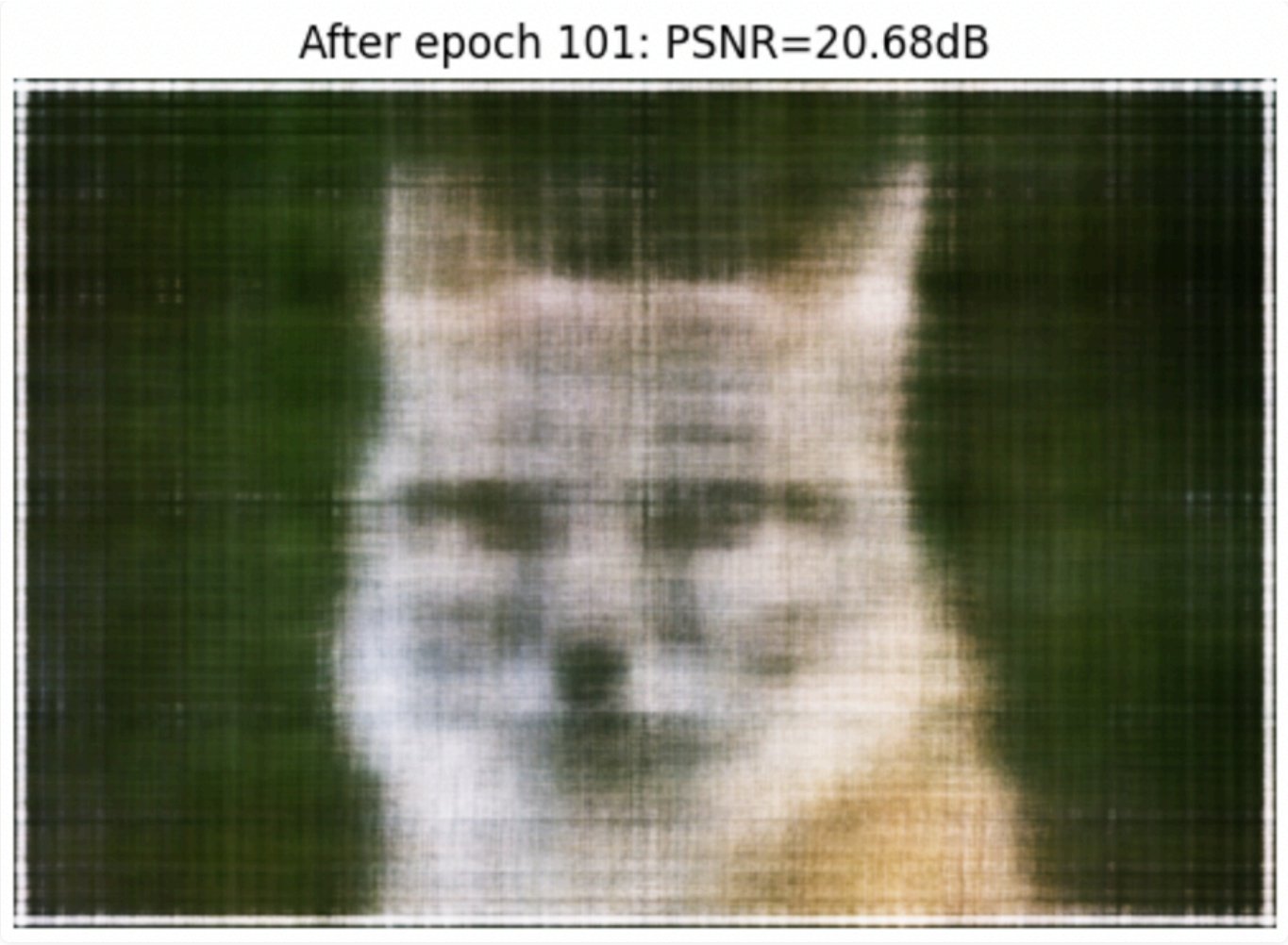
For this project, I implemented an Multi-layer Perceptron (MLP) to reconstruct a 2D image using a neural network. I started by building a positional encoding module, which transforms 2D image coordinates into a higher-dimensional representation using sinusoidal functions. This is essential for capturing fine details and patterns in the image. The MLP was designed with multiple hidden layers, activated by ReLU, and ending with a Sigmoid layer to predict RGB pixel values. The MLP takes the positional encodings as input and learns to map coordinates to their corresponding pixel colors.

To train the model, I created a custom dataset class that samples a batch of random pixel coordinates and their respective RGB values at each iteration. For supervision, I normalized the pixel colors to the range [0, 1] and used Mean Squared Error (MSE) as the loss function. The Adam optimizer was used for training, and I tracked performance using PSNR curve. During training, I observed that the network progressively reconstructed the image as it optimized, confirming that the positional encoding and MLP were effective in learning the 2D neural representation.

[Insert Varying Hyperparameters]

