Graduado en Ingeniería Informática

Optimización de hiperparámetros en el modelo de reconstrucción de campos de radiancia DirectVoxGo

Hyperparameter optimization for the radiance field reconstruction model DirectVoxGo

Realizado por
Ángel Luque Lázaro

Tutorizado por
Ezequiel López Rubio
Jorge García González

Departamento
Lenguajes y Ciencias de la Comunicación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre 2023

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN DE HIPERPARÁMETROS EN EL MODELO DE RECONSTRUCCIÓN DE CAMPOS DE RADIANCIA DIRECTVOXGO

HYPERPARAMETER OPTIMIZATION FOR THE RADIANCE FIELD RECONSTRUCTION MODEL DIRECTVOXGO

Realizado por
Ángel Luque Lázaro

Tutorizado por
Ezequiel López Rubio
Jorge García González

Departamento
Lenguajes y Ciencias de la Comunicación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2023

Fecha defensa: septiembre de 2023

Abstract

Neural Rendering is a method, based on neural networks and other techniques, capable of creating novel images and video footage based on preexisting scenes. The DirectVoxGo model is a Neural Rendering model for the reconstruction of 3D scenes, based on the NeRF technology, and featuring the optimisation of several voxel grids. With this, considerably lower execution times with comparable quality to NeRF are achieved.

In this project we perform a hyperparameter study on this model. In it, we analyse semantically what role these hyperparameters play in the execution of the model, with the aim to select those that we consider the most relevant for our experiments. Next, we choose the metrics to measure: quality (PSNR, SSIM and LPIPS), memory consumption and time of execution. We continue by designing a code, based on the original implementation of the model, that allows us to perform several consecutive tests and export the results. Following this, we analyse the results obtained and theorise about the possible optimal values for each analysed hyperparameter. We delve deeper into this by finding the Pareto optimal values, which provides us with the best-balanced values among ratios of metrics.

With these results, we create combinations of optimal values. We perform a second phase of experimentation with these combinations, finding the Pareto optimal configurations, and analysing semantically the results. From our investigation, we find two Pareto optimal configurations, one of which we think improves the results of the default configuration of the model.

We conclude that a hyperparameter study is an effective way of optimising Neural Rendering models, and, more specifically, the DirectVoxGo. We consider that, due to the high variability of the results, a further more extensive and exhaustive study would be recommended to confirm and extend our results. We also regard as interesting in the future to widen this investigation in scope, values, hyperparameters, datasets, models and techniques.

Keywords: optimization, hyperparameters, DirectVoxGo, Neural Rendering, Pareto optimal values, experimentation, metrics

Resumen

Neural Rendering es un método, basado en redes neuronales y otras técnicas, capaz de crear imágenes y vídeos nuevos basados en escenas preexistentes. El modelo DirectVoxGo es un software de Neural Rendering para reconstrucción de escenas en 3D, basado en la tecnología NeRF y consistente en la optimización de varias redes de vóxeles. Con esto, consigue tiempos de ejecución considerablemente menores a los de otros modelos NeRF con calidad comparable.

En este proyecto llevamos a cabo un estudio de hiperparámetros sobre este modelo. En él, analizamos semánticamente el rol que juegan estos hiperparámetros en la ejecución del modelo, para así seleccionar los que consideramos más relevantes para nuestros experimentos. Posteriormente, seleccionamos las métricas a medir: calidad (PSNR, SSIM y LPIPS), consumo de memoria y tiempo de ejecución. Proseguimos diseñando un código para la ejecución de pruebas, basado en la implementación original del modelo, y que nos permita realizar diferentes puebas consecutivas y exportar los resultados. A continuación, analizamos los resultados y, buscando los valores óptimos de Pareto, encontramos cuáles son los valores que mejor consiguen un balance entre pares de métricas.

Con estos resultados, creamos combinaciones de valores óptimos. Realizamos una fase de experimentación con estos, encontrando las combinaciones Pareto óptimas, y analizándolas semánticamente. De nuestra investigación, obtenemos dos combinaciones Pareto óptimas para las 6 gráficas, de las cuales una creemos que mejora los resultados de la configuración estándar de los hiperparámetros.

Concluimos que el estudio de hiperparámetros es una forma efectiva de optimizar modelos de Neural Rendering, y, más en concreto, el DirectVoxGo. Consideramos que, debido a la variabilidad de los resultados, sería conveniente realizar estudios de mayor duración y exhaustividad para confirmar y ampliar nuestros resultados. También consideramos interesante en el futuro ampliar esta investigación en escala, datos, modelos y técnicas.

Palabras Clave: optimización, hiperparámetros, DirectVoxGo, Neural Rendering, valores óptimos de Pareto, experimentación, métricas

Contents

1 Introduction **9**

1.1 Motivation 9

1.2 Objectives 10

1.3 Structure of the document 11

2 Literature Review and DirectVoxGo **13**

2.1 State of the art in Computer Vision 13

2.1.1 Convolutional Neural Networks (CNNs) 13

2.1.2 Object Detection 14

2.1.3 Semantic Segmentation 14

2.1.4 Transfer Learning 15

2.1.5 Generative Models and Style Transfer 15

2.2 State of the art for Neural Networks 16

2.2.1 Deep Learning Architectures 16

2.2.2 Transfer Learning 17

2.2.3 Generative Models 18

2.2.4 Reinforcement Learning 18

2.2.5 Ethical Considerations 18

2.3 State of the art in Neural Rendering 19

2.3.1 Neural Scene Representation and Rendering 19

2.3.2 Neural Face and Body Rendering 19

2.3.3 Real-time and Interactive Rendering 20

2.3.4 Artistic and Style Transfer Rendering 20

2.3.5 Ethical Considerations and Deepfakes 21

2.4 NeRF (Neural Radiance Fields) 22

2.5 DirectVoxGo 24

3 Technologies and Methodology **27**

3.1 Technologies used 27

3.2 High-level working of the code 28
3.3 Problems encountered 29
3.4 Hyperparameters selection 29
3.5 Measurement criteria 31

4 Single Hyperparameter Experimentation (phase I) 33

4.1 Single Hyperparameter Experimentation 33
4.1.1 Number of iterations in coarse training 34
4.1.2 Number of random rays per optimization step 38
4.1.3 Number of iterations in fine training 42
4.1.4 Number of voxels in coarse rendering 47
4.1.5 Number of voxels in fine rendering 51
4.2 Pareto optimal values 55
4.2.1 PSNR vs. Time 56
4.2.2 SSIM vs. Time 58
4.2.3 LPIPS vs. Time 59
4.2.4 PSNR vs. Memory 60
4.2.5 SSIM vs. Memory 60
4.2.6 LPIPS vs. Memory 61
4.2.7 Correction for outlier in memory measurement 62
4.3 Conclusions for this phase of experimentation 66

5 Combined Hyperparameter Experimentation (phase II) 69

5.1 Hyperparameter combinations 69
5.2 Measurement criteria 71
5.3 Results dispersion and Pareto optimal values 71
5.3.1 PSNR vs. Time 71
5.3.2 SSIM vs. Time 73
5.3.3 LPIPS vs. Time 74
5.3.4 PSNR vs. Memory 75
5.3.5 SSIM vs. Memory 77
5.3.6 LPIPS vs. Memory 78

5.4 Conclusions for this phase of experimentation 80

6 Conclusions and Futures Lines of Research 85

6.1 Conclusions 85

6.2 Future lines of Research 89

7 Conclusiones y Líneas Futuras 91

7.1 Conclusiones 91

7.2 Líneas Futuras 95

1 Introduction

1.1 Motivation

In the rapidly evolving landscape of computer graphics, conducting a hyperparameter study on the DirectVoxGo model, specifically within the realm of neural rendering, holds an undeniable importance. Neural rendering has emerged as a revolutionary paradigm, upending traditional techniques and offering a new frontier of possibilities in scene representation and visualization. However, performance and quality of such models heavily rely on the proper selection of hyperparameters. Against this backdrop, a hyperparameter study takes on newfound significance, offering the following reasons for excitement:

- **Unleashing Neural Networks' Potential**: Neural rendering models, like DirectVoxGo, rely on intricate neural network architectures to capture and synthesise complex scenes. A hyperparameter study delves into the interplay of network configurations, numbers of iterations, and optimization strategies, unveiling the nuances that drive the model's performance. With every experiment, we are closer to unleashing the full potential of neural networks in generating astonishingly realistic visuals [38].

- **Tailoring to Varied Real-World Scenarios**: In today's dynamic world, neural rendering is applied across a diverse spectrum of scenarios – from gaming and entertainment to architectural visualization and medical imaging. A comprehensive hyperparameter study acknowledges this diversity, seeking to fine-tune the DirectVoxGo model to excel in varied real-world contexts.

- **Advancing Real-Time Interactivity**: One of the most exhilarating aspects of neural rendering is its potential to bring real-time interactivity to the forefront. Through hyperparameter exploration, we strive to optimize parameters that directly impact rendering speed and memory consumption without compromising on visual quality. This

pursuit aligns with the growing demand for seamless and immersive experiences in applications like virtual reality and augmented reality [39].

- **Bridging the Gap Between Research and Application**: As neural rendering transitions from research labs to practical applications, the need for optimized models is paramount. A hyperparameter study bridges the gap between theoretical advancements and real-world impact. It propels neural rendering closer to mainstream adoption by refining models to meet the strict demands of production environments.

- **Fueling Iterative Innovation**: The iterative nature of a hyperparameter study fuels a virtuous cycle of innovation. Insights gained from each study feed into subsequent iterations, progressively refining the performance of contemporary neural rendering models. This cycle of improvement exemplifies the dynamic nature of neural rendering research and keeps researchers and practitioners at the forefront of technological advancement.

In sum, the excitement surrounding a hyperparameter study on the DirectVoxGo model within the domain of neural rendering stems from the convergence of cutting-edge technology, real-world application, and the pursuit of optimal performance. With each parameter tuned and each experiment conducted, we inch closer to realizing the full potential of neural rendering, establishing it as a transformative force in the world of computer graphics.

1.2 Objectives

The main objectives of this thesis project are:

- **Understanding DirectVoxGo**: The first objective is to delve into the fundamentals of the DirectVoxGo neural rendering model. This includes comprehending its architecture, underlying principles, and the workflow of the rendering process.

- **Hyperparameter Analysis**: Perform an in-depth investigation of the individual hyperparameters of DirectVoxGo. This involves testing various combinations of hyperparameter values, focusing on their impact on the model's output quality (measured by PSNR, SSIM, LPIPS), execution time, and memory utilization.

- **Optimization and Improvement**: Based on the hyperparameter analysis results, identify the most influential parameters and combinations that significantly enhance the rendering quality while optimizing the computational resources. The aim is to provide insights on how to improve the model's performance and efficiency for practical applications.

1.3 Structure of the document

This document is structured into several sections, each addressing a specific aspect of the project:

Chapter 1: Introduction. Provides an overview of the project, its motivation, and the objectives it pursues.

Chapter 2: Literature Review and DirectVoxGo Model. Surveys the existing literature on computer vision, neural networks, neural rendering models, DirectVoxGo, and related works in hyperparameter optimization for deep learning models. Specializes in talking about the NeRF model, as it is the most direct predecessor of our model of study, and has a had the biggest impact in the recent history of the field. Also, we will explain the high-level working of DirectVoxGo, with its techniques and phases, in order to better understand the hyperparameters we will further test.

Chapter 3: Technologies and Methodology. Dives into the technologies used in the project, a high-level analysis of the code generated, and then goes briefly over the problems encountered throughout the realisation of the project. Furthermore, it explains the hyperparameters chosen for the first phase of testing and the metrics that will be used.

Chapter 4: Individual Hyperparameter Analysis. Presents the results and insights gained from testing individual hyperparameters in isolation, showing graphs of the evolution of quality, time, and memory metrics over the five values tested. Later, we try to find the Pareto optimal values on the ratios of quality vs. time and quality vs. memory (six different ratios, considering the three different metrics of quality).

Chapter 5: Combined Hyperparameter Analysis. Explores the combined effects of selected hyperparameter combinations and their impact on rendering quality, execution time, and memory usage. These hyperparameters to combine are chosen based on the results of the previous phase of testing, and the methodology for obtaining the Pareto optimal values is

similar as before.

Chapter 6: Conclusions and Future Lines of Research. Summarizes the key findings and proposes optimized hyperparameter configurations to enhance the DirectVoxGo model's performance. Also, proposes future lines of research for the topic.

2

Literature Review and DirectVoxGo

2.1 State of the art in Computer Vision

Computer vision is a multidisciplinary field that enables machines to interpret and understand visual information from the world. It has seen remarkable progress due to advances in deep learning, increased computational power, and the availability of larger datasets. In this review, we will discuss some of the key developments and current trends in the field.

2.1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) have revolutionized computer vision by mimicking the visual processing that occurs in the human brain. They consist of layers of interconnected neurons that automatically learn hierarchical features from input images. Each layer detects increasingly complex patterns, allowing CNNs to excel at tasks like image classification, object recognition, and feature extraction. The key innovation is the convolution operation, which scans the input image with small filters to extract features. CNNs, such as AlexNet, VGG, and ResNet, have achieved remarkable accuracy in image classification, often surpassing human performance [17].

CNNs are also the foundation for various other computer vision tasks, including object detection and segmentation, where they form the backbone of many state-of-the-art models. These networks are trained on massive datasets, often comprising millions of labeled images, which enables them to generalize well to a wide range of visual recognition tasks.

2.1.2 Object Detection

Object detection involves identifying and localizing objects within an image or video. Several key developments have made this task more accurate and efficient:

- **Faster R-CNN**: This model introduced the Region Proposal Network (RPN), which generates potential object bounding boxes, combined with a CNN for object classification and refinement. It achieves state-of-the-art accuracy in object detection [31].
- **YOLO (You Only Look Once)**: YOLO is known for its real-time object detection capabilities. It divides an image into a grid and assigns each cell the responsibility of predicting bounding boxes and object classes simultaneously. YOLOv3, in particular, improved accuracy and speed [30].
- **SSD (Single Shot MultiBox Detector)**: SSD combines multiple layers with different scales for predicting object bounding boxes and classes in a single forward pass. This approach balances accuracy and speed and has been widely adopted in real-time applications [22].

Object detection is essential for tasks like autonomous driving, surveillance, and content-based image retrieval.

2.1.3 Semantic Segmentation

Semantic segmentation assigns a class label to every pixel in an image, enabling fine-grained object delineation. Relevant developments in this area include:

- **U-Net**: U-Net is a popular architecture for biomedical image segmentation. It combines a contracting path for feature extraction and an expansive path for pixel-wise prediction, making it effective for tasks like cell and tissue segmentation [32].
- **SegNet**: SegNet is an encoder-decoder architecture designed for efficient real-time segmentation. It has been applied to tasks such as road scene understanding and medical image analysis [1].

- **DeepLab**: DeepLab employs atrous convolution (also known as dilated convolution) and conditional random fields (CRFs) to capture fine details and improve segmentation results. It has been used for applications like image segmentation, object counting, and more [4].

Semantic segmentation has applications in medical imaging, autonomous navigation, and scene understanding for robotics.

2.1.4 Transfer Learning

Transfer learning involves using pre-trained neural networks as feature extractors for downstream tasks, even if those tasks have limited training data [37]. Some key aspects of transfer learning include:

- **Pretrained Models**: Pretrained models like VGG, ResNet, and Inception are trained on massive datasets (e.g., ImageNet) and serve as excellent feature extractors. Fine-tuning these models on specific tasks often leads to faster convergence and better performance.

- **Domain Adaptation**: Transfer learning can also involve adapting a model trained on one domain (e.g., natural images) to perform well in another domain (e.g., medical images) by fine-tuning the model with domain-specific data.

Transfer learning has democratized the development of deep learning applications, allowing researchers and practitioners to leverage the knowledge encoded in pretrained models.

2.1.5 Generative Models and Style Transfer

Generative models have enabled creative applications in computer vision:

- **GANs (Generative Adversarial Networks)**: GANs consist of a generator and a discriminator that are trained simultaneously in a game-theoretic setting. GANs have been used to generate realistic images, create deepfake videos, and perform image-to-image translation [11].

- **Variational Autoencoders (VAEs)**: VAEs are used for generating data and learning latent representations. They find applications in image generation, denoising, and anomaly detection [16].

- **Style Transfer**: Neural style transfer techniques combine the content of one image with the style of another, resulting in visually appealing artistic effects. Gatys et al.'s work demonstrated how to achieve this using deep neural networks [8].

Generative models and style transfer have expanded the scope of computer vision beyond traditional tasks, enabling the generation of creative content and artistic expression in the digital domain. They also have practical applications in data augmentation and image enhancement.

These trends represent the core developments in computer vision in recent times, reflecting both the foundational building blocks and the creative, innovative applications of the field. Ongoing research and developments continue to push the boundaries of what is possible in computer vision, offering exciting opportunities and challenges for the future. To stay current with the latest advancements, it is essential to follow recent literature and attend computer vision conferences and workshops.

2.2 State of the art for Neural Networks

Neural networks, particularly deep neural networks, have become the cornerstone of many artificial intelligence and machine learning applications. They have demonstrated remarkable success in various domains, including computer vision, natural language processing, and reinforcement learning. Here's an overview of some key developments and trends in neural networks:

2.2.1 Deep Learning Architectures

Deep learning architectures have played a pivotal role in the success of neural networks. They have evolved from shallow models to deep neural networks with numerous layers, enabling the extraction of complex features and representations. Here are some key developments:

- **Convolutional Neural Networks (CNNs)**: as mentioned before, CNNs are especially effective for image analysis and recognition, and therefore play a critical role in our field of computer vision. They leverage convolutional layers to automatically learn hierarchical features from images. This architecture has powered remarkable advancements

in computer vision, including image classification, object detection, and image segmentation [20].

- **Recurrent Neural Networks (RNNs)**: RNNs are designed to handle sequential data, making them ideal for tasks like natural language processing (NLP) and time series analysis. Variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have addressed the vanishing gradient problem and improved the modeling of long-range dependencies in sequences [12].

- **Transformers**: Transformers introduced a novel architecture based on self-attention mechanisms. They have had a profound impact on NLP tasks and have achieved state-of-the-art results in areas like language translation, sentiment analysis, and language understanding [41].

These architectures have not only advanced their respective domains but have also influenced the development of hybrid models that combine features from multiple architectures.

2.2.2 Transfer Learning

As aforementioned, transfer learning has revolutionized neural network training by allowing models pre-trained on large datasets to be fine-tuned for specific tasks. This approach significantly reduces the need for vast amounts of labeled data and training time. Two noteworthy developments are:

- **BERT (Bidirectional Encoder Representations from Transformers)**: BERT introduced bidirectional contextual embeddings, making it one of the most influential models in NLP. It achieved state-of-the-art performance on various NLP benchmarks and tasks [6].

- **GPT (Generative Pre-trained Transformer) Series**: Models like GPT-2 and GPT-3 demonstrated the power of massive-scale pre-training and fine-tuning for diverse natural language understanding tasks, including language generation, translation, and question answering. One of the most famous and prominent examples of these technologies nowadays is the chatGPT from OpenAI [29].

Transfer learning continues to be a driving force behind advancements in various domains beyond NLP, including computer vision and reinforcement learning.

2.2.3 Generative Models

Generative models aim to produce new data samples, making them integral to creative applications in neural networks. Two prominent classes of generative models, which we have already mentioned, are:

- **Generative Adversarial Networks (GANs)**: GANs consist of a generator and a discriminator network trained adversarially. They have been instrumental in image generation, style transfer, data augmentation, and the creation of deepfakes [11].
- **Variational Autoencoders (VAEs)**: VAEs are used for tasks like image generation, anomaly detection, and latent space exploration. They combine probabilistic modeling with neural networks to generate data samples [16].

These generative models have enabled creative content generation, artistic style transfer, and have practical applications in data augmentation and image enhancement.

2.2.4 Reinforcement Learning

Reinforcement learning (RL) combines neural networks with reinforcement learning algorithms to enable agents to learn optimal behavior through interaction with an environment. One of the most notable developments in RL is:

- **AlphaGo**: Developed by DeepMind, AlphaGo made headlines by defeating human world champions in the game of Go. It demonstrated the power of deep RL in mastering complex games, and AlphaGo's success has spurred research in RL and AI in general [34].

2.2.5 Ethical Considerations

As neural networks become more powerful, ethical considerations surrounding AI have gained prominence. Issues like bias, fairness, transparency, and accountability have come to the forefront. Research in this area is essential for responsible AI development and deployment.

- **"Algorithmic Bias Detectable in Amazon Delivery Service" by Gebru et al.**: This study highlights concerns related to bias and fairness in AI systems, emphasizing the importance of addressing ethical considerations in AI development [23].

These developments in neural networks underscore the transformative impact of deep learning on various domains, and they continue to shape the landscape of artificial intelligence and machine learning. Researchers and practitioners are continually pushing the boundaries of what neural networks can achieve, and staying up-to-date with the latest research is vital to remain at the forefront of this rapidly evolving field.

2.3 State of the art in Neural Rendering

Neural rendering techniques have advanced significantly, enabling the generation of highly realistic and controllable images and videos. These techniques bridge the gap between traditional computer graphics and data-driven approaches, allowing for more natural and flexible content creation.

