NCSR Demokritos - Deep Learning Assignment 2024

Giorgos Bouritsas
Aris Tsilifonis-mtn2323

Neural Radiance Field(NeRF): 3D Reconstruction and Novel View Synthesis

Neural Radiance Fields revolutionised the way to represent 3D scenes. Traditional 3D graphics technology, such as voxel grids, did not manage to capture a scene in such detail as NeRF. Other 3D graphic methods required a lot of expenses for cinematic studios and were significantly time consuming. Neural Radiance Field introduces an innovative way of 3D-rendering with deep neural networks, constituting it accessible to a larger number of people. The project is heavily based on the paper that introduced NeRF NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. It presented a volumetric rendering technique employing a fully-connected deep neural network, relying solely on a sparse set of images and corresponding camera poses. The model's output includes the emitted radiance, which encompasses volumetric density and view-dependent values. The model utilises input consisting of 5D positional data that follows camera rays. This positional data includes the 3D location (x, y, z) and viewing direction (θ, φ). It applies deep learning methods to project the produced colours and opacities into an image. In this assignment, we process the Tiny NeRF Dataset to create a 3D model of a toy bulldozer.

1. Theory

In this section, we will elaborate on the theoretical foundations that this work is based on.

1.1 Volumetric Rendering

$$C(r) = \int_{t_n}^{t_f} T(t)\, \sigma(r(t))\, c(r(t), d)\, dt \quad (1)$$

$$\text{Where } T(t) = exp\left(-\int_{t_n}^{t} \sigma(r(s))\, ds\right) \quad (2)$$

The rendering equation illustrates the colourness expectation C(r) of camera ray r(t) = o + t*d. Limits t_n and t_f are associated with near and far bounds. The t fluctuates between those limits. Here, o is the origin of the ray and d is the direction vector of the ray. $\sigma(r(t))$ is the volumetric density at point r(t). Higher density can result in the scattering or absorbtion of the light since more particles are present along the ray. The $c(r(t), d)$ is the colour that radiates from point r(t) in the direction d. It signifies the light emitted at each point along the ray.

The accumulated transmittance in equation (2) depicts the likelihood that the ray travels from t_n to t without encountering any other object(chance to be absorbed).

It decreases as the density σ along the path becomes greater. The exponential term shows the total amount of light absorbed as it travels inside the ray, whereas ds denotes a small segment of distance to calculate the light that is absorbed up to that point.

To sum up, the combination of accumulated transmittance, density and emitted radiance affect the colour the model outputs. By integrating the contributions of those values, we obtain the scene's colour.

It is worth noting that the luminosity of the colour is different in each direction (view-dependent). Figure 3 in the original paper proves the effect of viewing perspective on the quality of the output. The loss is derived from the original perspective image and is used to update the network.

1.2 Hierarchical Volume sampling

The equation (1) is in continuous form and cannot be converted as it is to code.

Therefore, we have to approximate it with a discrete mathematical equation:

$$\hat{C}(r) = \sum_{i=1}^{n}\left(T_i(1 - exp(-\sigma_i\delta_i)\, c_i\right)$$

Subtracting this from 1 gives the probability that the ray is absorbed at the i-th sample point

Where $T_i = exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right)$ in

$$t_i \sim u\left[t_n + (t_f - t_n)\frac{i-1}{N},\ t_n + (t_f - t_n)\frac{i}{N}\right]$$

The optical axis is divided into N segments and in each one we sample points uniformly randomly for the 3D rendering

$\delta_i = t_{i+1} - t_i$(distance of adjacent samples)

In the scene, there are empty spaces and occlusions that are not contributing to the rendering process.
The ECCV2020 paper introduces a way to sample more efficiently by optimising two neural networks at the same time, coarse and fine. In this implementation, we follow this approach:

1.2.1 Compute the CDF from the PDF:
We normalise weights,

$$p_i = \frac{w_i}{\sum_{j=1}^{N} w_j + \varepsilon}$$

where ε is a small value to avoid division by zero, and N is the number of weights.