Fox Image Progression


After epoch 1: PSNR=10.29dB


After epoch 101: PSNR=20.68dB


After epoch 201: PSNR=22.66dB


After epoch 301: PSNR=24.67dB


After epoch 401: PSNR=25.22dB


After epoch 501: PSNR=25.28dB


After epoch 601: PSNR=25.08dB

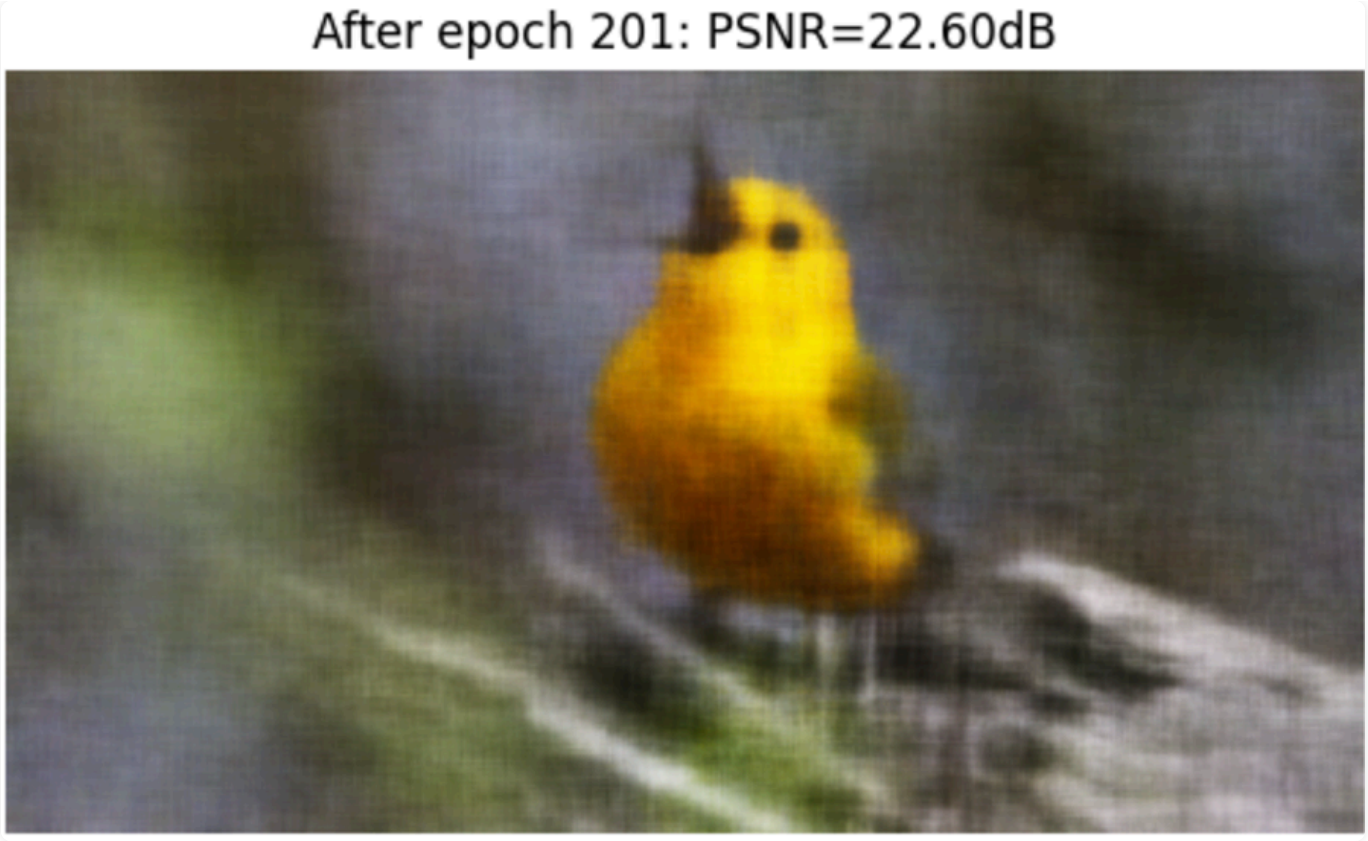

After epoch 701: PSNR=25.74dB


After epoch 801: PSNR=26.03dB

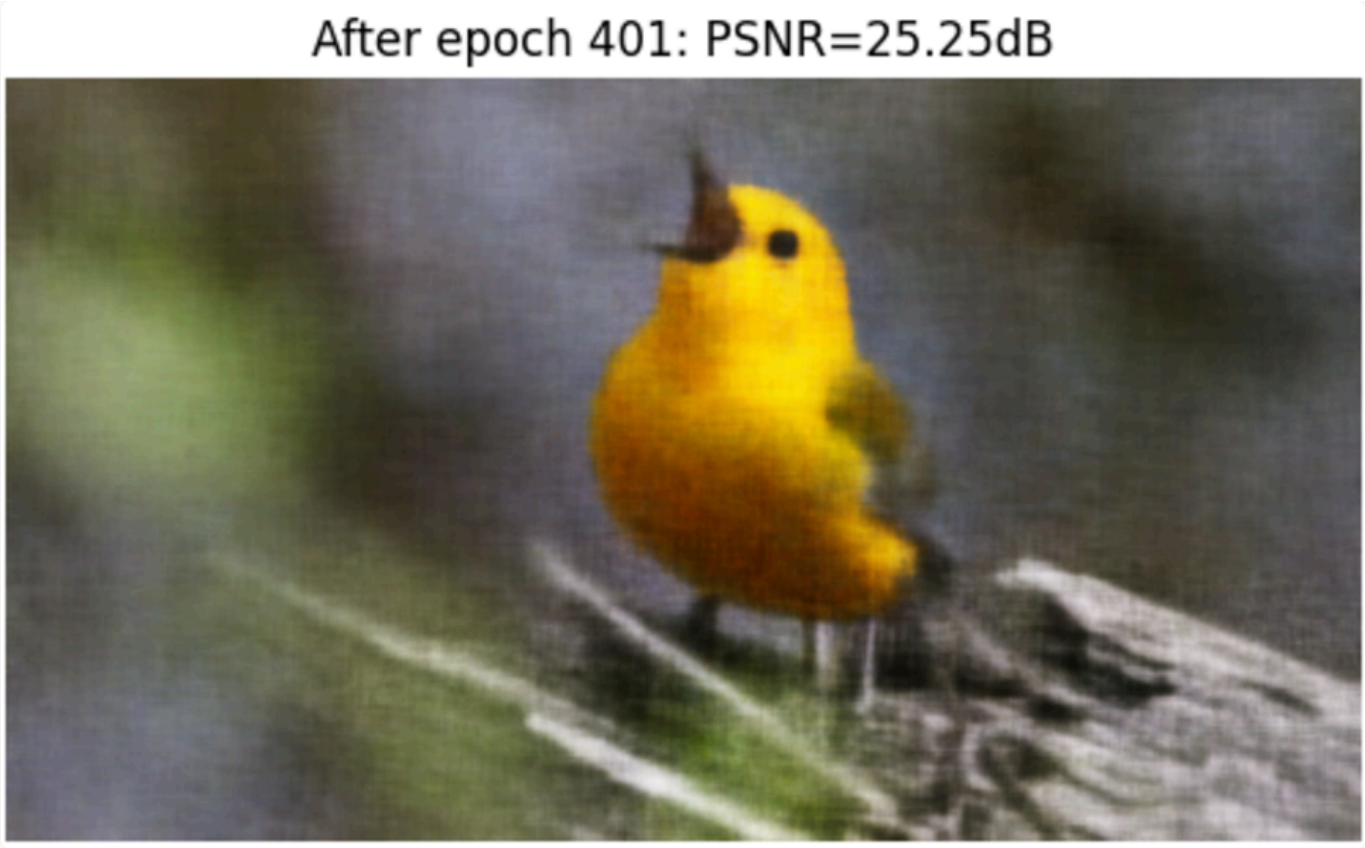
Bird Image Progression

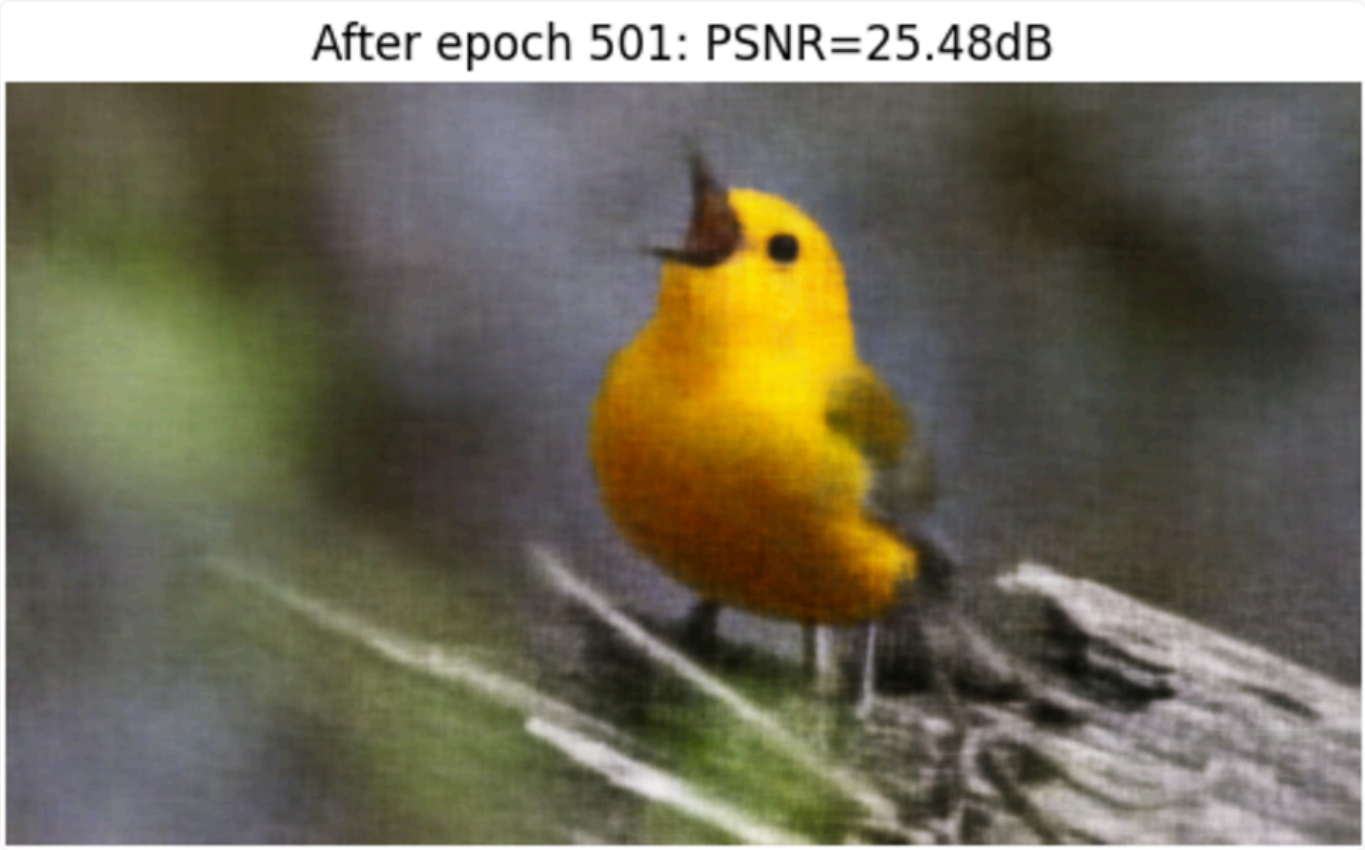
After epoch 1: PSNR=13.51dB

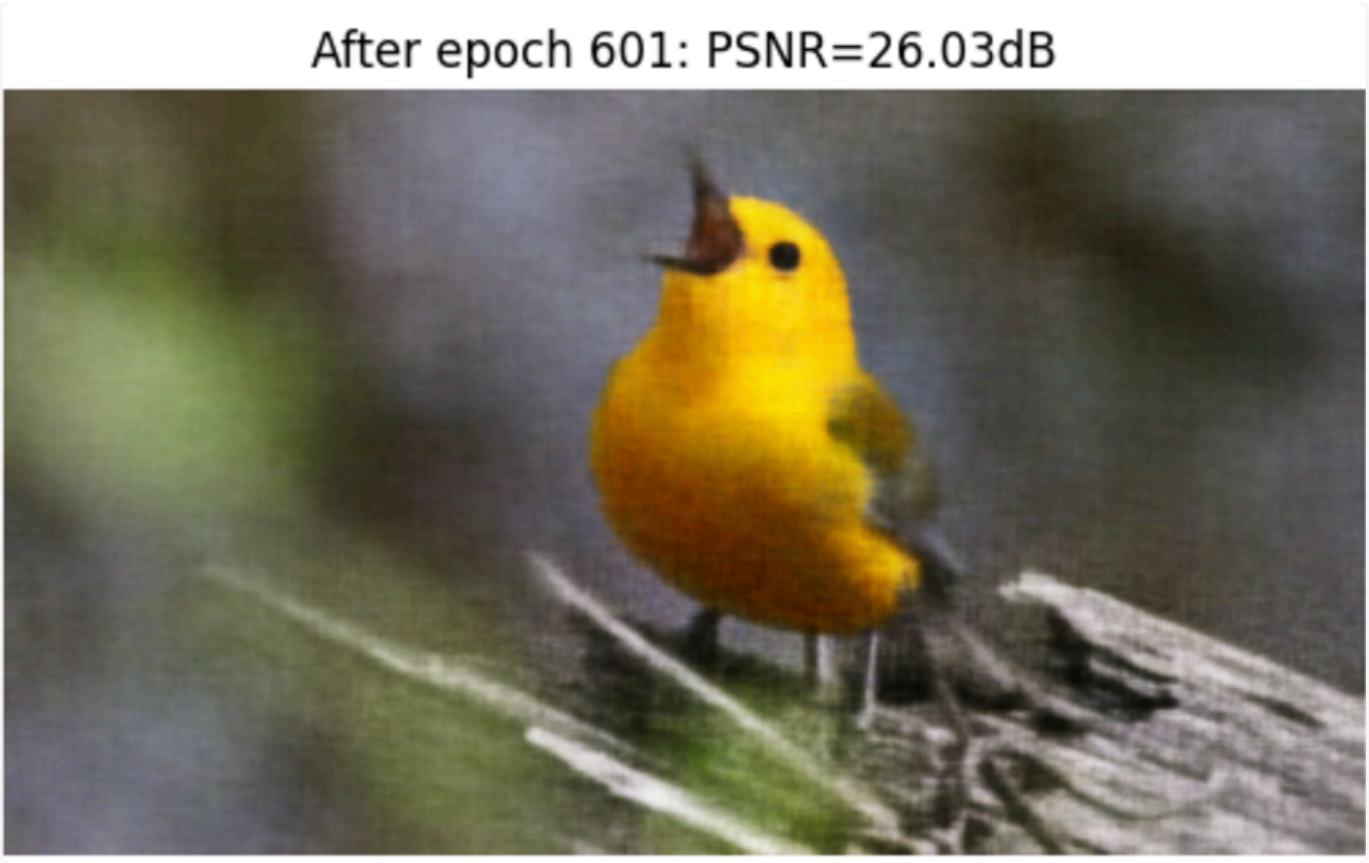
After epoch 101: PSNR=21.35dB


After epoch 201: PSNR=22.60dB

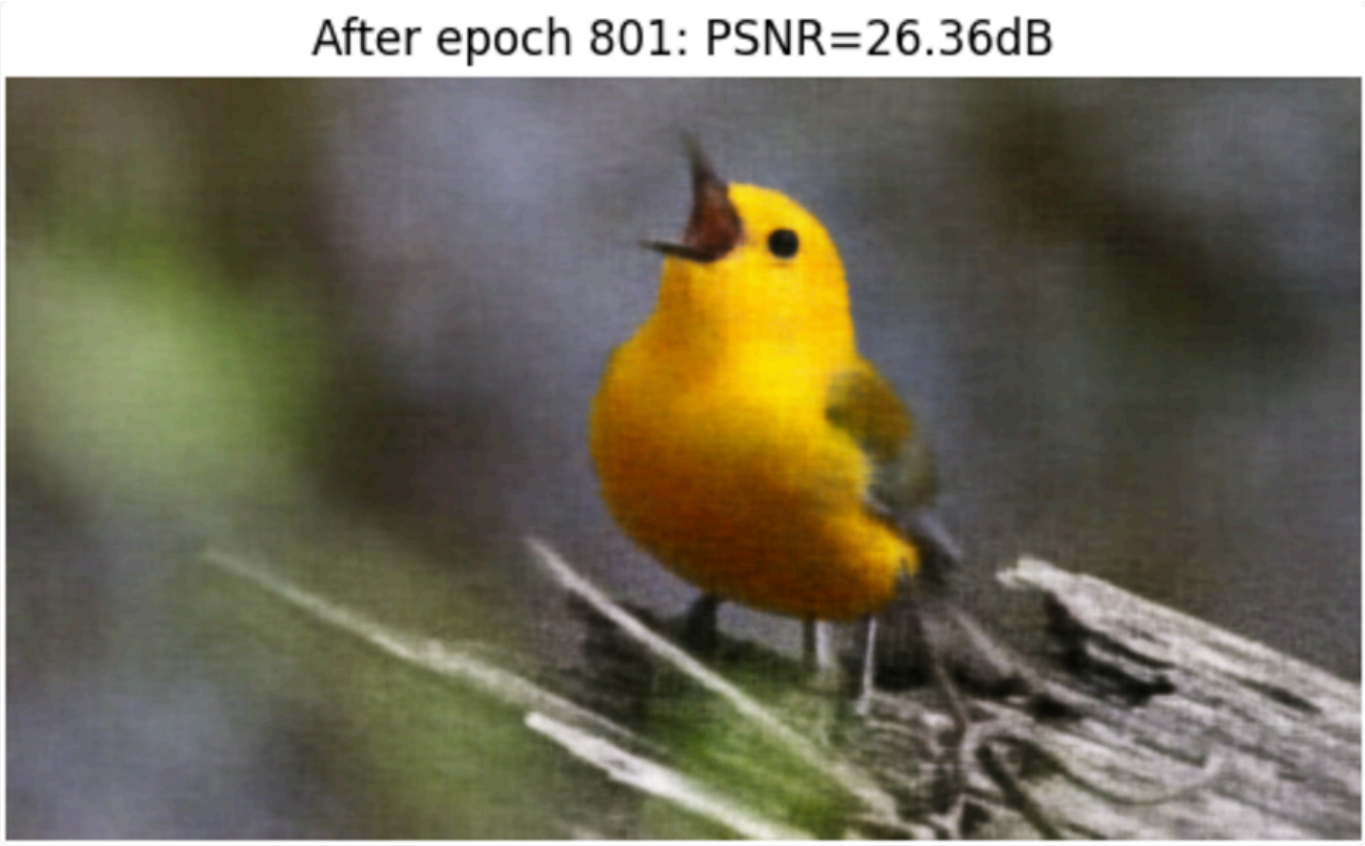

After epoch 301: PSNR=24.20dB

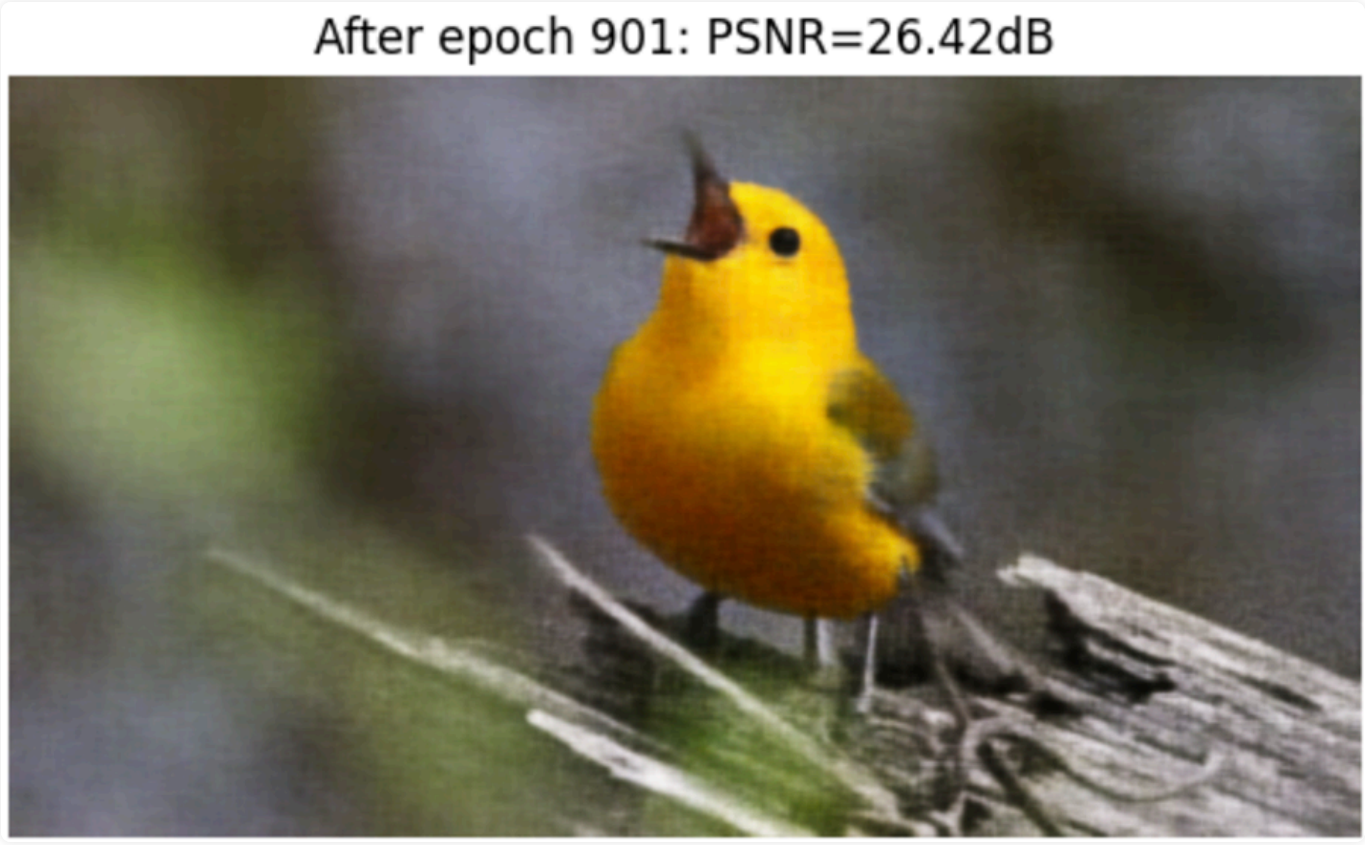

After epoch 401: PSNR=25.25dB


After epoch 501: PSNR=25.48dB

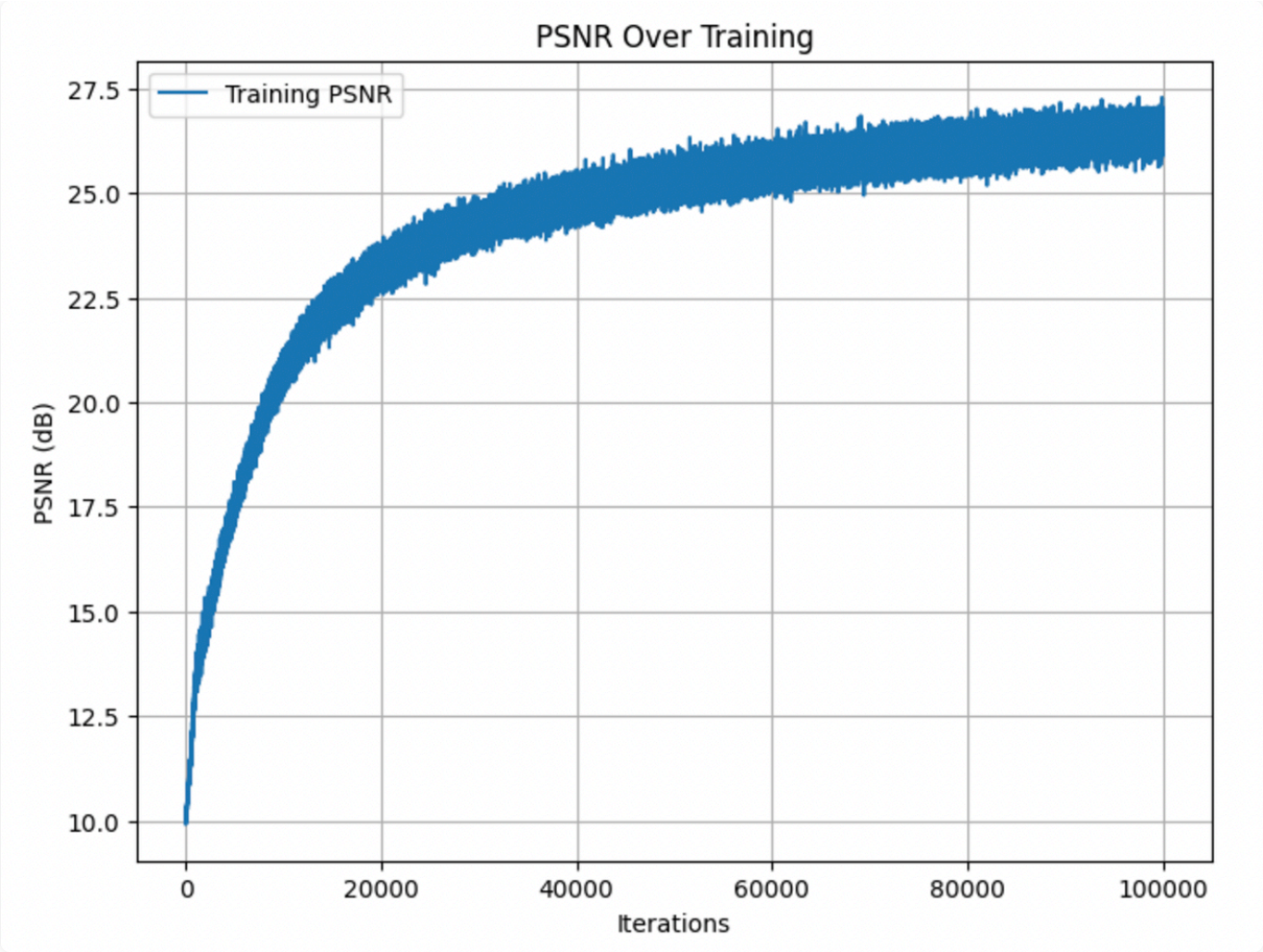

After epoch 601: PSNR=26.03dB