2.3.1 Neural Scene Representation and Rendering

In the field of neural rendering, the representation of 3D scenes and objects is a fundamental challenge. Traditional computer graphics rely on explicit geometric and material models, while neural rendering seeks to learn these representations directly from data. This is the realm where we will focus in this project.

A key development is the concept of Neural Radiance Fields (NeRF), which introduces a groundbreaking approach for scene representation and rendering. The NeRF framework has opened up possibilities for realistic 3D content creation, virtual tourism, and augmented reality applications. We will further analyse and explain this model in the next section.

2.3.2 Neural Face and Body Rendering

Neural rendering techniques are not limited to scenes but have also been applied to human faces and bodies. These applications are particularly relevant in fields like digital entertainment and virtual communication:

- **DeepFake and FaceSwap**: DeepFake and FaceSwap techniques use deep learning models to manipulate and synthesize facial expressions and identities. They have gained popularity for their ability to convincingly swap faces in videos and create realistic facial animations. While they have raised concerns about the potential misuse for deceptive purposes, they also have legitimate uses in entertainment and visual effects.

- **Volumetric Capture**: Volumetric capture systems use arrays of cameras and depth sensors to capture detailed 3D data of humans or objects. Neural networks are employed to refine and render these volumetric models, resulting in highly detailed and realistic character animations. This technology has applications in the gaming industry, virtual try-ons in e-commerce, and teleconferencing.

2.3.3 Real-time and Interactive Rendering

Efforts have been made to bring neural rendering techniques into real-time and interactive contexts, enabling applications like virtual reality and augmented reality:

- **NeRF in the Wild**: Extending the NeRF framework to real-world and uncontrolled environments is an ongoing research area. Researchers aim to make neural rendering applicable to a broader range of scenes, including those with complex lighting, dynamic objects, and variable conditions. Additionally, optimizations are being explored to improve the rendering speed for real-time applications, allowing for interactive and immersive experiences [24].

Real-time neural rendering has the potential to transform fields such as gaming, virtual tourism, and training simulations.

2.3.4 Artistic and Style Transfer Rendering

Neural rendering techniques have also found artistic applications, allowing artists and creators to explore new frontiers in digital art:

- **StyleGAN and StyleGAN2**: StyleGAN models utilize generative adversarial networks (GANs) to synthesize high-quality images with specific styles. They have been employed for art generation, style transfer, and the creation of novel characters. Artists

and researchers have leveraged these models to generate visually stunning and unique artworks [14] [15].

These artistic applications demonstrate how neural rendering can unleash creativity and provide new tools for digital artists.

2.3.5 Ethical Considerations and Deepfakes

As neural rendering techniques, especially deepfakes, have become more sophisticated, they have raised significant ethical concerns. Deepfakes are AI-generated content that can convincingly depict real people saying or doing things they never did. The potential for misuse, misinformation, and privacy invasion is a critical issue.

Efforts to address the ethical challenges of neural rendering include:

- **Detection and Mitigation**: Researchers are actively working on developing methods to detect deepfake content and differentiate it from genuine content. These techniques employ various cues, including inconsistencies in facial movements, unnatural artifacts, and analysis of biometric features.

- **Regulation and Policy**: Policymakers are considering regulations and policies to address the potential harms of deepfakes, including the misuse of deepfake technology for malicious purposes. Striking a balance between freedom of expression and responsible use is a complex challenge.

- **Awareness and Education**: Raising awareness about the existence and capabilities of deepfakes is essential. Educating the public about the potential for manipulation through AI-generated content helps individuals make informed judgments.

The ongoing discussion about deepfakes highlights the importance of responsible AI development, media literacy, and safeguards against malicious uses of neural rendering technology.

As a conclusion, neural rendering is a dynamic and rapidly evolving field with profound implications for computer graphics, entertainment, virtual reality, and beyond. While it presents exciting opportunities for creativity and innovation, it also poses complex ethical and security challenges that require ongoing research and responsible use. To stay current in this field, it

is crucial to follow the latest research developments and ethical discussions related to neural rendering.

2.4 NeRF (Neural Radiance Fields)

Neural Radiance Fields (NeRF) is a breakthrough technique in the field of computer graphics and neural rendering that has had a significant impact on how we represent and render complex scenes. NeRF was introduced in a paper titled "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis" by Ben Mildenhall et al. in 2020. It addresses the problem of reconstructing 3D scenes and rendering novel views of those scenes from a limited number of 2D images.

Traditional methods of scene representation and rendering often involve constructing 3D models from multiple images using techniques like photogrammetry or building meshes from depth sensors. However, these approaches can be computationally expensive, require careful calibration, and might struggle with capturing intricate details and complex lighting effects. NeRF offers a revolutionary alternative by using neural networks to directly model the volumetric scene's appearance and structure.

Here's the impact NeRF has had on neural rendering and scene representation:

- **Continuous Scene Representation**: NeRF introduced the concept of representing a scene as a continuous function that takes 3D coordinates as input and predicts the volumetric radiance (color and intensity) at those coordinates. This is in contrast to discrete representations like point clouds or voxel grids. This continuous representation allows for more precise modeling of scene geometry and appearance.

- **Novel View Synthesis**: One of the most impressive applications of NeRF is its ability to synthesize novel views of a scene from arbitrary camera angles. Once trained on a set of images capturing a scene from different viewpoints, NeRF can render high-quality images from any new viewpoint without the need for traditional rendering pipelines. This capability has broad implications for virtual reality, gaming, special effects, and architectural visualization.

- **Data Efficiency**: NeRF can achieve impressive results with relatively few images, making it appealing for scenarios where obtaining a large number of training views might be

challenging. Traditional 3D reconstruction techniques often require dense image data from multiple angles to produce accurate results, but NeRF can work well with a smaller dataset.

- **Handling Complex Lighting**: NeRF can handle complex lighting effects and materials, including shadows, reflections, and refractions. Traditional rendering methods often struggle to accurately capture these effects, but NeRF's neural network can learn to model these intricacies directly from the training data.

- **Challenges and Variants**: While NeRF brought about a significant advancement, it also introduced challenges such as training and inference speed due to the neural network's complexity. Researchers have since proposed various extensions and improvements, such as NeRF in the wild (handling uncontrolled images), NeRF++, and more efficient versions like PlenOctrees and Neural Scene Graphs.

- **Inspiration for Further Research**: NeRF has inspired researchers to explore novel ways of scene representation and rendering using neural networks. Variations of the original NeRF model have been developed to address limitations and improve efficiency, leading to a rich landscape of research in the broader field of neural rendering.

In summary, Neural Radiance Fields (NeRF) have had a profound impact on the field of neural rendering and scene representation by introducing a novel approach that allows for more accurate, detailed, and flexible scene modeling and rendering. Its innovative ideas have sparked further research and advancements in this exciting area of computer graphics.

The algorithm utilized employs a fully-connected deep network, which is non-convolutional in nature. The input to this network is a singular continuous 5D coordinate, comprising of spatial location (x, y, z) and viewing direction (θ, φ). The output of this network is the volume density and view-dependent emitted radiance (RGB colour) at the aforementioned spatial location.

The process of synthesizing views involves querying 5D coordinates along camera rays and utilizing traditional volume rendering techniques to project the resulting colors and densities into an image. As volume rendering is inherently differentiable, the sole input necessary for optimizing our representation is a collection of images with established camera poses. They

take advantage of the effective optimization of neural radiance fields for the purpose of rendering photorealistic novel views of complex scenes with intricate geometry and appearance. Furthermore, they present results that surpass previous achievements in the fields of neural rendering and view synthesis. [25]

However, one of the big problems with these models is that they take long times in training. The training process for a single scene for a NeRF model can range from hours to days. In contrast to this, one of the most important features of DirectVoxGo is the velocity of its training process.

With DirectVoxGo, training from scratch under 15 minutes for a single scene is achieved using a single GPU. This is done thanks to the use of voxels and the introduction of two simple yet non-trivial specific techniques. The first one would be the use of post-activation interpolation on voxel density, and the second one would be the robustification of the optimization process by imposing several priors. [35]

2.5 DirectVoxGo

Although the analysis of the inner workings of the DirectVoxGo model is not the main objective of this project, it is relevant to have a rough idea of the structure and steps.

In this model, the concept of voxel is used. A voxel is a three-dimensional (3D) pixel. It is the smallest unit of volume in a 3D grid and they are often used in computer graphics to represent 3D objects. Here, the representation of the scenes is divided into two voxel grids: one for the density and another one focused on the rest of features (mainly, view-dependant colour).

For the density voxel grid, two priors are introduced in order to achieve comparable quality with the NeRF process. First, densities are initialized to values very close to zero everywhere to avoid the geometry solutions being biased toward the cameras' near planes. Second, voxels that are visible to a higher number of views are given a higher learning rate than those that appear less in our sample views. It is shown in the study that the proposed solutions can successfully avoid the suboptimal geometry and work well on the five datasets. A process of post-activation is later introduced, achieving a successful capability of modeling sharp surfaces.

For the feature representation, which only happens in the fine detail phase, both an specific

feature voxel grid and a shallow MLP (multi-layer perceptron with only two hidden layers with 128 channels) are used forming a hybrid model of representation.

In general, the whole process is divided into two phases:

- **Coarse geometry searching:** as scenes are usually dominated by free space, this phase is used in order to find the 3D areas of interest in our representation, and have a coarse idea of their structure before getting to the fine and view-dependant detailing. With this, they achieve to greatly reduced the number of queried points in each ray in the later fine stage. Both of the aforementioned priors for the density voxel grid are applied in this stage, and later the regular scene representation with point sampling and interpolation is performed.

- **Fine detail reconstruction:** now that we have the optimized coarse geometry voxel grid from the previous phase, we can focus on a smaller subspace to reconstruct the surface details and view-dependant effects. The optimized density voxel grid is frozen at this stage. In this stage, a higher-resolution density voxel grid with post-activated interpolation is used. To model view-dependent color emission, we opt to use an explicit-implicit hybrid representation that comprises i) a feature voxel grid and ii) a shallow multi-layer perceptron (MLP) parameterized by the angle. The point sampling strategy is similar as before with some modifications. One of these main changes is that the known free space from the coarse density voxel grid is now skipped at sampling for our new voxel grids.

Having a rough idea about these two phases will be important to better understand the hyperparameters we will use in our testing.

With this process, DirectVoxGo achieves:

- Convergence speeds about two orders of magnitude faster than NeRF, reducing training time from 10−20 hours to 15 minutes using a single NVIDIA RTX 2080 Ti GPU.

- Visual representation quality comparable to NeRF at a rendering speed about 45x faster.

- No need for cross-scene pretraining.

- Grid resolution considerably smaller than in previous work.

Having fast convergence without quality lose in neural scene representation can have a big impact in systems that require high volumes of rendering and, potentially, real-time rendering. Additionally, it can have an effect in further research about the topic, as it is possible to perform more tests in a shorter amount of time.

3 Technologies and Methodology

3.1 Technologies used

The project leverages the following technologies to accomplish its goals:

- Python: the primary programming language used for implementing the DirectVoxGo model, conducting experiments, and data analysis [40].
- PyCharm: IDE used for programming in the Python language [13].
- PyTorch: the deep learning framework employed to build, train, and evaluate the DirectVoxGo neural rendering model [27].
- WSL (Windows Subsystem for Linux): the environment used for seamless integration between Windows and Linux, facilitating efficient experimentation and resource management. A Linux bash script has also been used for the repetition of executions in order to properly run the necessary tests.
- LaTeX: text composition system used for writing the report of this thesis [18].
- Overleaf: online software used for writing in the LaTeX language [3].
- GitHub: repository with all the generated code and asset can be found here[1] [9].

By combining the power of these technologies, this project aims to provide valuable insights into optimizing DirectVoxGo's hyperparameters and advancing the field of neural rendering.

[1] https://github.com/OrangelLuke/DirectVoxGO

3.2 High-level working of the code

For the code used for our testing, we used the original implementation of the model DirectVoxGo as a base [36]. This code is mainly in the language Python, and in usage of both Pytorch and CUDA technologies. The main modifications were made over the file *run.py*.

The first thing we did was getting most of the code for the execution out of the main function, and into a new separate function called *execute_everything*. This way, in only one execution of the main we could perform several tests by calling this function multiple times. Later, we created three other useful functions:

- ***load_results***: loads the results into a json file using dictionaries and the json library.
- ***measure_memory_usage***: measures the memory usage using the library psutil.
- ***check_value_is_done***: checks if a value has already been tested before proceeding to the execution. If the value has not been done, the execution takes place; otherwise, we go to the next value in the list.

Inside the main function, we have the rest of the code needed for executing the tests and the extraction of the measurements. The quality measurements are already provided by the original code, and the time measurement is extracted using the time library from Python. In case of an exception, we retry the execution a total of three times before aborting the main execution altogether.

As this limit is sometimes reached, and other type of errors that directly kill the program happen sometimes, we created a Linux bash script that allows us to rerun the command that kickstarts the main execution multiple times without us interceding. Thanks to our function that checks if a value has already been tested by looking at the results json file, when we start the main execution again, we will start where we last left.

To run the testing with the combinations of parameters, we duplicated the modified *run.py* into a file called *runCombo.py*, and adapted the code to the needs of multi-hyperparameter testing, while keeping the semantics of the program intact.

For the generation of the graphs, we use specific python scripts that automatically read the results from the json files and generate and export the graphs through the matplotlib library.

For the finding of Pareto optimal results, we take advantage of the oalib from the OApackage python package [7].

3.3 Problems encountered

The implementation of the DirectVoxGo model heavily relies on both the Pytorch framework and Nvidia's CUDA acceleration software. This directly implies certain difficulty due to the complexity of their installation. The versions for these two softwares need to match, as not all versions are compatible amongst them. At the same time, these versions need to be compatible with the graphics hardware in your computer. Achieving this has posed a great problem, as the newer software versions that had been used when designing the implementation of the model were not compatible with the older (and less powerful) hardware that we had at hand. When we finally seemed to reach a configuration that worked, we encountered that the hardware we had was not powerful enough to perform the testing in reasonable amounts of time, and when we tried to change into a server with a higher computational capability, the installation and versions problems resurfaced and persisted.

The final solution was the acquisition of newer and more powerful hardware (a new gaming laptop), closer to the technology used by the original developers. This ended up fixing our issues, but these had already taken months of the time dedicated to this project. This, added to the intrinsic dilated length of the project due to the long times the experimentation take, posed a serious setback in the realisation of this thesis.

3.4 Hyperparameters selection

The DirectVoxGo model has an extensive list of hyperparameters to determine its working, having a file with more than 100 lines of text in its implementation to define them all. Due to the extension of this project, we will only select a limited amount of them to include in our experimentation. For this, we will try to choose those that have a higher likelihood to have a larger impact on the results that we will measure, based on the semantics of the model we have explained before.

As we have mentioned, we have two distinct phases in the execution: coarse and fine phases. We will choose hyperparameters from both phases, so we can see what impact each

of these have on the results. If we look at the divisions within the list of hyperparameters, we can see that each of these phases is subdivided again in two: "train" and "model and render". We will also try to select hyperparameters from these two sections more or less evenly.

With all this in mind, we start by choosing the number of iterations in the training of both the coarse and fine phases. This is due to the fact that the number of iterations tends to have a huge impact in the quality of the result and, specially, in the time consumption of most artificial intelligence algorithms. In the training, but this time being a common value for both phases, we also have the hyperparameter for the number of random rays per optimization step (also called batch size). Knowing the high-level working of the model, and how these rays are surveyed to generate the result, this hyperparameter seems to also have a relevant impact in our result.

In terms of "model and render", we will chose the number of voxels as our hyperparameter, due to its likely impact in quality and memory. We can see that we have two values related to this, one being the expected number of voxels, and another one called "num_voxels_base", used to rescale delta distances. As these values are always the same in both phases for the default combination, we will also modify them concurrently to maintain the semantics of the model. Thus, we will consider these two hyperparameters as one.

Therefore, the 5 hyperparameter we will use in our experimentation will be:

- Number of iterations in coarse training
- Number of random rays per optimization step
- Number of iterations in fine training
- Number of voxels in coarse rendering
- Number of voxels in fine rendering

This selection does not imply that the rest of hyperparameters do not have a relevant impact on the results, but due to the limited scope of our study, we will stick with these ones and leave the rest as future lines of research.

3.5 Measurement criteria

Image quality measurements, such as PSNR, LPIPS, and SSIM, are used to assess the fidelity and similarity between reference (original) and processed (enhanced or distorted) images. Each metric captures different aspects of image quality, providing valuable information for evaluating the performance of image processing algorithms.

1. **Peak Signal-to-Noise Ratio (PSNR)**: PSNR is a widely used metric to quantify the quality of an image by measuring the difference between the reference image and the processed image in terms of signal power and noise. It calculates the ratio of the peak signal power to the mean squared error (MSE) between the two images on a logarithmic scale (usually in decibels). A higher PSNR value indicates a smaller difference between the images and, consequently, a higher quality image. While the PSNR is a straightforward calculation, it has limitations in terms of perceptual quality evaluation, as it does not fully account for human visual perception [26].

2. **Learned Perceptual Image Patch Similarity (LPIPS)**: LPIPS is a perceptual image quality metric that uses deep neural networks to learn image similarity. It evaluates the perceptual difference between images by comparing the feature representations of patches extracted from the reference and processed images. LPIPS takes into account human visual perception, making it a better indicator of image quality from a subjective standpoint. It is particularly advantageous when evaluating the quality of images that have undergone intricate transformations, such as image style transfer or image generation with deep learning models. The degree of perceptual similarity between images is inversely proportional to the LPIPS score [19].

3. **Structural Similarity Index (SSIM)**: SSIM is another perceptual metric that assesses the structural similarity between the reference and processed images. It considers three key components of human visual perception: luminance, contrast, and structure. The similarity index computed by SSIM is based on these components and ranges from -1 to 1, with a value of 1 indicating a perfect match. A higher SSIM value indicates a greater similarity between the images and superior quality. SSIM is extensively employed in

image compression and restoration tasks, as it closely corresponds to human visual perception [5].

In summary, PSNR is a classic metric based on pixel-wise differences, while LPIPS and SSIM are perceptual metrics that take human visual perception into account. When evaluating image quality, it's often beneficial to use a combination of these metrics to gain a comprehensive understanding of how an image processing algorithm performs.

Apart from these quality measurements, we will also measure the time and memory consumed in each execution, and show graphically these distributions.

With this information, we will later create six graphs with the ratios between quality and time and memory usage for all experiments. The graphs will be as follow: PSNR vs. time, SSIM vs. time, LPIPS vs. time, PSNR vs. memory, SSIM vs. memory, and LPIPS vs. memory. With the data in these graphs, we will attempt to find the Pareto optimal values in each of the configurations.

4 Single Hyperparameter Experimentation (phase I)

4.1 Single Hyperparameter Experimentation

In the first round of experimentation, we decide to test several values of each of the five chosen hyperparameters. Despite the long times that each of the tests take, we decide to try five different values for each hyperparameter, in order for the results to have certain validity and show relevant data and progression. This resulted in 25 different tests in this first phase. For each test, as aforementioned, we measure the time and memory used in the execution, as well as the three metrics of quality: PSNR, LPIPS and SSIM.

In each execution, we only modify one of the studied hyperparameters, and leave the rest with their default configuration. The goal of this is to figure out what effect each of the hyperparameter has on our measurements, with the intention of having a notion of how to better combine them in the future.

We keep the original logs that the implementation of the model generates, but we also export the most significant data into a JSON file to help us in the posterior processing of this information. The complete logs and files can be consulted in the GitHub repository of this project.

The obtained results are as follow.

4.1.1 Number of iterations in coarse training

For this hyperparameter, we tested the values: 500, 1000, 5000 (default), 10000, 20000.

