**Final Year Project – Blog**

# A brief explanation of NeRFs

08/10/2022

NeRFs (Neural Radiance Fields) are fully-connected deep neural networks (non-convolutional) that represent 3D scenes within their weights by training on a sample set of images. The input to the network is a 5D vector, , where is the 3D spatial location and is the 2D viewing direction (represented as a 4x4 matrix).

The output is a 4D vector, where is the colour (view-dependent) and is the volume density (the opacity of the material). The network can then be used to predict novel views by querying 5D coordinates along a camera ray for each pixel which results in an image of the new view.

## Viewing direction matrix

The viewing direction is represented in a matrix. represents the angle of rotation on the x-axis and represents the angle of rotation on the z-axis. (Rotation on the y-axis is not necessary). Given a 3D point of the camera, rotation and translation of the camera is

(using a single rotation matrix)

, where is the translation vector.

This process can be combined into the following matrix, where is a matrix, is a matrix, and the bottom row, makes the matrix square ().

## Rendering a NeRF from a particular viewpoint

1. Send a camera ray through the scene for each pixel of the view. Generate a sample of 5D coordinates (location, viewing direction) along different parts of the camera rayDiagram

   Description automatically generated
2. Input the 5D coordinate into the NeRF network and get, as output, the colour and volume density at each coordinate Diagram

   Description automatically generated
3. Accumulate the colours and volume densities for all the sampled coordinates and apply volume rendering to calculate the colour of each pixel and produce a 2D imageDiagram

   Description automatically generated

Since volume rendering is differentiable, gradient descent can be applied to train the NeRF by minimising the rendering loss (the difference in pixel values) between a predicted image and the ground truth observation of the view.

# NeRF from Nothing Implementation Notebook

16/10/2022

The NeRF from Nothing Notebook implements the NeRF algorithm from the original paper. The code is executed in Google Collab using the free version.

Link to article: <https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666>

I attempted to adjust the hyperparameters and see the various results from the network.

All experiments were run with the random seed set to and 1000 iterations.

The test image with id=101 (training data used images with ids in the range , so id=101 refers to a new view the model hasn’t seen).

The training process includes early stopping to restart training if the loss got stuck in a local minimum.

## Default hyperparameters

**Chart, line chart, histogram

Description automatically generated**

* Reduced the early stopping PSNR default from 10.0 to **5.0** to allow the training to continue as it kept restarting. This is reflected for all other experiments
* Time taken to complete – 5min 53s
* Output – loss (MSE) = 0.012801

## Number of samples per ray

**Chart, line chart, histogram

Description automatically generated**

* Increase from 64 to **100** (increasing any more will cause Out-Of-Memory error on free tier)
* Same increase applied for hierarchical sampling
* Takes longer to complete than default – 9min 16s
* Output has negligible difference – loss (MSE) = 0.013704

## Number of neurons/dimension per layer

**Chart, histogram

Description automatically generated**

* Increase from 128 to **200** (increasing any more will cause Out-Of-Memory error on free tier)
* Same increase applied for both coarse and fine models
* Takes longer to complete than default – 10min 40s
* Output has negligible difference – loss (MSE) = 0.011322

## Number of layers

**Chart, histogram

Description automatically generated**

* Increase from 2 to **4** (increasing any more will cause Out-Of-Memory error on free tier)
* Same increase applied for both coarse and fine models
* Takes longer to complete than default – 8min 58s
* Output looks slightly better – loss (MSE) = 0.0099412…

## Conclusion

Changing the hyperparameters doesn’t result in a massive change in the loss and output. In fact, they all took more time to perform 1000 iterations. Therefore, keeping to the defaults provided in the original implementation is a good decision.

# Using COLMAP To Get Camera Pose Data

24/10/2022

Given a set of unordered images of an object, building, or scene, COLMAP applies Structure-from-Motion to perform a 3D reconstruction. It estimates the camera position/angle and produces both sparse and dense models which can be viewed in mesh rendering software, such as MeshLab.

I began by testing how GUI software worked by using it on a collection of images I took of a chair in my room. I made sure that the camera settings (autofocus, ISO, white balance…) were kept fixed. There were 35 images in total.

**A screenshot of a video game

Description automatically generated**

After running COLMAP on medium quality, it produced a dense mesh which I opened in MeshLab. The overall shape of the chair and its colour came out fine, but the reconstruction missed certain regions. I realised this was because I didn’t have enough images covering the main front of the chair, nonetheless, the software did work by outputting a basic 3D reconstruction.

A picture containing graphical user interface

Description automatically generated

Next, I wanted to apply COLMAP to the OIVIO dataset. I was interested in knowing if it would be able to reconstruct a dark cave environment with poor lighting conditions.

The OIVIO dataset contains many thousands of images, therefore I selected around 400 from the start. The initial sequence is useful as the robot moves around in a circle which allows it to calibrate against the calibration chart placed at the beginning of the cave.