After epoch 701: PSNR=26.20dB

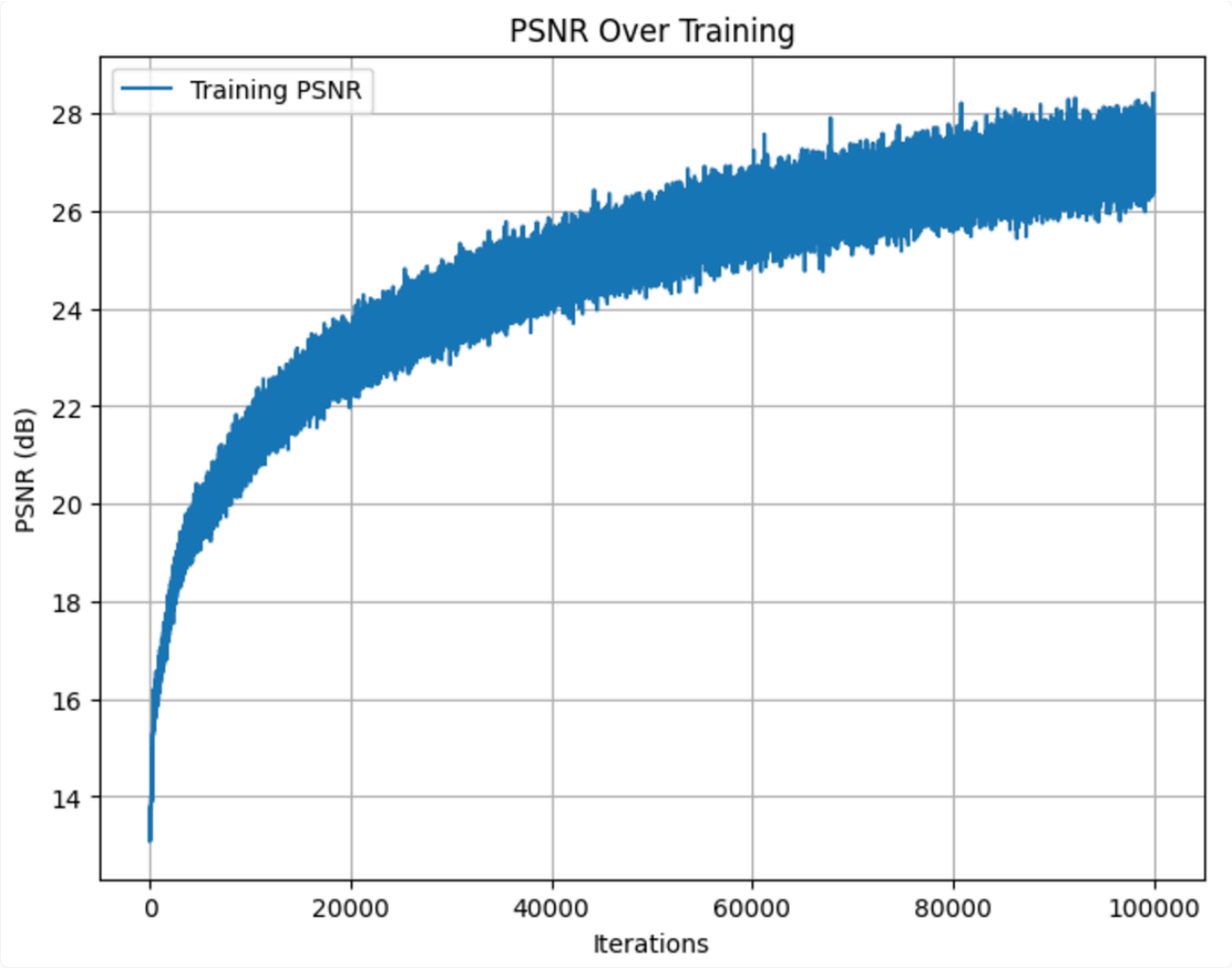




PSNRS


PSNR Over Training
Training PSNR
PSNR (dB)
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
0
20000
40000
60000
80000
100000
Iterations

Fox PSNR



Bird PSNR

Part 2: Fit a Neural Radiance Field from Multi-view Images

Part 2.1: Create Rays From Cameras

For this task, I implemented functions to generate rays from cameras, which is important for implementing Neural Radiance Fields (NeRF). The process involves transforming points between the world space and the camera space, converting pixel coordinates into camera coordinates, and finally calculating ray origins and directions.

First, I created the transform function, which transforms points from the camera space to the