In terms of the quality metrics, we obtained the following results:

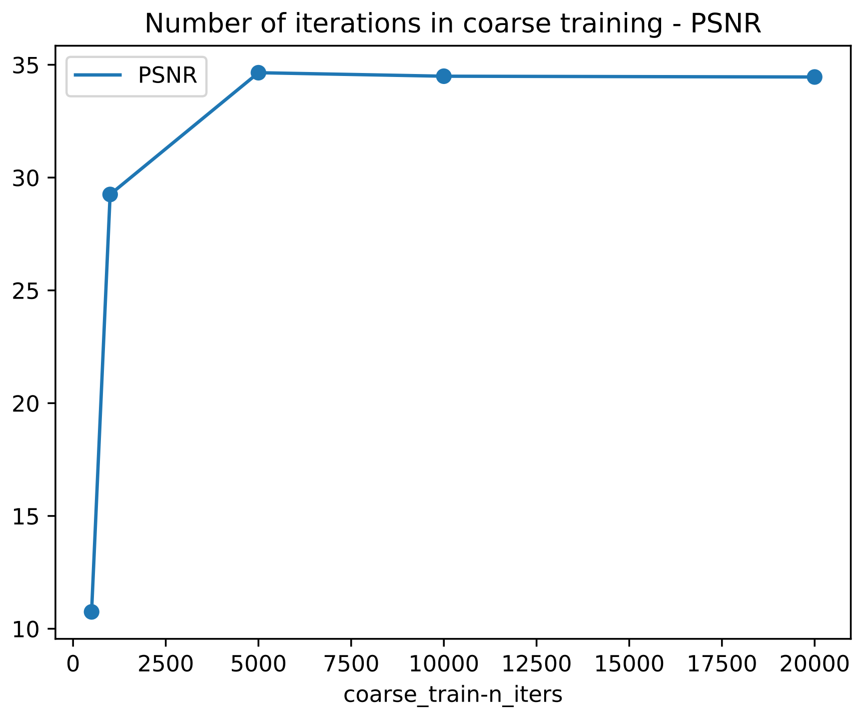

Figure 1: Number of iterations in coarse training - PSNR plot

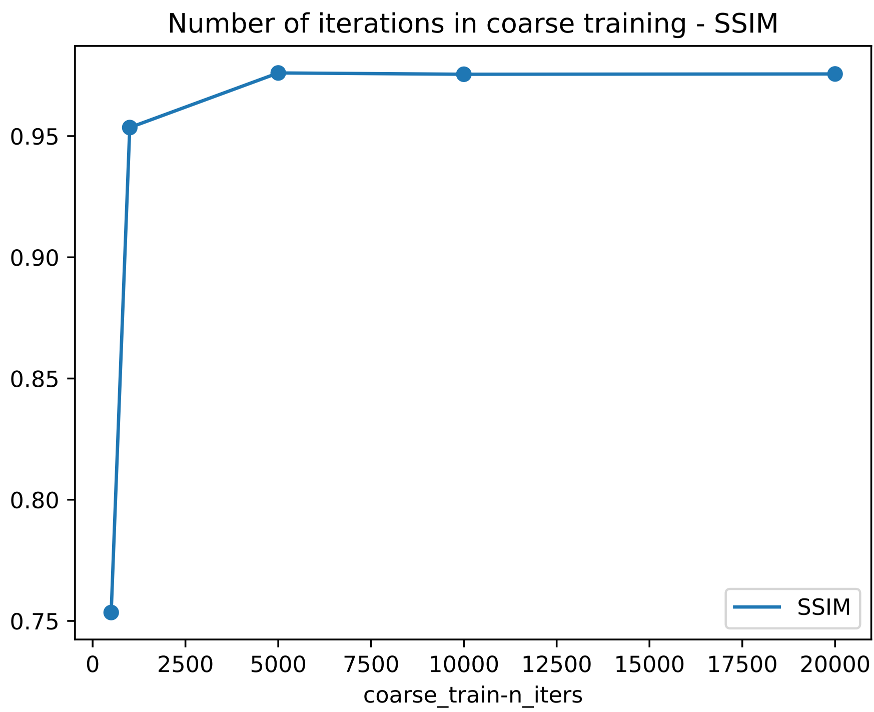

Figure 2: Number of iterations in coarse training - SSIM plot

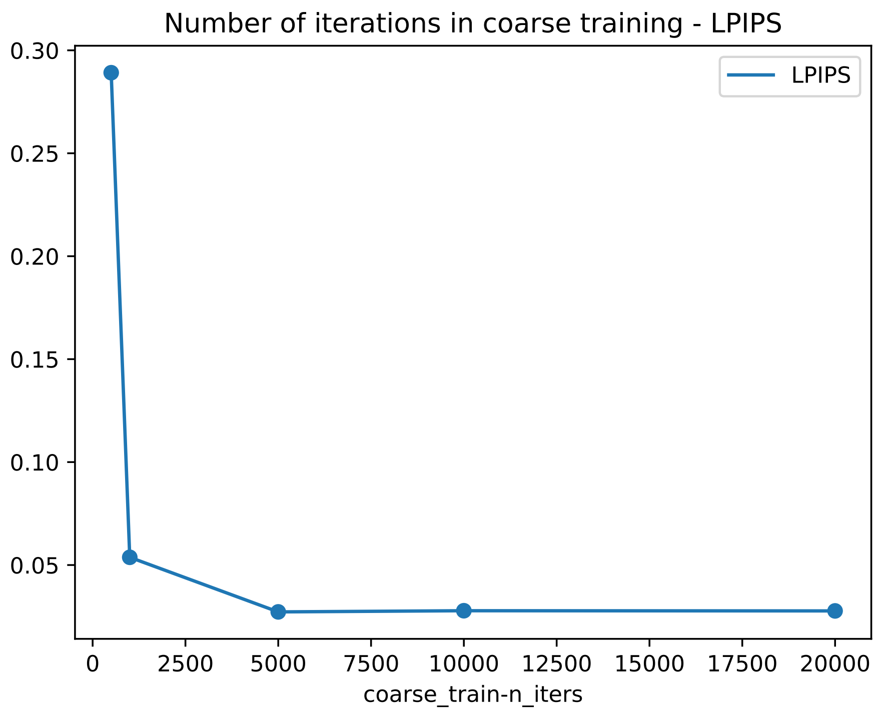

Figure 3: Number of iterations in coarse training - LPIPS plot

In terms of quality, the results are clear. The value that performs best is 5000 (default), which is also where the quality improvement stabilizes. Higher values barely improve the quality of the result, and lower values have a negative impact on it. The value 1000 could be seen as a compromise value, as it it is slightly worse in quality than the default, but a good performance in time and/or memory consumption could make up for that.

In terms of time taken in the execution, the result is this:

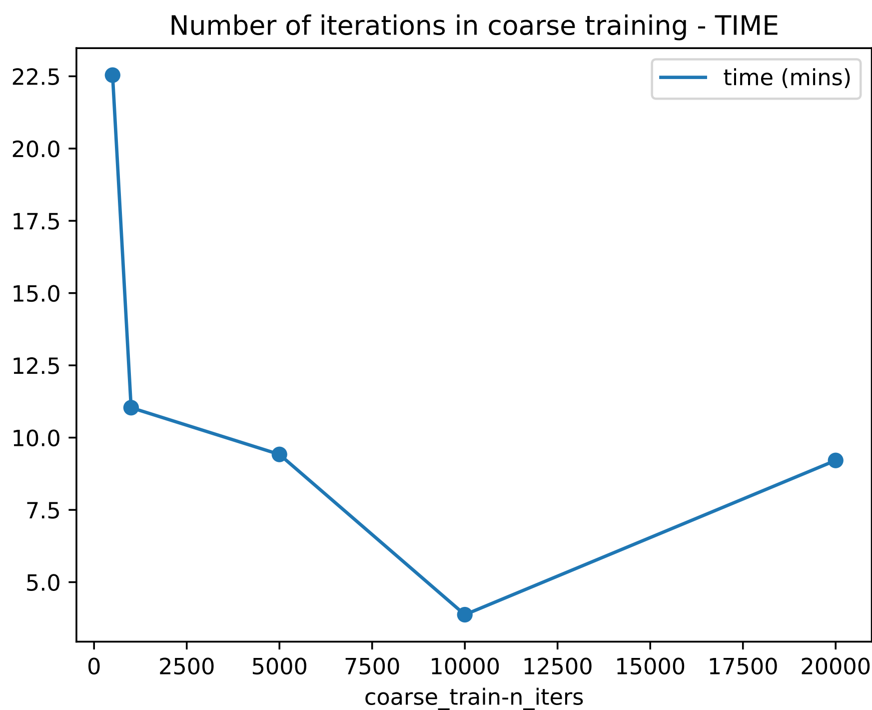

Figure 4: Number of iterations in coarse training - TIME plot

Rather counter-intuitively, the best performer in terms of time is the value 10000, better than the default (5000), which performs as good as the highest value (20000). This is interesting as 10000 also has a high quality value. The suspected compromise value (1000) here tests second worst.

Finally, in terms of memory usage:

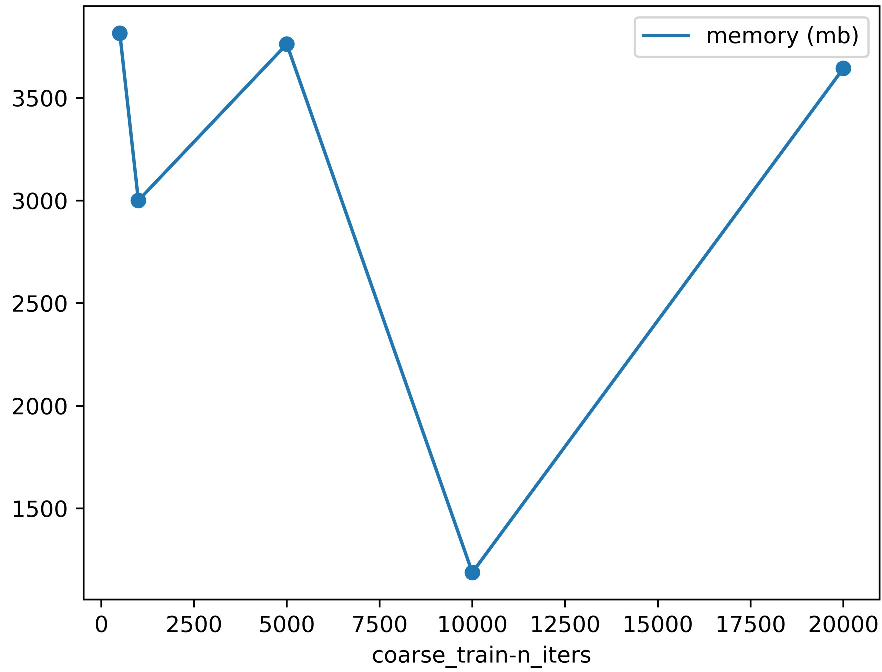

Figure 5: Number of iterations in coarse training - MEMORY plot

Visually, we can see a huge drop in memory usage for the value 10000, which does not seem to be coherent with the rest of data. Given the possibility that this could be an outlier, we retake the testing with said value, obtaining the following updated plot:

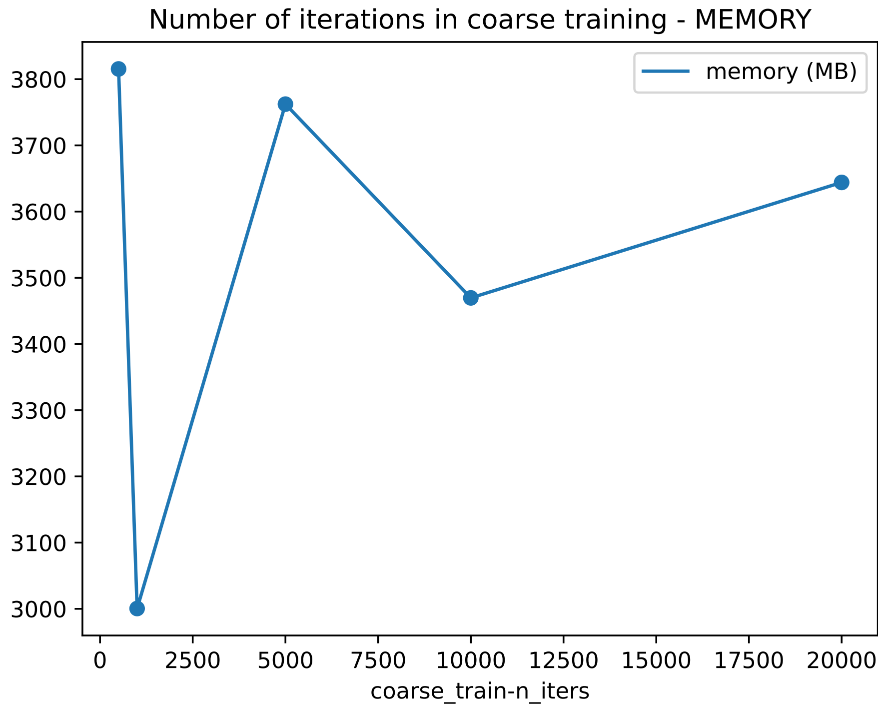

Figure 6: Number of iterations in coarse training - MEMORY updated plot

Now we can see, paying special attention to the scale in the Y-axis, that the former value was an outlier in the execution, and now the data is more coherent. This outlier could have been caused by cache or other type of temporal data from a former test that had been used in the new execution without being taken into account. From now on, we will use this updated data (instead of the old one with the outlier).

Although the differences are not very significant, the value that performs best in terms of memory usage would be 1000, below the default 5000 which tests the second highest. The winner of the time measurements, 10000, has the second lowest memory consumption.

It is hard to tell which of these values is the most optimal for this hyperparameter, but we can conclude that it can be either 1000, 5000 or 10000, depending on what we value the most (either quality, time or memory), thus discarding both 500 and 20000.

4.1.2 Number of random rays per optimization step

For this hyperparameter, the chosen values have been: 2048, 4096, 8192 (default), 16384, 32768.

In terms of the quality metrics, we obtained these results:

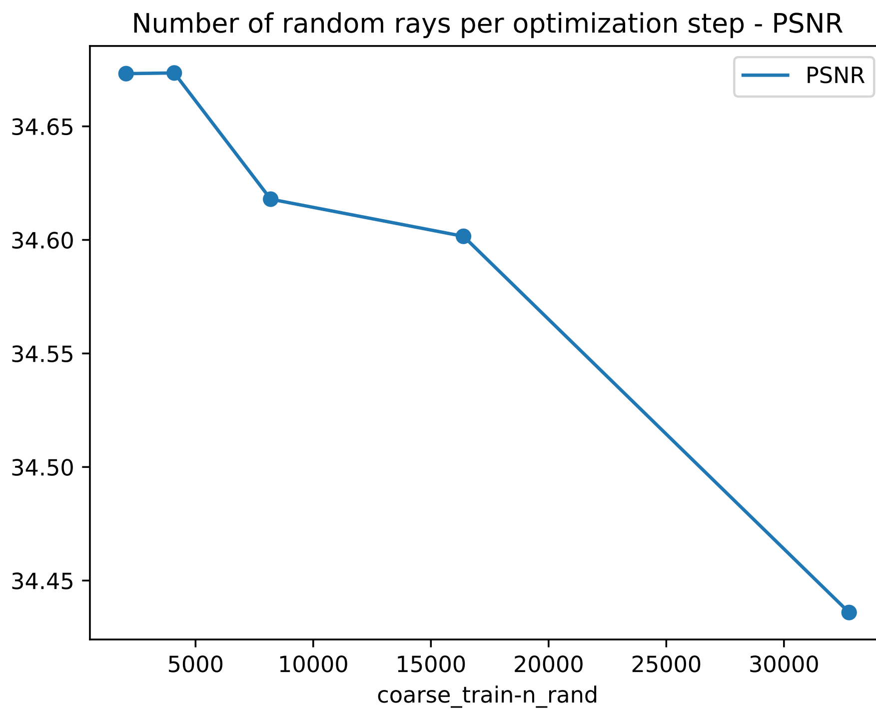

Figure 7: Number of random rays per optimization step - PSNR plot

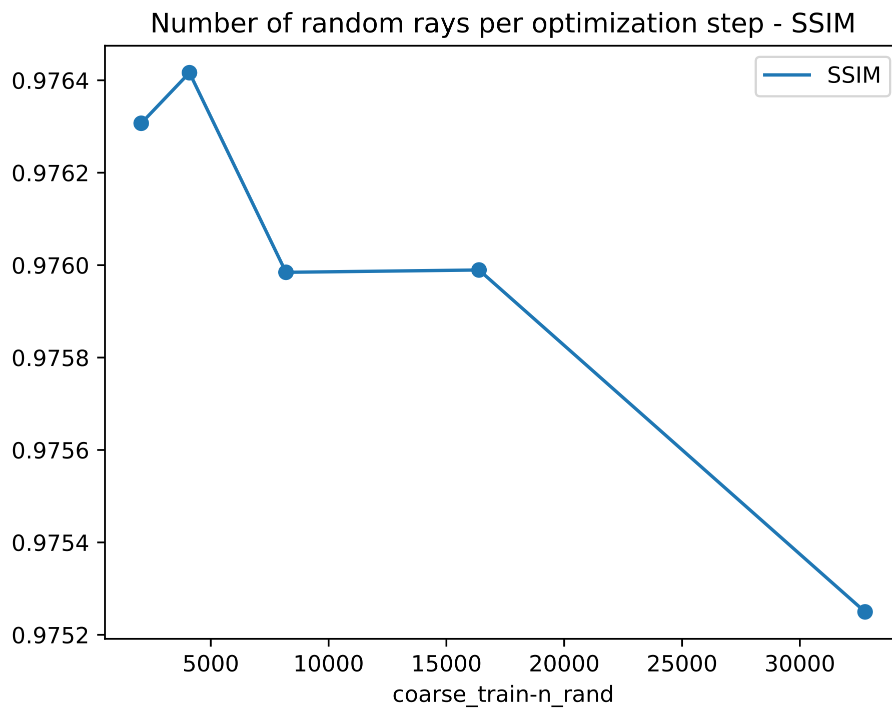

Figure 8: Number of random rays per optimization step - SSIM plot

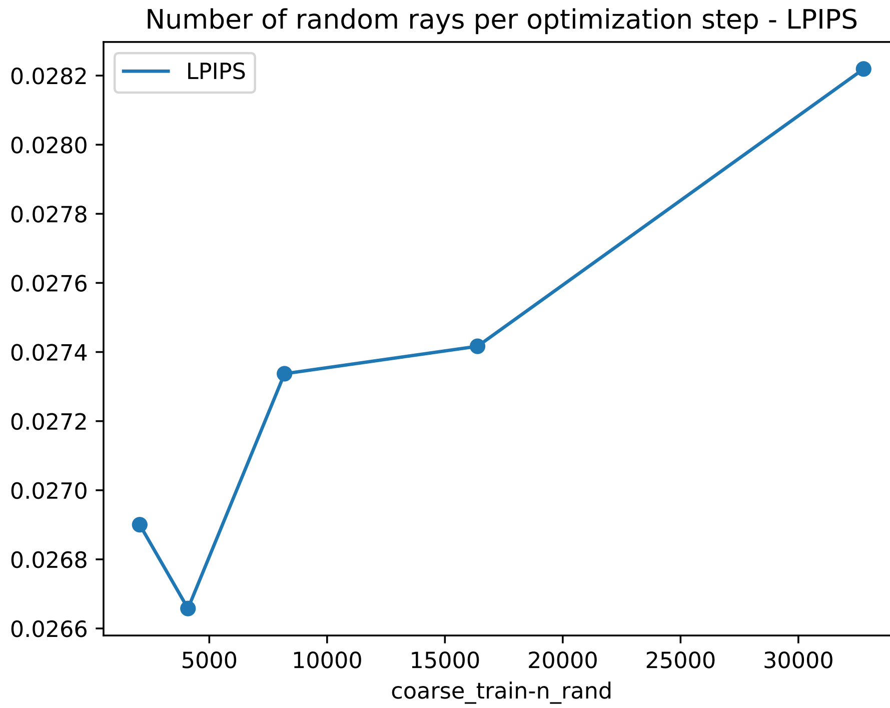

Figure 9: Number of random rays per optimization step - LPIPS plot

In these plots, it is clear to see that the best performing value in the three cases is 4096, closely followed by 2048, above the default value (8192). Additionally, the general tendency is that when we increase this hyperparameter above that optimal value, the quality decreases. We can also observe a valley in the values 8192 and 16384, achieving both similar quality.

When measuring the execution times, the following results are obtained:

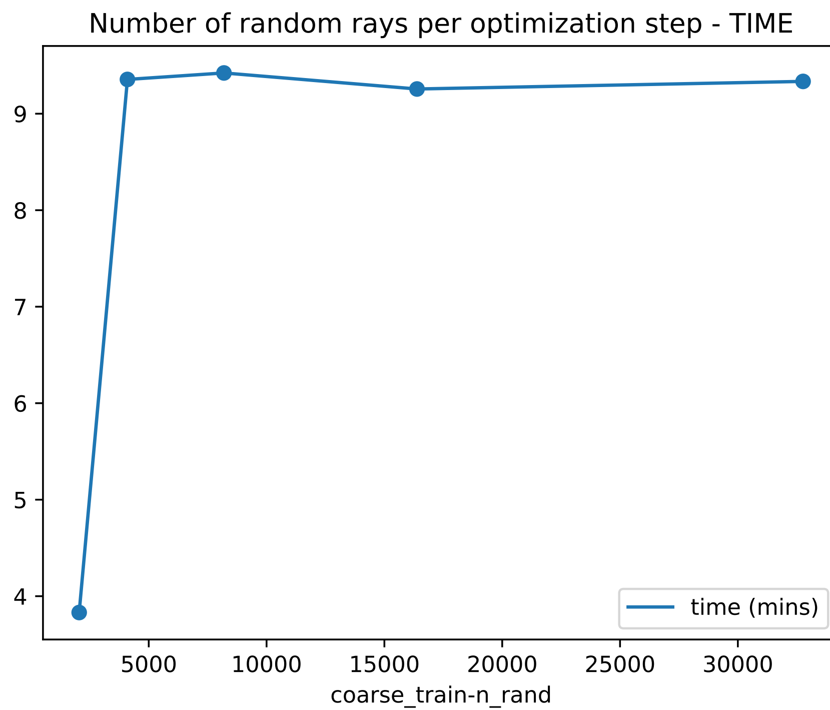

Figure 10: Number of random rays per optimization step - TIME plot

In consonance with the quality results, the best value among the tested is 2048. It is notable that it seems that above the value 4096, the time measurement stabilizes in a bit above the 9 minutes mark.

