NeRF

Sambaran Ghosal
Department of Electrical and Computer Engineering
UC San Diego
La Jolla, USA
sghosal@ucsd.edu

I. Introduction

Neural Radiance Field or NeRF is a breakthrough work in the field of deep learning related to 3D data generation. It is a method of generating novel views from given different camera poses. It has wide applications since using novel views of scenes, we can create 3D meshes, create simulation scenes etc.

II. Problem Formulation

Given a set of camera poses in the world frame, the aim is to create 3D scene images. During training, we are provided with images corresponding to the scene observed from a given camera pose. We use this image and pose to train the NeRF model, and during generation, we then use the camera pose to generate rays for the scene and using NeRF output and volume rendering concepts, accumulate the rgb values of each point in the pixel space. This then yields the generated test scene. We will now discuss in detail the training architecture models.

III. Technical Approach

A. Training

1) ***Getting rays and origins****:* During training, NeRF requires the ray origin and directions corresponding to the given camera pose. Each point in the image is then described by the equation

$$r(t) = o + td \tag{1}$$

where $r(t)$ is the position of a pixel, o is the ray origin and d is the direction vector or viewing direction from the camera to the pixel. t is a parameter ranging from t_n corresponding to the nearest plane of image formation and t_f corresponding to the farthest plane. For our work, we use a Stratified Sampling approach to generate 64 points uniformly along each ray of the image. Hence from one image, we have in total $H \times W \times 64 \times 3$ points sampled from the image. The viewing direction is represented as another 1×3 vector at each pixel. Hence we have a total of $H \times W \times 3$ matrix as the direction vector for each pixel.

2) ***Positional Encoding****:* Once we have the points sampled along each ray and corresponding directions, we pass the points and the direction vectors through a positional encoder that maps these lower dimensional vectors to a higher dimensional vector using harmonic embedding. For a given input x, the harmonic embedding is defined as

$$E(x) = \begin{bmatrix} x \\ sin(x) \\ cos(x) \\ sin(2x) \\ cos(2x) \\ .. \\ sin(2^{N-1}x) \\ cos(2^{N-1}x \end{bmatrix} \tag{2}$$

where N is the desired number of encoding dimension. So for each component of point (x, y, z) and each component of direction vector (d_1, d_2, d_3) we apply the encoding. If the number of positional encoding for position is N_{pos} and encoding dimension for direction is N_{dir}, the output dimension after encoding will be $3 \times (1 + 2N_{pos})$ for position and $3 \times (1 + 2N_{dir})$ for direction.

3) ***Neural Radiance Field Model Architecture****:* The main concept of NeRF is that a scene can be modeled as a collection of rgb values and density function at each point in the camera space. We can get these rgb values and the density values at each point in the space by using a non-Convolutional Neural Network (MultiLayer Perceptron) that takes in as input the encoded / or non encoded position and viewing direction at each point in the image space. The architecture used in our work is as follows :-

- We use 4 linear layers with 128 neurons at each layer. These layers take in as input the position of the points. We have a skip connection in the 2nd layer where we concatenate the output of this layer with the original position input.
- The output of the first 4 layers is then processed by two separate Linear Layers, one producing the density function, and the other producing a temporary output which is further processed as discussed below.
- The intermediate output is then concatenated with the viewing direction of the points, and then further processed by two separate linear layers which finally outputs the rgb values at the points. The rgb values and the density output are finally concatenated to give the final output of the NeRF MLP which will now be used first with volume rendering to generate scenes and then use the ground truth

images provided corresponding to different camera poses to train this model parameters.
The architecture flow is shown in Figure(2).

*4) **Volume Rendering**:* Once we have the rgb values and the density function value at each point, we use volume rendering concepts to generate the image scene. Let $c_i = (r, g, b)_i$ be the output color from the MLP for points along ray i, σ_i be the density function along the ray i. Then the obtained rgb values along ray is given as

$$\hat{C}(r) = \sum_{i=1}^{N} e^{-\sum_{j=1}^{i-1} \sigma_j \delta_j} (1 - e^{-\sigma_i \delta_i}) c_i \qquad (3)$$

where $\delta_i = t_{i+1} - t_i$ is distance between adjacent samples along the ray and t_i are generated by the stratified sampling algorithm along each ray. This equation accumulates rgb values along the ray depending on the density function which represents the chance that a given pixel is actually occluded.
For our work due to computation resource limitation and time limitation, we did not run the training using Fine Model that uses Hierarchial Sampling to even further generate points that are not occluded and hence have more impact in generating the scenes.
Hence, the overall training architecture flow is described in Figure(1).

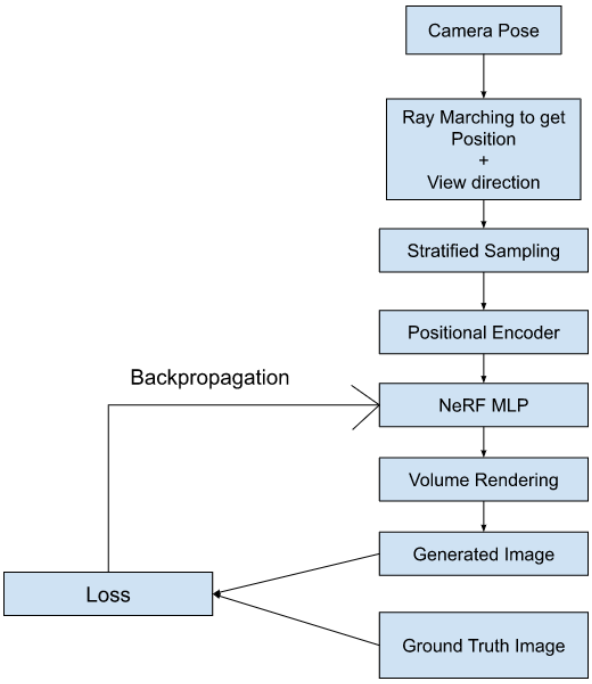

Fig. 1: Nerf pipeline

IV. Experiments

A. Train setting

During training, we use the following hyperparameter values : Learning rate is started from 4e-3 and decayed by 0.25 after every 50000 iterations. We only use coarse network and do not use fine network and no hierarchial sampling. The positional encoding dimensions are set at 5 for positions and 2 for viewing directions. The MLP has 4 layers with skip connection after second layer in addition to the bottleneck layers that are not changed anytime. The 4 linear layers are set with dimension 128. The network is trained on 100×100 sized images and the first 100 images for the training + validation set provied to us. Evaluation of the model is done every 500 iterations on an image not included in the training set and the progress is saved. We keep track of training and validation PSNR to evaluate the model performance.

Since NeRF has a high GPU requirement, we use a batch size of 4096 to select rays from a given image. This helps reduce the GPU usage slightly.

B. Training Progress

Figure (3) show the quality of performance over several iterations. The left hand side shows the generated image using NeRF on a validation pose, middle one is the ground truth image for that validation file and right side image shows the training psnr vs validation psnr.

C. Training with and without Positional Encoding

In this section, we study the affect of doing training with and without positional encoding on the position and directions. For the position encoding we use the same dimensions as above i.e. position encoding of 5 for (x, y, z) and directional encoding of 2. We show the 800×800 image reconstructed using model trained with and without encoding. We also report the PSNR on the validation image using the above.

The models are tested on validation images 1_val_0050.png and 1_val_0080.png. The resulting generated images are shown in Figure(4). The model with no encoder yield a PSNR of 21.4405 on the 1_val_0050.png and 20.0150 on 1_val_0080.png, whereas the model with the encoder yields a PSNR of 21.7359 on the 1_val_0050.png and 21.4636 on 1_val_0080.png.

The differences would be more clear in the image if we had also used the hierarchial sampling and fine model output too. But we can observe in the left figure, we can see that the grass in the right hand side of the upper left image is represented better than in the right image showing that with positional encoding we are able to capture different objects. Similar behaviour is also observed in the bottom image where grasses in the surface and tub are better represented in the left image than in the right image.

D. Validation PSNR

Once the training is done, we use the NeRF model to generate 800×800 sized images on poses labeled '1_val_0020.txt', '1_val_0040.txt', '1_val_0050.txt', '1_val_0095.txt' that were

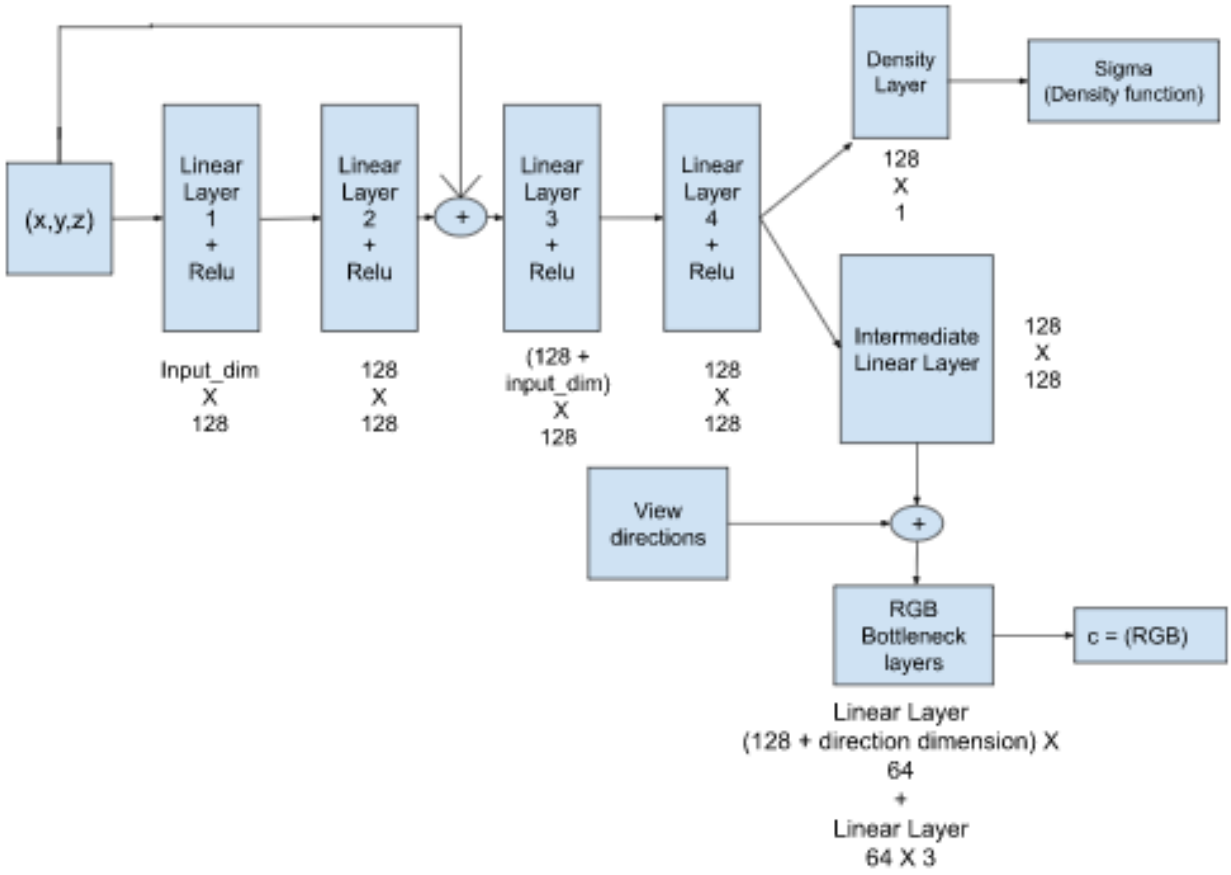

Fig. 2: NeRF MLP Architecture

Validation Image	PSNR
1_val_0020.png	23.84
1_val_0040.png	22
1_val_0050.png	21.12
1_val_0095.png	22.05

TABLE I: PSNR of selected validation images

not part of the training image and the validation image during training. The generated images vs ground truth images are shown in Figure(5) and corresponding PSNR is shown in Table(I).

Using our trained NeRF, we are able to achieve an average PSNR of **22.06**. We get a low PSNR which can be credited due to the fact that we used only a linear layer of 128 feature dimensions, a total of 4 linear layers, a lower encoding dimension of 5 and 2, and also not using the fine model for further quality improvement. We can see that our generated images although having very similar scenes, are blurry compared to the sharp quality of the ground truth images.

E. Test Settings

To generate the test images, we simply use the given pose of the camera to get all rays and directions. Then we generate the total 800×800 image by generation 4 blocks of 400×400 images corresponding to upper left, upper right, bottom left and bottom right blocks of the image and concatenate these accordingly to get the full image. This way at a given instant we are using only 1/4 GPU memory compared to if we did generate the image at once which would quite possibly lead to GPU OOM issues.

F. Test image generation

Finally, using the trained model, we generate novel views for the camera poses provided in 2_test_0000 , 2_test_0016 , 2_test_0055 , 2_test_0093 , and 2_test_0160. These test images are attached in the zip file along with the submission.

V. Acknowledgement

I have written the code by referring to available online NeRF implementations from [1], [2] and [3]. I acknowledge that I have not copied the full code directly and have understood

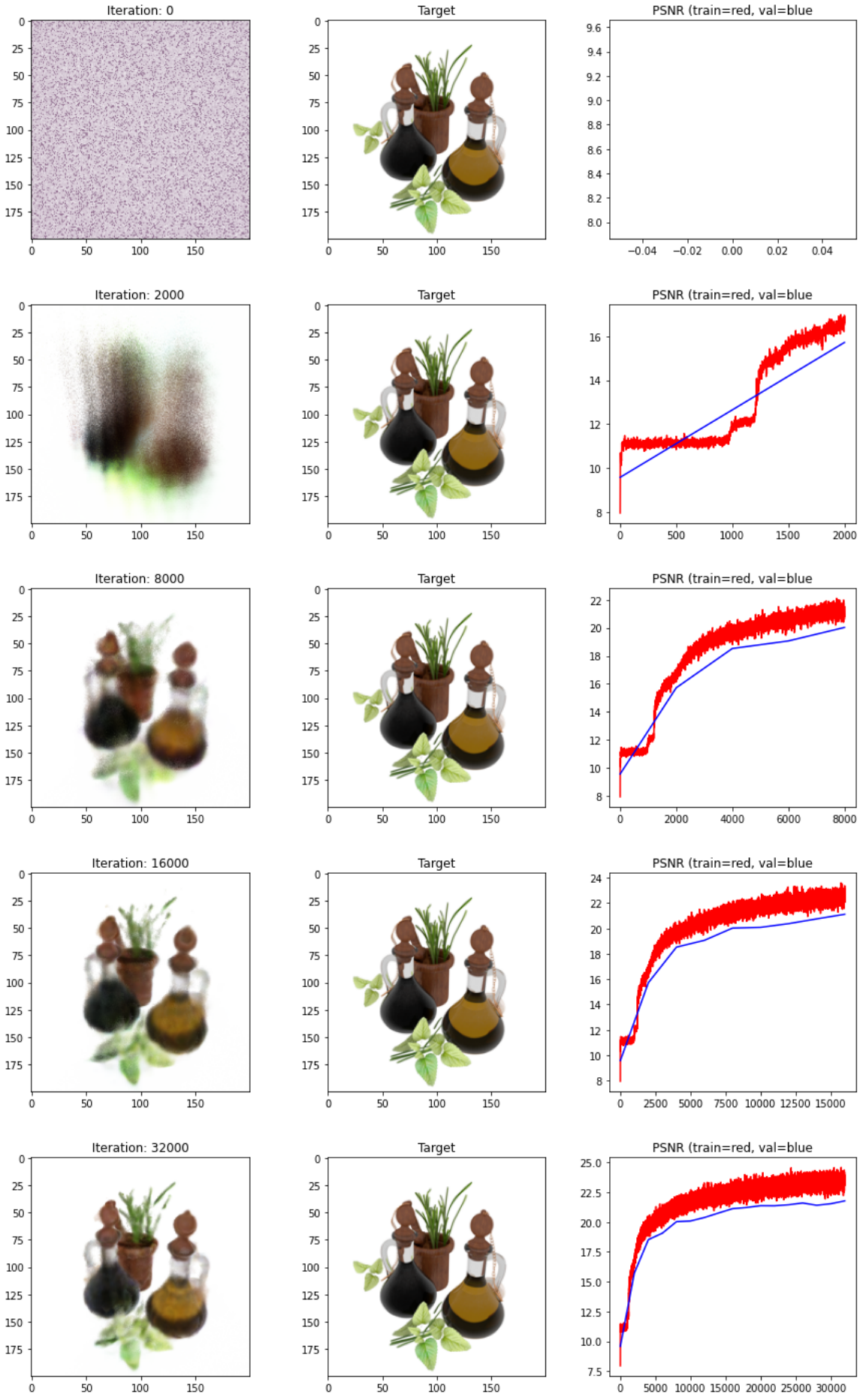

Fig. 3: Validation image progress over epochs

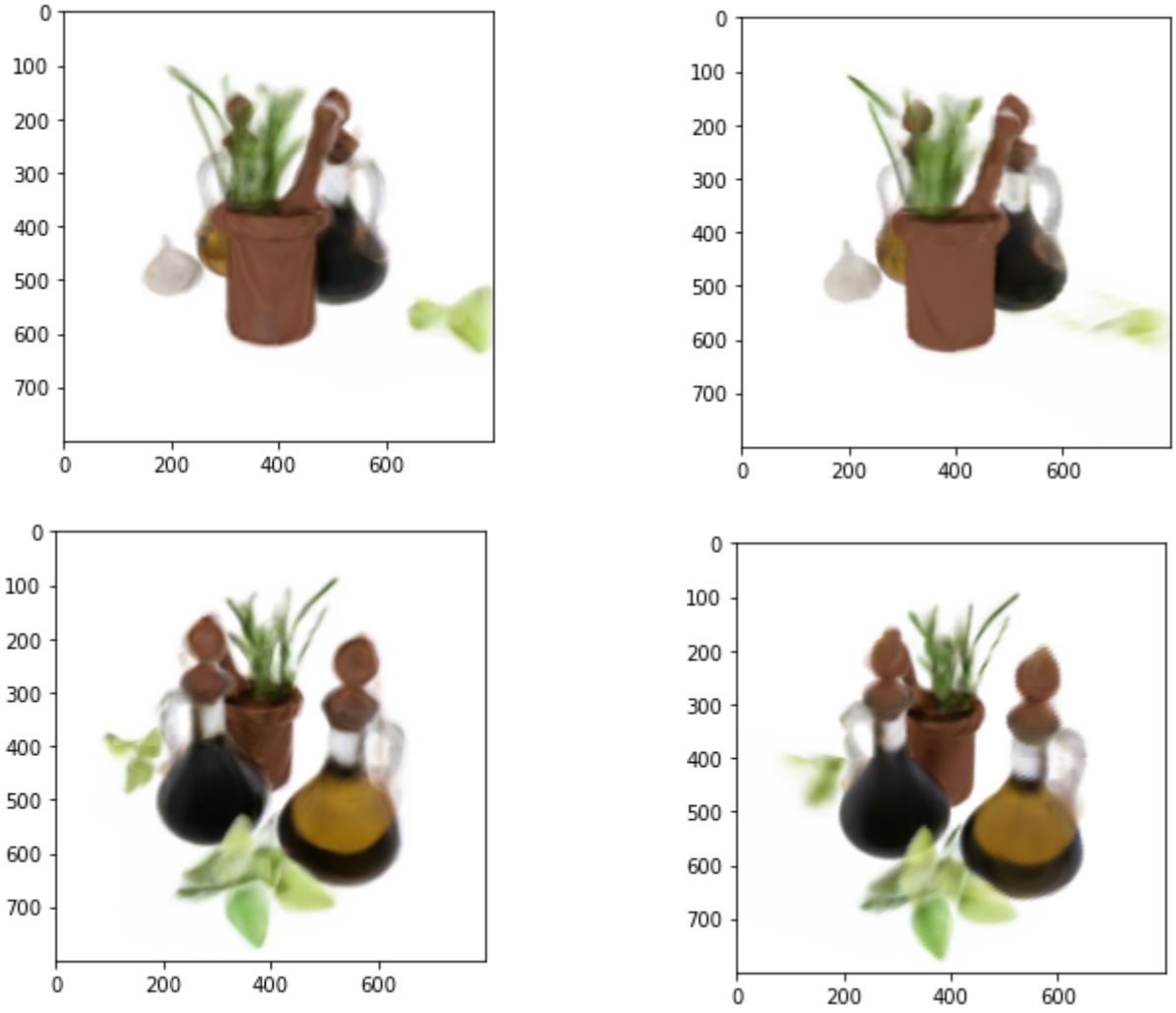

Fig. 4: Generate image with positional encoder (left) and without encoding (right)

each step in the given implementations and written down the implementations in my own way.

I would also like to acknowledge that I had fruitful discussion and debugging sessions with my colleague Mr Albert Liao. We discussed key implementation features, concepts and also helped each other with pytorch debugging sessions.

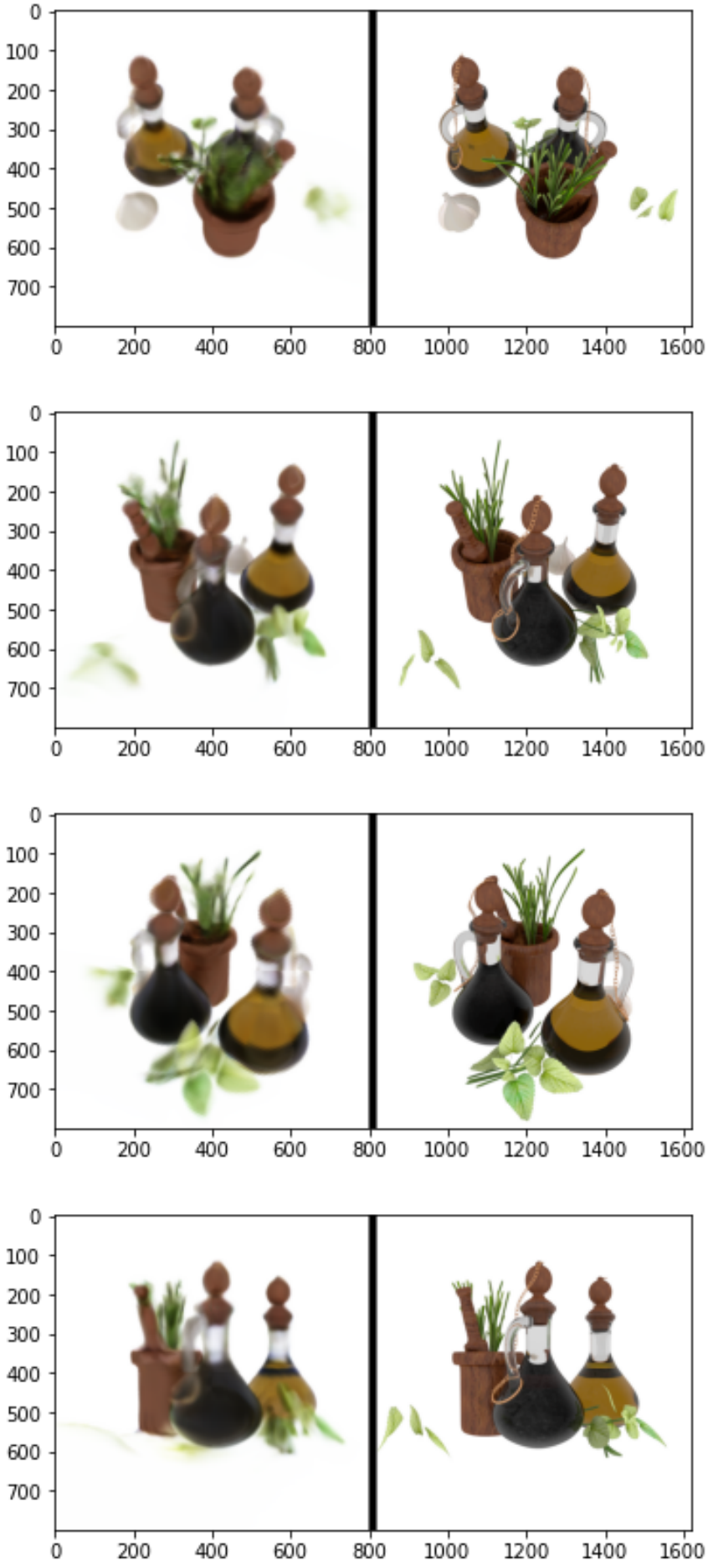

Fig. 5: Generated images using NeRF(left) and ground truth images (right)

REFERENCES

[1] "'Mildenhall et al. NeRF: Representing Scenes as Neural Radiance Fields for View, https://github.com/bmild/nerf Synthesis"'

[2] "'https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666"'

[3] "'https://github.com/yenchenlin/nerf-pytorch"'