world space using a camera-to-world transformation matrix (c2w). This function supports batched inputs and ensures consistency by performing homogeneous transformations and converting back to Cartesian coordinates. Next, the pixel_to_camera function converts 2D pixel coordinates into 3D camera coordinates by applying the pinhole camera model using the intrinsic matrix K. This step allows us to project a pixel in image space into a 3D point along a ray in camera space, with a fixed depth s.

Finally, the pixel_to_ray function computes the ray origins and directions for each pixel. The origin of each ray is simply the camera's position in world space, obtained by transforming the origin of the camera space (0, 0, 0). The ray direction is computed by projecting a pixel into the world space using the transform and pixel_to_camera functions, and then normalizing the vector pointing from the camera origin to this world-space point.

Part 2.2: Sampling

Part 2.2.1: Sampling Rays From Images

Here, I implemented the sample_random_rays function to randomly sample rays from multiple training images. The function selects N_rays rays by first sampling random image indices and then selecting random pixel coordinates (u, v) within each image. To ensure accurate ray computation, a 0.5 offset is added to the pixel coordinates to account for the pixel center.

Using the selected pixel coordinates and their corresponding camera-to-world transformation matrices (c2ws_train), the function computes the ray origins and directions with the pixel_to_ray function. It also retrieves the RGB color values of the sampled pixels from the training images. This approach efficiently generates ray data for training, allowing the model to optimize over rays sampled from the entire dataset of images.