Finally, considering the memory usage, the results are:

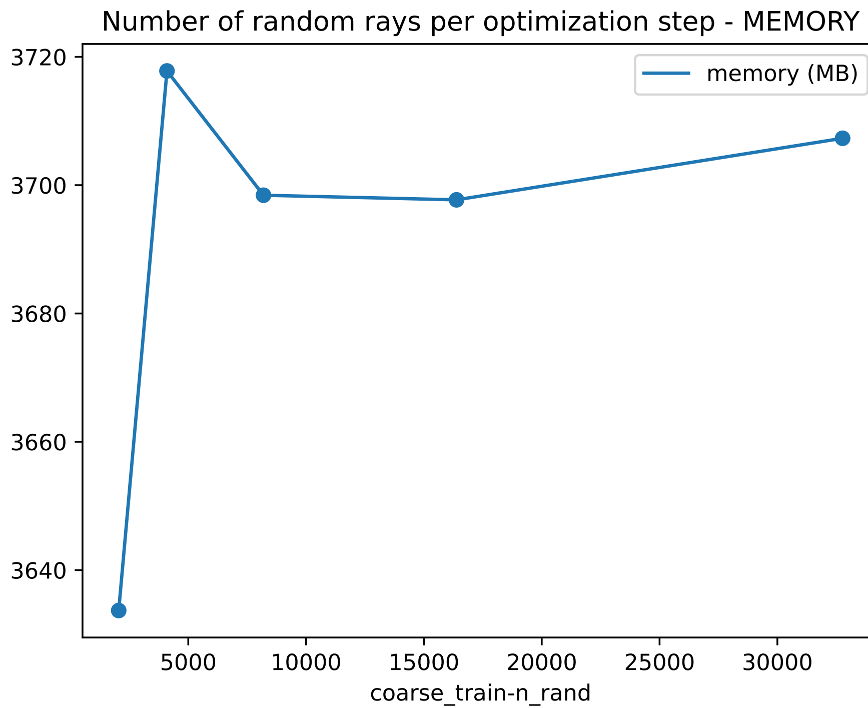

Figure 11: Number of random rays per optimization step - MEMORY plot

In this case, the best result is achieved, as before, by 2048. The value 4096, that had decent in terms of quality and time, here achieves the worst value.

In this case, the best achieving hyperparameter is clearly 2048, as it performs best in all metrics. Behind that, we can also consider the default value, 4096, as it is arguably the second best-balanced result.

4.1.3 Number of iterations in fine training

For this hyperparameter, the chosen values were: 1000, 5000, 20000 (default), 40000, 80000.

When measuring the quality of the output, we obtained the following results:

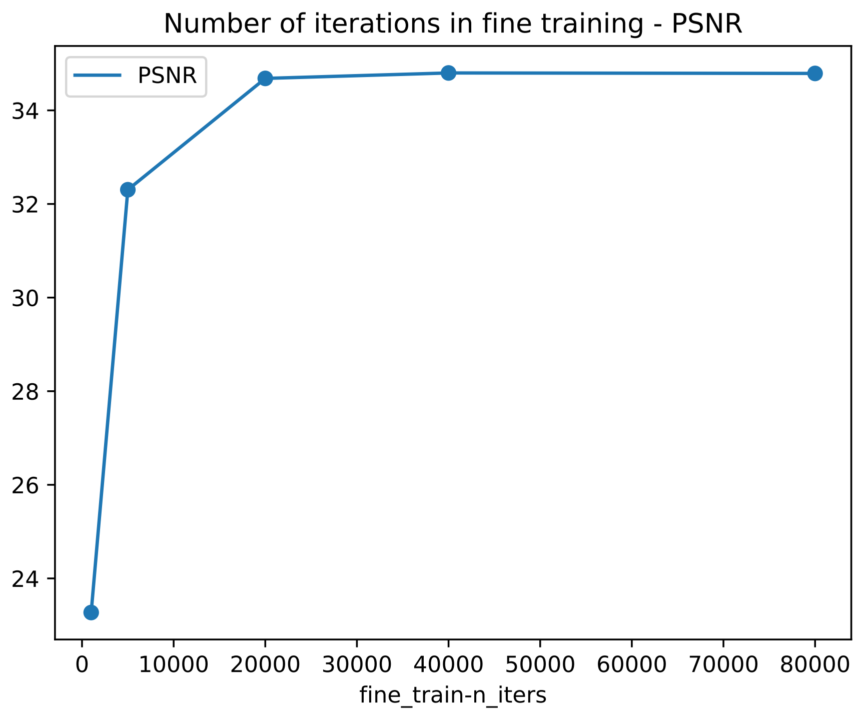

Figure 12: Number of iterations in fine training - PSNR plot

Figure 13: Number of iterations in fine training - SSIM plot

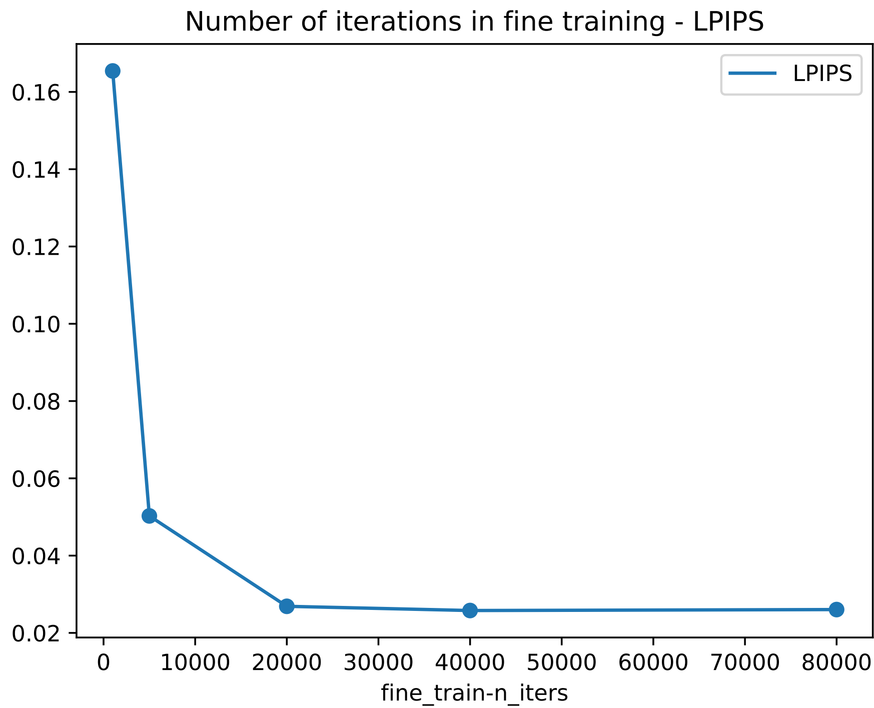

Figure 14: Number of iterations in fine training - LPIPS plot

The results show that the quality stabilizes as its highest in the default value (20000). Lower values negatively affect the quality (in different measure), and highest values barely make the results better. As in the coarse phase, a compromise value could be the immediate lower, 5000, depending on its performance in the rest of measurements.

Regarding time measurements, the result is as follows:

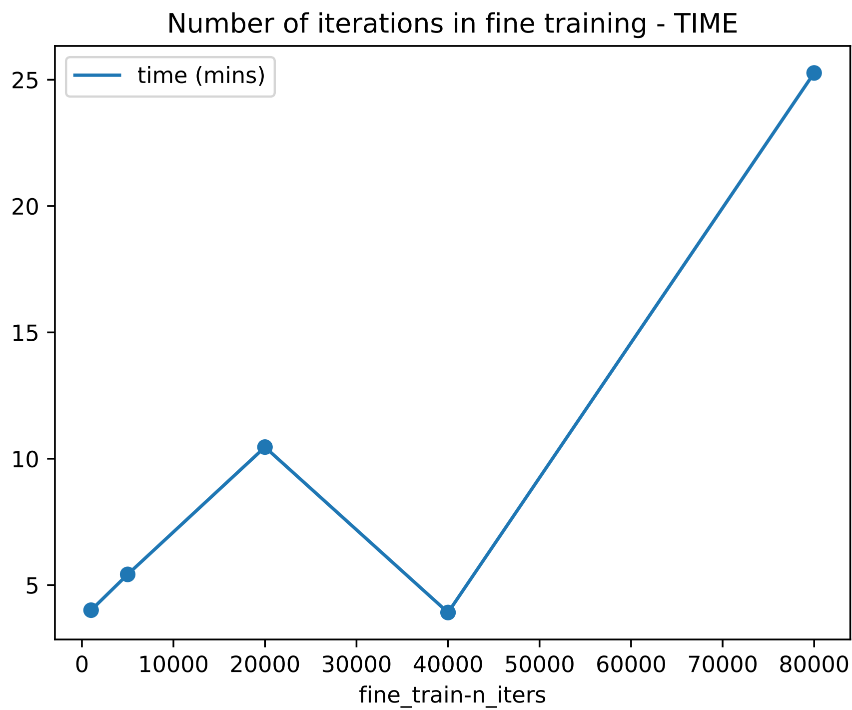

Figure 15: Number of iterations in fine training - TIME plot

The lowest (and therefore better) values are the result of values 1000 and 40000. The worst value is the highest number of iterations, as expected; but the default value, which tested high in the quality measures, gives a surprising worse results than values above it. Here we can see that our comprise value, 5000, offers a better result than the default, giving insight that we could find a more balanced optimal result with it.

Lastly, regarding memory usage:

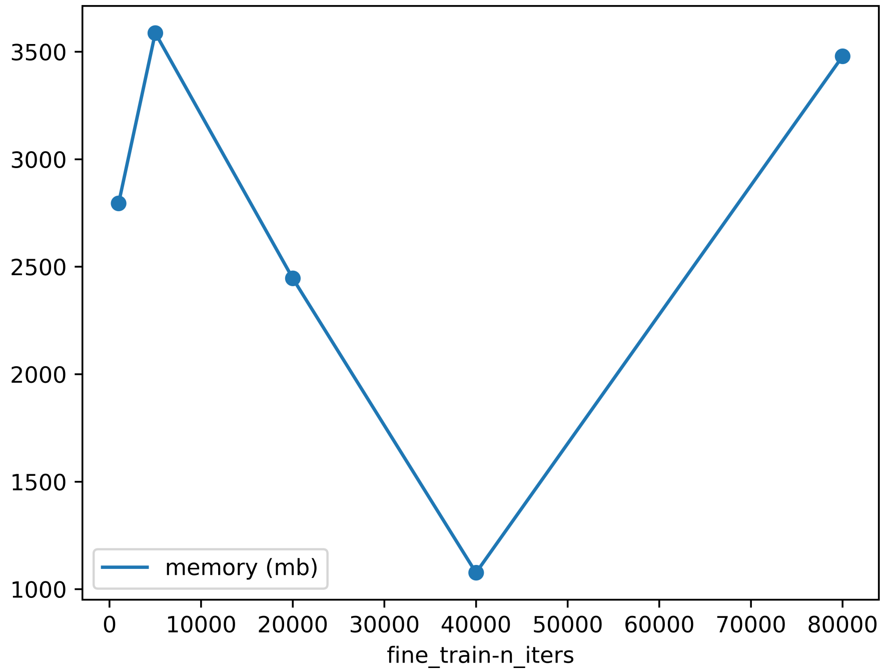

Figure 16: Number of iterations in fine training - MEMORY plot

Visually, we can see a significant drop in memory usage for the value 40000, not seeming to be coherent with the rest of data. Given the possibility that this could be an outlier, we retake the testing with said value, obtaining the following updated plot:

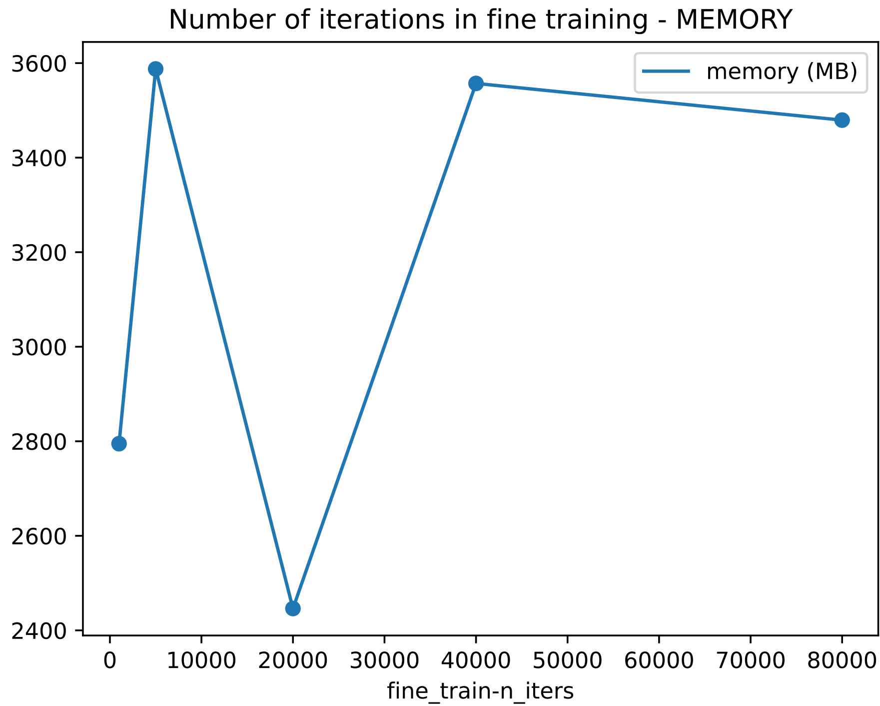

Figure 17: Number of iterations in fine training - MEMORY updated plot

Opposite to before, the measures are closer among them (special attention to the Y-axis values) and more reasonable. From now on, we will use this updated data (instead of the old one with the outlier).

The best result is achieved by the default value for this hyperparameter (20000), and our predicted compromise value achieves the worst result. It is because of this that we can see that the optimal value for this hyperparameter would be the default one (20000), given its balance in all three tests.

4.1.4 Number of voxels in coarse rendering

As aforementioned, as the the value for the hyperparameters of *num_voxels* and *num_voxels_base* in the default settings is the same in all cases, we have kept it equal as well. Considering this, the values selected for these two hyperparameters have been: 256000, 512000, 1024000 (default), 2048000, 4096000.

Regarding quality, we obtain:

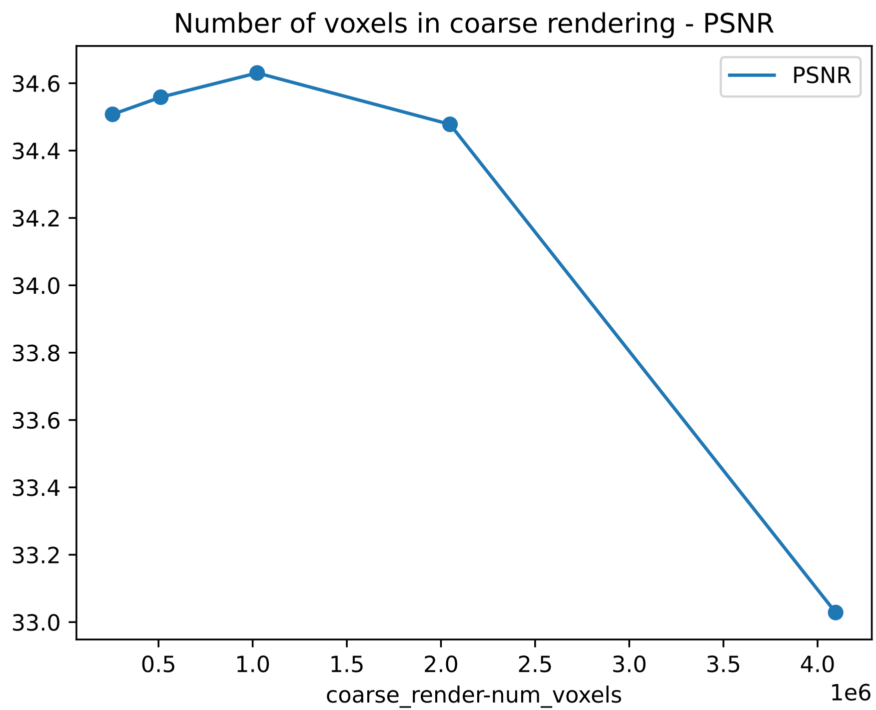

Figure 18: Number of voxels in coarse rendering - PSNR plot

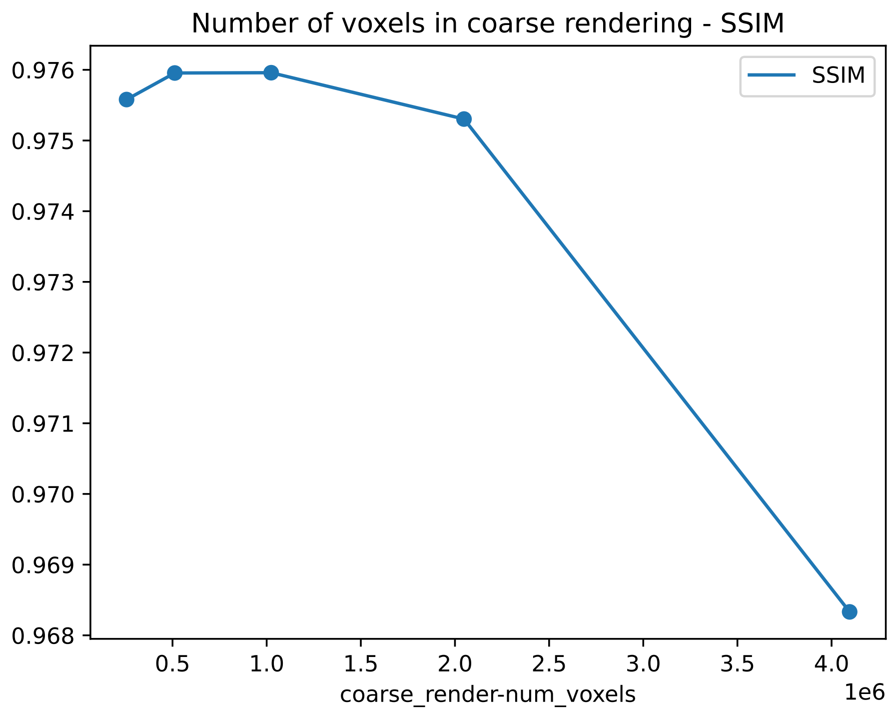

Figure 19: Number of voxels in coarse rendering - SSIM plot

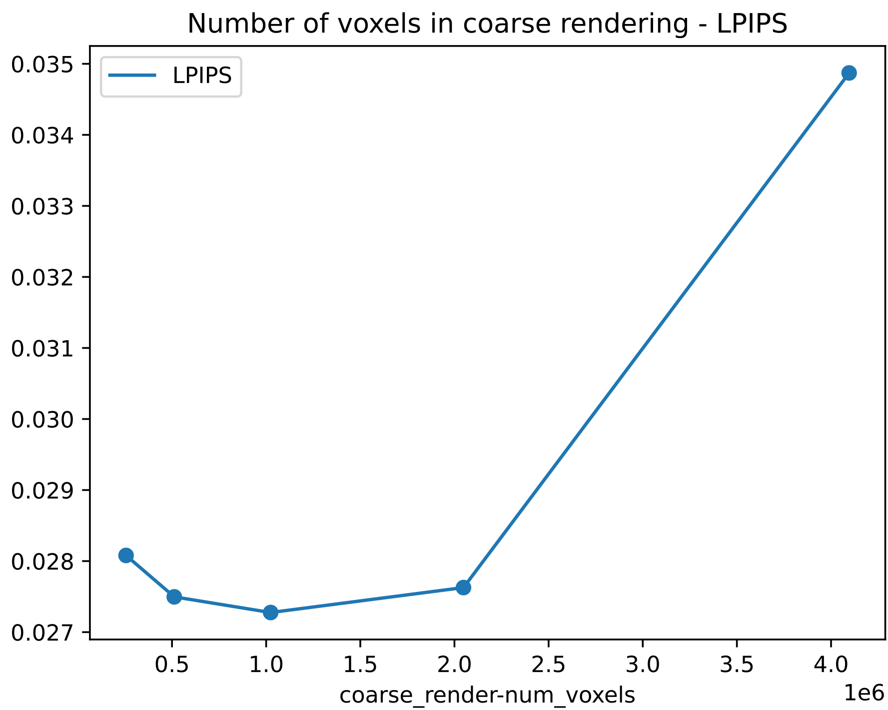

Figure 20: Number of voxels in coarse rendering - LPIPS plot

We observe that, although the first 3-4 values are rather stable, with little difference among them, we always peak at the default value (1024000). When we increase considerably this value, we start to lose quality.