A picture containing text, electronics, black, keyboard

Description automatically generated

Map

Description automatically generated

The quality of the 3D reconstruction is noticeably better than the chair. The advantage of this dataset is that the images are taken at regular intervals, therefore, capturing the motion of the robot in detail. This allows COLMAP to better estimate the camera pose information. Additionally, COLMAP works better when the scene contains a lot of detail. The texture of the cave walls, as well as the shadows, were mapped really well in the reconstruction.

# Converting COLMAP Data into a NeRF format

27/11/2022

Once COLMAP extracts pose data from a set of images, it stores it in binary files. In order to use it for training a NeRF, we need a extrinsics pose matrix, the image data , and the focal length intrinsics.

## 4x4 Pose Matrix explained

The affine transformation pose matrix is defined as:

The top-left sub matrix is the rotation matrix and the right column is the translation vector . The transformation rotates and translates a given vector.

The final rotation matrix is a product of individual rotation matrices for each axis, .

## Quaternions and Rotation Matrix

COLMAP chooses to store the camera rotational data as quaternions which are a vector alternative to the rotation matrix. Quaternions are an extension of a complex number to 4 dimensions. Given a quarternion, , it is made up of a real part, , and 3 imaginary parts, , for the 3 axes of rotation. Therefore, a quarternion vector can represent a rotation around any axis.

The Python library package SciPy provides a function to convert from a quarternion to its rotation matrix equivalent, so I used it on the COLMAP data.

## World-to-Camera vs Camera-to-World coordinates

COLMAP stores the quarternion and translation vectors in the world-to-camera (W2C) format. However, a NeRF requires the data to be in the camera-to-world (C2W) format.

To convert from W2C to C2W, I needed to get the negative inverse rotation matrix. Rotation matrices are orthogonal, so their inverse is the same as their transpose, then I multiplied it by . I also needed to multiply each negative inverted rotation matrix with its corresponding translation vector to get a new translation vector in the C2W space.

## COLMAP Data Format

Although the binary data cannot be read directly, COLMAP also provides a feature to convert these to text files. The data is separated into 3 files: cameras.txt, images.txt, points3D.txt. I am only interested in cameras.txt (the camera intrinsics) and images.txt (the camera extrinsics for each image).

**Camera data format:**

* CAMERA\_ID
* MODEL
* WIDTH
* HEIGHT
* FOCAL
* PRINCIPAL POINT OFFSET X
* PRINCIPAL POINT OFFSET Y

**Images data format:**

* IMAGE\_ID
* QW
* QX
* QY
* QZ
* TX
* TY
* TZ
* CAMERA\_ID
* FILENAME

## Process to get data for NeRF

1. Read cameras.txt and store the focal length intrinsic
2. Read images.txt for the filename of each image. Use OpenCV to open each image and store them all in a Numpy array of size . They are stored in the same order as the text file to ensure the images match their pose in the pose matrix
3. Get the quaternions from images.txt in the format . Pass all quaternion vectors to the SciPy function Rotation.from\_quat(quaternions).as\_matrix() to get the matrix of rotation matrices of size
4. Get the translations from images.txt in the format .
5. Convert the data from World-to-Camera to Camera-to-World coordinates
   1. The rotation matrices are transposed and negated,
   2. The translation vectors are multiplied by the new rotation matrices,
6. Column stack the new rotation matrices and translation vectors and add the bottom row to create the ) pose matrix
7. Save the image matrix, pose matrix, and focal\_length into a Numpy zip file

## Visualisation of Pose Data

Once I computed the pose matrix, I could visualise the origin and directions of each camera. In figure , I am showing the camera rays for the tractor dataset. The tractor dataset is one of the standard datasets used to train a NeRF and was created synthetically which means the tractor is not real but created in software. Therefore, the camera directions were generated from a hemisphere and the ground plane can clearly be seen.

On the other hand, the building dataset is a set of images of a real-life building. Since I used COLMAP to get the camera poses, there is no concept of a ground plane, as can be seen in Figure . Fortunately, this is not an issue for training a NeRF as they also don’t have a concept of a ground surface.

A brick building with trees in front of it

Description automatically generated with medium confidenceA picture containing text

Description automatically generatedChart, radar chart

Description automatically generatedChart, scatter chart

Description automatically generated

Figure 2 - Poses of building dataset

Figure 1 – Poses of tractor dataset

# Training a NeRF on OIVIO

14/01/2023

* Started using the NeRF implementation from GitHub which uses Python scripts instead of a notebook, which allows for larger models trained using higher computing resources
* Trained a NeRF model on the building dataset which generated a 360-degree video of the building
* Trained two separate NeRF models on different parts of the OIVIO dataset. One was for the starting location that contained the calibration QR code and the other was for a tunnel sequence

# Exploring through a NeRF model of OIVIO

22/01/2023

* Used CV2 to take in keyboard input (WASD) that allowed us to explore through the NeRF model’s reconstruction
* Depending on what key was inputted, the pose matrix would be updated to a new viewpoint which was rendered and displayed
* Going forwards/backwards increased/decreased the translation column by a scalar multiple of the current translation vector. This allowed the camera to move in the direction it was pointing instead of on an axis
* Going left/right involved finding the orthogonal vector and moving in that direction. This required performing a cross product between the translation vector and the y-axis basis vector