Part 2.2.2: Sampling Points along Rays

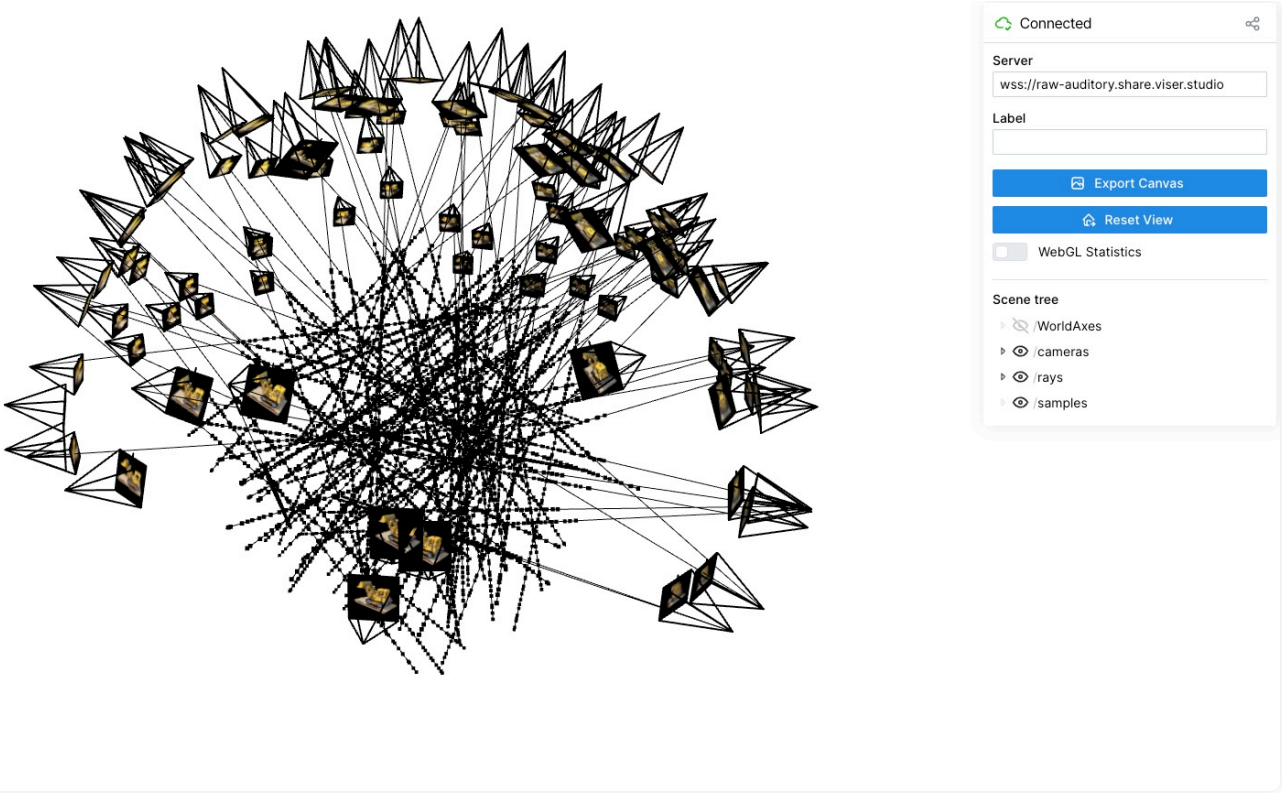
To sample 3D points along rays, I implemented the sample_along_rays function. This function breaks each ray into N_samples uniformly spaced points between a near and far range (default near=2.0, far=6.0). Using torch.linspace, I generate the initial sample positions (t_vals) along each ray. To ensure all regions along the ray are visited during training, I introduce a small random perturbation to these sample positions by shifting them within their respective intervals.

The 3D coordinates of the sampled points are computed using the ray origin (rays_o) and direction (rays_d). By combining the origins and directions with the perturbed t_vals, we create a set of 3D points that are distributed along each ray.

Part 2.3: Putting the Dataloading all Together

For this part, I defined a dataloader to enable sampling of rays from multi-view images, incorporating camera intrinsics and extrinsics. The dataloader randomly samples pixel coordinates from the training images and uses those coordinates to generate ray origins and directions in 3D space. This is done by converting 2D pixel coordinates into camera coordinates and then transforming them into world coordinates using the c2w matrix. The pixel colors corresponding to the sampled coordinates are also retrieved to serve as ground truth supervision during training.

To verify my implementation, I used the viser library to visualize the rays, cameras, and sampled 3D points. First, I plotted the camera frustums for all training images, providing a clear view of their spatial layout. Then, I visualized the sampled rays originating from the cameras and extending into the scene, ensuring they remained within the camera frustum. Finally, the sampled points along the rays were visualized as a 3D point cloud. By combining these visualizations, I was able to confirm that all components—rays, samples, and cameras—were working together as expected.



Part 2.4: Neural Radiance Field

For this step, I implemented another NeRF model to predict the density (sigma) and color of sampled points in 3D space. The model extends the 2D implementation from earlier, incorporating 3D world coordinates and view directions as inputs. These inputs are processed using positional encodings (PE) to capture high-frequency details, with separate encoding layers for the spatial coordinates (L=10) and view directions (L=4).

The network itself is an MLP with eight hidden layers of width 256, including a skip connection at the fourth layer to reintroduce the original positional encodings. This trick ensures the model does not “forget” the input features over the depth of the network. The output is divided into two branches: one predicts the density value (sigma), constrained to non-negative values using ReLU, and the other predicts a 256-dimensional feature vector. This feature vector is concatenated with the view direction encodings and passed through another smaller MLP to produce RGB colors in the range [0, 1], using a Sigmoid activation function.

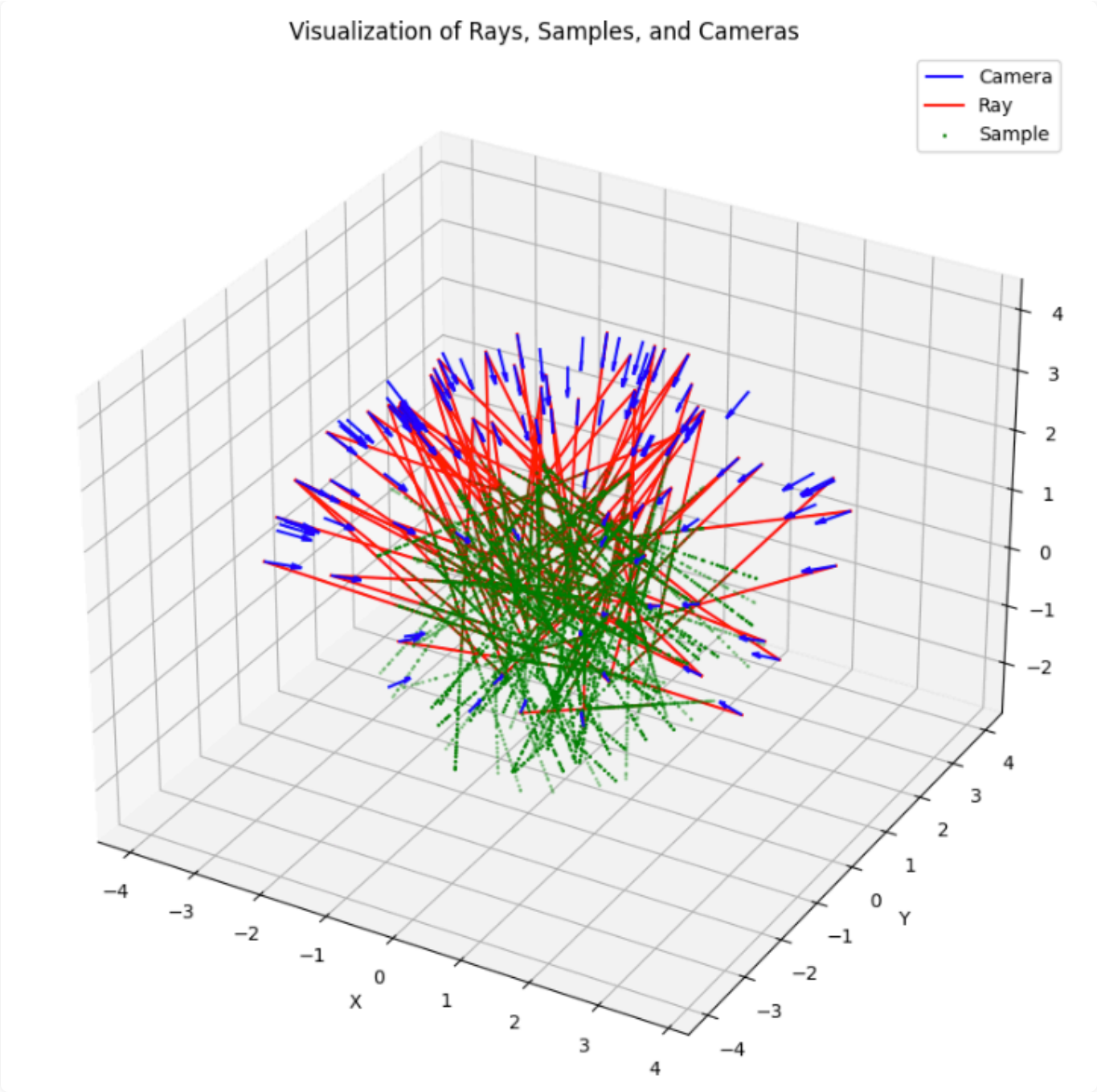
Part 2.5: Volume Rendering

For the volume rendering implementation, I developed a function that computes the final rendered colors for a batch of rays passing through a scene, based on the discrete approximation of the volume rendering equation. The function takes as input the densities, colors, and the step size between sampled points along the rays, and it outputs the final colors observed at each ray. This involves calculating weights for each sample along the ray, which are determined by the alpha compositing values and accumulated transmittance, and blending these weighted colors with an optional background color.

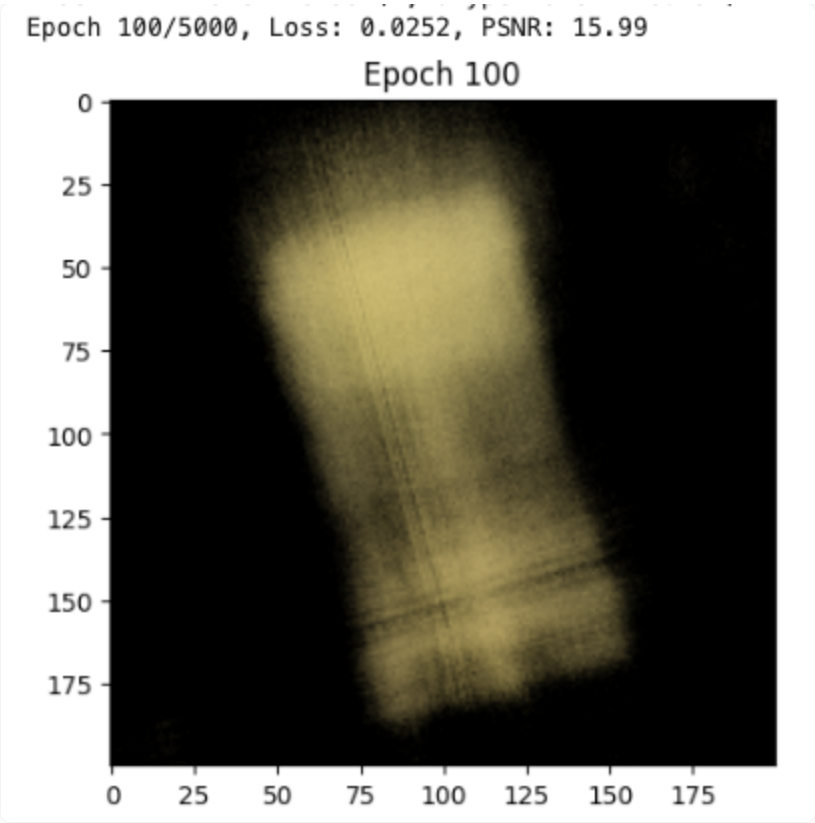
The core process involves three steps: computing alpha values from the density values, determining transmittance via cumulative products of the (1-alpha) values, and using these to compute weights for each sample. These weights are then applied to the colors predicted by the network to generate the final rendered image. An additional blending step incorporates the contribution of the background color based on the accumulated transmittance at the farthest sample.

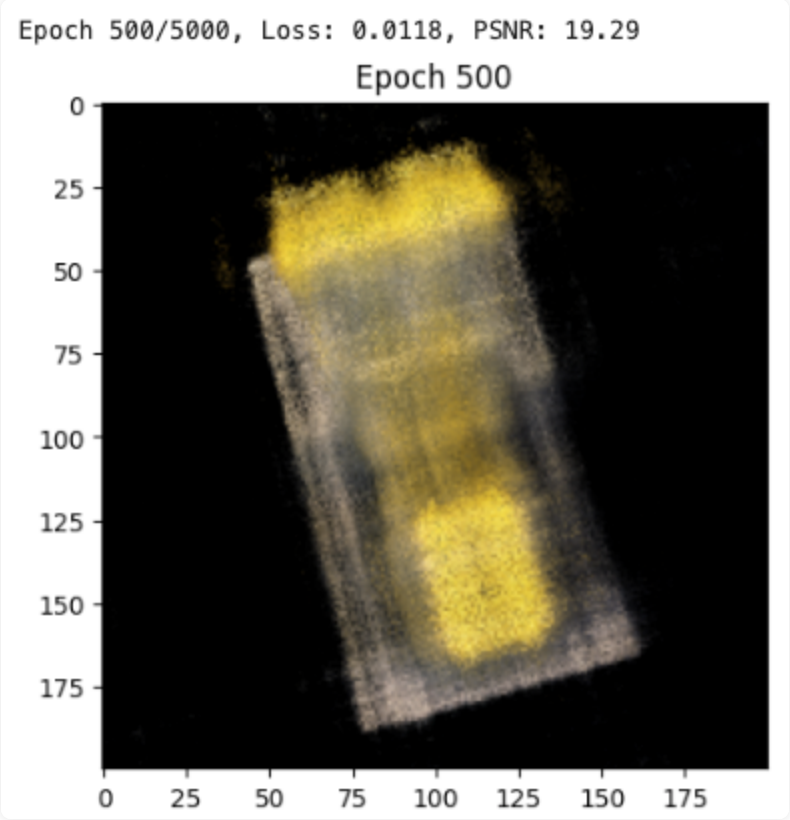
I extended this approach to create a rendering pipeline that visualizes novel-view animations. By sampling points along rays from multiple test camera poses and passing them through the trained neural radiance field model, I rendered images frame by frame. I then stitched these frames together into a smooth animation. For my bells and whistles requirement, I enabled the use of a customizable background color.

Visualization of Rays, Samples, & Cameras at a Training Step

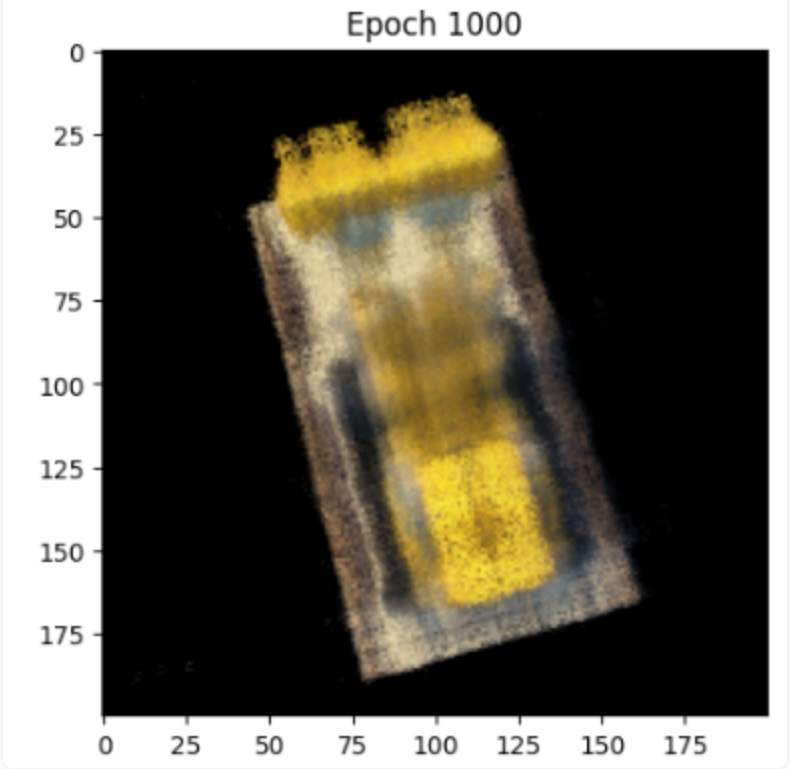


Views Across Epochs

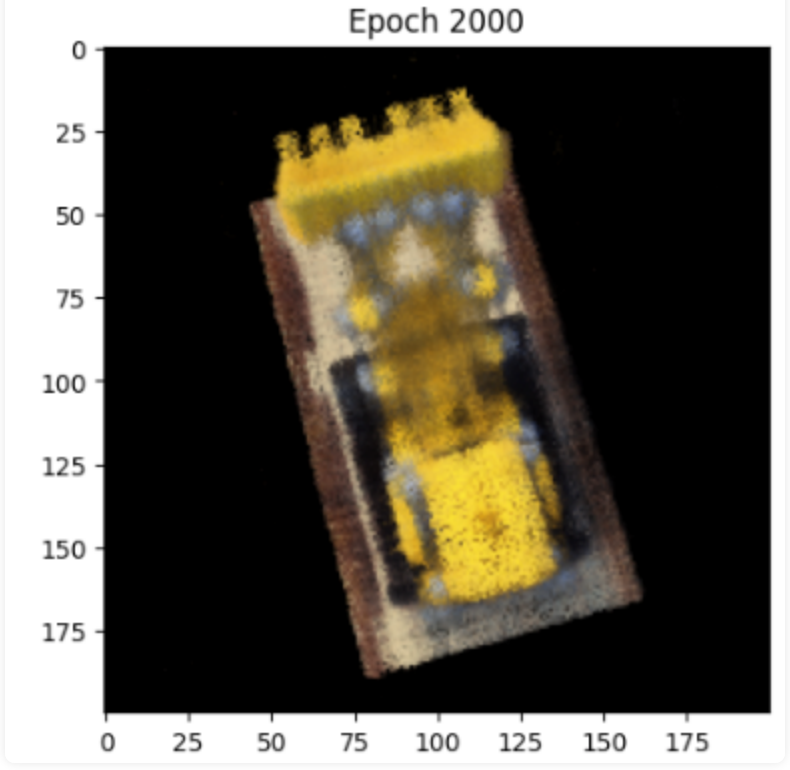

Epoch 100/5000, Loss: 0.0252, PSNR: 15.99
Epoch 100
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175


Epoch 500/5000, Loss: 0.0118, PSNR: 19.29
Epoch 500
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175

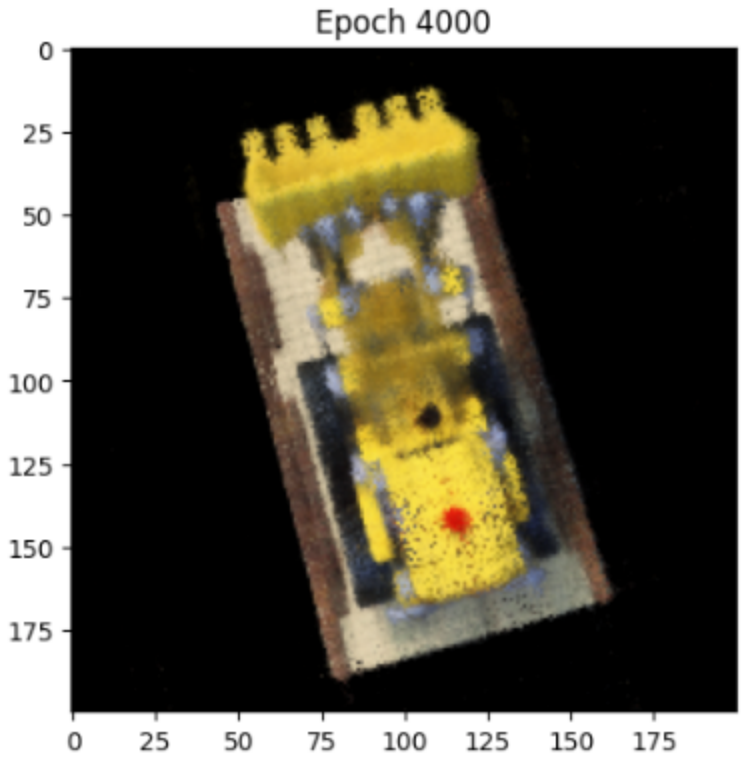
Epoch 1000/5000, Loss: 0.0070, PSNR: 21.56



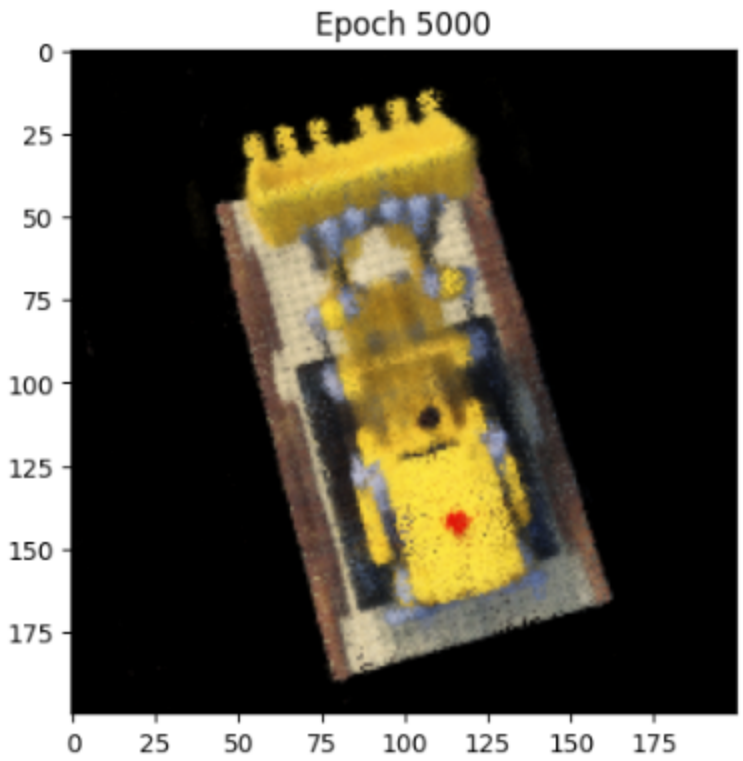
Epoch 2000/5000, Loss: 0.0060, PSNR: 22.19



Epoch 4000/5000, Loss: 0.0040, PSNR: 23.94



Epoch 5000/5000, Loss: 0.0034, PSNR: 24.69



0:00 / 0:02

0:00 / 0:02

Bells & Whistles: Different Background Color