In terms of time consumption, we find the following:

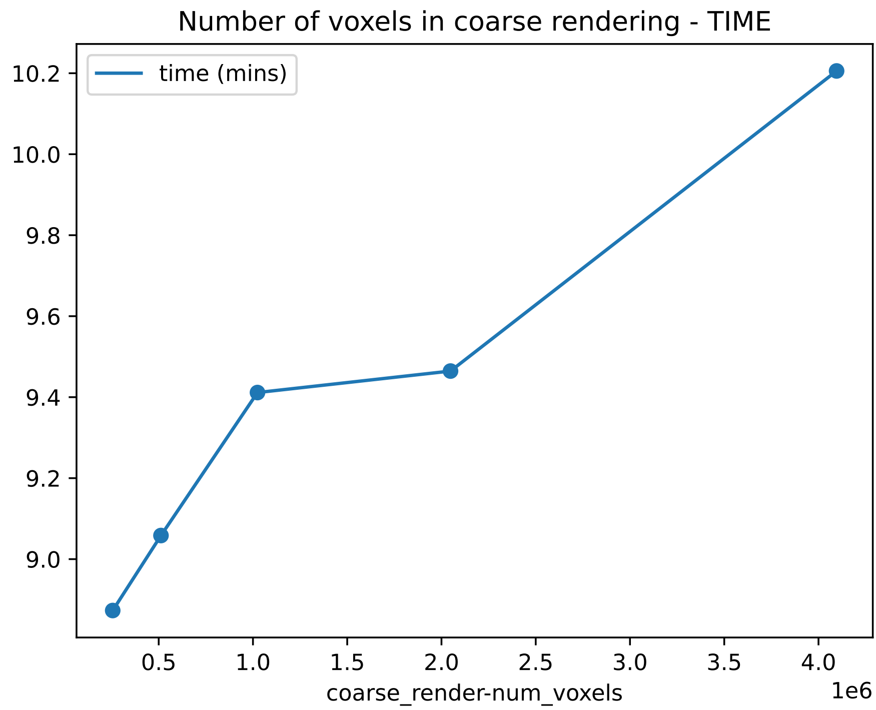

Figure 21: Number of voxels in coarse rendering - TIME plot

As intuitive, the time of executions grows with the increase of the number of voxels. However, we can see there is a sort of valley around the middle values where the ratio of increase between the two magnitudes is smaller.

Finally, regarding the memory consumption, we have as our results:

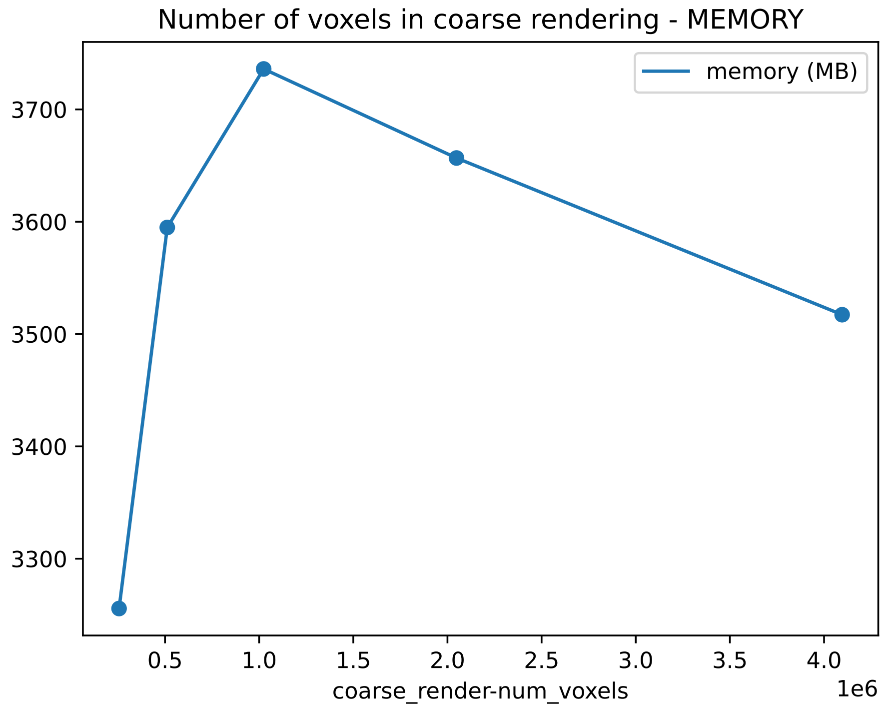

Figure 22: Number of voxels in coarse rendering - MEMORY plot

Opposite to the general tendency, here it is the default value (1024000) the one that has the highest memory consumption, although the different with other values is not that high. The one with the lowest memory consumption is the smaller value, being, in this case, 256000.

When choosing an optimal value here we would have to make a compromise, and depending whether we want to sacrifice some quality or some memory, we could either choose the default value (1024000) or the next one (2048000).

4.1.5 Number of voxels in fine rendering

Again, we remark that, as the the value for the parameters of *num_voxels* and *num_voxels_base* in the default settings is the same in all cases, we have kept it equal as well. Considering this, the values tested for these two hyperparameters have been: 10**3, 40**3, 80**3, 160**3 (default), 320**3.

In terms of quality measurements, the results are as follows:

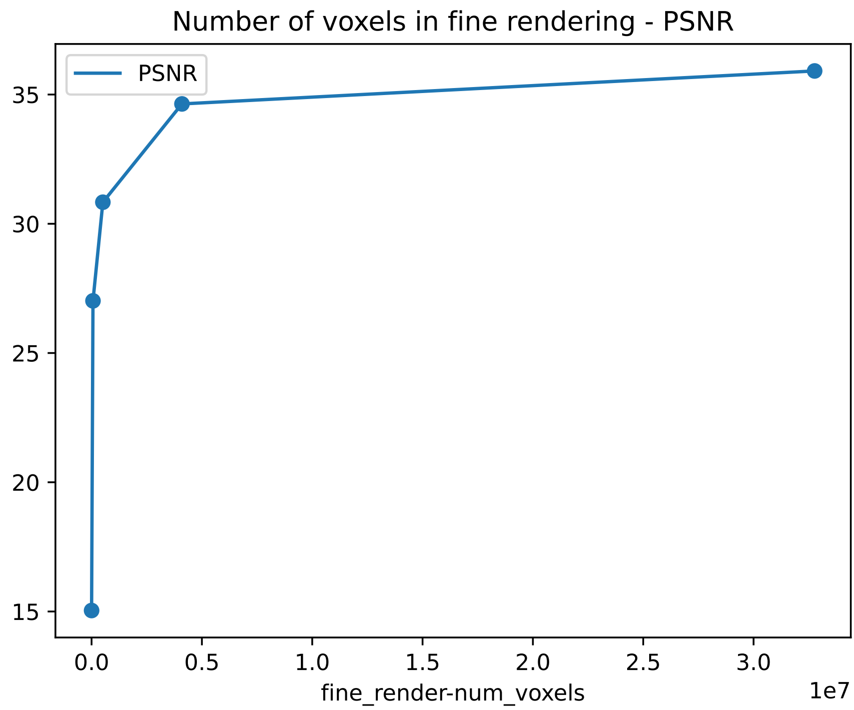

Figure 23: Number of voxels in fine rendering - PSNR plot

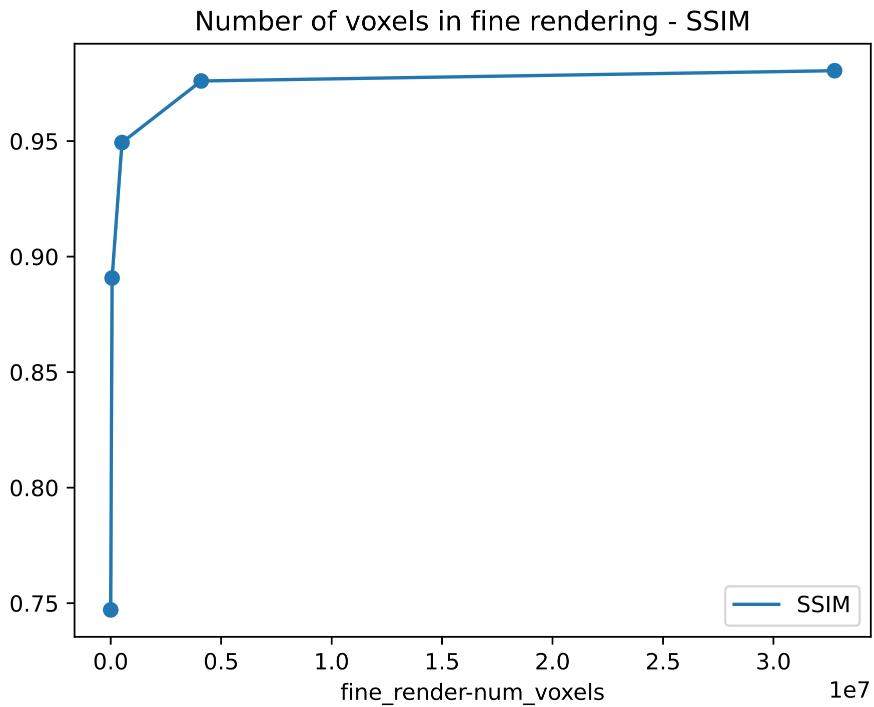

Figure 24: Number of voxels in fine rendering - SSIM plot

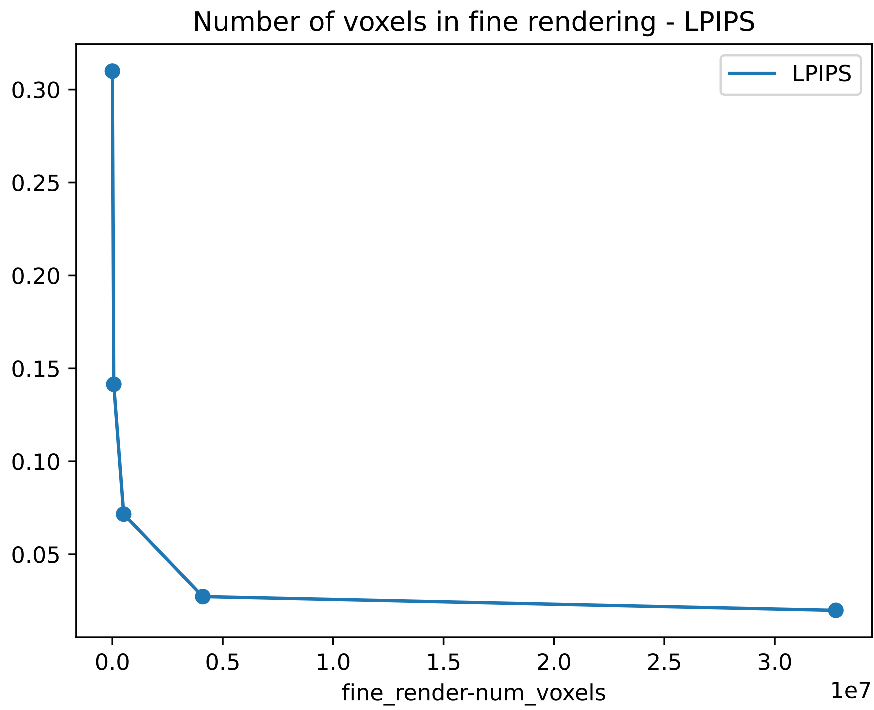

Figure 25: Number of voxels in fine rendering - LPIPS plot

Although the highest value in the metrics would be the last one (320**3), the quality becomes stable in its highest marks in the default value (160**3), making this one the best value in these terms.

When measuring the time of the executions, we find the following:

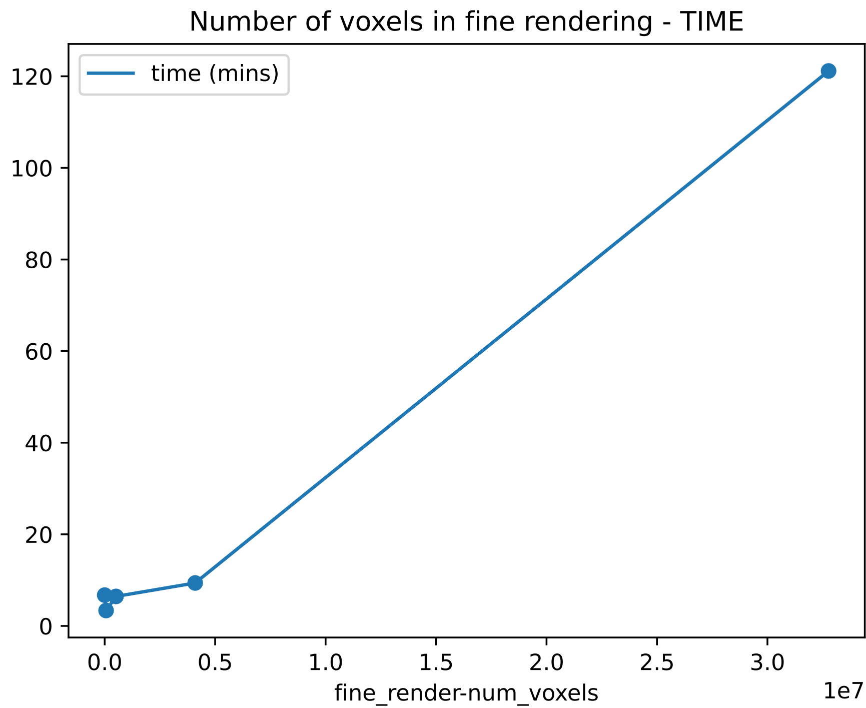

Figure 26: Number of voxels in fine rendering - TIME plot

Again, as expected, the time of executions grows approximately with the increase of the number of voxels. When we are testing values as high as 360**3, we reach almost 2 hours of execution time, which is unacceptable for a model whose biggest feature is being fast and having small execution times.

Finally, measuring the memory usage, we obtain:

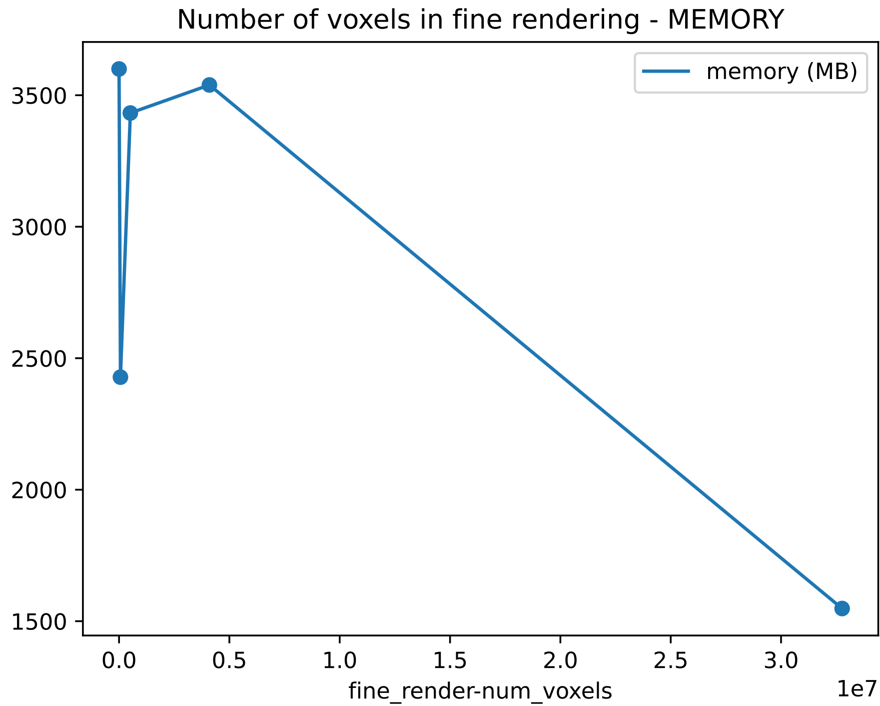

Figure 27: Number of voxels in fine rendering - MEMORY plot

The memory usage drops in the second value, and appears stable in the first, third and fourth value we tested. It decreases again in the highest and fifth value, but, as we said before, we have already discarded such value due to its unacceptably high time usage.

The second value (40**3) offers a better result in terms of memory and time usage, but we sacrifice some quality. On the other hand, the default one (160**3) offers the best reasonable quality measurement, and the worsening in the time and memory consumption is not catastrophic. It comes down, once again, to a matter of priorities.

4.2 Pareto optimal values

Pareto optimality, also known as Pareto efficiency or Pareto optimality, is a concept in economics and game theory that describes a state in which no individual or entity can be made better off without making someone else worse off. It is named after the Italian economist Vilfredo Pareto, who first introduced the idea in the early 20th century.

In a simplified two-agent scenario, Pareto optimality can be illustrated as follows: given a set of possible allocations of resources between two individuals, an allocation is considered

Pareto optimal if it is not possible to reallocate resources in a way that would increase the welfare or utility of one individual without reducing the welfare or utility of the other individual.

The set of all Pareto optimal allocations is known as the “Pareto front” or “Pareto frontier.” The Pareto front represents the boundary of the feasible solutions that achieve the highest possible welfare for one individual without negatively affecting the other individual’s welfare [2].

In more complex scenarios with multiple individuals or criteria to consider, the concept of Pareto optimality remains the same. An allocation is Pareto optimal if no improvement can be made for any individual or criterion without sacrificing the well-being of at least one other individual or criterion.

The Pareto front is essential in multi-objective optimization problems, where there are multiple conflicting objectives that need to be optimized simultaneously. In these cases, finding the Pareto front helps decision-makers understand the trade-offs between different objectives and choose the most suitable solution based on their preferences and priorities.

For example, in an urban planning context, the Pareto front might represent a set of city plans that achieve different balances between factors such as housing density, green spaces, transportation efficiency, and public amenities. Each plan along the Pareto front is considered Pareto optimal because it cannot be improved in one aspect without worsening another.

Overall, Pareto optimality and the Pareto front are crucial concepts in various fields to evaluate the efficiency and trade-offs of different solutions when multiple criteria are involved.

Knowing this, we will use this technique to find the tests that best balanced the quality measurements versus the time and memory consumption. This way, we will obtain 6 plots and values, representing PSNR vs. Time, SSIM vs. Time, LPIPS vs. Time, PSNR vs. Memory, SSIM vs. Memory, LPIPS vs. Memory.

As an observation, for values that we intend to minimize instead of maximize, meaning LPIPS, Time and Memory, we will make these values negative.

4.2.1 PSNR vs. Time

Having PSNR as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

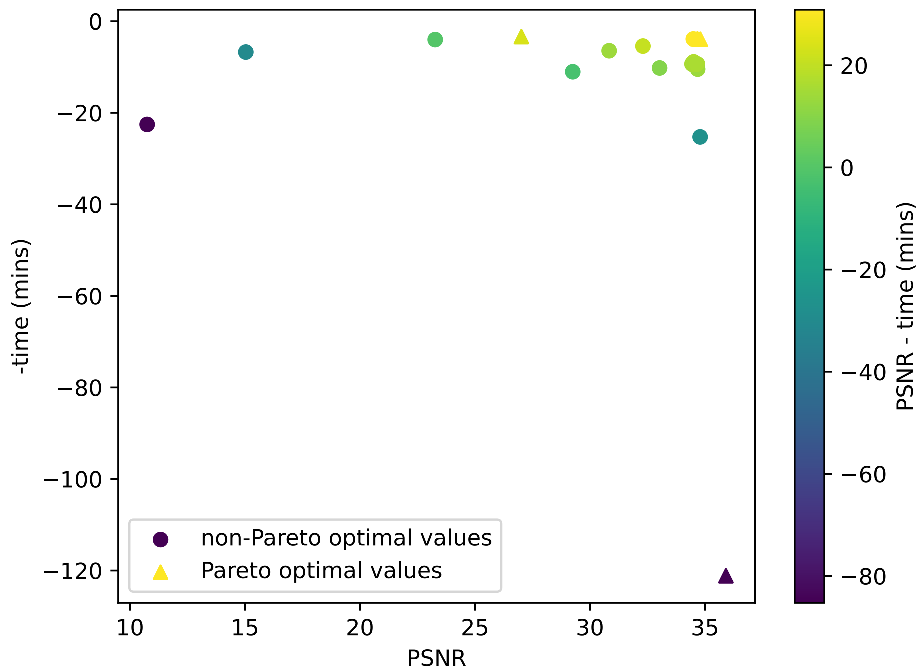

Figure 28: Pareto optimal values for PSNR vs. TIME plot

The four Pareto optimal values identified are the ones corresponding to:

- Number of random rays per optimization step: 2048
- Number of iterations in fine training: 40000
- Number of voxels in fine rendering: 40**3
- Number of voxels in fine rendering: 320**3

All of these valued but one we selected as potentially optimal values in our previous phase of research, which makes these results coherent. The other value, 40000 iterations in fine training, can be explained seeing how well it performs in the metrics of both quality and time (figures 12, 15), being its high memory usage (figure 17) the factor that lead us to initially discard it. This same situation is bound to keep happening in all the selections of Pareto optimal values when we compare quality and time.