We calculate cumulative distribution function (CDF):

$$cdf_i = \sum_{j=1}^{i} pdf_j \quad ,\ cdf_i \sim F_i = [F_0, F_1, \ldots, F_n]$$

The function also ensures that the CDF starts from zero:

$$cdf = concatenate(0, cdf)$$

1.2.2 Draw uniform random samples

$$u_k \sim uniform(0, 1),\ k = 0, 1, 2, \ldots\ M$$

1.2.3 Map Uniform Samples to the CDF (Inverse Transform Step):

$$BinIndex\,(u_k) = min\left\{i \mid F_i \le u_k < F_{i+1}\right\}$$

$$LowerIndex\ (k) = max\left\{BinIndex(u_k) - 1,\ 0\right\}$$

$$UpperIndex(k) = min\left\{BinIndex(u_k),\ N\right\}$$

(Bins are considered the intervals in the cdf)

1.2.4 interpolate to find the Fine sample points
By utilising the indices found in the previous step, we interpolate to find sample values:

$$t = \frac{u_k - cdf_{lower}}{cdf_{upper} - cdf_{lower}}$$

The interpolation ensures that the sample points follow the original distribution of the weights

$$s_k = u_{lower} + k * (u_{upper} - u_{lower})$$

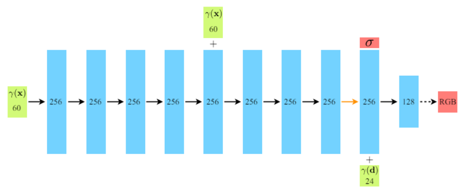
Where u are the midpoints of the bins used to discretize a distribution. The midpoints are used to map the uniform samples provided to the cdf back to the original points of the distribution.
Instead of performing uniform random sampling, we employ inverse transform sampling to obtain a more accurate representation of the original distribution. This algorithm is detailed in section 5.2 of the original NeRF paper. It could be considered as Importance Sampling since we find the most important points in the distribution to get more accurate rendering by concentrating more samples from the area of interest in the scene. As you will see below, it drastically improves the quality of the rendered 3D model.

1.3 Positional Encoding
Another important aspect of the rendering process is the positional encoding calculation. Previous research work indicated that neural networks have a tendency to focus on lower frequency data. Since we want to represent high-definition scenes with great detail, we need to shift the focus of the neural network to high frequency data.
This can be achieved by constructing vector:

$$\gamma(p) = (sin(2^0\pi p), cos(2^0\pi p), \ldots, sin(2^{L-1}\pi p), sin(2^{L-1}\pi p))$$

where p can be either position or direction vector. Value π can be omitted in the code since it did not provide improvement. Even if the distance of two points in the original space is significantly close, we can distinguish the two points by positional encoding method. As a result, clearer pictures can be generated.

1.4 Deep Neural Network Architecture

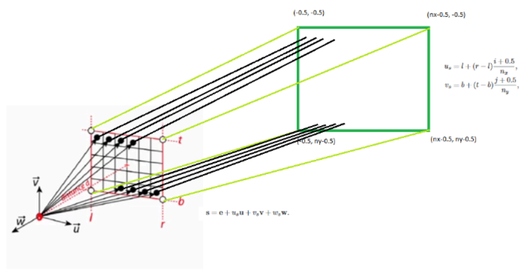


The input to the network is a positional encoding of 3D coordinates, denoted as $\gamma(x)$. The input vector has dimensionality of 60. The other part of the network consists of several fully connected layers, each with 256 width. These layers process the encoded input by applying ReLU activation function. After each pass of a layer, a neural network can learn even more complex features. The network incorporates skip connections that concatenates the original position encoding with the current output and feeds the combined structure to the network again. This can assist training since the initial information can be preserved through deeper layers of the network. In the last part of the architecture, we observe that the network splits into two branches. The first uses a fully connected layer to predict the volumetric density σ (as alpha in the code). The density value (σ) determines how much light is absorbed at a point in space. The second branch processes additional γ(d) input which is the positional encoding of viewing direction with dimensionality 24. Orange arrow represents no activation function. Dashed arrow represents the Sigmoid activation function. The symbol "+" in the schema, illustrates the concatenation of the features. The feature colour vector passes through an additional fully connected layer, first with 256 size, followed by a reduction to 156 neurons. The final output is an rgb colour, normalised between $[0, 1]$. In this way we enable 3D-rendering from sparse 2D images.

Another important aspect of the rendering process is the ray sampling technique.

1.5 Ray Tracing

Several fundamental theories of graphics are used to create an implementation of this structure. We need to generate ray origins and directions for a pinhole camera model given the camera parameters and transformation matrix. In this concept, we sample rays according to the corresponding pixels in the image plane.



(-0.5, -0.5)
(nx-0.5, -0.5)
(-0.5, ny-0.5)
(nx-0.5, ny-0.5)
$u_s = l + (r-l)\frac{i+0.5}{n_x}$,
$v_s = b + (t-b)\frac{j+0.5}{n_y}$,
$\mathbf{s} = \mathbf{e} + u_s\mathbf{u} + v_s\mathbf{v} + w_s\mathbf{w}$.
v
w
u
t
b
l
r

This process involves the transformation of 3D coordinates to camera coordinates and then to image coordinates. To achieve image formation we need to create a pixel grid corresponding to the image plane. Then, we can obtain directions by converting image coordinates. This can be calculated with the equations:

For x coordinate,

$$directions[i, j, 0] = \frac{\left(j - \frac{width}{2} + 0.5\right)}{FocalLength}$$

For y coordinate,

$$directions[i, j, 1] = -\frac{\left(i - \frac{height}{2} + 0.5\right)}{FocalLength}$$

Z coordinate is set as -1 which shows that therays are pointing forward in the camera space

We adjusted the pixel centers to be centered around (0, 0) and we scaled by the focal length.

Then we can transform directions to world space using the rotation part of the camera to the world matrix. This is a four by four matrix and we extracted the first 3 rows and 3 columns which contain the rotation part. By multiplying with the transpose of the camera to world matrix we get ray directions

$RayDirections = directions * CameraToWorld[:3, :3]^T$

To get the ray origins, someone need to use the translation part of the matrix (the camera position in the world space) :

$CameraToWorld[:3, 3]$. In the implementation, we repeat the camera's position to match the dimension of the direction vectors.

2. Implementation details

In order to get the rgb value and the weights that our rendering process requires, we need to adjust our implementation accordingly.

Sampled points that are passed in **get_rgb_weights()** function have shape (batch_size, num_samples,3). Batch_size is the number of rays being processed currently. Num_samples is the number of sample points along each ray. The 3 represents the (x, y, z) coordinates of each sample point. We flattened these points (Batch_size*num_samples,3) and encoded them properly to pass them to the neural network for processing. Optionally, we can use viewing direction to enhance the model's efficiency in rendering scenes. Some tensors operations are applied to ensure that directions are expanded properly to match the correct shape for the neural network(similar shapes as before).

Z values in the implentation deal with the depths of sample points along each ray in volumetric rendering. They show the positions of the sampled points along each ray. We computed the distances between consecutive samples in this way:

$$\Delta z_i = z_{i+1} - z_i$$

Delta holds the distances between two consecutive points. We handled a boundary condition by concatenating a very large value at the end of each ray to indicate that the last point contributes insignificantly in formation of the weights. By calculating the euclidean norm of each direction vector, we can scale effectively delta values like this:

$$\Delta z_i = \Delta z_i * \|directions_i\|_2$$

If specified by the user we can add noise to the density produced by the model. This added randomness can boost the robustness of the network by making it less susceptible to overfitting and improving generalisation.

$$\sigma_i' = \sigma_i + noise_{std} * R_i, \ where \ R_i \sim N(0, 1)$$

We scale random variables by $noise_{std}$ which is typically a value inside [0,1].

Finally we can compute the weights and the transmittance, according to equations mentioned in the theory. To compute:

$$a_i = 1 - exp(-\sigma_i \Delta z_i)$$

We need to concatenate a tensor of ones along the last dimension(we add one more column-(batch_size,num_samples+1), meaning that no absorption has occurred before the first sample point. Then transmittance will be calculated by

$$T_i = \prod_{j=0}^{i-1} (1 - a_j)$$

We added a small number to prevent the factor inside the parentheses from ever becoming zero. In cumulative product calculations a zero value could cause all following values to be zero which is undesired.

The weight of each sample point i is computed by:

$$w_i = a_i * T_i$$

The weight represents the contribution of the sample point to the rendered colour.

Transmittance[:, :-1] excludes the last value of the transmittance tensor because it was added to handle the boundary condition mentioned previously.

Regarding the **rendering** function in the code, we followed a specific approach. Firstly, we generated uniform samples along the rays between near and far boundaries.

Then we use a formula to calculate sample

$$SamplePoint = origin + t * direction$$

Where t is the depth. The shapes of the tensors are :

origins: Tensor of shape (batch_size, 3)

directions: Tensor of shape (batch_size, 3)

z_vals: Tensor of shape (batch_size, uniform_samples) which contains the uniform samples. We have to unsqueeze those for proper broadcasting. The resulting tensors are

Origins: shape (batch_size, 1, 3)

Directions: shape (batch_size, 1, 3)

Uniform t (samples): shape (batch_size, uniform_samples, 1). The output points have shape (batch_size, uniform_samples, 3) and can properly be fed to the model. Then we calculate midpoints, which are placed

$$mid\ z\ vals_i = 0.5 * (z_i + z_{i+1})$$

The weights associated with the midpoint values, excluding the first and last weights since we handle them differently.
Regions with higher predicted densities receive greater weights, indicating they are more likely to significantly impact the final rendered image. We want to sample more from those regions of the images. If we opt to use importance sampling, we have to concatenate initial z values with the important z values. These values were generated by inputting the weights produced by the network and the initial midpoints to the hierarchical sampling function. The new set of points is being fed to the network again. The weights of the network are not updated in this pass since we want to save gpu memory. Finally, we compute the three final outputs. The colour, the depth and the accumulated weights map. The value that we care most is the one for the predicted colour.

$$ColourMap = \sum_{i=1}^{N}(final\,weights_i * rgb_i)$$

Final_rgb has tensor shape (batch_size, num_samples, 3). Final weights tensor has the shape of (batch_size, num_samples), denoting the importance of the sampled point. The line in the code T.sum(final_weights.unsqueeze(-1) * final_rgb, dim=-2) converted the shape from (batch_size, num_samples, 3) to (batch_size, 3). The contributions of all sampled points along the ray are summed to produce the final RGB colour for each ray, which is essentially the colour map.

3. Metrics

Several metrics were applied to measure the performance of the algorithm. MSE loss measures the average squared difference between the predicted and the actual value, according to the equation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

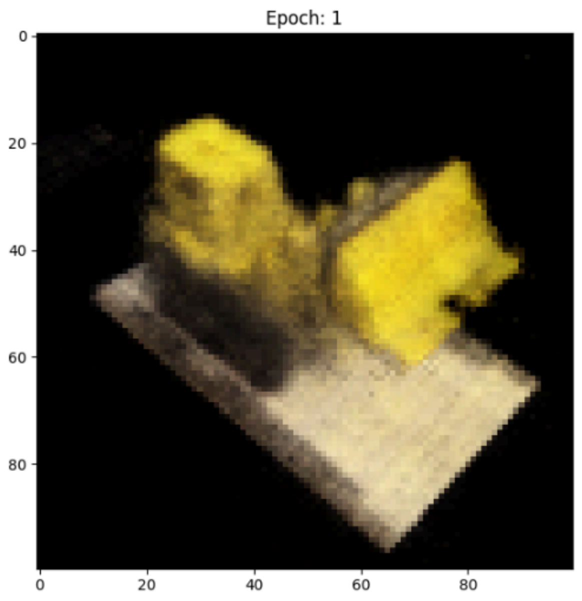
In this way, we can test if the predicted output matches the ground truth data. The loss value should be approaching zero for better results.
Then, we utilised peak signal to noise ratio. It provides a measure of similarity between the predicted ground truth image. Higher PSNR values indicate better performance

$$PSNR = 10 * log_{10}\left(\frac{MAX^2}{MSE}\right)$$

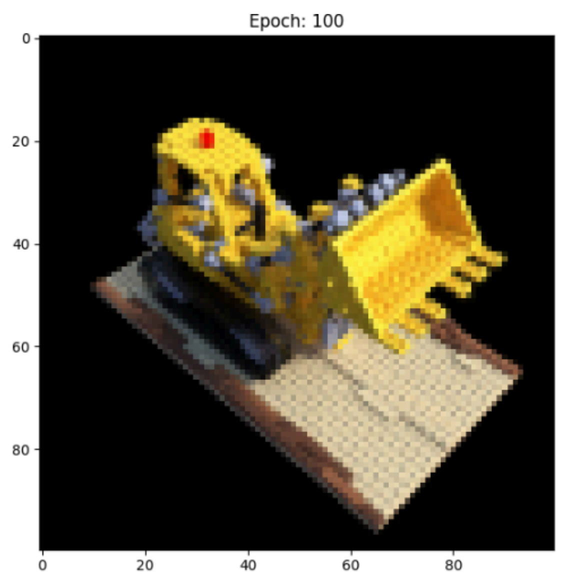
PSNR is calculated after evaluating the model on the test dataset. Finally, a predicted image was used to represent the model's ability to reconstruct the test scene based on the rays sampled from the test camera pose. For this purpose, we sampled rays from an image that did not exist in the train dataset and run the model to predict the RGB values along these rays.

4. Results

The results are based on the metrics that were provided. On our experiments we used 100 images of the toy bulldozer of size 100*100 pixels to construct a 3D scene of it.
Reproduced test image on the first epoch was

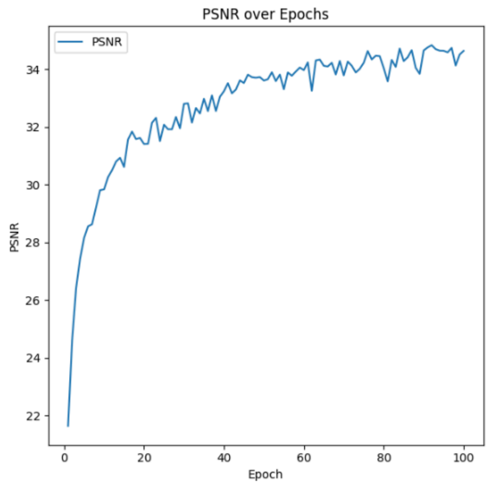


PSNR score=21.6334
Average loss per epoch = 0.02323991668383881
After 100 epochs of the experiment, the model managed using hierarchical sampling, this output.
PSNR score=34.6326
Average loss per epoch = 0.00018255940395632574
Without hierarchical sampling the resulting PSNR of the predicted test image was:
PSNR score=30.3533
Average loss per epoch = 0.00019438861233881698
It is satisfying but considerably lower compared to the inverse transform sampling method. Below, we depict the predicted image at the end of the experiment

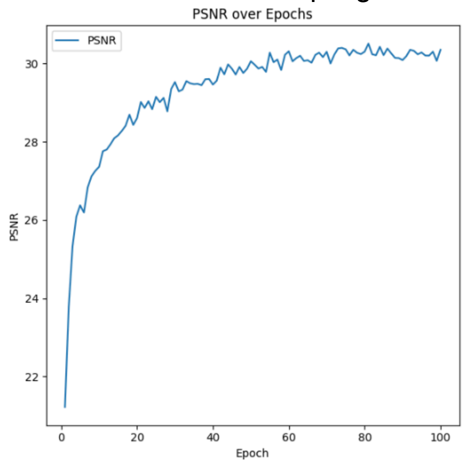


Regarding the other metrics used, the test PSNR achieved a higher peak when hierarchical sampling was applied. Additionally, the image similarity score improved more rapidly compared to the simpler version.

With hierarchical sampling

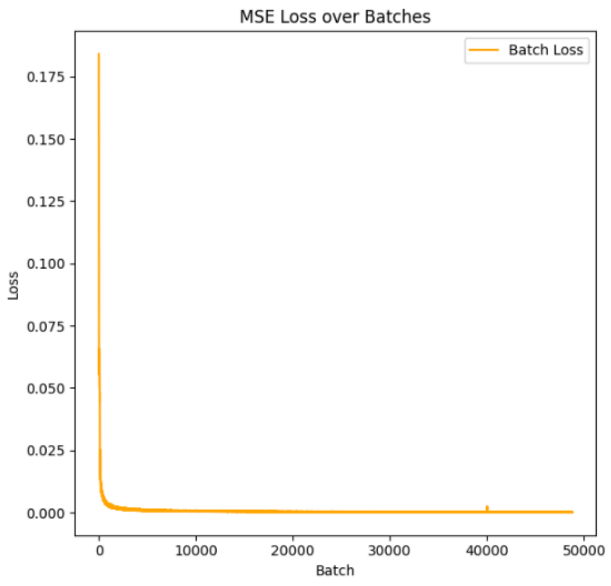


Without hierarchical sampling



It is evident from the graph that at 20 epochs the complete version reached PSNR 32 while the simpler one around 32, indicating the effect of the sampling method.

The MSE loss increased in both of the experiments, which is a good sign that the models are performing well and there is no overfitting.



Finally, a 360 video was created to demonstrate the ability of our model to render 3D scenes. A screenshot of the video is provided to show that the object is now on 3 dimensions instead of 2

[Toygif]()

The 360 video code generates a video by rendering frames of a 3D scene from different camera angles using the NeRF model that we trained. It rotates the camera around the scene, samples rays for each camera position, and uses the NeRF model to predict the RGB values for the sampled rays. The frames are then assembled into a video file. For this purpose, we compute the camera-to-world transformation matrix for each angle. After that, we sample rays for the current camera position. The results illustrate that the model is very efficient at depicting a model in three dimensions since the object in the produced video looks close to the real one, while the metrics that were used verify this conclusion.
About the parameters that were used, they are not the same as the original paper since we had to handle a smaller number of lower resolution images. The number of batches is 488 and each batch has 2048 rays on the first experiment. On the other one, we used 244 batches of rays and each batch contains 4096 rays in each epoch. We understand that when the batch size is smaller, the accuracy increases significantly. Unfortunately, due to gpu limitations we were not capable of training with higher resolution images that could have created even more realistic 3D scenes. Depth bounds were (2, 6) , based on the original tiny NeRF implementation. The dataset shapes were:
image_data.shape: (106, 100, 100, 3)
106 images of 100*100 size with 3 colour channels
pose_data.shape: (106, 4, 4)
106 different camera orientations, the 4*4 camera translation matrix
focal_length: 138.88887889922103
Focal length depends on the distance between the lens and the object. It can affect the field of view, the magnification and the depth of field. It concerns how the rays are spread out from the camera's origin and is part of calculations in our programme about ray sampling. Finally, viewing directions are used and play a crucial role in the good results. There are 24 viewing input channels as defined in the original paper. The time that each epoch required was 20 seconds in the GTX 4090 and 2 minutes in T4 and L4 colab GPUs.

5. Conclusions

The NeRF model produced high definition and photorealistic 3D scenes. It is an innovative way to reconstruct 3D scenes using relatively simple networks. About the implementation, several aspects could be improved. Since NeRF models are GPU intensive by nature. If the GPUs become even better then the 3D rendering will improve since it will require less time. On this assignment, we faced high running times at the beginning of the implementation and it was a great challenge to manage to reduce them to a reasonable time. We believe that some rays are higher correlated than others and the NeRF model is capable of recognizing these relations even more frequently with more recent versions. Are the MLPs really the best option? There are a lot of emerging deep learning architectures that seem really promising, like Kolmogorov-Arnold Network (KAN). Is the way that we are incorporating viewing directions on the MLPs optimal? There might be an even better way. Considering sampling, mip-NeRF introduced a different way of sampling with cones which proved even more efficient that NeRF. Sparse voxel structure accelerated the rendering by skipping the voxels containing no relevant scene content. Kilo-NeRF also provided much faster rendering times compared to the initial versions. The future of NeRF looks really impressive and we anticipate the latest advancements in that field.

References

https://arxiv.org/pdf/2003.08934
https://en.wikipedia.org/wiki/Inverse_transform_sampling
https://www.youtube.com/watch?v=rnBbYsysPaU&ab_channel=BenLambert
https://computergraphics.stackexchange.com/questions/7980/problem-of-understanding-the-coordinate-systems-involved-in-ray-tracing
https://www.cvlibs.net/publications/Reiser2021ICCV.pdf
https://arxiv.org/abs/2103.13415
https://colab.research.google.com/github/bmild/nerf/blob/master/tiny_nerf.ipynb#scrollTo=6XurcHoCj0FQ
https://github.com/bmild/nerf
https://github.com/kwea123/nerf_pl
https://github.com/NVlabs/instant-ngp
https://github.com/awesome-NeRF/awesome-NeRF