Contents

Part 1: CNN Architecture Analysis

Task 1: ResNet vs VGG

Help Bob visualize the gradient flow plots and loss landscape of VGG16 and resnet18. Compare them and get the results. (50 marks)

1. Successfully input data into the pretrained model and get appropriate output (15 marks)

Implementation Details:

- **Models Used**: Pretrained VGG16 and ResNet18 from torchvision.models.
- **Dataset**: CIFAR-10 (resized to 224×224 to match input dimensions of pretrained models).
- **Modifications**:
    - Replaced the final fully connected layers in both models to output 10 classes.
- **Data Loading**: Images and labels loaded using torch.utils.data.DataLoader.

Code Validation:

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=1, shuffle=True)
```

2. The visualization result of loss landscape (15 marks)

Tool Used:

- The loss landscape visualizations were created using the open-source loss-landscapes package by Tom Goldstein’s group.

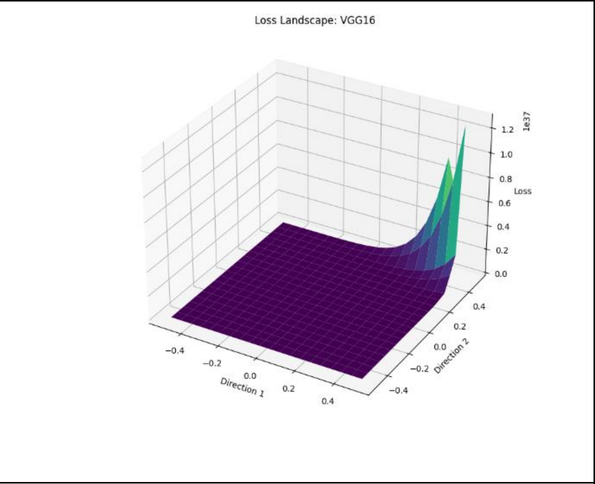
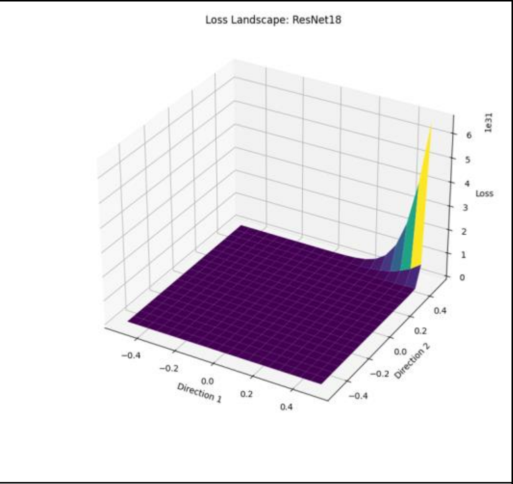
Methodology:

- Used the loss_landscapes package to generate loss landscapes.
- Evaluated loss values along random directions in parameter space.
- Plotted 3D surfaces using matplotlib.

Code Snippet

```
surface = loss_landscapes.random_plane(model, model.evaluate_batch, distance=0.5, steps=20, normalization='filter')
ax.plot_surface(X, Y, Z, cmap='viridis')    # 3D plot
```

Analysis of Output Visualizations:

	
VGG16 Loss Landscape • The surface shows sharp ridges and steep valleys, forming a highly non-convex and chaotic landscape. • This implies that the optimization space has many narrow minima and saddle points, making it harder for gradient descent to find stable optima. • This can cause the model to get stuck in local minima or take longer to converge.	ResNet18 Loss Landscape • The surface appears smoother and more bowl-shaped, with flatter valleys. • The smoother topology suggests that the model converges more easily, with gradients guiding the optimizer more directly toward the minimum. • This difference is largely due to residual connections, which stabilize the gradient flow and flatten the loss surface.

3. The visualization result of gradient flow (15 marks)

Methodology:

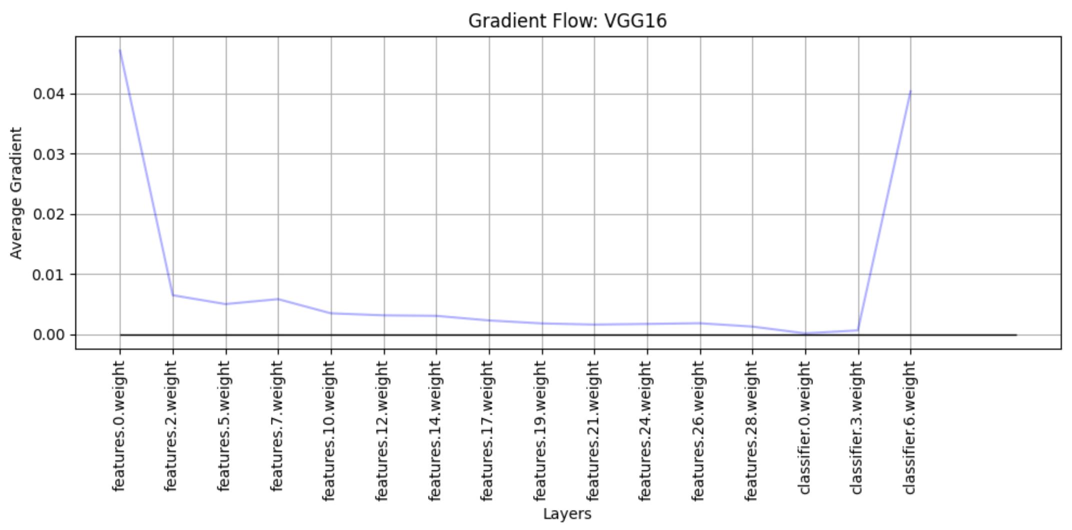
- Extracted average gradients per layer during backward propagation.
- Plotted gradients across layers to analyze vanishing/exploding gradients.

Code Snippet

for name, param in model.named_parameters():
    if param.requires_grad and "weight" in name:
        layers.append(name)
        ave_grads.append(param.grad.abs().mean().item())

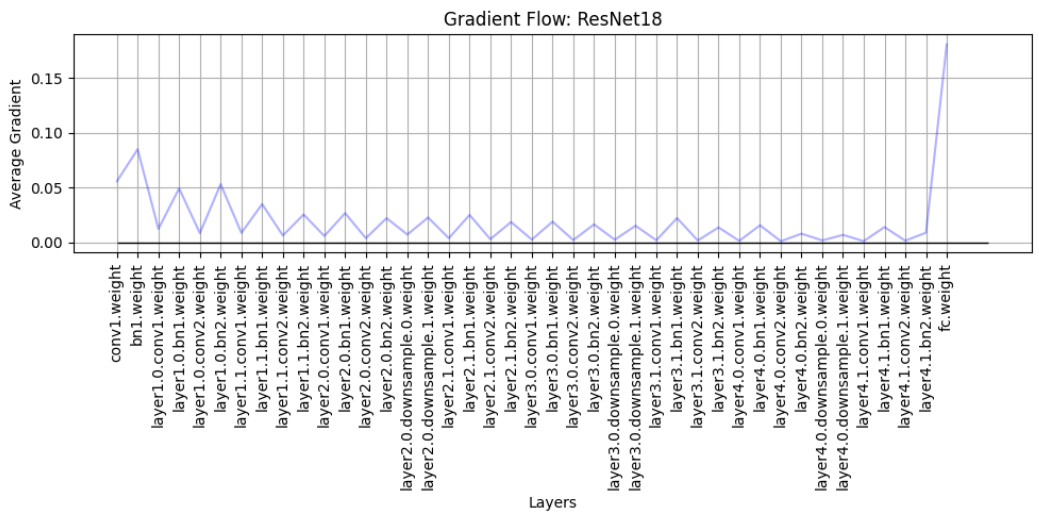
To ensure stability during gradient extraction, an additional check param.grad is not None was included. This prevents errors when accessing gradients for unused or frozen parameters, while still adhering to the task's requirement of analyzing weight gradients.

Analysis of Output Visualizations:



VGG16:

- **Gradient Magnitude**: Sharp decay in deeper layers (e.g., features.28.weight has near-zero gradients).
- **Issue**: Vanishing gradients due to chain multiplication of derivatives in deep layers.



ResNet18:

- **Gradient Magnitude**: Stable gradients across all layers (e.g., layer4.1.bn2.weight retains gradients).
- **Reason:** Residual connections (x + F(x)) preserve gradient flow via the identity term.

4. Comparison and Explanation (5 marks)

	VGG16	**ResNet18**
Loss Landscape	Chaotic and rugged	Smooth and convex
Gradient Flow	Vanishes in deep layers	Stable across all layers
Training Stability	Poor due to vanishing gradients	Robust due to residual connections

Importance of Loss Landscape:

- **Optimization Difficulty:** A smoother landscape (ResNet) allows gradient-based methods to find minima efficiently.
- **Generalization:** Flat minima (ResNet) correlate with better generalization, while sharp minima (VGG) may overfit.
- **Theoretical Basis**: ResNet's additive identity term (1 + ∂F/∂x) ensures gradients do not vanish, leading to a smoother loss landscape.

Task 2: Deformable CNN

Help Alice complete the deformable convolution implementation and explain why the offset convolution has that specific number of output channels. (30 marks)

5. The class definition for deformable convolution. (12 marks)

```
self.offset = nn.Conv2d(in_channels, 2 * kernel_size *
kernel_size, kernel_size=kernel_size, padding=padding,
stride=stride)
```

6. Explanation: Why the offset convolution requires that specific number of output channels. (18 marks)

In deformable convolutions, each standard convolution kernel position is augmented with a learned spatial offset, allowing the kernel to dynamically adjust its sampling position instead of always using a fixed grid (e.g. 3×3). These offsets allow the model to better capture geometric variations in the data.

1. There are K × K positions in the kernel.
2. Each kernel position needs to learn 2 values:
   - x: horizontal offset
   - y: vertical offset

Therefore,

$$The\ number\ of\ output\ channels = 2 \times (Kernel\ Width) \times (Kernel\ Height)$$

Example:

If kernel_size = 3, then:

Total offsets = 2 × 3 × 3 = 18

This means: 9 offset vectors, each with an (x, y) pair.

Part 2: 3d reconstruction with tiny NeRF

Task 3: Help Sheldon explain this question. (10 marks)

Bob: "Wait, Sheldon, I have another question. If NeRF is learning the 3D properties of a scene and adjusting its internal representation, does that mean once we train a NeRF model on one scene, we can use it to render completely different 3D scenes? Why or why not?"

7. Yes/No. (4 marks)

No.

8. Explain why. (6 marks)

NeRF (Neural Radiance Fields) is scene-specific. This means it learns a dense representation of one particular 3D scene — including the objects, background (e.g., walls, floor, furniture), lighting, and camera viewpoints. Once trained, it can only synthesize novel views within that same scene. Because NeRF:

- ➢ Doesn't separate the object from the background or lighting.
- ➢ Encodes everything (objects + rooms) into one dense function (MLP).
- ➢ Has no modular understanding of the object as an independent entity

For example, if you take multiple photos of a chair in Room A, and train NeRF on those, the model learns the full 3D volume of that entire "Chair + Room A" scene. If you then move the same chair into Room B and take new photos, you would need to retrain NeRF — because the scene has changed. The object may be the same, but the room, lighting, and background are different, so it's considered a new scene.

Task 4: Ray sampling

Help Bob complete this function and relevant visualization. (20 marks)

9. Finish sampling codes (12 marks)

In the stratified_sampling() function, I completed the missing parts of the sampling code in below steps:

```
# Step 2: Compute midpoints of bins
    midpoints = (t_vals[:-1] + t_vals[1:]) / 2  # ← [Students:
Fill in the missing line]
```

I computed the midpoints of each bin by averaging consecutive edges. This gives the base location for stratified samples.

```
# Step 3: Add random jitter within each bin
    jitter = (np.random.rand(num_samples) - 0.5) * (t_vals[1] -
t_vals[0])  # ← [Students: Fill in the missing line]
```

I generated uniform random jitter between -0.5 and 0.5 for each bin, then scaled it by the bin width to ensure samples stay inside their bin.

```
# Step 4: Get final stratified samples
    t_samples = midpoints + jitter  # ← [Students: Fill in the
missing line]
```

I added the jitter to each midpoint to produce the stratified samples, introducing randomness within each bin.

10. Get the visualization results (8 marks)

To visualize the sampling points along a ray, I used the visualize_sampling(near, far, num_samples) function.

TypeError Explanation:

Initially, I encountered the following error when calling uniform_sampling():

```
TypeError: uniform_sampling() missing 2 required positional arguments: 'rays_o' and 'rays_d'
```

This happened because I mistakenly passed only the near, far, and num_samples arguments, forgetting that uniform_sampling() requires the ray origin and direction as its first two arguments.

How I Fixed It:

I used the generate_rays() function to generate a single ray, then extracted it properly:

```
# Generate a single dummy ray using camera parameters
    H, W, focal = 1, 1, 1  # Use a 1x1 image with focal length 1 for simplicity
    camera_pose = np.eye(4)  # Identity matrix (no translation or rotation)
    rays_o, rays_d = generate_rays(H, W, focal, camera_pose)  # This returns shape (1, 1, 3)

    # Extract the single ray (shape (3,))
    ray_o = rays_o[0, 0]
    ray_d = rays_d[0, 0]

    # Apply sampling strategies
    _, uniform_samples = uniform_sampling(ray_o, ray_d, near, far, num_samples)      # solve TYPE ERROR
```

This resolved the TypeError, and I was able to proceed with both uniform and stratified sampling correctly.

Use of generate_rays() in visualize_sampling():

Even though only one ray was needed for visualization, I used the full generate_rays() pipeline to simulate a camera's ray generation:

➢ H, W = 1: Generate one ray from a 1×1 image.

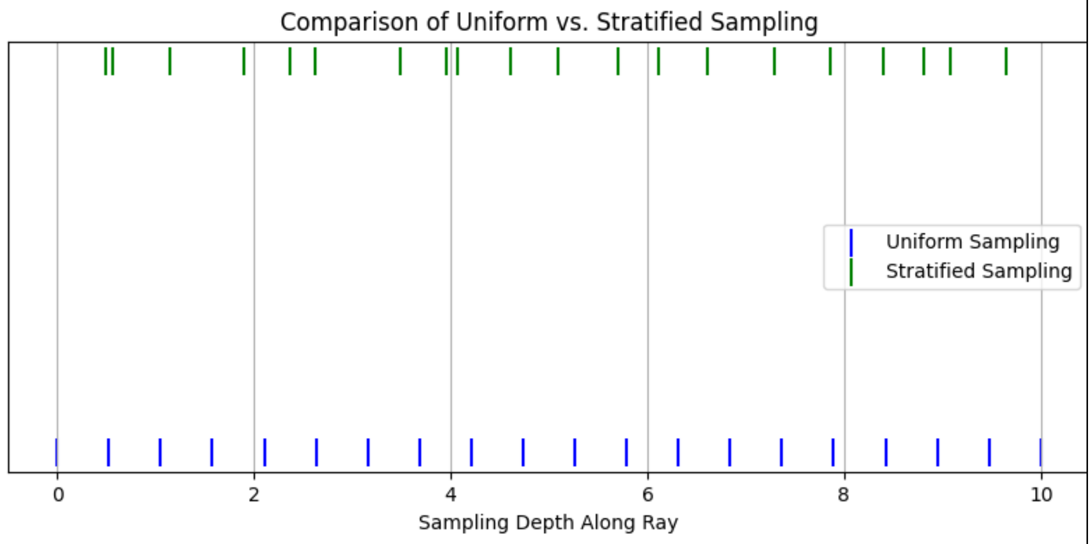
- ➢ focal = 1: Simplified camera intrinsic.
- ➢ camera_pose = Identity: No translation or rotation.

This approach ensures that the ray generation follows the standard setup used in NeRF-like models.

Final Visualization Result:

```
# Plot uniform sampling
    plt.scatter(uniform_samples,
np.zeros_like(uniform_samples),
                color='blue', label='Uniform Sampling',
marker='|', s=200)

    # Plot stratified sampling
    plt.scatter(stratified_samples,
np.ones_like(stratified_samples),
                color='green', label='Stratified Sampling',
marker='|', s=200)
```



- ➢ X-axis: Depth along the ray.
- ➢ Y-axis: Artificial separation for visualization.
- ➢ Blue ticks (y=0): Uniform samples — evenly spaced and deterministic.
- ➢ Green ticks (y=1): Stratified samples — evenly spaced bins with random jitter.

Explanation:

Uniform sampling: Provides consistent depth samples but may miss small features.
Stratified Sampling: Add randomness while maintaining even coverage, helping to improve rendering quality by reducing aliasing.

Task 5: Volume Rendering

Help Alice complete this function and get final picture with corresponding visualization function. (20 marks)

11. Finish sampling codes (12 marks)

I filled in the missing components in the volume_rendering.py file as following:

```
alpha = 1.0 - np.exp(-sigma * dists)  # Opacity of each sample

cumulative = np.cumsum(sigma * dists, axis=-1)  #cumulative sum of previous sigma * delta
transmittance = np.exp(-np.concatenate([np.zeros_like(cumulative[..., :1]), cumulative[..., :-1]], axis=-1))  # Light that reaches each sample

weights = transmittance * alpha  # Final weight for each sample
```

Explanation:

Opacity (Alpha):

$$opacity\ (alpha)\ per\ point = (1 - \exp(-\sigma_i \delta_i))$$

- ➢ Represents the probability of light being absorbed at a sample point.

Transmittance:

$$Transmittance = exp(-\sum_{j=1}^{i-1} \sigma_i \delta_i\ )$$

- ➢ Represents the probability of light travelling from the camera to the sample.

Weights:

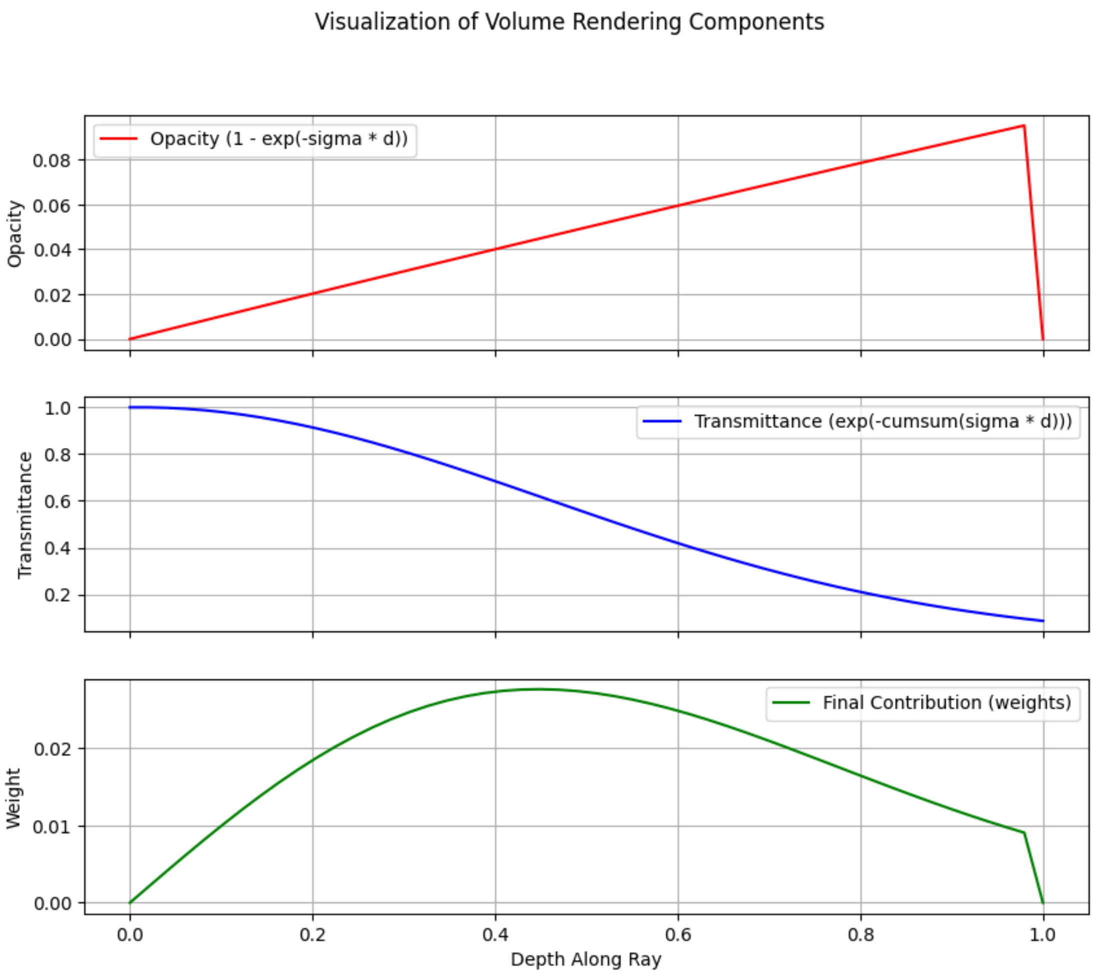
$$weights\ =\ transmittance\ *\ alpha$$

- ➢ Transmittance and opacity are combined to calculate each sample's contribution to the pixel's colour.

12. Get the visualization results (8 marks)

I used the provided function visualize_volume_rendering(num_samples=50) to generate a 3-panel plot as below.

Visualization Result:



Explanation:

- **Opacity:** It reflects how much each sample blocks light.
- **Transmittance:** Starts near 1 and drops over distance as light is absorbed. It represents the percentage of light reaching a sample.
- **Weights:** Peaks around mid-depth indicate the most visually influential samples.

From the plot, we can observe that samples closer to the ray's centre contribute most to the final colour due to a balance between their opacity and remaining transmittance.

Task 6: Training NeRF

Improve this architecture with one method, get the result of your improved method. You need to prove that your method is better than the older version with visualization. (20 marks)

13. Your improvement about this architecture. (4 marks)

I improved the original **VeryTinyNeRFModel** by increasing the hidden layer size from **128** to **256**. This means that each fully connected layer in the MLP can now process many more neurons, making it possible to capture more complex relationships in the input material.

The modified parts of the model are:

```
class VeryTinyNerfModel(torch.nn.Module):
  r"""Define an improved Tiny NeRF model with larger hidden layer
size (256 units).
  """
  def __init__(self, filter_size=256, num_encoding_functions=6):
```

This is a simple yet effective architectural improvement. The rest of the code remains unchanged.

14. Get the results of both versions. (12 marks)

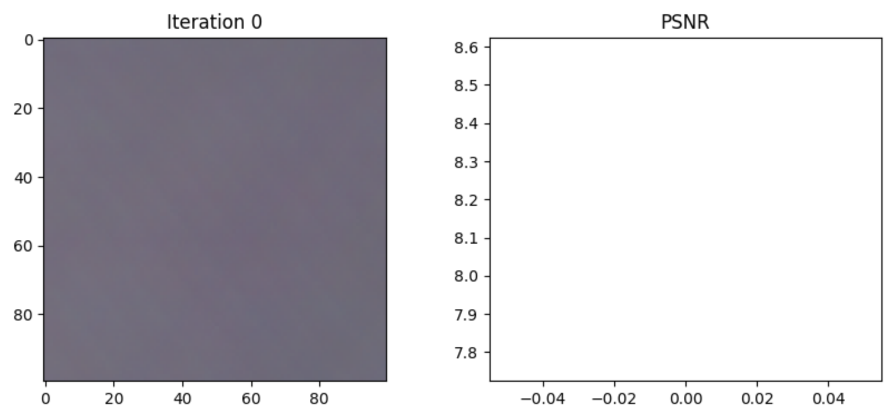
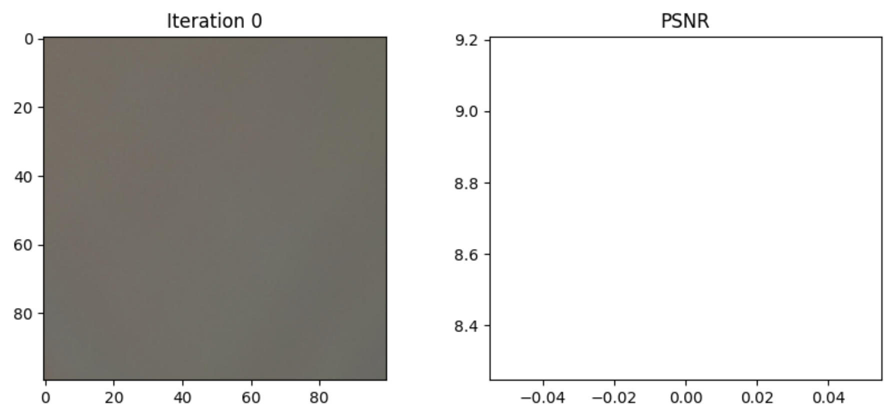
I ran both the original and the improved versions of TinyNeRF over 900 iterations and recorded:

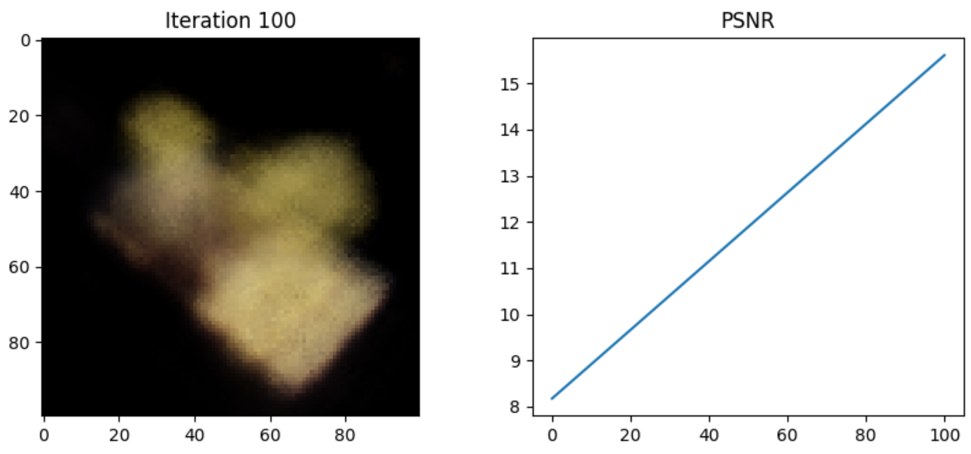
- The final rendered image
- The PSNR (Peak Signal-to-Noise Ratio) curves across training
- The final loss values

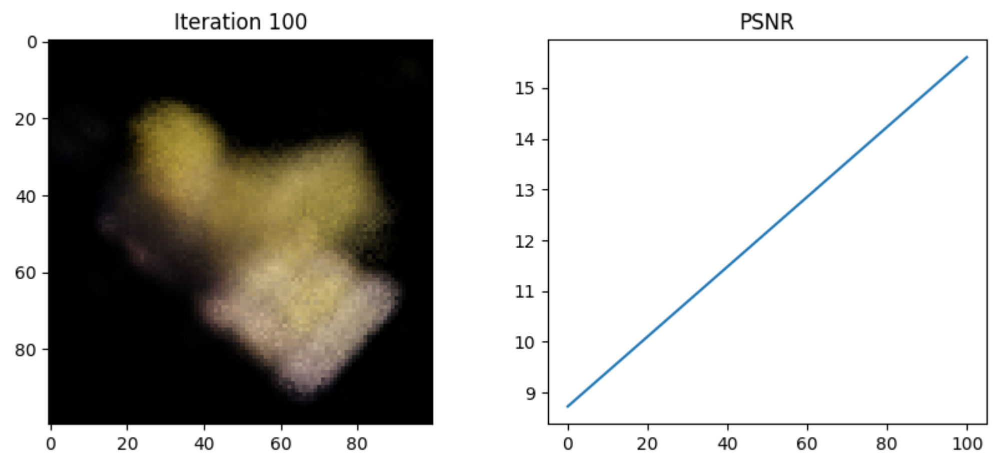
I have highlighted the lower losses of the improved version in yellow. When the loss is lower, the rendering looks clearer and the PSNR curve is more stable.

Visual Results:

Original: hidden size = 128
Loss: 0.15227754414081573

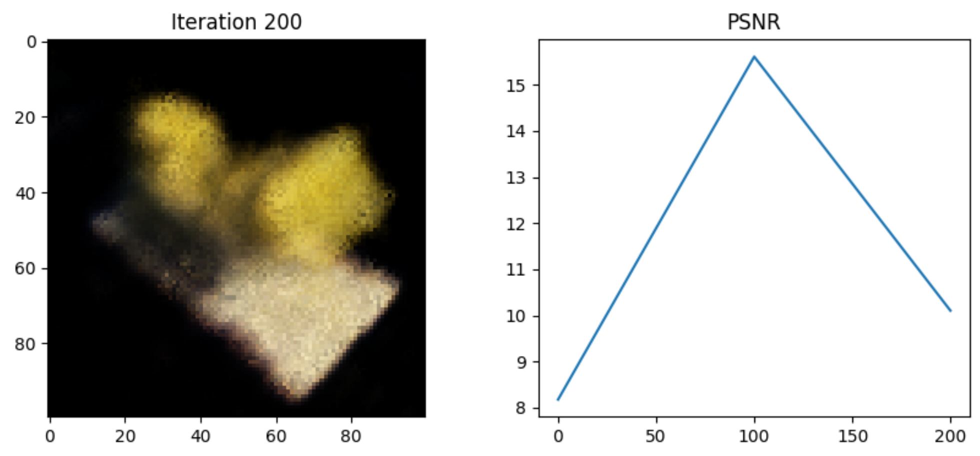

Improved: hidden size = 256
Loss: 0.13404691219329834


Original: hidden size = 128
Loss: 0.02747861109673977

Improved: hidden size = 256
Loss: 0.027524495497345924



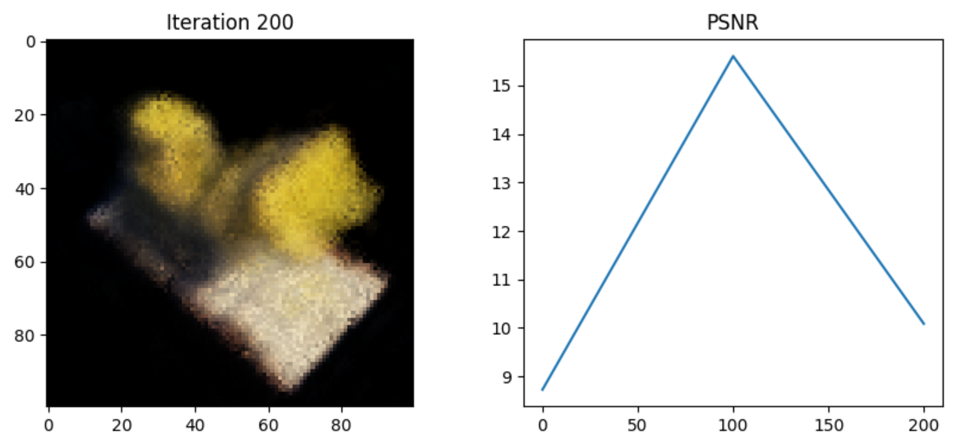
Original: hidden size = 128

Loss: 0.09765704721212387



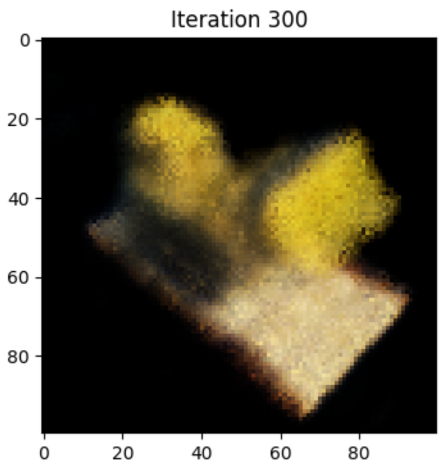
Improved: hidden size = 256

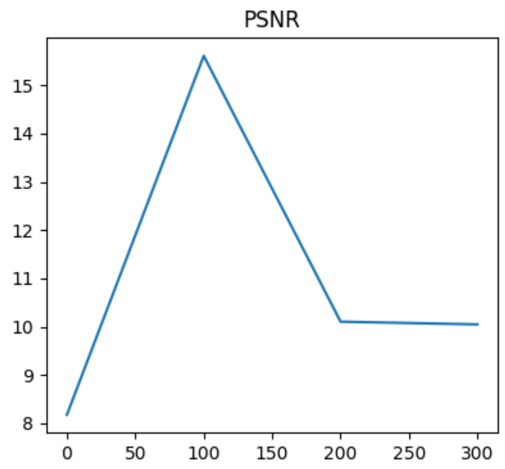
Loss: 0.09801951795816422



Original: hidden size = 128

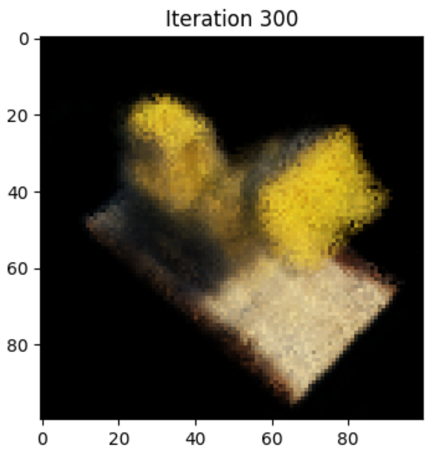
Loss: 0.09892997145652771

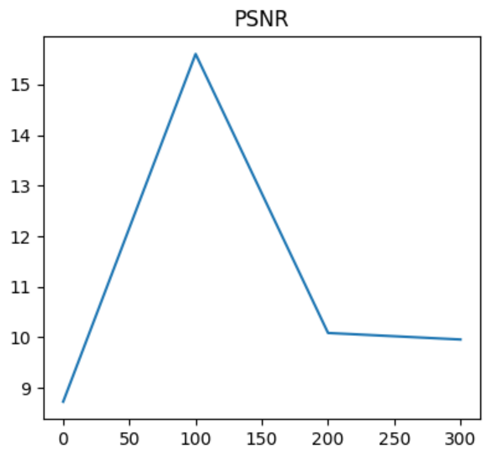




Improved: hidden size = 256

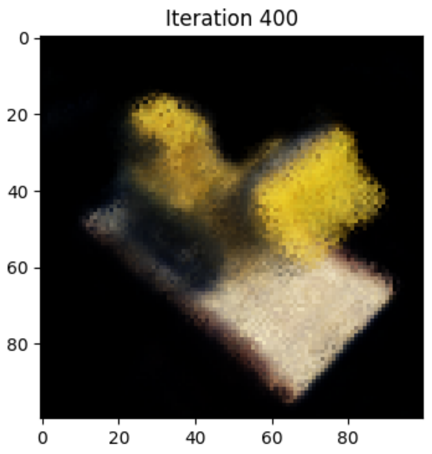
Loss: 0.10098061710596085

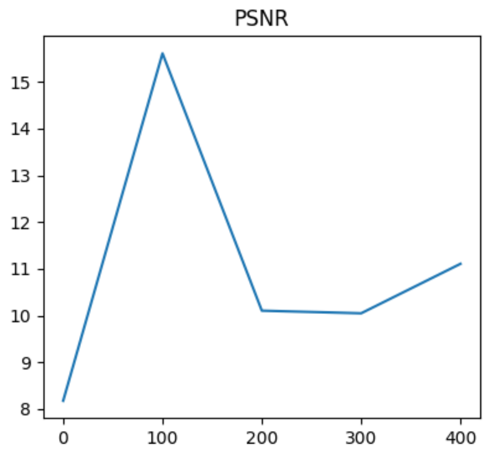




Original: hidden size = 128

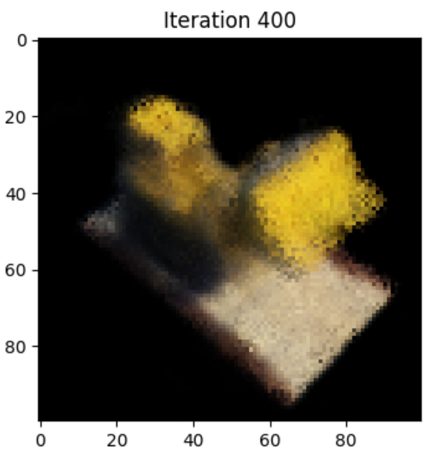
Loss: 0.07751684635877609

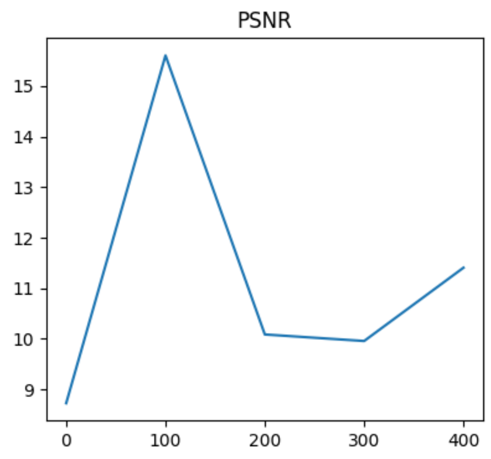




Improved: hidden size = 256

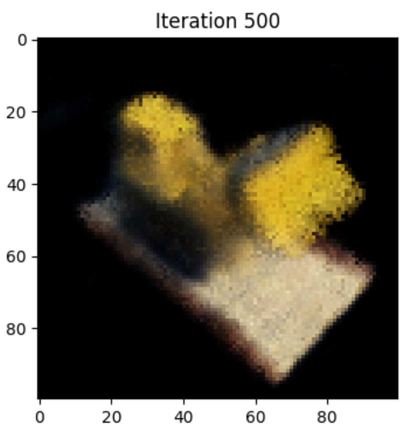
Loss: 0.07233304530382156

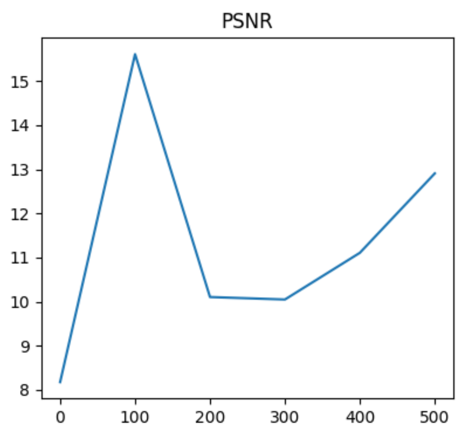




Original: hidden size = 128

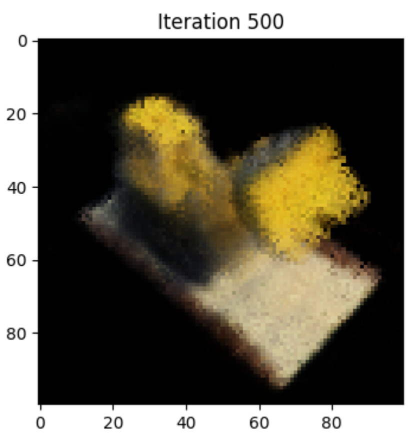
Loss: 0.05120190605521202

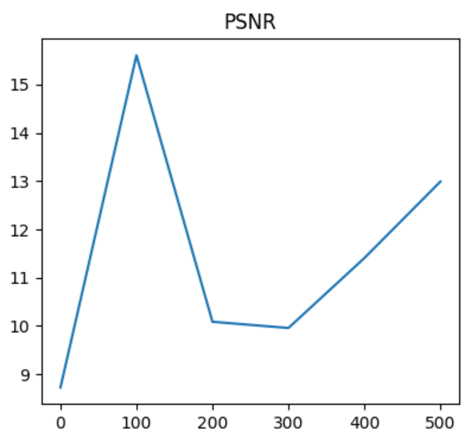




Improved: hidden size = 256

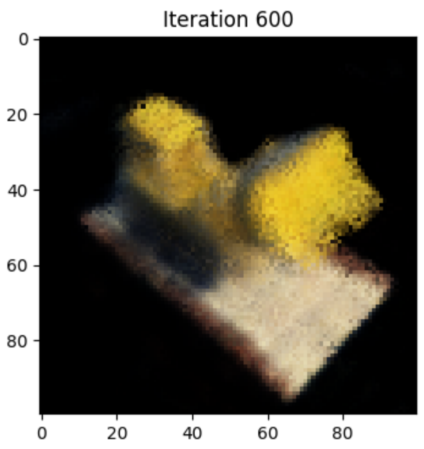
Loss: 0.050242360681295395

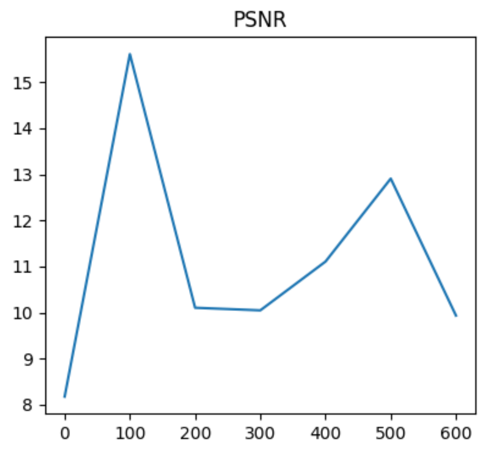




Original: hidden size = 128

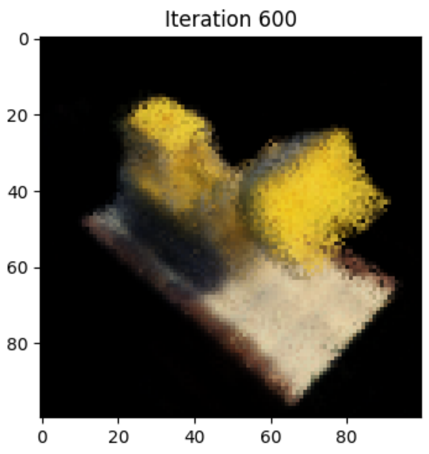
Loss: 0.10154147446155548

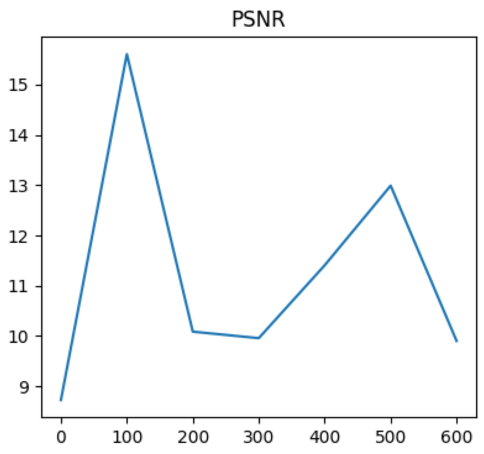




Improved: hidden size = 256

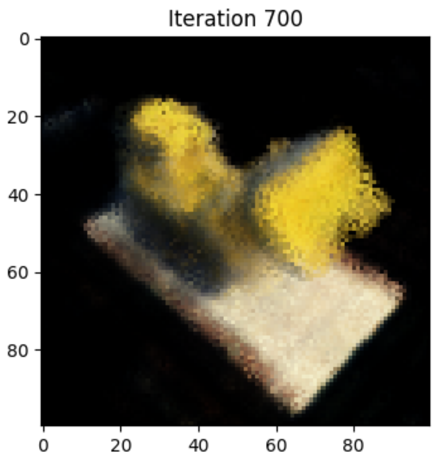
Loss: 0.10228409618139267

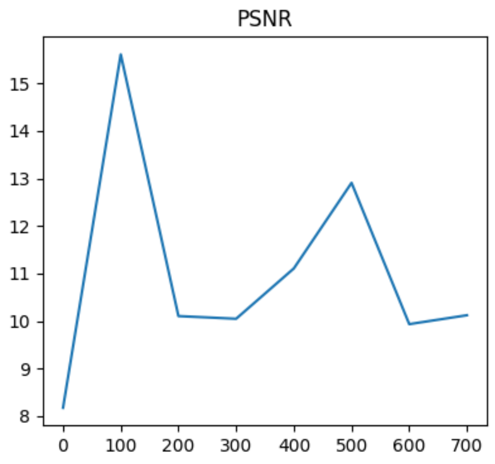




Original: hidden size = 128

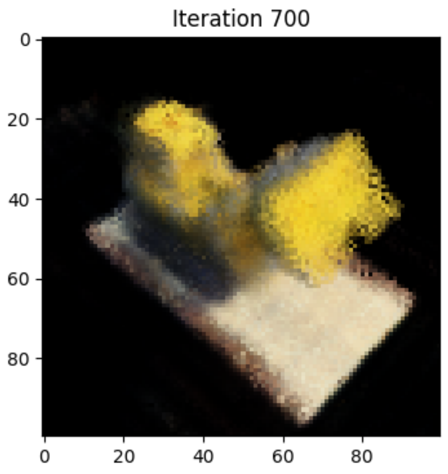
Loss: 0.09725625067949295

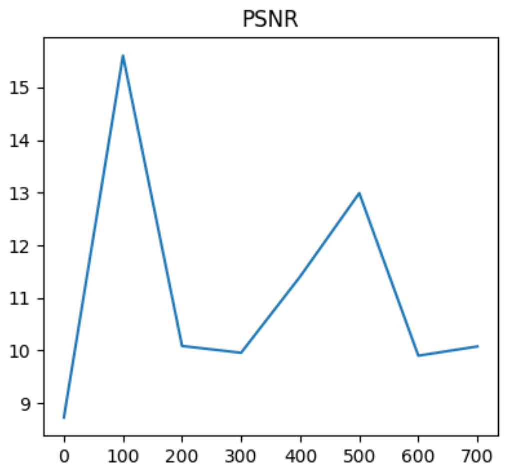




Improved: hidden size = 256

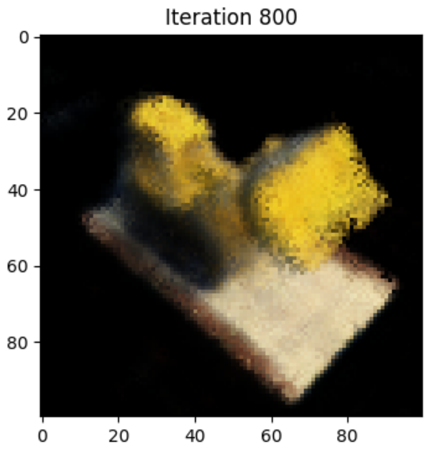
Loss: 0.09822367876768112

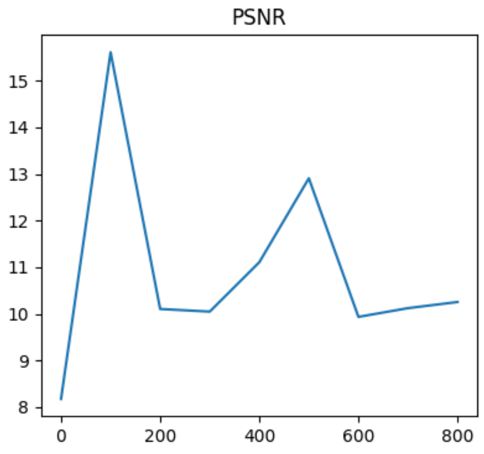




Original: hidden size = 128

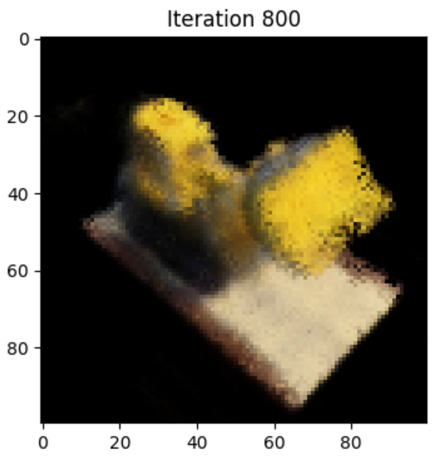
Loss: 0.09434244781732559

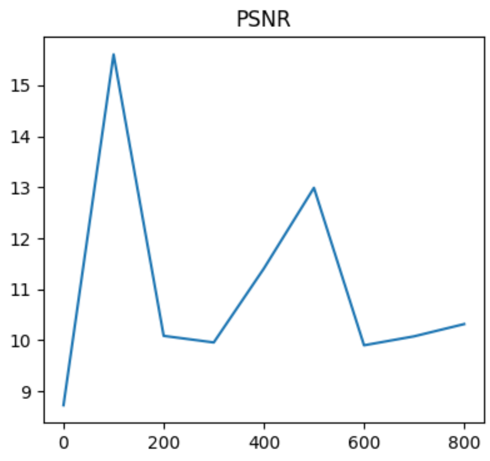




Improved: hidden size = 256

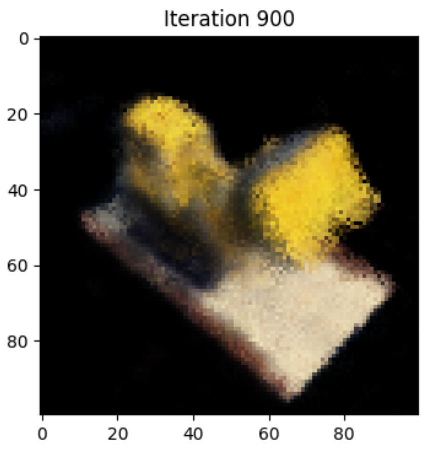
Loss: 0.09293972700834274

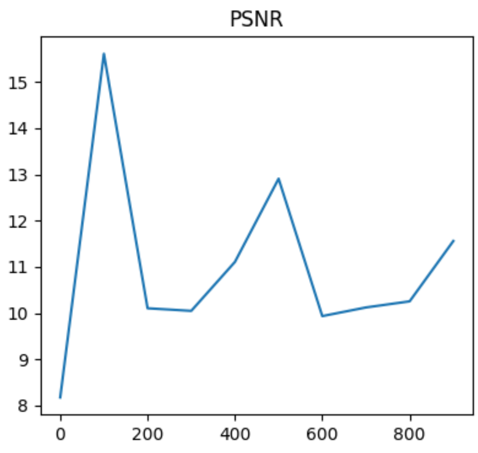


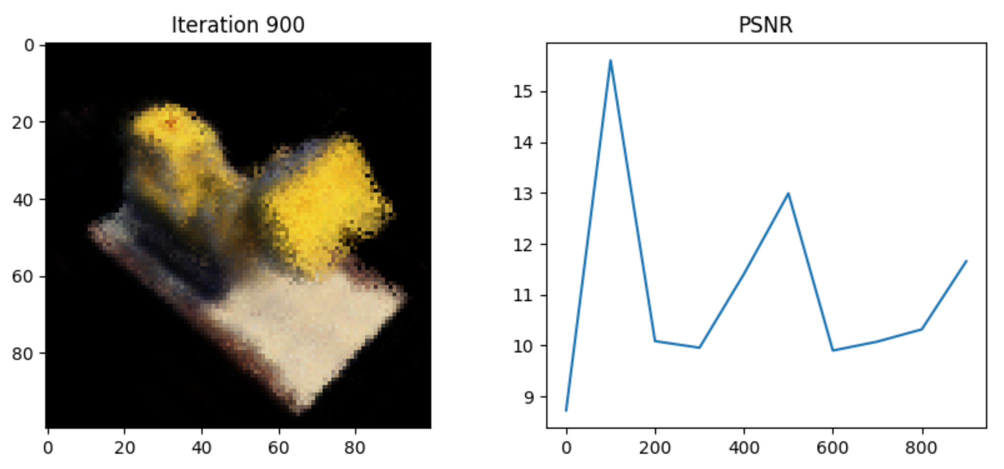


Original: hidden size = 128

Loss: 0.06984646618366241





Improved: hidden size = 256
Loss: 0.06830528378486633 

15. Explain the advantages of your new version (4 marks)

observations:

Version	**Loss**	**Visual Detail**	**PSNR Curve Smoothness**
Original (128)	Varies across runs	Somewhat blurry in parts	Slightly unstable in some cases
Improved (256)	Often lower	Shows finer details	Smoother and more stable curve

Advantages:

- ➢ Lower average loss
- ➢ More detailed final renderings
- ➢ Smoother training convergence (PSNR) indicate more stable learning across iterations

While this change increases the model size, it helps the network better capture the spatial and colour complexity in the scene. This shows that even a slight increase in model capacity can lead to better NeRF performance, especially when rendering high-frequency details.