4.2.2 SSIM vs. Time

Having SSIM as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

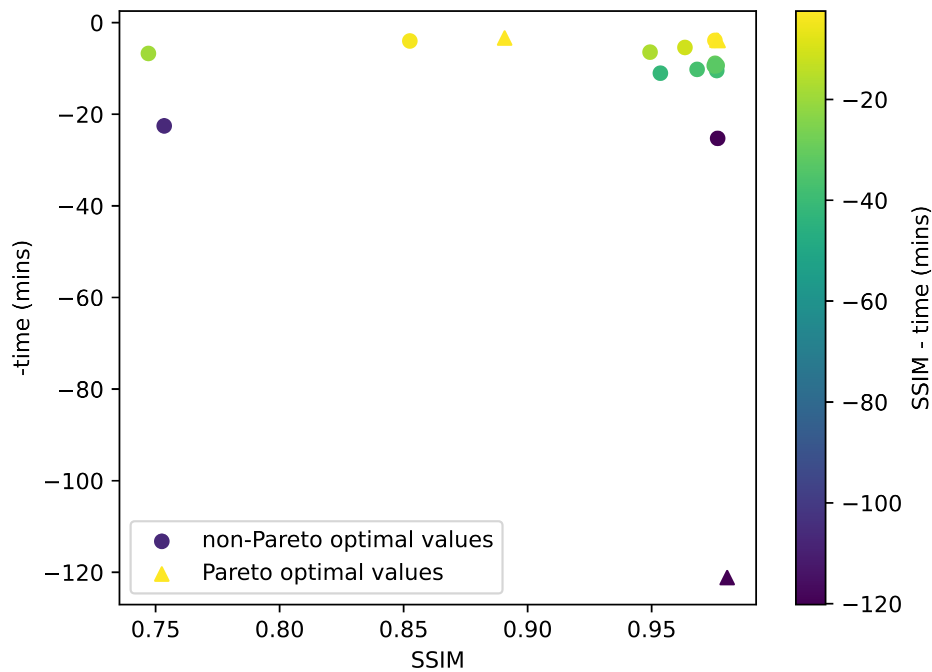

Figure 29: Pareto optimal values for SSIM vs. TIME plot

The Pareto optimal values identified are the ones corresponding to:

- Number of random rays per optimization step: 2048
- Number of iterations in fine training: 40000
- Number of voxels in fine rendering: 40**3
- Number of voxels in fine rendering: 320**3

Here we obtain the same results as before. This is due to the similarity in results between the three metrics of quality.

4.2.3 LPIPS vs. Time

Having -LPIPS as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

Figure 30: Pareto optimal values for LPIPS vs. TIME plot

The four Pareto optimal values identified are the ones corresponding to:

- Number of random rays per optimization step: 2048
- Number of iterations in fine training: 40000
- Number of voxels in fine rendering: 40**3
- Number of voxels in fine rendering: 320**3

As before, the same results as obtained with the same explanation attached. The similarity between the three metrics of quality further verifies that these metrics are trustworthy and the results are reliable because of this cross-verification.

4.2.4 PSNR vs. Memory

Having PSNR as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

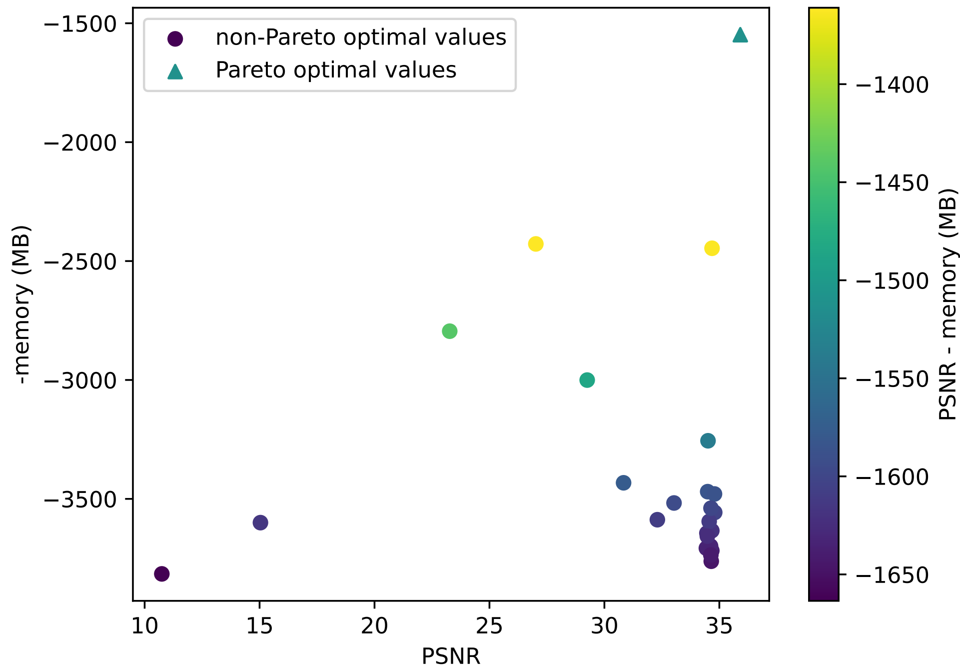

Figure 31: Pareto optimal values for PSNR vs. MEMORY plot

The only Pareto optimal value identified corresponds to:

- Number of voxels in fine rendering: 320**3

4.2.5 SSIM vs. Memory

Having SSIM as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

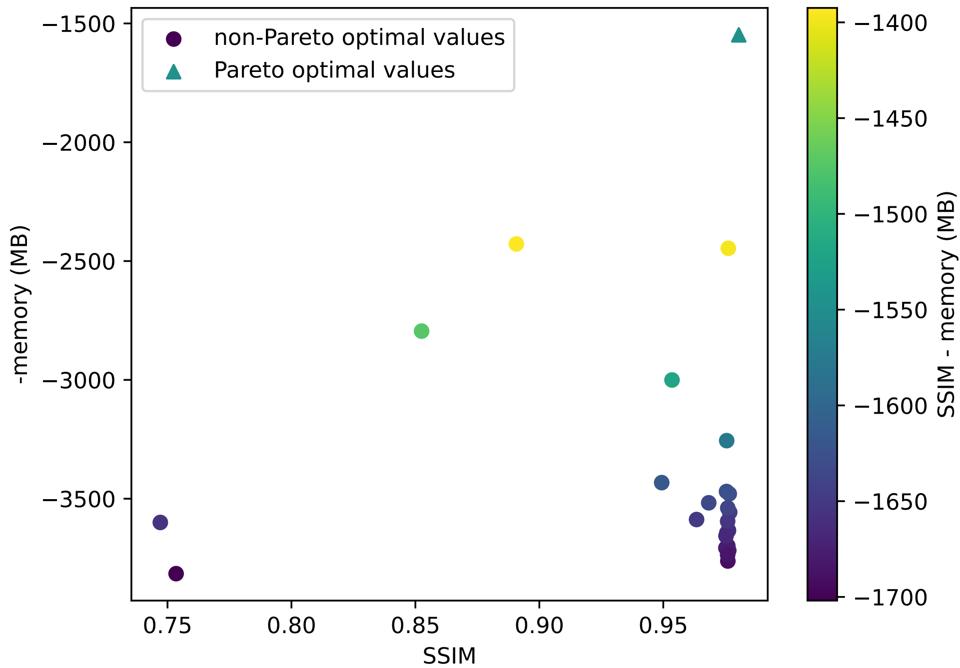

Figure 32: Pareto optimal values for SSIM vs. MEMORY plot

The only Pareto optimal value identified corresponds to:

- Number of voxels in fine rendering: 320**3

4.2.6 LPIPS vs. Memory

Having -LPIPS as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

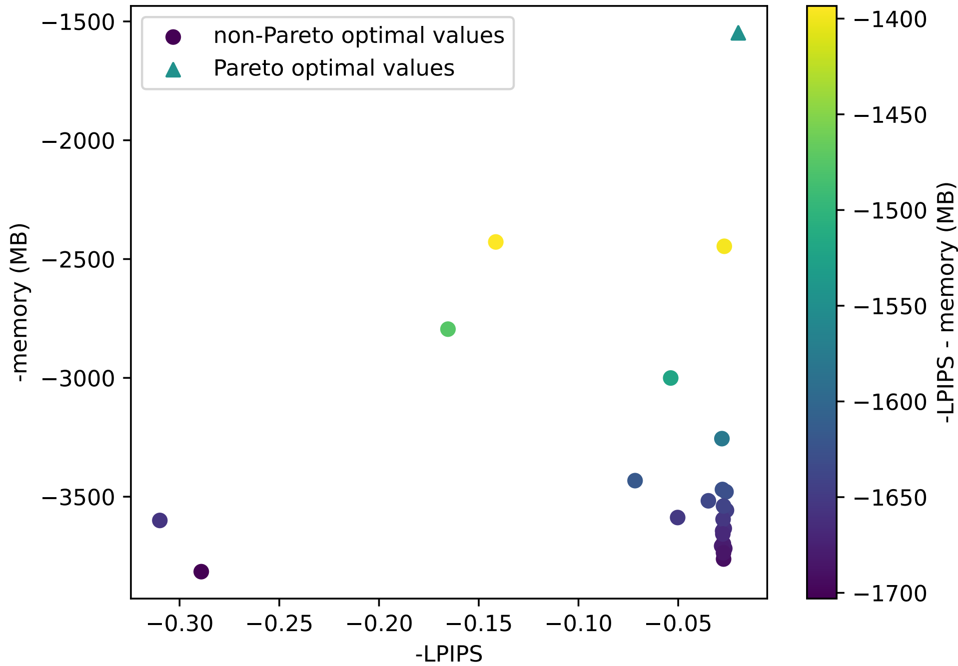

Figure 33: Pareto optimal values for LPIPS vs. MEMORY plot

The only Pareto optimal value identified corresponds to:

- Number of voxels in fine rendering: 320**3

4.2.7 Correction for outlier in memory measurement

In the search for Pareto optimal values in the graphs related with memory consumption, in the three cases, we obtain that the only Pareto optimal value found is the one with 320**3 as the number of voxels in the fine rendering phase. This is coherent with the results we have, as this value, even if it does not improve it, maintains a high level of quality (figures 23 - 25), and has a significantly lower memory consumption than the rest of the values (figure 27). However, in terms of time consumption, this value offers a really bad results, reaching times around the 2 hours mark, where all the other values offer results under 20 minutes (figure 26). Considering the DirectVoxGo is a fast neural rendering model, whose biggest feature is its low execution times without losing quality (compared with other neural rendering models), it is very likely that we will discard this value for our further phases of testing. Given this, and in order to obtain more useful results in the realm of memory consumption that are hidden by this outlier,

we will re-calculate the Pareto optimal values for memory consumption leaving this specific experiment out. Considering this, the results are as follow.

PSNR vs. Memory Having PSNR as our X-axis and -memory (MB) as our Y-axis, and having eliminated our outlier, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

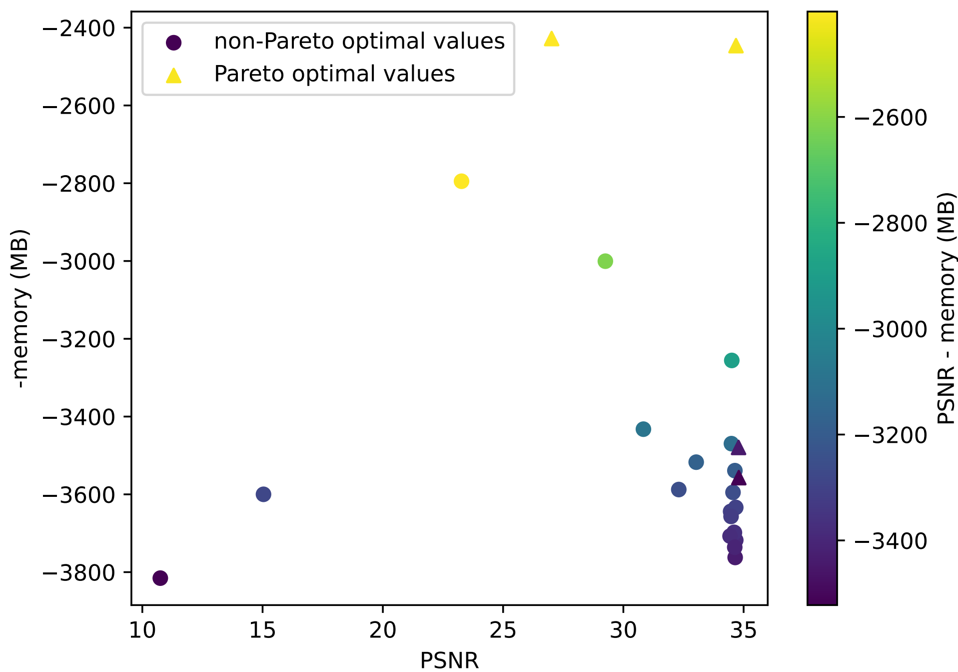

Figure 34: Pareto optimal values for PSNR vs. MEMORY plot without the outlier

The Pareto optimal values identified corresponds to:

- Number of iterations in fine training: 20000
- Number of iterations in fine training: 40000
- Number of iterations in fine training: 80000
- Number of voxels in fine rendering: 40**3

Here, 20000 as the number of iterations in fine training and 40**3 as the number of voxels in fine rendering are expected values, as we identified them before as potentially optimal values

in their realm. However, the introduction of the value 80000, and the re-appearance of the value 40000, both in the number of iterations in fine training, can be surprising. However, there's an explanation for this. Both values have a high measurement in all quality metrics, as high as the value 20000, sitting around 34.7 for the PSNR measurement (figure 12). While it is true that they both have drastically worse results in memory than the default value (for 20000 the consumption sits around 2400 MBs, while for the other two values, it tests higher than 3400 MBs), this value of around 3500 MBs of memory consumption is still on or even below the average of memory consumption for all values in all hyperparameters. Adding the fact that the memory consumption is not penalized against their competitors, and that a PSNR value of around 34.7 is slightly above most other measurements, it's reasonable that these values are found here.

It is also remarkable that 2048 random rays per optimization step and 160**3 voxels in fine rendering, which were previously chosen as Pareto optimal values, have dissapeared now. The reasoning is simple: these values have a poor performance in memory consumption, consuming around 364 MBs and 354 MBs respectively.

SSIM vs. Memory Having SSIM as our X-axis and -memory (MB) as our Y-axis, and having eliminated our outlier, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

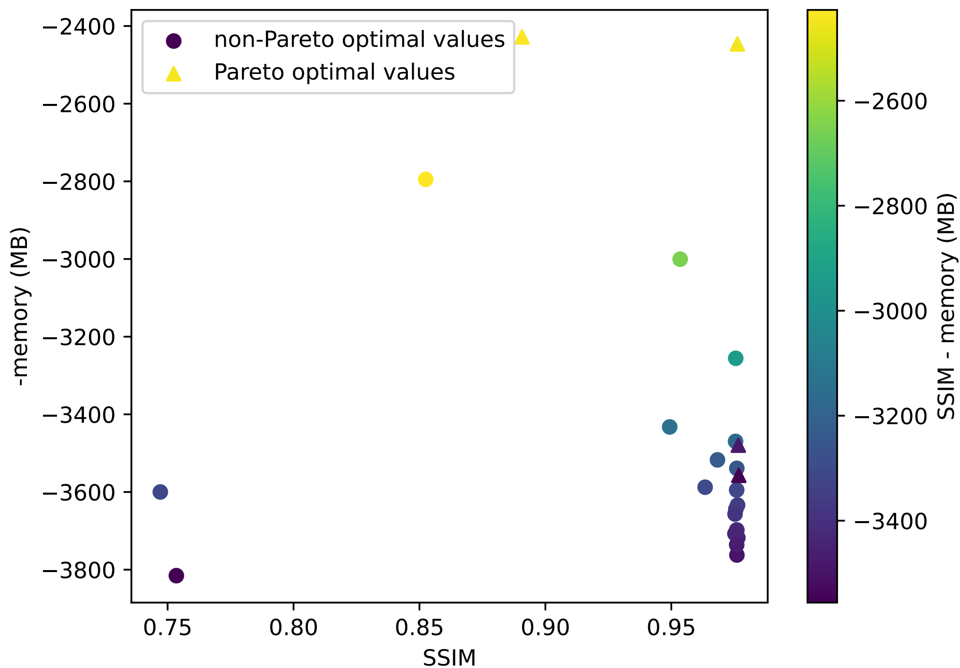

Figure 35: Pareto optimal values for SSIM vs. MEMORY plot without the outlier

The Pareto optimal values identified corresponds to:

- Number of iterations in fine training: 20000
- Number of iterations in fine training: 40000
- Number of iterations in fine training: 80000
- Number of voxels in fine rendering: 40**3

As we have obtained the same results, and given the parity in the three quality measurements, the aforementioned explanations apply here also.

LPIPS vs. Memory Having -LPIPS as our X-axis and -memory (MB) as our Y-axis, and having eliminated our outlier, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

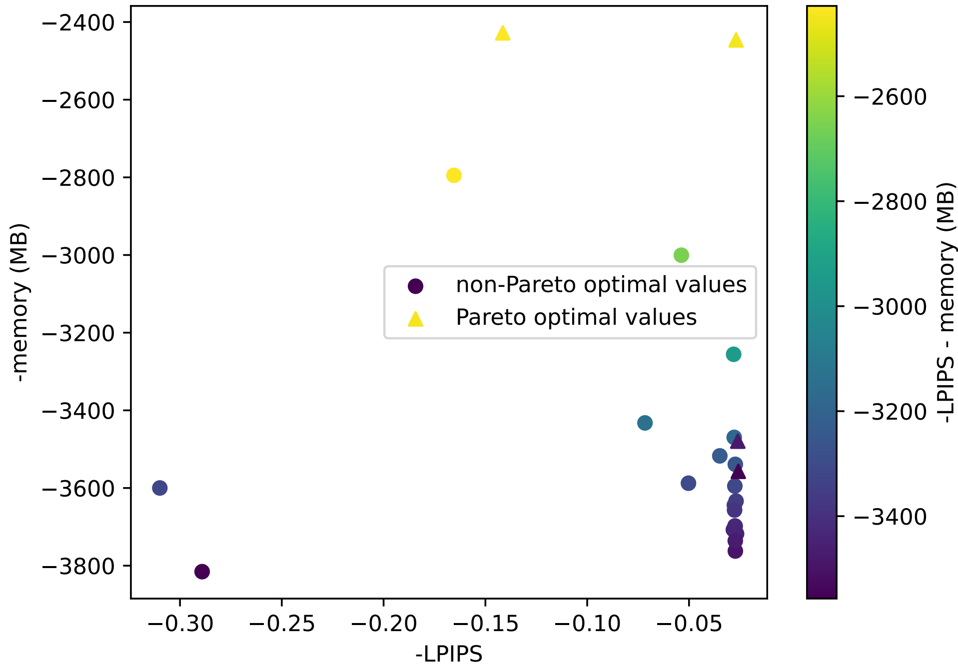

Figure 36: Pareto optimal values for LPIPS vs. MEMORY plot without the outlier

The Pareto optimal values identified corresponds to:

- Number of iterations in fine training: 20000
- Number of iterations in fine training: 40000
- Number of iterations in fine training: 80000
- Number of voxels in fine rendering: 40**3

Once again, we obtain the same results with the same reasoning behind them. It is remarkable also that only parameters regarding the fine phase appear here, showing that, the reduced relevance that the coarse phase could still have in the time measurements, disappears when talking about memory.

4.3 Conclusions for this phase of experimentation

The results for this phase of experimentation are surprising. We have analyzed 5 different values in 5 different hyperparameters, and we have made an individual analysis of each of

these. However, when searching for the Pareto optimal values, only 3 of these hyperparameters have appeared. These are the number of random rays per optimization step, the number of iterations in fine training, and the number of voxels in fine rendering. This offers us our first conclusion, which is that **the number of iterations in coarse training and the number of voxels in coarse rendering have a lesser impact on the results**. Following this thread, this could also indicate that the **coarse phases have a smaller impact on the final results than the fine phases**.

We have also seen that, when considering the hyperparameter individually, the general trend is that the default value chosen by the developers offers a result among the best. Nevertheless, this is not always the case when finding the Pareto optimal values. This does happen in the case of the number of iterations in fine training, where 20000 (default value) appears as a Pareto optimal value when regarding the memory consumption, as much as the value 80000, but the value 40000 for this hyperparameter appears in all cases (considering both time and memory usage). Nonetheless, **the default values for number of rays per optimization step and the number of voxels in fine rendering are not among the Pareto optimal in any of the 6 graphs**. This is coherent with the data from the individual testing, as the default value was not the best result in the case of the rays per optimization step (figures 7, 8, 9); and in the case of the number of voxels in fine rendering, although the default value was the best (figures 18, 19, 20), it was second worst in both the time testing (figure 21) and memory usage (figure 22).

According to these results, the best values for parameters are:

- Number of iterations in fine training: 40000
- Number of voxels in fine rendering: 40**3

These values are chosen as Pareto optimal values in all six comparisons. In our next phase, we will perform tests with the combination of most of these and the rest of Pareto optimal values.

5 Combined Hyperparameter Experimentation (phase II)

5.1 Hyperparameter combinations

In this phase of experimentation we have the objective of finding the combinations of values for the hyperparameters we are testing that offer a better result in our metrics. To achieve this, we will test executions with these combinations, and analyze the results as we did in our previous phase: comparing the quality with the time and memory consumption. However, now we will not try with arbitrary values as before, but we will use the results we obtained earlier.

To make our combinations, we will use the values that have appeared as Pareto optimal values in any of the graphs. These would be:

- Number of rays per optimization step: 2048
- Number of iterations in fine training: 20000, 40000, 80000
- Number of voxels in fine rendering: 40**3, 320**3

Note that only the three hyperparameter that appeared in the Pareto optimal values are being considered, and the rest will be left in their default values, as we consider they do not have a significant impact in the result.

