CSE291 - Assignment 3 (Novel View Synthesis)

Saqib Azim (A59010162)
UC San Diego
sazim@ucsd.edu

Dec 9, 2022

1 Questions

a. What is a radiance field? What information is included in a radiance field? How is neural radiance field (NeRF) different?

- A radiance field is a function that describes how light transport occurs through a 3D volume. It describes the direction of light rays moving through every coordinate $p = (x, y, z) \in \mathbb{R}^3$ in 3D space and in every direction d, described either as angles (θ and ϕ) or as a unit vector $d = [d_x, d_y, d_z] \in \mathbb{R}^3$. Collectively they form a 5D feature space that describes light transport in a 3D scene.

- The neural radiance field (NeRF), inspired by the above representation, is a method for synthesizing novel views of complex scenes. More specifically, it attempts to approximate a function F_θ that maps from this 5D feature space into a 4D space consisting of color $c = (R, G, B)$ values and a volume density $\sigma \in \mathbb{R}_+$ (represents the likelihood that the light ray at this 5D coordinate space is terminated (e.g. by occlusion)). The standard NeRF is thus a function of the form $F_\theta : (p, d) \rightarrow (c, \sigma)$.

b. What is ray marching?

- Ray marching is a class of rendering methods or solvers for 3D computer graphics where rays are traversed in an iterative manner, and each ray is divided into smaller ray segments in order to sample some function at each iterative step. This function can encode volumetric data for volume ray casting, distance fields for intersection finding of surfaces, among various other information.

- Consider a surface defined in terms of a distance field $f(x, y, z) = 0$ such that $\|\nabla f\| = 1$. The function f can also represent negative distances and is referred as a Signed Distance Function (SDF) which means that when $f(x, y, z) > 0$, the point (x, y, z) lies outside the surface and when $f(x, y, z) < 0$, it lies inside the surface.

- The main principle of ray marching is very similar to ray tracing. For each pixel of the screen, we cast a ray spreading from the camera center to the pixel. However instead of computing ray surface intersections by solving an equation, we iterate through the generated ray step by step and check whether we intersect a surface at each step by evaluating the scene SDF at the current location. More precisely, for a given ray $r(t) = r_o + tr_d$, where r_o is the camera center and a given surface, we compute the SDF of the surface $f(r(t))$ and check whether it equals 0. If not, we increase t by a given amount δ_t. We iteratively keep doing it until

Given a radiance field, how is each pixel calculated? (This is called the render equation.) Write down your render equation in a concrete math expression with clarified notations.

- A radiance field represents a 3D scene using the volume density (σ) and directional emitted radiance (RGB) at any point (x, y, z) in the 3D space. The color of any ray passing through the scene can be rendered using classical volume rendering techniques.

- The volume density $\sigma(p)$ at any point $p \in \mathbb{R}^3$ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location p.

- Let's assume that a ray is defined as $r(t) = r_o + tr_d$, where r_o and r_d are the ray origin and directions respectively. The expected color $C(r)$ of a camera ray r is given by -

$$C(r) = \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t), d)dt \quad \text{where} \quad T(t) = \exp\Big(- \int_{t_n}^{t_f} \sigma(r(s))ds\Big) \tag{1}$$

  where t_n and t_f are the near and far distance bounds. The function $T(t)$ is the accumulated transmittance or the probability of ray travelling from t_n to t without hitting any particle.

- For each pixel (i, j) in an image captured from a pose $p \in R^{4\times4}$ in the world, we first estimate the ray (origin and direction) that originates from the 3d scene, enters the camera center and falls upon the pixel (i, j). In order to trace the ray, we assume the ray origin to be the position of camera in world coordinates $p[: 3, 3]$ and the ray direction as the direction of the line joining the pixel to the camera pinhole expressed in world coordinates.

- Once we have ray origin r_o and ray direction r_d for the pixel (i, j), we can parameterize the ray using $r(t) = r_o+tr_d$. To estimate the color at this pixel, we can use the render equation in Eq 1. Practically, it is not possible to calculate the integral as it requires sampling infinite points along the ray $r(t)$. Therefore, numerically, we estimate this continuous integral using stratified sampling approach where we first partition the interval $t \in [t_n, t_f]$ into N equally spaced bins and then sample one point uniformly at random from each bin.

$$t_i \sim U\Big[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{n}(t_f - t_n)\Big] \tag{2}$$

- Once we have discrete sampled points along the ray $r(t_i) = r_o + t_i r_d$, we can evaluate our trained MLP network at these 3D points. The MLP network takes as input a 3d point in space and a ray direction vector. One thing to note is that even though we use discrete set

of samples to estimate the integral, stratified sampling is able to represent a continuous scene because it results in the MLP being evaluated at continuous positions during the training optimization.

- The discretized version of the expected color is given by -

$$\hat{C}(r) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i \delta_i))c_i \quad \text{where} \quad T_i = \exp\Big(-\sum_{j=1}^{i-1} \sigma_j \delta_j\Big) \tag{3}$$

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples. Therefore, for each of the N sampled points along the ray, we feed it into our NeRF MLP network which outputs RGB value c_i and density σ_i. Finally, we use these sample outputs to estimate $C(r)$ for ray r which is also the color value at pixel (i, j).

(c) What is positional encoding? What is the purpose of positional encoding in NeRF models? Train your model without positional encoding and compare the results. You need to show at least two pairs of examples.

- Positional encoding refers to encoding or mapping a set of input coordinates in a continuous space to a higher-dimensional space using high-frequency functions.
- In the NeRF paper, the authors have reformulated the NeRF network F_θ as a composition of two functions $F_\theta = F_\theta^{'}(\gamma(p))$ used the following positional encoding - $\gamma : R \to R^{2L}$

$$\gamma(p) = [\sin(2^0\pi p), \cos(2^0\pi p), \sin(2^1\pi p), \cos(2^1\pi p), ..., \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)] \tag{4}$$

- In the NeRF model, the authors realized that feeding just position (xyz) and direction (θ, ϕ) coordinates to the multilayer perceptron network results in renderings that perform poorly at representing high frequency variation in color and geometry.
- Additionally, deep networks are biased towards learning lower frequency functions. Using positional encoding helps to circumvent the bias that neural networks have towards lower frequency functions, therefore, allowing NeRF to represent sharper (high frequency) details more easily.
- It enables the MLP in the NeRF model to more easily approximate a high frequency function.
- Mapping the inputs to a higher dimensional space using high frequency functions before passing them to the network enables better fitting of data containing high frequency variations, aiding the model to learn better and produce sharper models.
- We trained two models - one with positional encoding and another without positional encoding. Results and visualization can be found in experiment section.

(d) Is it possible to extract scene geometry (e.g., depth) information from a trained NeRF model? Describe your method in detail. Implement your method and show two depth maps generated by your method.

- Yes, it is possible to extract scene geometry or depth information from a trained NeRF model.
- **Method for extracting scene depth information:** The process of estimating depth is very similar to that of computing RGB values for each pixel as described in question 2. For each pixel (i,j), we first compute the ray origin and ray direction wrt world coordinates. This gives us the ray from the scene to the pixel (i,j). Along this ray, we sample N uniformly random points. Then we feed these N ray positions along with the corresponding ray direction to the MLP network which output directional emitted radiance (RGB) value and volume density (σ). Using these N set of RGB values and density values, we compute the weights and multiply these by corresponding t_i values along the ray. Finally, we take the sum to of all these values to get depth at pixel (i,j).

(e) What are the major issues you find when using NeRF? List at least 2 drawbacks. For each of them, propose a possible improvement. You are encouraged to check follow-up papers of NeRF, but you should cite these works if borrowing their ideas.

Major Issues using NeRF (my observations): These issues are my observations. Some of them may not be find these as issues or problems.

- Training is slow and time-consuming and compute-intensive. As we increase training image size, the training becomes extremely difficult and compute-intensive. Hence, I had to train on 200x200 images instead of original 800x800 images.
- Rendering / Synthesizing novel images is also slow and time-consuming. In addition, rendered images often have blurry effects and visual artifacts.
- The NeRF model is inflexible. That is, one model trained on one scene cannot be used for other scenes. For each scene in the 3d space, it requires training separate networks. The training time is roughly 15-20 hrs on medium sized images (400x400) to achieve convergence. This seems a very daunting and time-consuming task to learn NeRF models for large number of different scenes.

Potential Improvements:

- Neural Sparse Voxel Fields (NSVF) is a neural scene representation that enables fast, high-quality rendering which is not dependent on a specific viewpoint. It works by defining voxel-bounded implicit fields organized in a sparse network of cells, and progressively learns voxel structures in each cell of the network. It can render new views much faster by skipping voxels with no scene content. This technique makes NSVF over ten times faster than the original NeRF.
- KiloNeRF addresses the problem of slow rendering in NeRF, mainly related to the need to query a deep MLP network millions of times. KiloNeRF separates the workload among thousands of small MLPs, instead of one large MLP which needs to be queried many times. Each small MLP represents part of a scene, enabling 3X performance improvement with lower storage requirements, and comparable visual quality.

- Mip-NerF extends the original NeRF model with the objective of reducing blurring effects and visual artifacts. NeRF uses a single ray per pixel, which often causes blurring or aliasing at different resolutions. Mip-NeRF uses a geometric shape known as a conical frustum to render each pixel, instead of a ray, which reduces aliasing and makes it possible to show fine details in an image, and reduces error rates by 17-60% while increasing rendering speed by 7% as compare to NeRF.

2 Report

2.1 Model details and Explanations:

I have used an MLP network to approximate a function $F_\theta : (x, y, z, \theta, \phi) \rightarrow (c, \sigma)$ to represent a continuous scene representation. The model architecture consists of a simple N layered ($N = 4, 8$) fully connected neural network with feature dimension (experimented with 128/256). There is a residual connection at layer N/2. The feature obtained after layer N-1 is divided into 2 branches. In the first branch, the feature is passed through a linear layer to produce volume density $\sigma \in \mathbb{R}_+$. In the second branch, the feature map is forwarded through the Nth layer and the output is concatenated with the input direction vector. This combined feature map is passed through a final single layer to produce RGB values. After each MLP layer, we used a ReLU activation function.

2.2 Hyperparameters

- Learning rate: Used a starting learning rate of 0.0005 and exponentially decreased the learning rate at each iteration by a factor of $(0.1)^{i/250000}$, where i is the training iteration. I tried changing initial learning rate to 0.001 and 0.005, but observed that 0.0005 was giving best performance among these.
- Number of training iterations: Trained all models for 200k iterations. In some cases, when the initial results were poor, I only trained those cases for 100k iterations
- Number of MLP layers (N): Used $N = 4$. Initially, I was using 8 MLP layers but observed better PSNR with 4.
- Number of MLP channels/layer: In my initial experiments I was using 256 as mentioned in the paper. Later, reduced it to 128 and observed better results and faster training (win-win)
- Ray batch size: 1032 rays/batch. The paper suggests 4096 rays/batch but due to limited compute capacity I was running into CUDA related error. Hence, I decreased it to 1032.
- Optimizer: Adam
- Number of bins/samples along each ray: 64

- Image size: For training, I resized images by factor 4 to 200×200. For validation also, I am using resized (200×200) images for computing PSNR. For rendering test poses, I am rendering 800×800 sized images
- Training/Validation data size: Used 150 samples for training (all 100 train images + first 50 validation images (0000-0049)) for training. Remaining 50 validation samples (0050-0099) were used for validation during training.
- Positional encoding size: Used L=10 for position coordinates (x, y, z) and L=4 for direction coordinates (d_x, d_y, d_z) (Model-1 & Model-2). Experimented with $L = 5$ and $L = 2$ as well (Model-3).
- Near (t_n) and Far (t_f) bounds: $t_n = 0$ and $t_f = 5.0$.
- Chunk size: Due to RAM and CUDA memory limitation, I had to chunkify the input before feeding it into the network. We used a chunk size of 1024 rays.

The training process and experiments were completed on Google Colab and Jupyter Notebook. Model Checkpoints were saved at every 10k iterations. Validation on 50 samples is performed at every 20k iterations.

report_experiments

December 9, 2022

[36]:
```
import matplotlib.pyplot as plt
from PIL import Image
from IPython.display import Video
```

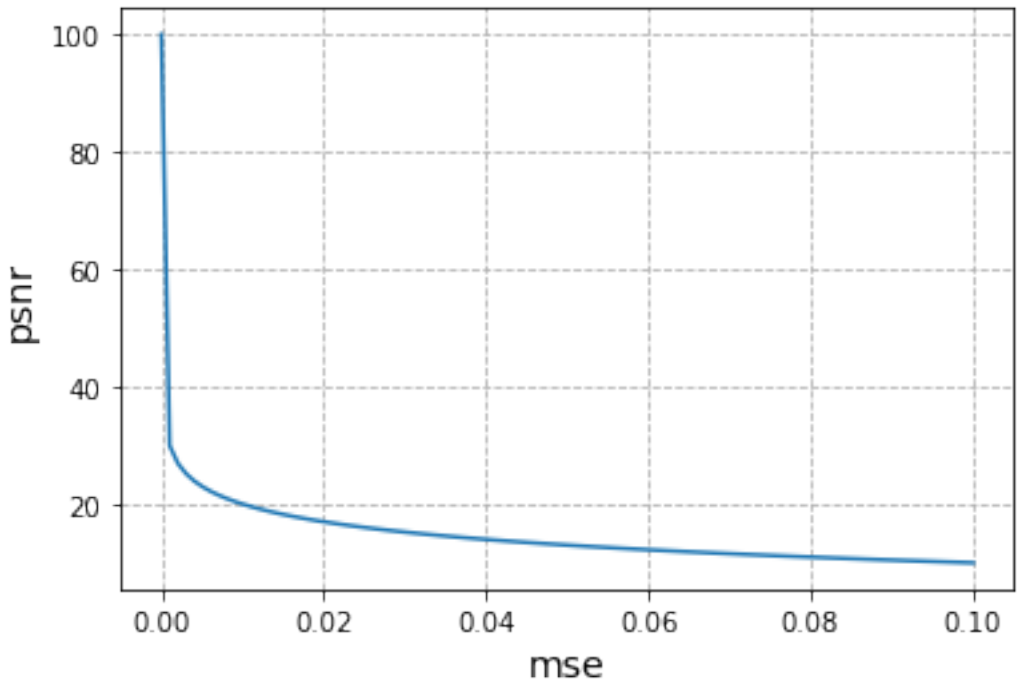
0.0.1 Peak Signal-to-Noise Ratio

$\text{PSNR} = -10 \log_{10}(\text{MSE})$

[35]:
```
mse = np.linspace(0., 0.1, 100)
psnr = -10. * np.log10(mse+1e-10)

fig = plt.figure()
plt.plot(mse, psnr)
plt.grid(linestyle='--')
plt.xlabel("mse", fontsize=14)
plt.ylabel("psnr", fontsize=14)
```

[35]:
```
Text(0, 0.5, 'psnr')
```



1 Experiments

1.1 Experiment - Effect of Positional Encoding

- Trained the network for 200k iterations on 150 training samples (100 train + 50 val) and performed validation on the remaining 50 validation samples.
- Denote: Model-1 with positional encoding and Model-2 without positional encoding.
- Clearly, Model-1 performs better than Model-2 based on the validation set PSNR.
- Model-1 Validation PSNR > 26, Model-2 Validation PSNR > 23

1.2 Model-1 (with positional encoding) & Model-2 (without positional encoding)

1.3 Model-1 vs Model-2 Loss Plot

[57]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd/trainloss_plot.png"))
plt.title("Model-1 Training Loss", fontweight="bold")

fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd_noembed/trainloss_plot.
↪png"))
```

```
plt.title("Model-2 Training Loss", fontweight="bold")
```

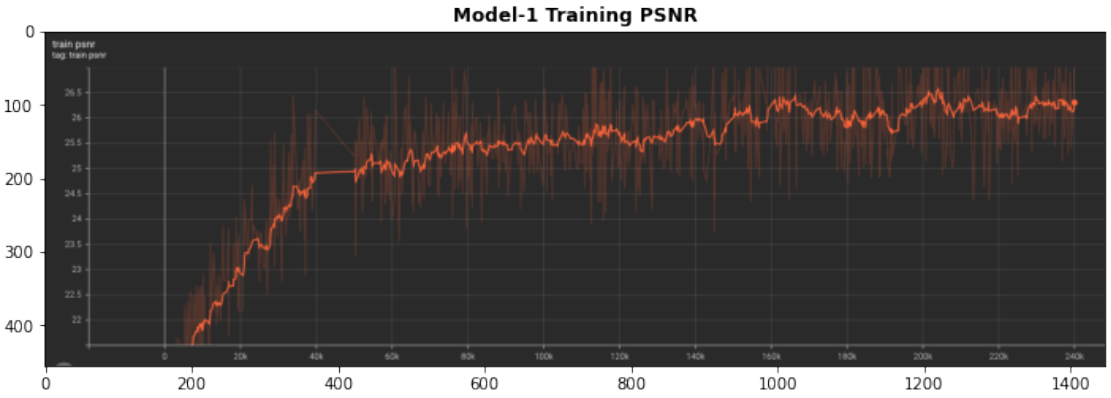
[57]: Text(0.5, 1.0, 'Model-2 Training Loss')

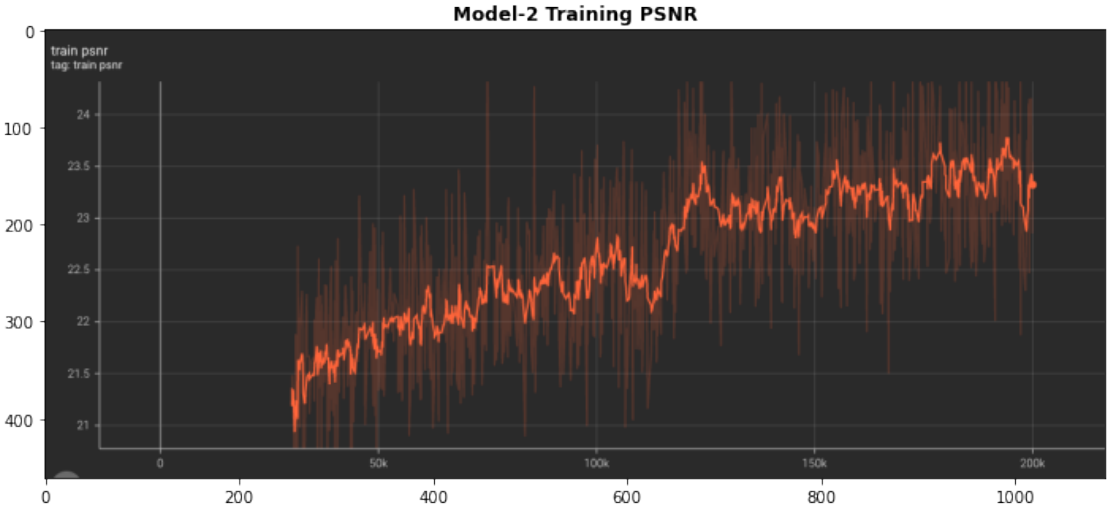
1.4 Model-1 vs Model-2 Training PSNR Plot

[58]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd/train_psnr_plot.png"))
plt.title("Model-1 Training PSNR", fontweight="bold")

fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd_noembed/train_psnr_plot.
 ↪png"))
plt.title("Model-2 Training PSNR", fontweight="bold")
```

[58]:
```
Text(0.5, 1.0, 'Model-2 Training PSNR')
```



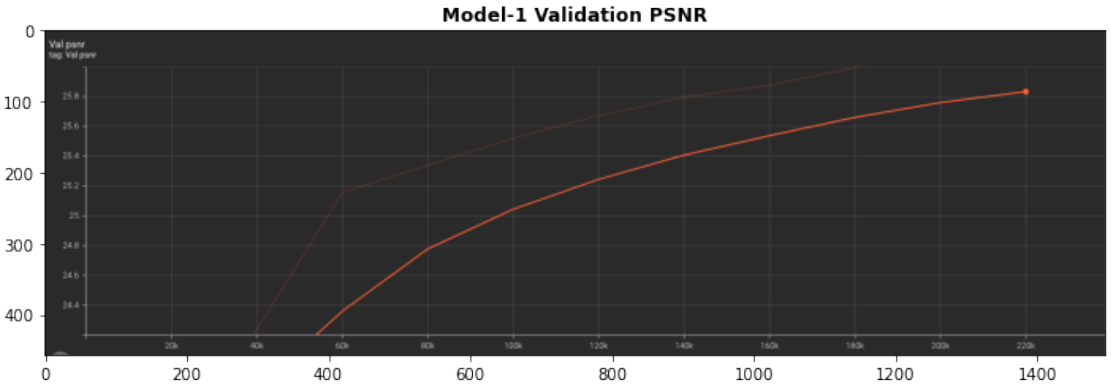


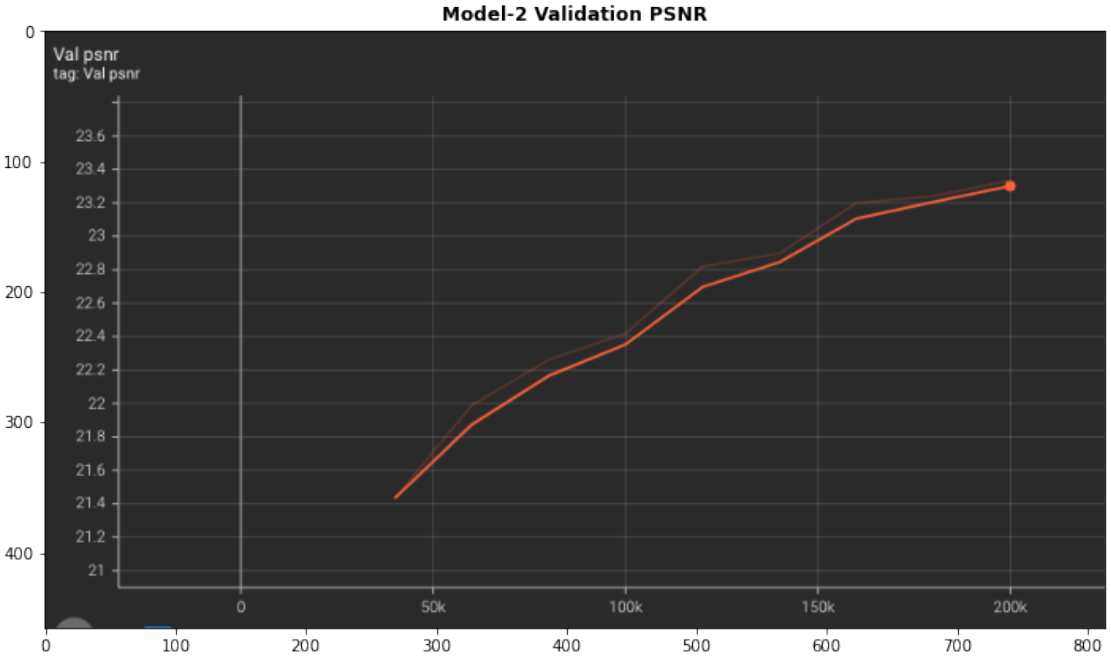
1.5 Model-1 vs Model-2 Validation PSNR Plot

[59]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd/val_psnr_plot.png"))
plt.title("Model-1 Validation PSNR", fontweight="bold")

fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp001_whitebkgd_noembed/val_psnr_plot.
↪png"))
plt.title("Model-2 Validation PSNR", fontweight="bold")
```

[59]:
```
Text(0.5, 1.0, 'Model-2 Validation PSNR')
```



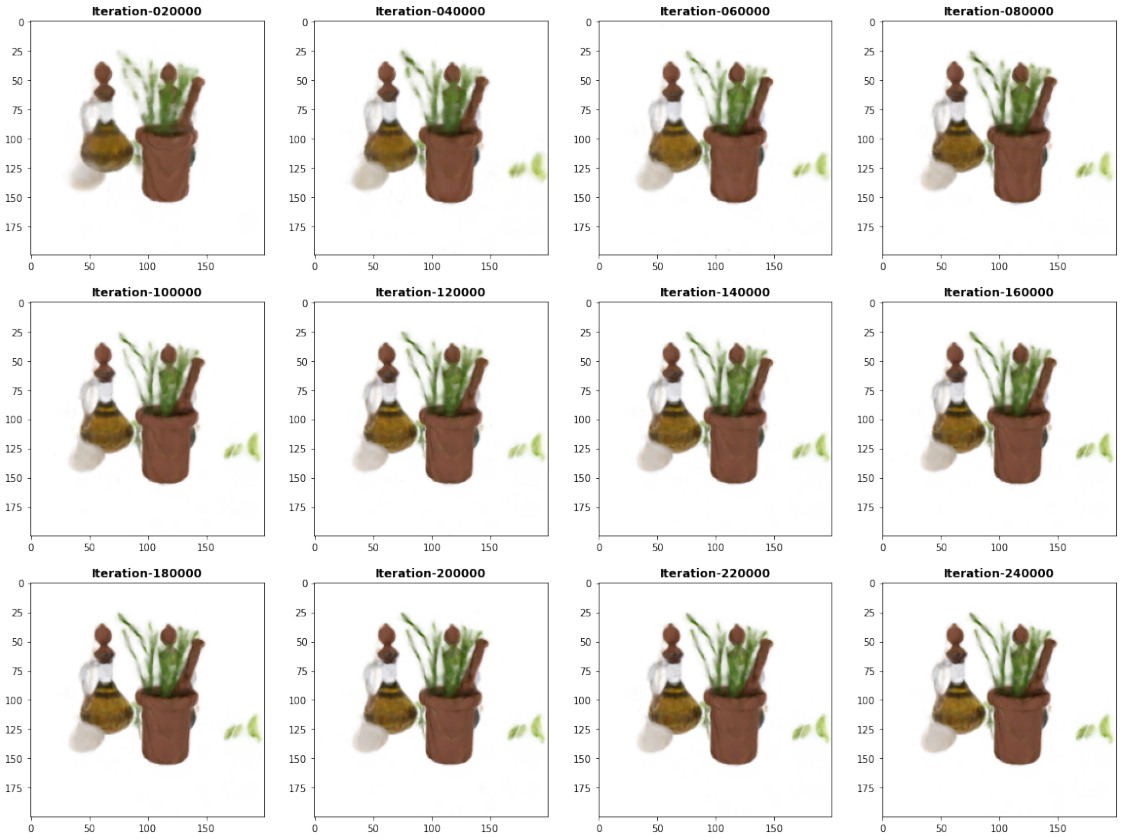


1.6 Model-1: Synthesized Validation Images during training : 1_val_0075.png

[44]:
```
fig = plt.figure(figsize=(20, 15))

file_lst = ["020000", "040000", "060000", "080000", "100000", "120000",␣
 ↪"140000", "160000", "180000", "200000", "220000",
            "240000"]

for i in range(len(file_lst)):
    plt.subplot(3,4,i+1)
    plt.imshow(Image.open("../logdir/finalexp001_whitebkgd/
 ↪valset_"+file_lst[i]+"/025.png"))
    plt.title("Iteration-"+file_lst[i], fontweight="bold")
```

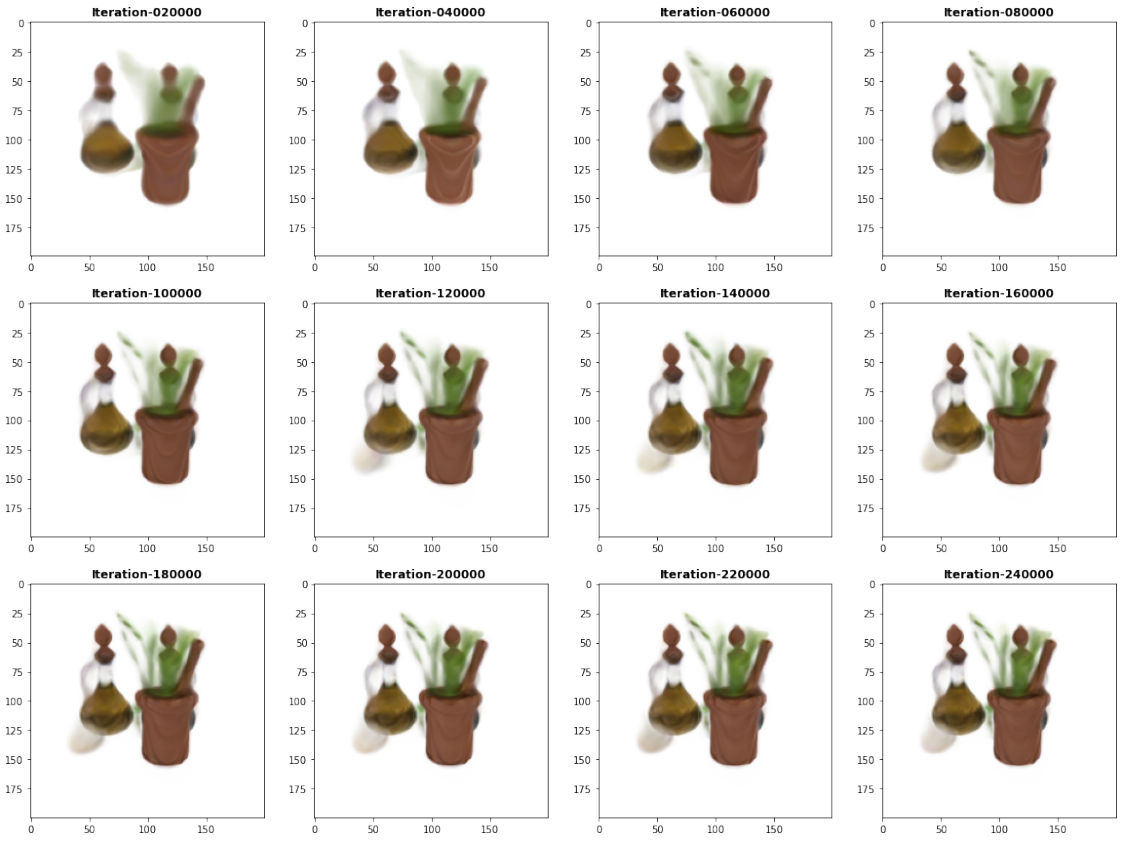


1.7 Model-2: Synthesized Validation Images during training : 1_val_0075.png

[43]:
```
fig = plt.figure(figsize=(20, 15))

file_lst = ["020000", "040000", "060000", "080000", "100000", "120000",␣
 ↪"140000", "160000", "180000", "200000", "220000",
            "240000"]

for i in range(len(file_lst)):
    plt.subplot(3,4,i+1)
    plt.imshow(Image.open("../logdir/finalexp001_whitebkgd_noembed/
 ↪valset_"+file_lst[i]+"/025.png"))
    plt.title("Iteration-"+file_lst[i], fontweight='bold')
```



1.8 Model-1 Synthesized Images: 1_val_0050 - 1_val_0099

[40]:
```
Video("../logdir/finalexp001_whitebkgd/finalexp001_whitebkgd_valset_240000_rgb.
 ↪mp4")
```

[40]: <IPython.core.display.Video object>

1.9 Model-2 Synthesized Images: 1_val_0050 - 1_val_0099

[45]:
```
Video("../logdir/finalexp001_whitebkgd_noembed/
 ↪finalexp001_whitebkgd_noembed_valset_240000_rgb.mp4")
```

[45]: <IPython.core.display.Video object>

1.10 Model-1 Synthesized Test Images - 2_test_(0000, 0016, 0055, 0093, 0160)

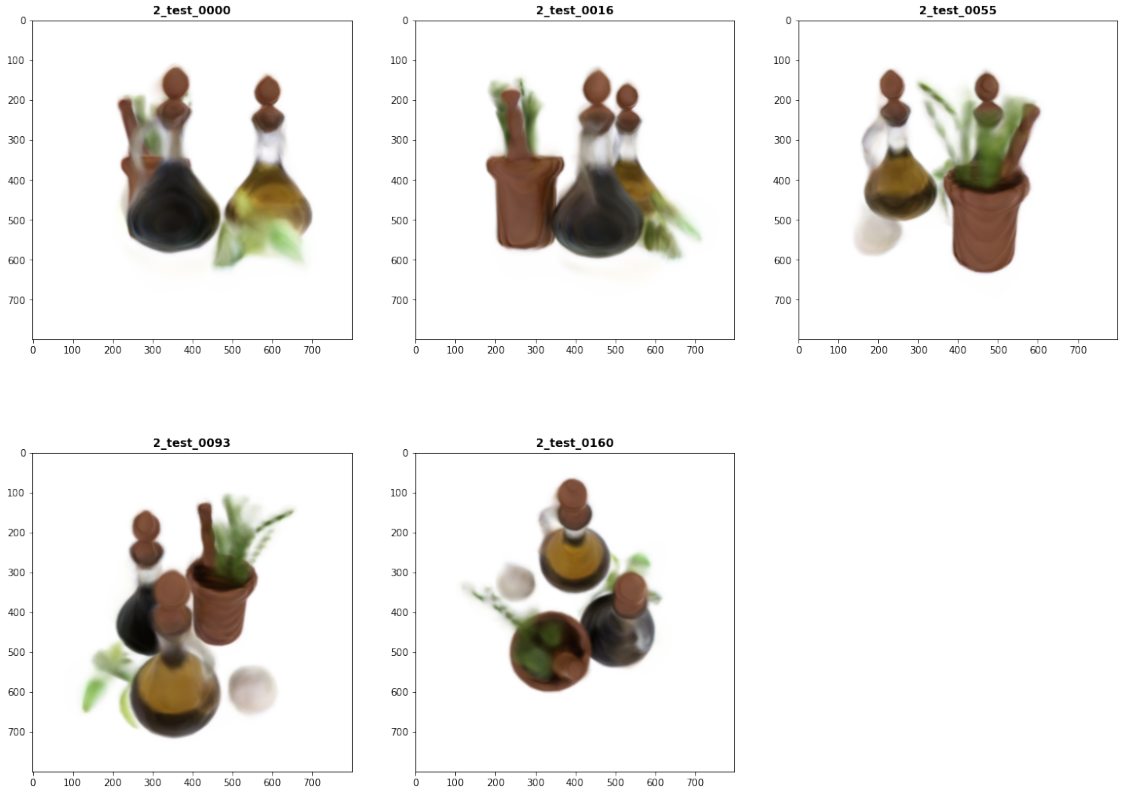
[50]:
```
fig = plt.figure(figsize=(20, 15))
file_lst = ["2_test_0000", "2_test_0016", "2_test_0055", "2_test_0093",␣
 ↪"2_test_0160"]

for i in range(len(file_lst)):
    if i == 5:
        break
    plt.subplot(2, 3, i+1)
    plt.imshow(Image.open("../logdir/finalexp001_whitebkgd/testset_250000/
 ↪"+file_lst[i]+".png"))
    plt.title(file_lst[i], fontweight='bold')
```

1.11 Model-2 : Synthesized Test Images - 2_test_(0000, 0016, 0055, 0093, 0160)

[51]:
```
fig = plt.figure(figsize=(20, 15))
file_lst = ["2_test_0000", "2_test_0016", "2_test_0055", "2_test_0093",␣
 ↪"2_test_0160"]

for i in range(len(file_lst)):
    if i == 5:
        break
    plt.subplot(2, 3, i+1)
    plt.imshow(Image.open("../logdir/finalexp001_whitebkgd_noembed/
 ↪testset_250000/"+file_lst[i]+".png"))
    plt.title(file_lst[i], fontweight='bold')
```

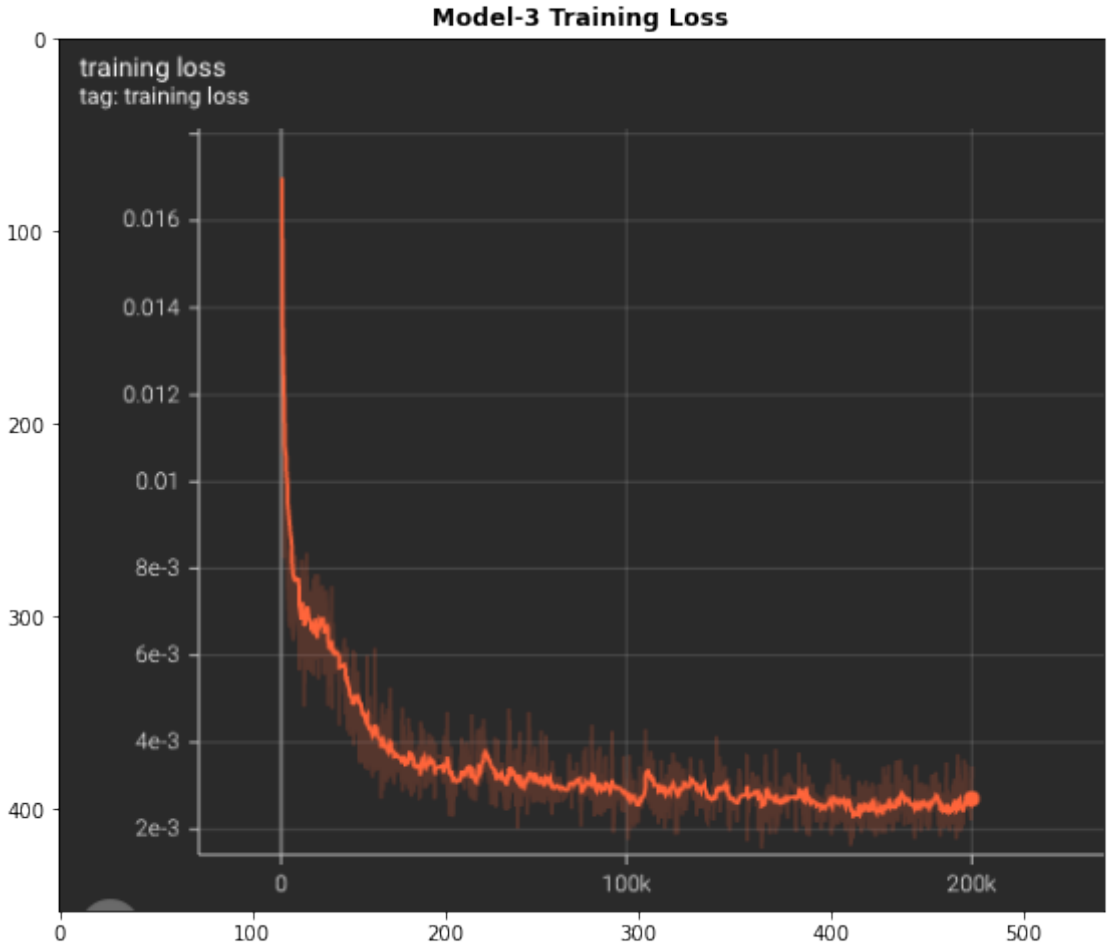


1.12 Experiment - Reducing the number of positional encoding parameters

- Used L = 5 for encoding position coordinates instead of L=10 used for Model-1 and 2
- Used L = 2 for encoding direction coordinates instead of L=5 used for Model-1 and 2
- Denote this trained model by Model-3
- Trained this model for 200k iterations

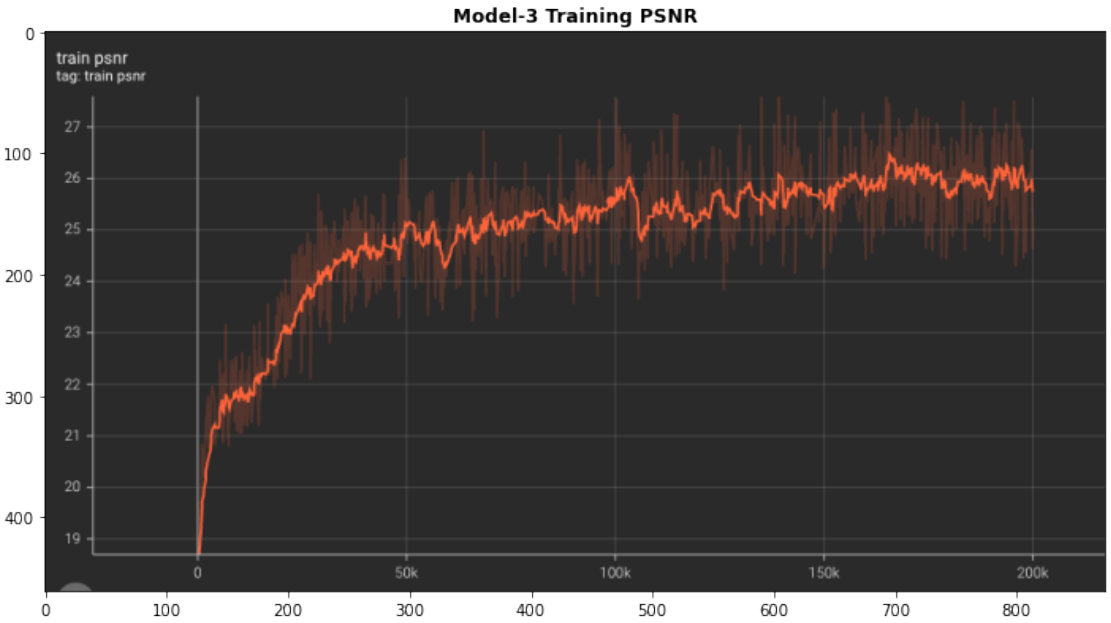
[60]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp002_whitebkgd/train_loss_plot.png"))
plt.title("Model-3 Training Loss", fontweight="bold")
```

[60]:
```
Text(0.5, 1.0, 'Model-3 Training Loss')
```



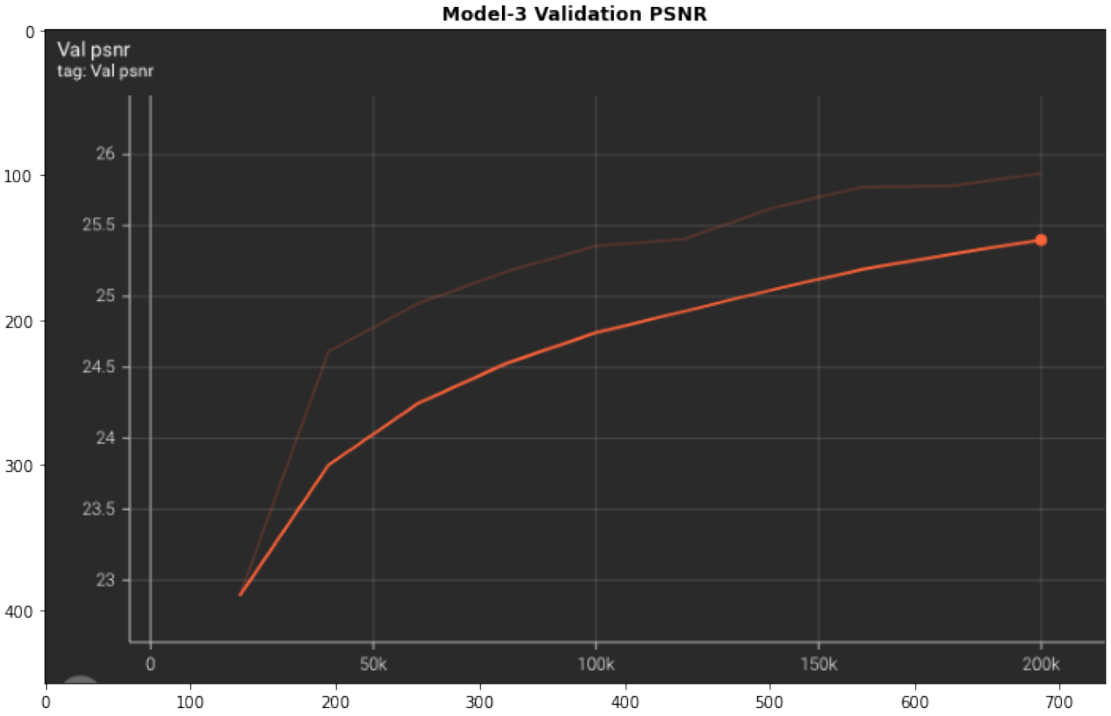
[61]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp002_whitebkgd/train_psnr_plot.png"))
plt.title("Model-3 Training PSNR", fontweight="bold")
```

[61]: Text(0.5, 1.0, 'Model-3 Training PSNR')



[62]:
```
fig = plt.figure(figsize=(12,8))
plt.imshow(Image.open("../logdir/finalexp002_whitebkgd/val_psnr_plot.png"))
plt.title("Model-3 Validation PSNR", fontweight="bold")
```

[62]: Text(0.5, 1.0, 'Model-3 Validation PSNR')

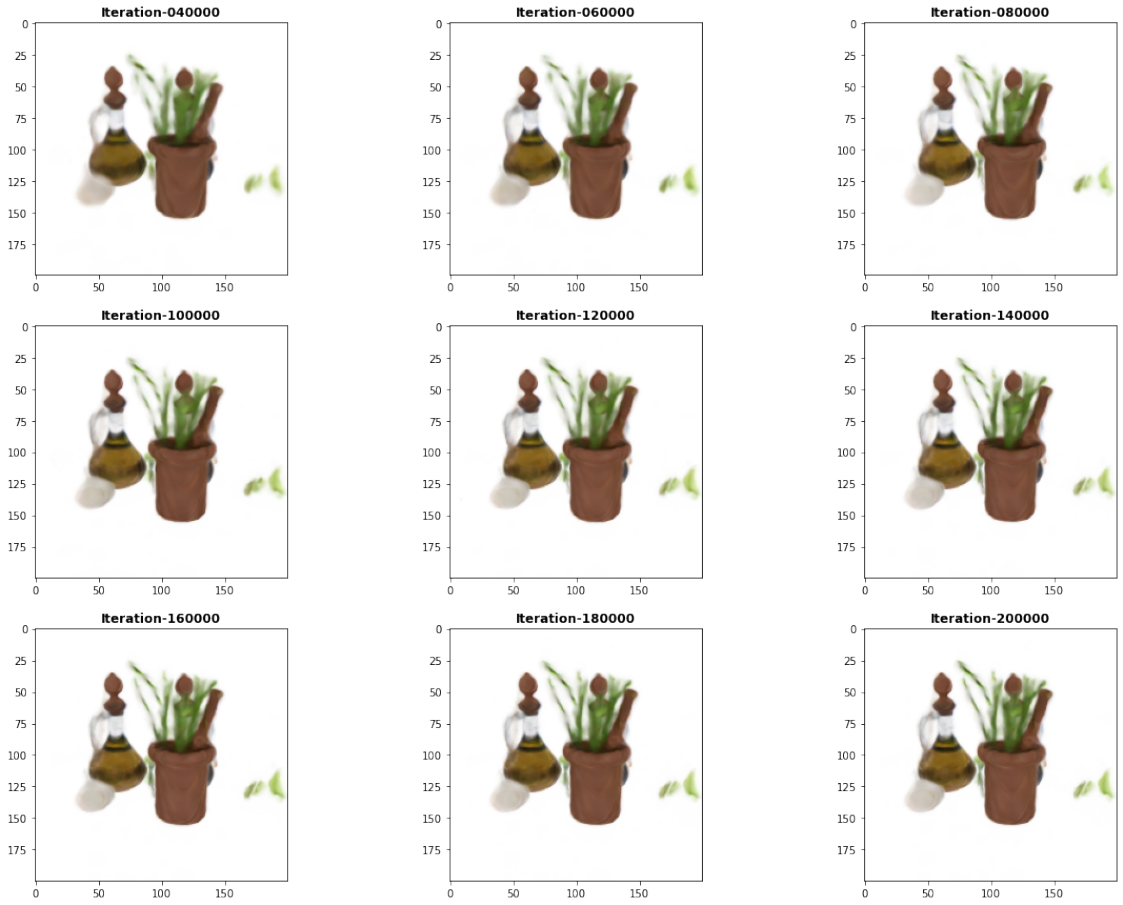


1.13 Model-3 Synthesized Validation Images

[47]:
```
fig = plt.figure(figsize=(20, 15))

file_lst = ["040000", "060000", "080000", "100000", "120000", "140000",␣
 ↪"160000", "180000", "200000"]

for i in range(len(file_lst)):
    plt.subplot(3,3,i+1)
    plt.imshow(Image.open("../logdir/finalexp002_whitebkgd/
 ↪valset_"+file_lst[i]+"/025.png"))
    plt.title("Iteration-"+file_lst[i], fontweight='bold')
```



1.14 Model-3 Synthesized Images: 1_val_0050 - 1_val_0099

[52]:
```
Video("../logdir/finalexp002_whitebkgd/finalexp002_whitebkgd_valset_200000_rgb.
↪mp4")
```

[52]:
```
<IPython.core.display.Video object>
```

3 References

1. https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666

2. Mildenhall, Ben and Srinivasan, Pratul P. and Tancik, Matthew and Barron, Jonathan T. and Ramamoorthi, Ravi and Ng, Ren, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis", 2020

3. https://github.com/yenchenlin/nerf-pytorch

4. https://www.youtube.com/watch?v=CRlN-cYFxTk

5. Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, Christian Theobalt; Neural Sparse Voxel Fields, 2020

6. Christian Reiser, Songyou Peng, Yiyi Liao, Andreas GeigerKiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs

4 Acknowledgement

I would like to thank all the TAs for their help on Piazza. I would like to thank all my classmates for having very helpful discussions on Piazza. Finally, a very big shout out to Prof. Hao Su for conducting such a wonderful course and I learnt a lot from the lectures and assignments. In addition, I would like to thank the authors of github repo in [3], as I referenced their repo for understanding and implementing some of the functions. Thank you.