As we have mentioned before, when increasing the number of voxels in fine rendering to 320**3, we reach times that are not reasonable for either testing or a real execution, which can be over two hours (when most of the other values are below 20 minutes). As the DirectVoxGo prides in being a model that offers quality results in lower execution times, we will discard this value for practical reasons and going against the principles of our model.

We will also add to the mix the default values for these hyperparameters, as we also want to try combinations where only one or two of the hyperparameters are altered (and to keep a hyperparameter unaltered is to assign its default value). Therefore, the final set of values that will be combined and tested are:

- Number of rays per optimization step: 2048, 8192 (default)
- Number of iterations in fine training: 20000 (default), 40000, 80000
- Number of voxels in fine rendering: 40**3, 160**3 (default)

This results in 12 different combinations that, in this order of hyperparemters, are as follow:

- values[0] = [2048, 20000, 40**3]
- values[1] = [2048, 20000, 160**3]
- values[2] = [2048, 40000, 40**3]
- values[3] = [2048, 40000, 160**3]
- values[4] = [2048, 80000, 40**3]
- values[5] = [2048, 80000, 160**3]
- values[6] = [8192, 20000, 40**3]
- values[7] = [8192, 20000, 160**3]
- values[8] = [8192, 40000, 40**3]
- values[9] = [8192, 40000, 160**3]
- values[10] = [8192, 80000, 40**3]
- values[11] = [8192, 80000, 160**3]

5.2 Measurement criteria

To maintain coherence and the capability of comparing and verifying our new results with our former ones, we will keep our measurement criteria intact. This is that we will keep Peak Signal-to-Noise Ratio (PSNR), Learned Perceptual Image Patch Similarity (LPIPS) and Structural Similarity Index (SSIM) as the metrics for quality. We will also pit each of these against the time and memory consumption, exactly as we did in our previous experimentation phase.

5.3 Results dispersion and Pareto optimal values

Having tested our combinations, the results are as follow.

5.3.1 PSNR vs. Time

Having PSNR as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

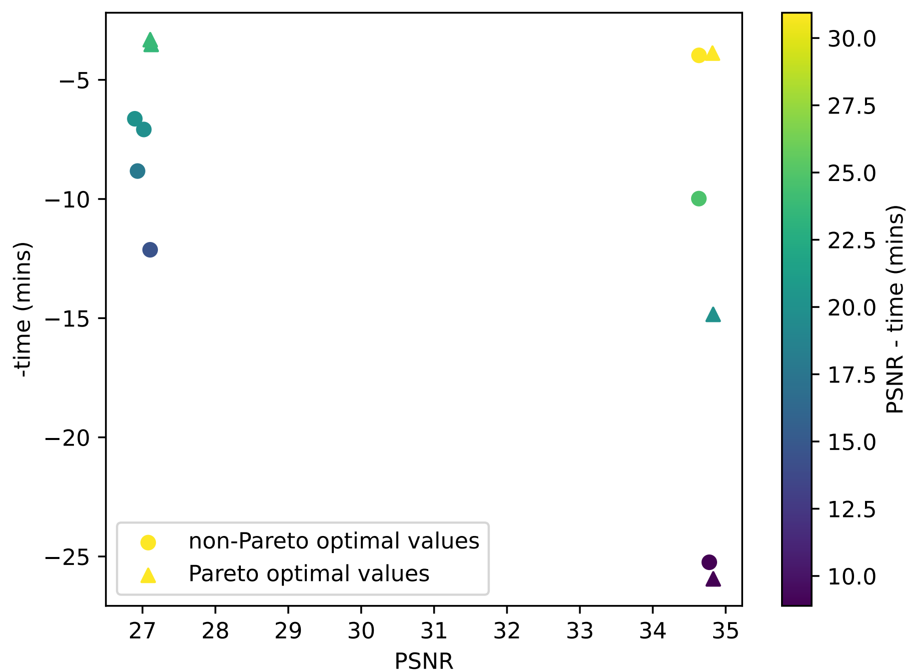

Figure 37: Pareto optimal values for PSNR vs. TIME plot

The Pareto optimal combinations identified are the ones corresponding to:

- Configuration 1
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 3
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 40**3
- Configuration 4
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 80000
  - Number of voxels in fine rendering: 40**3
- Configuration 5
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 80000
  - Number of voxels in fine rendering: 160**3

5.3.2 SSIM vs. Time

Having SSIM as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

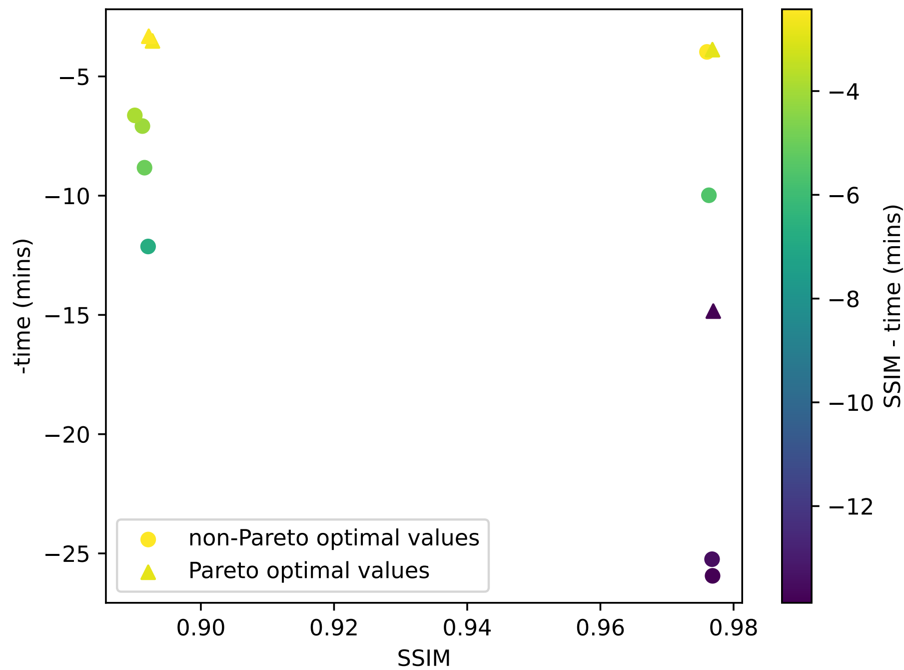

Figure 38: Pareto optimal values for SSIM vs. TIME plot

The four Pareto optimal combinations identified are the ones corresponding to:

- Configuration 1
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3

- Configuration 3
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 40**3
- Configuration 4
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 80000
  - Number of voxels in fine rendering: 40**3

5.3.3 LPIPS vs. Time

Having -LPIPS as our X-axis and -time (mins) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

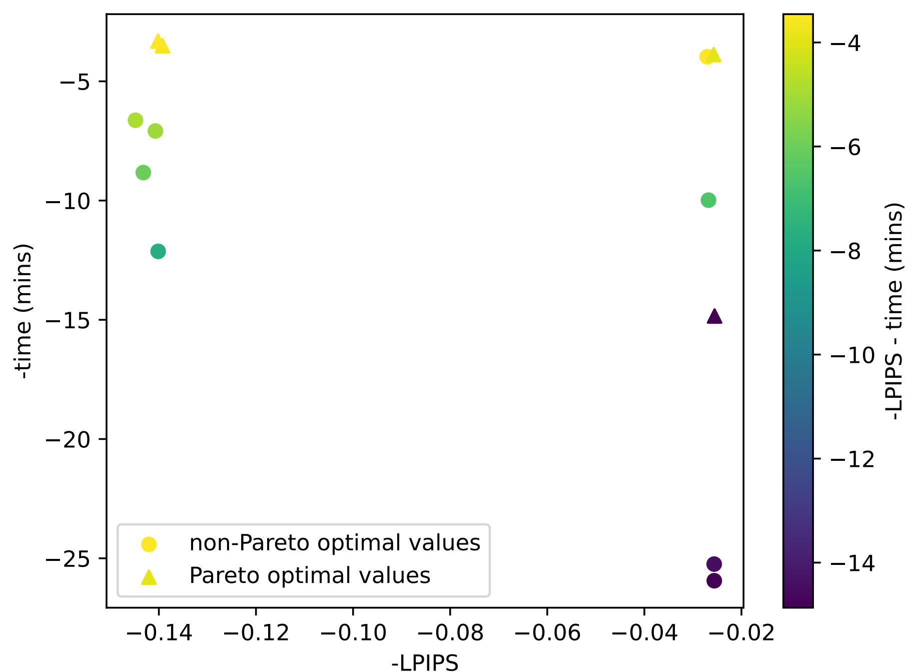

Figure 39: Pareto optimal values for LPIPS vs. TIME plot

The Pareto optimal combinations identified are:

- Configuration 1
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 3
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 40**3
- Configuration 4
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 80000
  - Number of voxels in fine rendering: 40**3

5.3.4 PSNR vs. Memory

Having PSNR as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

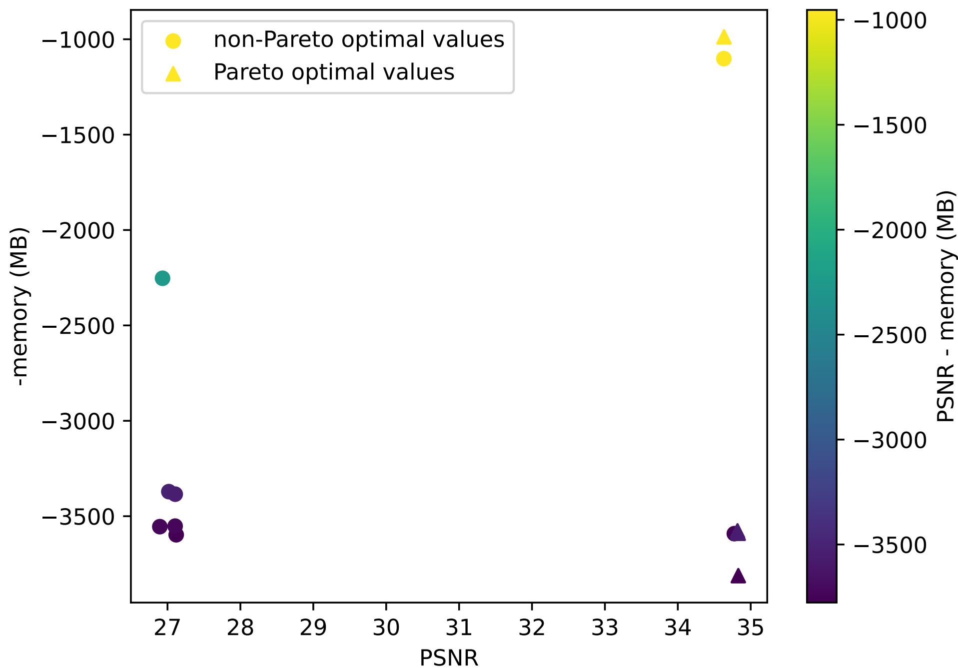

Figure 40: Pareto optimal values for PSNR vs. MEMORY plot

The Pareto optimal combinations identified correspond to:

- Configuration 1
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 20000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 3
  - Number of random rays per optimization step: 2048

– Number of iterations in fine training: 40000

– Number of voxels in fine rendering: 160**3

- Configuration 4

  – Number of random rays per optimization step: 8192

  – Number of iterations in fine training: 80000

  – Number of voxels in fine rendering: 160**3

5.3.5 SSIM vs. Memory

Having SSIM as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

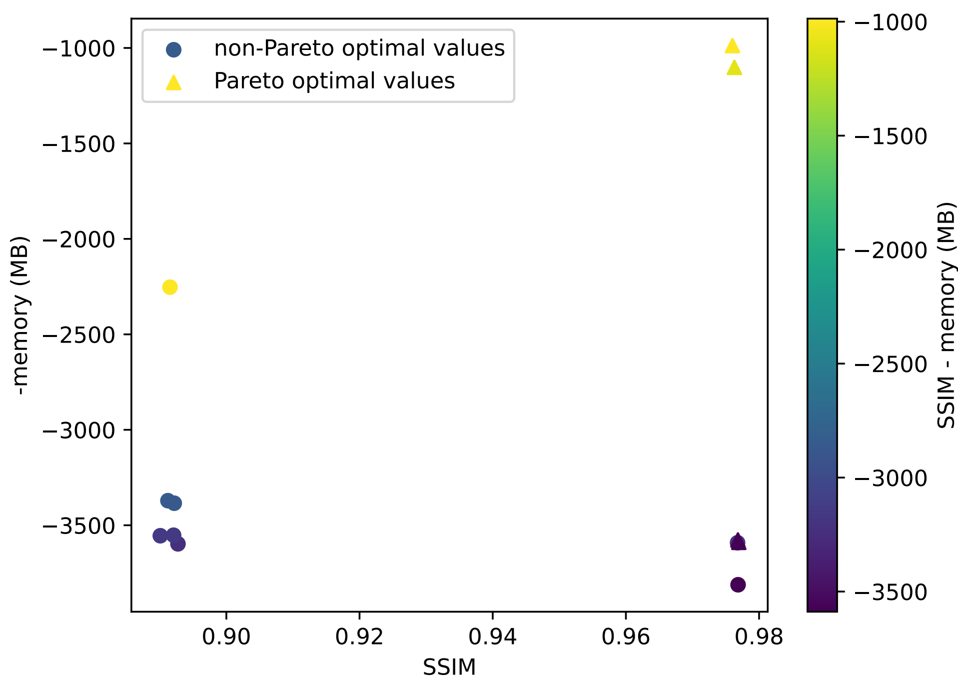

Figure 41: Pareto optimal values for SSIM vs. MEMORY plot

The Pareto optimal combinations identified correspond to:

- Configuration 1

  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 20000
  - Number of voxels in fine rendering: 160**3

- Configuration 2

  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 20000
  - Number of voxels in fine rendering: 160**3

- Configuration 3

  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3

- Configuration 4

  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3

5.3.6 LPIPS vs. Memory

Having -LPIPS as our X-axis and -memory (MB) as our Y-axis, the distribution of results and the Pareto optimal values (identified in triangular marks) are these:

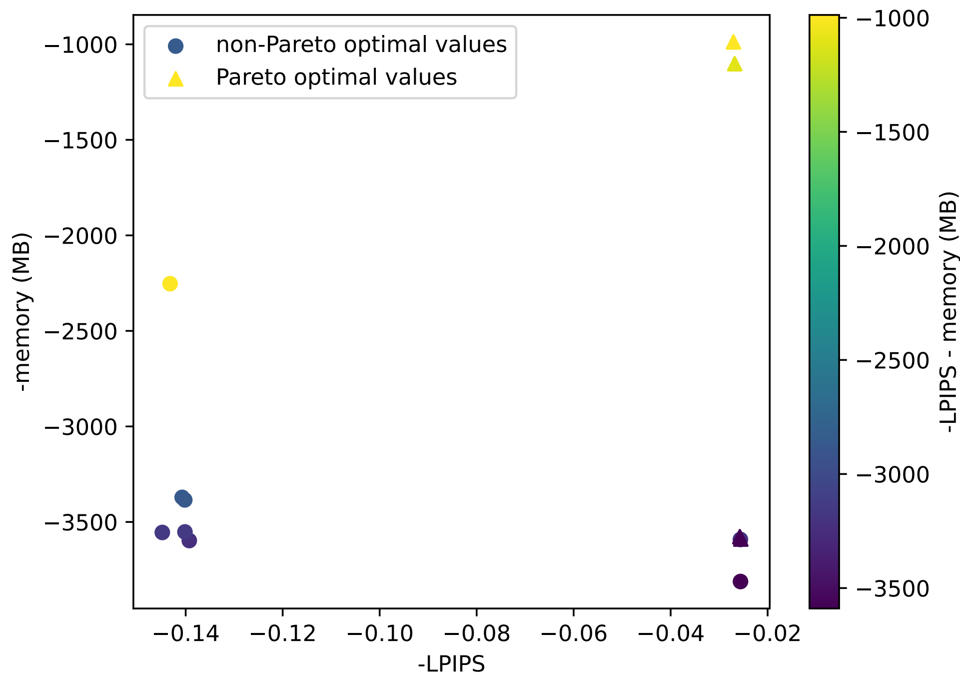

Figure 42: Pareto optimal values for LPIPS vs. MEMORY plot

The Pareto optimal combinations identified correspond to:

- Configuration 1
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 20000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 20000
  - Number of voxels in fine rendering: 160**3
- Configuration 3
  - Number of random rays per optimization step: 8192

  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 4
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3

5.4 Conclusions for this phase of experimentation

Once we have our results, it is convenient to see the number of appearances of each one of the tested combinations in the set of Pareto optimal values in each of the graphs.

We start with the graphs relating time and the different quality measurements:

N_RAND	N_ITERS	NUM_VOXELS	APPEARANCES
2048	20000	4,00E+04	0
2048	20000	1,60E+05	0
2048	40000	4,00E+04	0
2048	40000	1,60E+05	3
2048	80000	4,00E+04	3
2048	80000	1,60E+05	0
8192	20000	4,00E+04	0
8192	20000	1,60E+05	0
8192	40000	4,00E+04	3
8192	40000	1,60E+05	3
8192	80000	4,00E+04	0
8192	80000	1,60E+05	1

Figure 43: Hyperparameter combinations and their appearances in the Pareto optimal values of the graphs related with time

As visible, there are four combinations with the higher number of appearances (3, meaning they appeared as Pareto optimal values in all the graphs). We can see that these feature both

values for the hyperparameter of random rays per optimization step in the same proportion. Also, they feature the same proportion for the two values in the number of voxels in fine rendering. Finally, we can see only two out of three values in the number of iterations in the fine training phase appear, and not in the same proportion. There results lead us to conclude that while the number of random rays per optimization step and number of voxels in fine rendering may not be that important in terms of time consumption, the number of iterations in fine training is key. However, as we are considering combinations, is not safe to make assumptions about the role individual parameters play.

Next, we take a look at these values for the graphs related with memory consumption:

N_RAND	N_ITERS	NUM_VOXELS	APPEARANCES
2048	20000	4,00E+04	0
2048	20000	1,60E+05	2
2048	40000	4,00E+04	0
2048	40000	1,60E+05	3
2048	80000	4,00E+04	0
2048	80000	1,60E+05	0
8192	20000	4,00E+04	0
8192	20000	1,60E+05	3
8192	40000	4,00E+04	0
8192	40000	1,60E+05	3
8192	80000	4,00E+04	0
8192	80000	1,60E+05	1

Figure 44: Hyperparameter combinations and their appearances in the Pareto optimal values of the graphs related with memory

As opposite as in the time consumption cases, here we start to have appearances of the combinations where the number of iterations in fine training is 20000, while combinations with 40**3 voxels in fine rendering drop to 0 appearances, prompting us to assume that in terms of memory usage, the key factors are the number of iterations in fine training (as before), but also the number of voxels in fine rendering.

There are three combinations that appear in all three graphs as Pareto optimal values, two

of them overlapping with the time results, and a new one (that had zero appearances before).

To reach a global conclusion, we will now observe a table adding the results of the former two:

N_RAND	N_ITERS	NUM_VOXELS	APPEARANCES
2048	20000	4,00E+04	0
2048	20000	1,60E+05	2
2048	40000	4,00E+04	0
2048	40000	1,60E+05	6
2048	80000	4,00E+04	3
2048	80000	1,60E+05	0
8192	20000	4,00E+04	0
8192	20000	1,60E+05	3
8192	40000	4,00E+04	3
8192	40000	1,60E+05	6
8192	80000	4,00E+04	0
8192	80000	1,60E+05	2

Figure 45: Hyperparameter combinations and their appearances in the Pareto optimal values of all the graphs

As aforementioned, there are two combinations that have appeared as Pareto optimal values in all six cases. These are:

- Configuration 1
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3
- Configuration 2
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in fine rendering: 160**3

Based on our experimentation, we can say that, among the many tested, **these two are the combinations that offer the best results in balancing quality, time consumption and memory usage**.

Analyzing other aspects of the table, we can see that those combinations that had zero appearances all have individual values that have appeared in combinations with more appearances, leading us to believe that the combination of values, and not only the individual hyperparameters, have a weight on the results.

In terms of distributions per hyperparameter, we can see

- Random rays per optimization step: there are 11 appearances of the value 2048, and 15 for the value 8192, which is fairly evenly distributed, further suggesting that, choosing between these two specific values, this hyperparameter has little impact in our results.
- Number of iterations in fine training: the value 20000 appears 20% of the times (5), 40000 is featured 60% (15), and 80000 has 20% of the appearances (5). This shows a predominance for the value 40000, which explains that it is the one featured in both of the winning combinations.
- Number of voxels in fine rendering: in this case, we have a 6:19 ratio in favour of the value of 160**3 (24% vs. 76%). This again, offers no doubt as to why 160**3 is the value featured in both of the winning combinations.

6

Conclusions and Futures Lines of Research

6.1 Conclusions

In this study we have analyzed the hyperparameters of the model of scene reconstruction through neural rendering DirectVoxGo. For this purpose, we have worked over the original implementation of the project [36]. The project uses the Python library PyTorch, combined with Nvidia's CUDA technology. The installation for the latter in particular, and for the whole project in general, has resulted in numerous problems for us. In general, in both our personal computer as well as the server we have been given access, it was difficult to find versions of Pytorch and CUDA that were compatible among them, with our hardware and with the original code. Additionally, it was also challenging to find graphics and processing hardware advanced enough to meet the recommended requirements for our testing.

In terms of the obtained results, we have achieved two combinations of hyperparameters that perform better than the standard configuration offered by the original developers. These are:

- Configuration 1
  - Number of iterations in coarse training: 5000
  - Number of random rays per optimization step: 2048
  - Number of iterations in fine training: 40000
  - Number of voxels in coarse rendering: 1024000
  - Number of voxels in fine rendering: 160**3

- Configuration 2
  - Number of iterations in coarse training: 5000
  - Number of random rays per optimization step: 8192
  - Number of iterations in fine training: 40000
  - Number of voxels in coarse rendering: 1024000
  - Number of voxels in fine rendering: 160**3

For comparison, the default values for this same hyperparameteres are:

- Number of iterations in coarse training: 5000
- Number of random rays per optimization step: 8192
- Number of iterations in fine training: 20000
- Number of voxels in coarse rendering: 1024000
- Number of voxels in fine rendering: 160**3

Let's analyze now the performance results obtained for these configurations in our testing. For the data corresponding to the default configuration, we have chosen for each metric the median of the values from the 5 tests with the standard configuration that took place during the first phase of testing (in the individual testing, each time we tested the default value for each parameter), discarding in this case the value from the second phase of testing, as it showed unusual values in the time and memory measurements.

- Default configuration
  - Time (seconds): 565.1386282444
  - Memory (bytes): 3698429952
  - Quality - PSNR: 34.63700866699219
  - Quality - SSIM: 0.9760104574970425
  - Quality - LPIPS: 0.027261049784719944

- Configuration 1
  - Time (seconds): 890,4655954837799
  - Memory (bytes): 3589492736
  - Quality - PSNR: 34,82859399318695
  - Quality - SSIM: 0,9769523837598433
  - Quality - LPIPS: 0,0255326937045902
- Configuration 2
  - Time (seconds): 232,69829750061035
  - Memory (bytes): 3576987648
  - Quality - PSNR: 34,818851840496066
  - Quality - SSIM: 0,9768228257034578
  - Quality - LPIPS: 0,02573369013145566

In terms of quality, we can see in the three metrics that "configuration 1" tests slightly above the rest, although this different is pretty small. This way, we could say that all three configuration have comparable quality.

Regarding time consumption, we obtain more interesting results. We can see that "configuration 1" has an execution time considerably longer than the other two, timing slightly below 15 minutes. This is 58% higher than the standard configuration (aprox. 9.4 minutes), and roughly 283% higher than "configuration 2" (a bit below 4 minutes). A duration this high can entail a great disadvantage for a system that prides in its speed. Considering the other two results, "configuration 2" is the fastest, being 143% faster than default, which represents a considerable difference.

In the realm of memory consumption we encounter a greater parity. The most notable difference between values in this field is between the standard configuration and "configuration 2", and is only slightly above 100 MBs.

Taking everything into account we can see that "configuration 2" improves the results of the default configuration in every metric, having a special effect on time consumption. This

is particularly relevant in a model whose main feature is velocity (while maintaining similar quality). Henceforth, we can conclude that "configuration 2" is the optimal configuration among the tested values.

However, an important considering would be the relevant variability in the results among tests with the same configuration. During our experimentation, we have identified outliers (especially during the memory measurements), which have been corrected by retaking the involved tests. Additionally, even when there weren't outliers identified, due to the intrinsic workings of the system, the randomness that artificial intelligence entails and external factors, we obtain a more or less relevant variability in the obtained results testing the same values, as can be noted in the 5+1 tests of the default configuration. All this leads us to conclude that, even if we have obtained a preliminary result in our experimentation of an specific configuration that improves the performance of the model, a more exhaustive and extensive experimentation would be needed to reach more solid conclusions. Nevertheless, our results help us state that it would be possible to enhance the performance of our model through the tuning of the hyperparameters.

Another consideration that we can take into account is that the importance of memory and time usage must be adjusted to the activity where the system is being applied. We could have cases where we are interested in prioritising quality and we would have more freedom with time and memory, like, for example, when we would want to render an environment in high quality to use it later. On the other hand, there can be situations when we need a fast and portable result, and we are willing to sacrifice some quality, which could be the case for a the rendering of environments in real time in a video game.

Additionally, we can infer that the quality of the result is the measurement that remains the most stable throughout the testing, which leads us to think that hyperparameter tuning is more interesting to achieve improvements in performance (time and memory consumption) than in quality. Knowing this is relevant, given that in the current situation in computer vision and neural rendering, when achieving similar results (which are not necessarily equal), performance is more vehemently valued.

6.2 Future lines of Research

The study of hyperparameters in neural rendering models and, specifically, of the DirectVoxGo model is a promising and prosperous line of research.

As aforementioned, the first future line of research would be performing studies of greater extension and exhaustivity to confirm the results obtained in our study. It would be convenient to perform a larger number of tests in even more controlled environments, finding middle values (as could be the median or the arithmetic average) and crossing these results and conclusions with those obtained in our experimentation.

As intuitive, another possible way to continue our investigation would be to widen the scope of our study to a larger number of values and hyperparameters. Due to the extension of this thesis, we have only selected five values for each hyperparameter, but this number could be bigger, circling around those values that test the best until reaching the optimal value for each case. The same happens with the choosing of the hyperparameters themselves, given that we have only stuck with the five that we considered the most significant, but there are much more that could a have a bigger or smaller impact on the results and the metrics we are testing. Additionally, it could also be interesting to test different sets of images, being these either the other ones provided by the original developers or new ones taken just for the study.

Another natural step in our investigation would be to consider the new advances that have taken place regarding the DirectVoxGo model. Recently, this very year, a new article introducing the DirectVoxGo++ model has been published [28]. It would be relevant to study the modification that this article proposes in contrast with what we have analysed, and apply a similar procedure of hyperparameter tuning to this new algorithm, analysing if we obtain different and/or better results.

Widening even more the scope of our research, we could apply a similar study to other contemporary models of neural rendering, as could be the recent models MixVoxels [42] or DyNeRF [21]. It could even be interesting to consider a similar approach on the original base model of NeRF [25], although this would take a larger portion of time given the dilated duration of its executions.

Finally, a more creative future line of research would be the application of any artificial intelligence algorithm to the identification of optimal combinations of hyperparameters. In this

case, considering the nature of our problem, we think that it would be specially interesting to attempt the use of genetic algorithms [33] [10]. In this situation, the cromosome would be the specific combination of hyperparameters, being each gene one of them, and the feedback for our evolution process would be provided by the results of testing each of these combinations, "killing" the cromosomes that perform the worst and mixing those that test higher in our metrics. This, or any other artificial intelligence approach, could potentially give birth to better combinations that those we can find by hand, and could also result effective when managing large amounts of hyperparameters and values.

7
Conclusiones y Líneas Futuras

7.1 Conclusiones

En este estudio, hemos analizado los hiperparámetros del modelo de generación de escenas mediante neural rendering DirectVoxGo. Para ello, hemos trabajado sobre la implementación original del proyecto [36]. El proyecto usa la librería PyTorch, combinado con la tecnología CUDA de Nvidia. La instalación de esta última en concreto, y del proyecto en general, ha dado numerosos problemas en la instalación. En general, tanto en nuestro equipo personal como en el servidor al que se nos dio acceso, fue complicado encontrar versiones de PyTorch y CUDA que fueran compatible con nuestro hardware y lo usado en el código del proyecto. Adicionalmente, también fue un reto encontrar un hardware gráfico y de procesamiento que llegara al alto nivel requerido para realizar las pruebas necesarias.

En cuanto a los resultados obtenidos, hemos conseguido encontrar dos combinaciones de hiperparámetros que arrojan mejores resultados que la configuración estándar ofrecida por los desarrolladores. Estas son:

- Configuración 1
  - Número de iteraciones en el entrenamiento grueso (coarse training): 5000
  - Número de rayos aleatorios por paso de optimización: 2048
  - Número de iteraciones en el entrenamiento fino (fine training): 40000
  - Número de vóxeles en la generación gruesa (coarse rendering): 1024000
  - Número de vóxeles en la generación fina (fine rendering): 160**3
- Configuración 2

- Número de iteraciones en el entrenamiento grueso (coarse training): 5000
- Número de rayos aleatorios por paso de optimización: 8192
- Número de iteraciones en el entrenamiento fino (fine training): 40000
- Número de vóxeles en la generación gruesa (coarse rendering): 1024000
- Número de vóxeles en la generación fina (fine rendering): 160**3

Por comparación, los valores por defecto para estos mismos parámetros son los siguientes:

- Número de iteraciones en el entrenamiento grueso (coarse training): 5000
- Número de rayos aleatorios por paso de optimización: 8192
- Número de iteraciones en el entrenamiento fino (fine training): 20000
- Número de vóxeles en la generación gruesa (coarse rendering): 1024000
- Número de vóxeles en la generación fina (fine rendering): 160**3

Analicemos ahora los resultados de rendimiento obtenidos por estas configuraciones en nuestras pruebas. Para los datos de la configuración estándar, hemos elegido para cada métrica la mediana de los valores de las 5 pruebas con la configuración estándar que hemos realizado durante la primera fase de pruebas (en las pruebas individuales, cada vez que probábamos el valor estándar para ese parámetro), descartando en principio en este caso el valor de la segunda fase de pruebas, ya que arroja unos valores inusuales en las mediciones de tiempo y memoria.

- Configuración estándar
  - Tiempo (segundos): 565.1386282444
  - Memoria (bytes): 3698429952
  - Calidad - PSNR: 34.63700866699219
  - Calidad - SSIM: 0.9760104574970425
  - Calidad - LPIPS: 0.027261049784719944
- Configuración 1

  - Tiempo (segundos): 890,4655954837799
  - Memoria (bytes): 3589492736
  - Calidad - PSNR: 34,82859399318695
  - Calidad - SSIM: 0,9769523837598433
  - Calidad - LPIPS: 0,0255326937045902

- Configuración 2
  - Tiempo (segundos): 232,69829750061035
  - Memoria (bytes): 3576987648
  - Calidad - PSNR: 34,818851840496066
  - Calidad - SSIM: 0,9768228257034578
  - Calidad - LPIPS: 0,02573369013145566

En cuanto a la calidad, podemos ver en las tres métricas que la "configuración 1" es ligeramente mejor que las otras dos, aunque esta diferencia es bastante pequeña. De esta forma, poderíamos decir que las tres configuraciones tienen calidad comparable.

En cuanto al tiempo, obtenemos los resultados más interesantes. Podemos ver que la "configuración 1" tiene una duración considerablemente mayor que las otras dos, siendo esta de algo menos de 15 minutos. Esto es un 58% mayor que la configuración estándar (aprox. 9,4 minutos) y aproximadamente un 283% mayor que la "configuración 2 (algo por debajo de 4 minutos). Una duración tan superior puede suponer un gran detrimento en un sistema que tiene como característica principal su velocidad. En cuanto a las otras dos configuraciones, la "configuración 2", que es la más rápida, es un 143% más veloz que la estándar, lo cual representa una diferencia considerable.

En el campo de la memoria encontramos una mayor paridad. La mayor diferencia entre valores en este campo es entre la condiguración estandar y la 2, y es algo superior a 100 MBs.

Podemos ver que la configuración 2 mejora los resultados de la configuración estándar en todas las métricas, haciendo especial hincapié en la duración de la ejecución. Esto es especialmente relevante en un modelo cuya característica principal es su velocidad (manteniendo la

calidad). De esta forma, podemos concluir que la configuración 2 sería la óptima dentro de los valores con los que hemos experimentado.

Sin embargo, una consideración importante seria la variabilidad de los resultados entre las pruebas con los mismos valores. Durante nuestra experimentación, hemos identificado outliers (especialmente en la medición de la memoria), que hemos tenido que corregir retomando la prueba. Adicionalmente, incluso cuando no se trata de outliers, debido al funcionamiento intrínseco del sistema, la aleatoriedad que se da en el campo de la inteligencia artificial y factores externos, obtenemos una variabilidad más o menos relevante en los resultados obtenidos probando los mismos valores, como podemos ver en las 5+1 pruebas de la configuración estándar. Todo esto nos lleva a concluir que, aunque hemos obtenido un resultado preliminar en nuestra experimentación de una configuración concreta que mejora el rendimiento del modelo, sería necesaria una experimentación más exhaustiva y dilatada en el tiempo para llegar a conclusiones más estables. No obstante, nuestros resultados nos ayudan a afirmar que sería posible encontrar un mejor rendimiento del modelo mediante el afinamiento de los hiperparámetros.

Otra consideración que podemos tener en cuenta es que la importancia del uso de memoria y tiempo debe ajustarse a la actividad a la que se esté aplicando el sistema. Podemos tener casos en los que nos interese priorizar la calidad y tengamos mayor libertad en cuanto al uso de tiempo y memoria, como, por ejemplo, cuando queramos renderizar un entorno en alta calidad para usarlo con posterioridad. También podemos encontrarnos otros casos en los que necesitemos un resultado rápido y portable, y podamos sacrificar algo de la calidad, como podría ser el renderizado de entornos en tiempo real en un videojuego.

Adicionalmente, podemos extraer que la calidad del resultado es la medida que se mantiene más estable cuando modificamos los hiperparámetros del modelo, lo cual nos lleva a concluir que el afinamiento de parámetros es más interesante para conseguir mejoras en rendimiento (uso de tiempo y memoria) que en la calidad. Esto es relevante, ya que en el panorama actual de visión por computador y “neural rendering”, a resultados similares (que no necesariamente iguales), se tiende a valorar con más vehemencia el rendimiento del modelo.

7.2 Líneas Futuras

El estudio de hiperparámetros en los modelos de "neural rendering" y, en concreto, del DirectVoxGo es una vía prometedora con miras hacia el futuro.

Como hemos mencionado anteriormente, la primera línea futura de investigación sería realizar estudios de mayor extensión y exhaustividad para confirmar los resultados obtenidos en nuestro estudio. Sería conveniente realizar un mayor número de pruebas en entornos aún más controlados, buscar valores medios (como puede ser la mediana o la media aritmética) y cruzar esos resultados y conclusiones con los obtenidos en nuestra experimentación.

Como es intuitivo, otra posible forma de continuar nuestra investigación sería ampliar el ámbito de nuestro estudio a un mayor número de valores e hiperparámetros. Debido a la extensión del proyecto, solo hemos seleccionado cinco valores para cada hiperparámetro, pero este número podría llegar a ser mayor, acercándose más a aquellos valores que arrojen mejores resultados, hasta encontrar el óptimo en cada caso. Lo mismo ocurrre con la elección de hiperparámetros en sí, ya que nos hemos quedado con los cinco que consideramos más significativos, pero hay muchos más que pueden tener un impacto mayor o menor en el resultado y las métricas que medimos. Adicionalmente, también sería interesante realizar pruebas con otros conjuntos de imágenes distintos, ya sean los otros que incluyen los desarrolladores del modelo, o alguno nuevo que se tome para el estudio.

Otro paso natural en nuestra investigación sería considerar los nuevos avances llevados a cabo con respecto al modelo DirectVoxGo. Recientemente, este mismo año, se ha publicado un artículo que presenta el modelo DirectVoxGo++ [28]. Sería relevante estudiar las modificaciones que este modelo propone con respecto al que nosotros hemos analizado, y aplicar el procedimiento de afinamiento de hiperparámetros que hemos realizado, viendo así si obtenemos resultados diferentes y/o mejores.

Ampliando aún más el ámbito de nuestra investigación, podríamos aplicar un estudio similar al actual a otros modelos contemporáneos de "Neural Rendering", como podrían ser los recientes modelos MixVoxels [42] o DyNeRF [21]. Incluso podría ser interesante el realizar un estudio similar en el modelo base de NeRF [25], aunque esto supondría una mayor extensión temporal debido a la dilatada duración de las ejecuciones.

Finalmente, una futura línea de investigación más creativa sería la aplicación de algún al-

goritmo de inteligencia artificial a la identificación de combinaciones óptimas de parámetros. En este caso, debido a la naturaleza de lo que queremos obtener, pensamos que sería especialmente interesante probar el uso de algoritmos genéticos [33] [10]. En este caso, el cromosoma sería una combinación concreta de hiperparámetros, siendo cada gen uno de ellos, e iríamos retroalimentando el proceso evolutivo con los resultados de hacer pruebas con cada cromosoma, “matando” los cromosomas con peores resultados y mezclando aquellos que sean más óptimos. Este, o cualquier otro enfoque utilizando algoritmos inteligentes, pueden dar lugar a combinaciones mejores que las que podríamos encontrar “a mano”, y pueden resultar efectivos a la hora de manejar un gran número de hiperparámetros y valores.

References

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* (2017).

[2] Nicholas Barr. *Economics of the welfare state*. Oxford University Press, USA, 2020.

[3] Robert Browder and Carrie Cross. “Welcome to Overleaf: A Brief Overview of Opportunities”. In: (2018).

[4] Liang-Chieh Chen et al. “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* (2018).

[5] Pranjal Datta. *All about structural similarity index (SSIM): Theory + code in Pytorch*. Mar. 2021. URL: https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e.

[6] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[7] Pieter Thijs Eendebak and Alan Roberto Vazquez. “OApackage: A Python package for generation and analysis of orthogonal arrays, optimal designs and conference designs”. In: *The Journal of Open Source Software* 4.34 (Feb. 2019). DOI: 10.21105/joss.01097.

[8] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. *A Neural Algorithm of Artistic Style*. 2015. arXiv: 1508.06576 [cs.CV].

[9] github. *GitHub*. 2020. URL: https://github.com/.

[10] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[11] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[13] JetBrains. *Pycharm Community edition*. URL: https://github.com/JetBrains/intellij-community/tree/master/python.

[14] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2019. arXiv: 1812.04948 [cs.NE].

[15] Tero Karras et al. *Analyzing and Improving the Image Quality of StyleGAN*. 2020. arXiv: 1912.04958 [cs.CV].

[16] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML].

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems (NIPS)*. 2012.

[18] Leslie Lamport et al. *LATEX| User's Guide and Reference Manual"*. Addison-Wesley, 1994, 1996.

[19] *Learned Perceptual Image Patch Similarity (LPIPS) - PyTorch-Metrics 1.1.0 documentation*. URL: https://torchmetrics.readthedocs.io/en/stable/image/learned_perceptual_image_patch_similarity.html.

[20] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[21] Tianye Li et al. *Neural 3D Video Synthesis from Multi-view Video*. 2022. arXiv: 2103.02597 [cs.CV].

[22] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *European Conference on Computer Vision (ECCV)*. 2016.

[23] Andreas K. Maier et al. "Learning with known operators reduces maximum error bounds". In: *Nature Machine Intelligence* 1.8 (Aug. 2019), pp. 373–380. DOI: 10.1038/s42256-019-0077-5. URL: https://doi.org/10.1038%2Fs42256-019-0077-5.

[24] Ricardo Martin-Brualla et al. *NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections*. 2021. arXiv: 2008.02268 [cs.CV].

[25] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*. 2020.

[26] Manasa Nadipally. "Chapter 2 - Optimization of Methods for Image-Texture Segmentation Using Ant Colony Optimization". In: *Intelligent Data Analysis for Biomedical Applications*. Ed. by D. Jude Hemanth, Deepak Gupta, and Valentina Emilia Balas. Intelligent Data-Centric Systems. Academic Press, 2019, pp. 21–47. ISBN: 978-0-12-815553-0. DOI: https://doi.org/10.1016/B978-0-12-815553-0.00002-1. URL: https://www.sciencedirect.com/science/article/pii/B9780128155530000021.

[27] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[28] Daniel Perazzo et al. "DirectVoxGO++: Grid-based fast object reconstruction using radiance fields". In: *Computers Graphics* 114 (2023), pp. 96–104. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2023.05.027. URL: https://www.sciencedirect.com/science/article/pii/S0097849323000924.

[29] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[30] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[31] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *arXiv preprint arXiv:1506.01497* (2015).

[32] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *arXiv preprint arXiv:1505.04597* (2015).

[33] Jeffrey R. Sampson. “Adaptation in Natural and Artificial Systems (John H. Holland)”. In: *SIAM Review* 18.3 (July 1976), pp. 529–530. DOI: 10.1137/1018105. URL: https://doi.org/10.1137/1018105.

[34] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[35] Cheng Sun, Min Sun, and Hwann-Tzong Chen. *Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction*. 2022. arXiv: 2111.11215 [cs.CV].

[36] sunset1995. *directvoxgo: Direct voxel grid optimization for fast radiance field reconstruction*. June 2022. URL: https://github.com/sunset1995/DirectVoxGO.

[37] Chuanqi Tan et al. “A survey on deep transfer learning”. In: *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III 27*. Springer. 2018, pp. 270–279.

[38] Ayush Tewari et al. “Advances in neural rendering”. In: *Computer Graphics Forum*. Vol. 41. 2. Wiley Online Library. 2022, pp. 703–735.

[39] Ayush Tewari et al. “State of the art on neural rendering”. In: *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, pp. 701–727.

[40] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[41] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[42] Feng Wang et al. *Mixed Neural Voxels for Fast Multi-view Video Synthesis*. 2022. arXiv: 2212.00190 [cs.CV].

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga