# Enhancing CodeNeRF Predictions Using Transfer Learning and Input Segmentation

*Filip Balucha*


4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2023

# Abstract

Approaches that learn 3D scene representations using 2D supervision are a hot topic [11, 18]. These methods brought into the spotlight datasets such as SRN [41], which provides 2D renders of 3D objects. We propose Part-SRN, a novel dataset that augments SRN with part segmentations from PartNet [27], the largest dataset of segmented 3D models. With Neural radiance fields (NeRF) [23] at the forefront of current inverse graphics research, addressing NeRF's limitations is of particular importance. One such limitation is their restriction to a single scene. CodeNeRF [16] and pixelNeRF [60] are two promising works that allow NeRF to capture multiple scenes. CodeNeRF relaxes supervision requirements and is able to generate novel scenes at inference time, but it suffers from an incomplete code release and performs worse than pixelNeRF [15, 16]. The pixelNeRF architecture is more complex, and it does not support generating novel scenes [16, 60]. We present two methods to improve CodeNeRF's performance: EnCodeNeRF and SegCodeNeRF. EnCodeNeRF brings ideas from the better-performing pixelNeRF into CodeNeRF, while keeping CodeNeRF's distinguishing ability to generate novel scenes. SegCodeNeRF builds on EnCodeNeRF to inspect the potential of incorporating part-level information in NeRF-based architectures. While both modifications perform on par with CodeNeRF, we suggest directions likely to yield improvements.

# Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Filip Balucha*)

# Acknowledgements

I would like to thank Professor Hakan Bilen, my project supervisor, for having a clear vision throughout the project, offering useful ideas that kept the project moving and for his consistent guidance despite balancing an extremely busy schedule.

I would also like to thank Dr Octave Mariotti for introducing me to EfficientNet and PhD student Thomas Walker for useful discussions about CodeNeRF.

I would like to thank my family for their unconditional support throughout my studies, and my friends for believing in me at times when I did not. Ďakujem.

Finally, I would like to appreciate the open-source community for their selfless contributions to frameworks such as PyTorch Lightning, PyTorch, and blender, which this project relies on.

# Table of Contents

# Chapter 1

# Introduction

Neural Radiance Fields (NeRF) [23] was a watershed work in the field of implicit scene representations. NeRF was the first method to learn high-quality 3D scene representations using only 2D images as supervision [23]. As of 2022, 250 follow-up pre-prints based on NeRF were published, 100 of which were accepted in renowned computer vision conferences [11]. These works either extend the use of NeRFs from 3D reconstruction and novel view synthesis to new domains [11] or target NeRF's shortcomings [24].

A particular limitation of NeRF is that it must be trained anew for every scene, incurring significant computational cost [23]. pixelNeRF [60] offers a viable solution, albeit at the cost of an impenetrable architecture. CodeNeRF offers a trivial extension of the NeRF architecture [16]. While less performant than pixelNeRF, it relaxes inference-time supervision requirements and offers a way to generate novel scenes.

We look for a way to keep CodeNeRF's extensibility while achieving a pixelNeRF level of performance. We then inspect how additional supervision in the form of part-level segmentation impacts the performance of a NeRF-based architecture. To the best of our knowledge, this has not been done for a NeRF-based architecture before.

## 1.1 Contributions

(a) Reference (b) Official CodeNeRF (c) Reference (d) Ours

Figure 1.1: We address numerous bugs in the open-sourced CodeNeRF implementation [15], such as omitted activation function and discarding training signal.

We make the following contributions:

1. We propose architectural modifications to improve CodeNeRF's performance. We introduce EnCodeNeRF and SegCodeNeRF. SegCodeNeRF translates ideas from the better-performing pixelNeRF [60] into CodeNeRF, while keeping CodeNeRF's relaxed inference-time supervision requirements and ability to generate novel scenes [16]. SegCodeNeRF builds on EnCodeNeRF to inspect the potential of incorporating part-level information in NeRF-based architectures (Figure 1.2). While our work performs only on par with CodeNeRF, it highlights the extensibility of CodeNeRF and suggests directions for potentially fruitful follow-up work.
2. We rewrite CodeNeRF in a readable and extensible way that readily scales to multiple accelerators (Figure 1.1).
3. We introduce Part-SRN, the first dataset to augment the popular SRN dataset [40] with part annotations from PartNet [27], the largest collection of part-annotated 3D models. We also formalise the evaluation protocol for the popular SRN dataset [40], which is not defined in literature.

## 1.2 Results

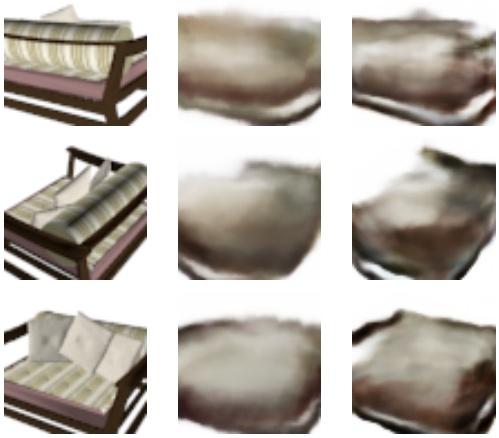



Figure 1.2: In the first half of the training procedure, SegCodeNeRF (ours) outperforms CodeNeRF even in the face of complex geometry.

## 1.3 Report structure

Chapter 2 motivates NeRFs, explains their weaknesses, and covers the background necessary to understand the proposed modifications. Chapter 3 explains existing datasets and introduces the novel Part-SRN dataset. In preparation for experiments, Chapter 4 explains our implementation of CodeNeRF. Chapters 5, 6 then propose and evaluate EnCodeNeRF and SegCodeNeRF, respectively. Finally, Chapter 7 proposes ways to improve the performance of EnCodeNeRF and SegCodeNeRF.

# Chapter 2

# Background

To motivate implicit neural scene representations, we compare them to classical approaches (Section 2.1.2). In Section 2.2, we focus on NeRF, the state-of-the-art framework in this area, and explain its shortcomings along with relevant follow-up work. Targeting a particularly limiting shortcoming of NeRFs, we introduce pixelNeRF and CodeNeRF in Sections 2.3 and 2.4. We conclude with a brief introduction of EfficientNet (Section 2.6), a family of convolutional encoders which lies at the heart of our proposed architectural modifications.

## 2.1 Preliminaries

### 2.1.1 Rendering

Rendering is the process of generating images of a 3D scene. The scene is defined by a scene representation (Figure 2.1, right) [17]. *Differentiable* rendering is a rendering technique that allows one to compute the gradients of a rendered image with respect to the input scene representation [17]. This makes the rendering technique optimisable using gradient-based optimisation methods. *Inverse* rendering reverses the rendering process. It is the process of reconstructing an unknown scene representation using rendered images (Figure 2.1, left) [19, 22]. If the inverted renderer uses a gradient-based method for optimisation, differentiable rendering can be used to obtain the gradients.

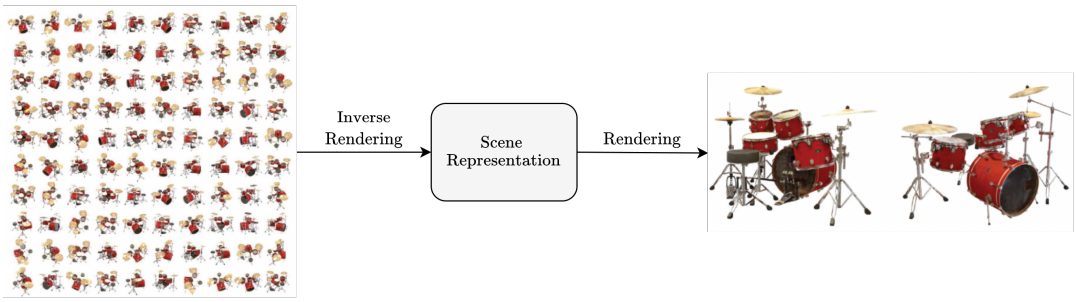



Figure 2.1: During rendering, a scene representation is used to produce views of a scene (right). Inverse rendering takes such views and reconstructs the underlying scene representation (left). (Figure adapted from [12].)

### 2.1.2 Scene representations

A scene representation describes a scene in terms of parameters such as geometry, materials and light sources [17]. We discuss four data representation classes commonly used in computer graphics in the context of differentiable rendering.

(a) Mesh (b) Voxel (c) Point cloud

Figure 2.2: Scene representations rely on different data representations. We illustrate three of the four main ones, omitting implicit scene representations, whose representations are problem-specific. (Figure adapted from [12].)

#### 2.1.2.1 Mesh-based scene representations

Meshes represent a scene using a set of vertices, interconnected using surfaces (Figure 2.2a). During rendering, the surfaces are projected onto the camera. Mesh-based rendering methods are non-differentiable [17]. [17] identify methods to introduce differentiability, though these compute *approximate* gradients.

#### 2.1.2.2 Voxel grid-based scene representations

Voxel grids split 3D space into cubes of uniform size, called voxels (Figure 2.2b). Each voxel encodes information about its point in space (e.g., transparency, material). Voxel-based renderers are differentiable [17], however their space complexity depends on resolution: a higher resolution requires a finer sampling of 3D space. Recent voxel grid-based representations require over 15 GB to store a realistic scene [23].

#### 2.1.2.3 Point cloud-based scene representations

Similar to voxel grids, point clouds represent scenes using a set of points with feature vectors (Figure 2.2c). Point clouds can be non-uniformly distributed in space. Thus, points can be omitted on low-density regions, resulting in a lower memory footprint. There are several rendering techniques for point clouds, some of which are differentiable. However, the non-uniform representation of space requires modelling the influence of point cloud points on a sampled point in space. This makes point cloud-based rendering challenging [17].

#### 2.1.2.4 Implicit neural scene representations

Implicit scene representations capture scene parameters in neural network weights. Similar to voxel grids, implicit representations densely model scene parameters. Since scene parameters are encoded in a fixed-size model, the memory footprint of implicit scene representations is constant with respect to resolution [17]. If a differentiable neural network is used, the rendering method can be designed to be differentiable.

In light of recent advancements in neural architectures, there has been increased interest in implicit neural representations [17]. DeepSDF [30] uses neural networks to model the distance to surface boundaries to obtain object silhouettes. SRN [41] models feature vectors combined to produce renders. NeRF [23] simply outputs model volume density and RGB colour, producing state-of-the art renders.

#### 2.1.2.5 Summary of scene representations

Table 2.1 summarises our discussion of neural scene representations.

| Representation | Space-efficiency | Render quality | Differentiable rendering |
|---|---|---|---|
| Mesh | ✓ | ✓ | ✗ |
| Voxel grid | ✗ | ✓ | ✓ |
| Point cloud | ✓ | ✗ | ✓ |
| Implicit | ✓ | ✓ | ✓ |

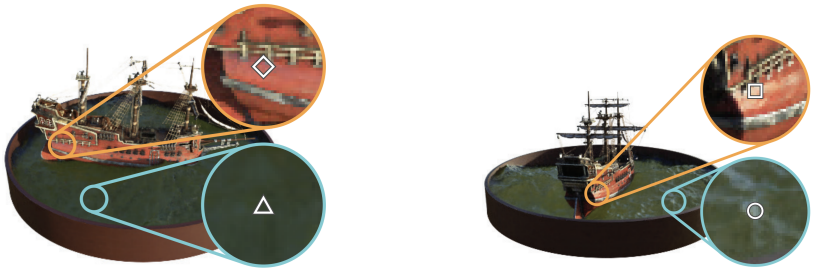
Table 2.1: From among the main scene representations, only implicit representations offer space efficiency and good render quality while supporting differentiable rendering.

## 2.2 Neural radiance fields

Neural radiance fields (NeRF) presented the first implicit neural scene representation able to render photorealistic views of real scenes and objects at high resolutions. NeRF has since become the cornerstone of a rich body of research [11, 18].

### 2.2.1 Scene representation

NeRF represents a scene using a fully-connected network $F_\Theta$. $F_\Theta$ maps a 3D spatial location $\mathbf{x}$ and viewing direction $\mathbf{d}$ to volume density $\sigma$ and RGB colour $\mathbf{c} = (r, g, b)$ [23].



(a) View A (b) View B

Figure 2.3: The colour of a point in space depends on viewing direction. View A and B display a point on the water surface and on the galleon's hull, respectively. The viewing direction in View A is aligned with incident light rays. In View B, it is at an angle, and the points exhibit specular reflection. (Figure adapted from [23].)

Intuitively, volume density does not depend on the viewing direction $\mathbf{d}$, while Colour $\mathbf{c}$ does (Figure 2.3). Accordingly, the NeRF architecture is divided into a shape and a texture subnetwork. The shape subnetwork $F_\Theta^S$ takes as input the 3D spatial location $\mathbf{x}$ and outputs volume density and an intermediate feature vector $\mathbf{v}$. The texture subnetwork $F_\Theta^T$ takes as input $\mathbf{v}$ and the viewing direction $\mathbf{d}$, and it outputs the RGB colour $\mathbf{c}$.

$$
\begin{aligned}
F_{\Theta_s}^S &: \mathbf{x} \rightarrow (\mathbf{v},\ \sigma) \\
F_{\Theta_t}^T &: (\mathbf{v},\ \mathbf{d}) \rightarrow \mathbf{c} \\
F_\Theta &: (\mathbf{x},\ \mathbf{d}) \rightarrow (\mathbf{c},\ \sigma)
\end{aligned}
$$

### 2.2.2 Positional encoding

The authors report that during initial experiments, NeRF was unable to model high-frequency variation in color and geometry [23]. [32] have shown that deep neural networks learn low-frequency functions. [46] discuss this in the context of fully-connected networks learning to represent images by mapping 2D coordinates to pixel colours. The authors propose to map MLP inputs into a higher-dimensional space before passing them to the network in a technique called Fourier feature encoding. Accordingly, in NeRF the spatial location $\mathbf{x}$ and viewing direction $\mathbf{d}$ are encoded in a higher-dimensional space using the encoding function [23]:

$$
\gamma(p) = \left(\sin\left(2^0\pi p\right),\ \cos\left(2^0\pi p\right),\ \cdots,\ \sin\left(2^{L-1}\pi p\right),\ \cos\left(2^{L-1}\pi p\right)\right)^\top
$$

$\gamma(\cdot)$ is applied to each component in the input vector respectively, and the outputs are concatenated. $L$ was empirically determined to be 10 for $\gamma(\mathbf{x})$ and 4 for $\gamma(\mathbf{d})$.

### 2.2.3 Rendering

Having covered NeRF at a high-level, we now discuss how NeRF can be used to render images. NeRF casts a ray through each pixel, queries the network $F_\Theta$ at a sampled set of

points and composes the outputs to obtain pixel colour. This technique is called volume rendering (Figure 2.4).

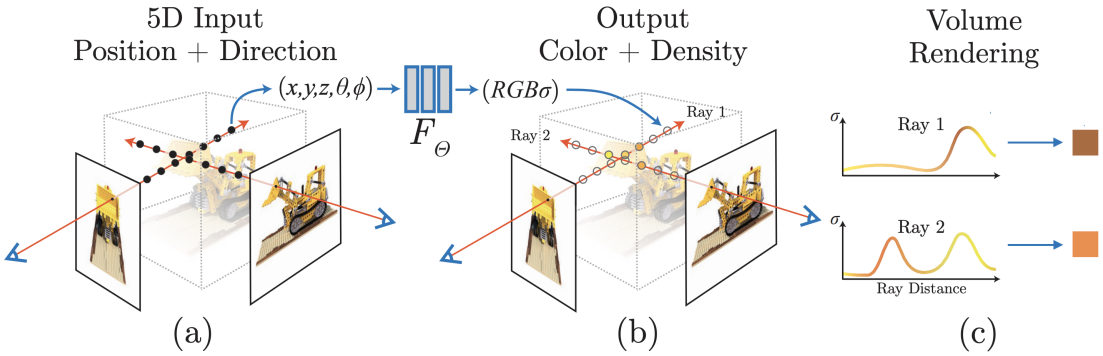



Figure 2.4: To render an image, NeRF casts a ray through each pixel and samples points along it (a). The network is queried at each point, producing a sequence of volume densities and colours (b), combined to produce pixel colour (c) [23]. (Adapted from [23].)

#### 2.2.3.1 Pixel colour

A ray is represented using the line segment:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d},\ t \in [t_n, t_f]$$

where $\mathbf{o}$ is the camera's centre of projection, and $t_n$ and $t_f$ delimit the points that are visible by the camera [1]. The ray $\mathbf{r}(t)$ is partitioned into $N$ equally-sized bins. Within each bin, a sample $t_i$ is drawn uniformly at random:

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

The sampling is repeated during each forward pass. Thus, in the limit, the network is evaluated at continuous positions. The model is queried at the corresponding point $\mathbf{r}(t_i)$. This produces a sequence of volume densities and colours $((\sigma_i, c_i))_{i=1}^N$. The projected pixel colour of $\mathbf{r}(t)$ is computed using:

$$\hat{\mathbf{C}}(\mathbf{r};\ \Theta) = \sum_{i=1}^{N} T_i \alpha_i \mathbf{c}_i \tag{2.1}$$

where

$$\alpha_i = (1 - \exp(-\sigma_i \delta_i)), \text{ where } \delta_i = t_{i+1} - t_i \tag{2.2}$$

represents how much light is contributed by the segment from which $t_i$ was drawn, and

$$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{2.3}$$

represents the amount of light blocked by preceding ray segments [23] (Figure 2.5). We refer the keen reader to an illuminating tutorial [24] by the authors of NeRF for an intuitive derivation of these equations using probabilistic concepts.

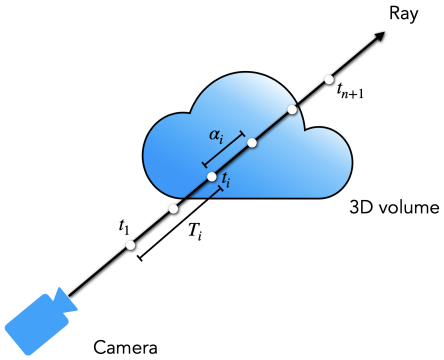



Figure 2.5: For a ray cast from the camera, $T_i$ represents how much light is blocked by preceding ray segments, and $\alpha_i$ captures how much light is contributed by the ray segment from which the sample $t_i$ was drawn [23]. (Figure retrieved from [24].)

#### 2.2.3.2 Differentiability

NeRF produces renders by casting rays through pixels, sampling points along each ray, and compositing outputs of the model at each point onto a pixel, as shown in Equation 2.1. We expand the equation used to produce pixel colours (Equation 2.1) using its constituents (Equations 2.2, 2.3) to highlight the connection to model inputs:

$$\begin{aligned}\hat{\mathbf{C}}(\mathbf{r};\,\Theta) = \mathbf{C}(\mathbf{o}+t\mathbf{d};\,\Theta) &= \sum_{i=1}^{N} T_i \alpha_i \mathbf{c}_i \\ &= \sum_{i=1}^{N} \left( \prod_{j=1}^{i-1} \exp(-\sigma_j \delta_j) \right) (1-\exp(-\sigma_i \delta_i))\mathbf{c}_i \end{aligned} \tag{2.4}$$

where $(\mathbf{c}_i, \sigma_i) = F_\Theta(\mathbf{r}(t_i), \mathbf{d})$. By inspection, Equation 2.4 is differentiable with respect to the model outputs $(\mathbf{c}_i, \sigma_i)$. The model $F_\Theta$ uses a differentiable neural architecture. Thus, the NeRF rendering procedure is differentiable, and hence suitable for optimisation using gradient-based methods.

#### 2.2.3.3 Hierarchical sampling

NeRF trains a coarse and a fine network. The coarse network uses a uniform sampling procedure and learns the distribution of volume in the 3D space. The fine network uses the coarse network as a prior when sampling to allocate more samples to regions with higher density and hence a larger impact on the final render (Equation (2.1)). The hierarchical sampling procedure is of little relevance to our work, so we refer the reader to the original work [23] for more details.

### 2.2.4 Architecure

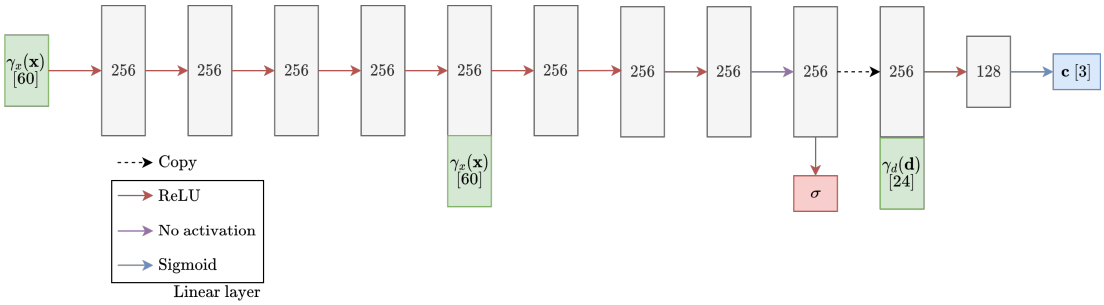



Figure 2.6: The NeRF architecture uses a fully-connected network to represent a scene. At any given point in space, the scene is represented using volume density $\sigma$ and colour $\mathbf{r}$.

The NeRF architecture uses a fully-connected model to represent scenes (Figure 2.6). The network is queried and the outputs are combined as described in Section 2.2.3.1. This architecture is strikingly simple compared to concurrent work, such as SRN [41]. SRN uses a fully-connected network to represent scenes, a recurrent model that dictates the sampling process, and a convolutional network that combines scene representations seen along a ray [41].

### 2.2.5 Training

Using a set of images with camera poses, NeRF minimises the L2 loss between the observed and the predicted image:

$$\min_{\Theta} \sum_{\mathbf{r}\in\mathcal{R}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2 \tag{2.5}$$

where $\mathcal{R}$ is the set of rays corresponding to the image, and $\mathbf{C}(\mathbf{r})$ is the observed pixel colour [23]. We remark that pixels are usually processed in chunks:

$$\sum_{\mathbf{r}\in\mathcal{R}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2 = \sum_{c} \underbrace{\sum_{\mathbf{r}\in\mathcal{R}_c} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2}_{\text{Chunk}}$$

where $\bigcup_c \mathcal{R}_c = \mathcal{R}$. This limits the number of computational graphs stored in memory at any time, reducing memory requirements.

### 2.2.6 Limitations and follow-up work

NeRF presented a paradigm shift in the field of implicit neural representations, outperforming concurrent methods. NeRF is now the backbone of the majority of neural rendering work. Yet, as we now discuss, it has several limitations.

We now discuss the ones relevant to our work. We refer the keen reader to a survey of neural field-based methods [56]. For works based on NeRF, we refer the reader to a

survey paper [11] and a code repository [18] that links the implementation for several NeRF-based works.

#### 2.2.6.1 Rendering and training speed

To render an image, the network must be queried at $\sim 100$ points samples per pixel. Generating a single render takes $\sim 30$ seconds [23]. This limits the use of NeRF-based models in real-time rendering pipelines, such as the ones used in the gaming industry. Follow-up work has targeted the compute per sample, reducing rendering time to fractions of a second. InstantNGP [28] cache network outputs using hash maps. KiloNeRF [33] uses a divide-and-conquer approach, dividing the large network into multiple smaller ones.

Training a NeRF takes at least 12 hours per scene on a single Tesla V100 research-grade GPU. By contrast, some discretised approaches are trainable within minutes [23]. Since training speed directly depends on the number of network queries, the approaches targeting rendering speed simultaneously reduce training speed.

#### 2.2.6.2 Generalisation across scenes

A NeRF model represents a single scene. If one wants to model a different yet related scene, a new NeRF model must be trained from scratch. This is exacerbated by NeRFs' large training cost.

Concurrent work on SRNs [41] achieves generalisation across scenes by equipping each scene instance with a latent code. Follow-up work on CodeNeRF [16] similarly uses per-instance latent codes. When combined with intermediate NeRF features, these allow for cross-scene generalisation.

#### 2.2.6.3 Camera pose supervision

NeRF requires camera poses for each input image. While synthetic datasets (e.g. SRN [38]) provide this information, on real-life datasets, additional preprocessing is required to extract camera poses (e.g. using the COLMAP structure-from-motion package [35]). CodeNeRF is able to infer camera pose at inference time [16], while ViewNeRF [21] is able to infer camera poses in a fully self-supervised manner.

## 2.3 pixelNeRF

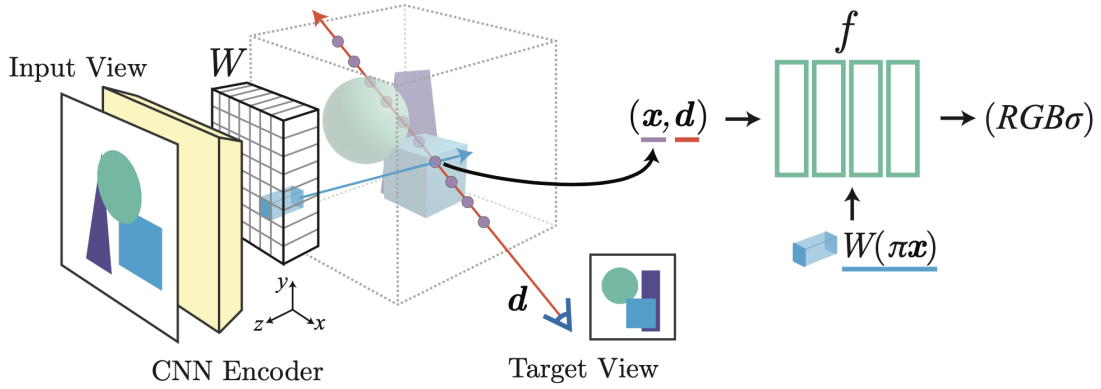



Figure 2.7: pixelNeRF augments NeRF with a CNN encoder-based prior. Given a position $\mathbf{x}$ and viewing direction $\mathbf{d}$, a feature is extracted from the feature volume $\mathbf{W}$ corresponding to one or two training viewpoints, randomly selected from among 50, and input into a modified NeRF model $f$. The output RGB and volume density σ then feed into a volume-rendering pipeline. (Adapted from [60].)

pixelNeRF equips NeRF with a scene prior. This allows it to reuse weights across scenes, eliminating NeRF's restriction to a single scene [60]. NeRF applies to novel view synthesis, where a view of a *known* scene is generated from a new viewpoint. By contrast, pixelNeRF's prior extends its applicability to few-shot estimation. In few-shot estimation, views of an *unknown* scene are generated using few input images.

To extract scene specifics, pixelNeRF passes input images through a pre-trained ResNet34 convolutional encoder, pre-trained on the ImageNet-1K dataset. Intermediate features are extracted from the encoder, upsampled using bilinear extrapolation to match the image size, and concatenated to obtain a pixel-aligned feature grid [59, 60]. The feature grid is then combined with NeRF's intermediate features (Figure 2.7). We refer the reader to the original work [60] for further exposition of the combining procedure.

pixelNeRF outperformed related approaches (SRN [41], DVR [29]) by a significant margin, so we do not discuss those here.

### 2.3.1 Limitations

pixelNeRF eliminates NeRF's limiting applicability to single scenes. However, it has limitations. Firstly, pixelNeRF still requires supervision in the form of camera poses for every input view. Secondly, pixelNeRF does not offer a way to leverage the decoupling of scene-specific and shared information. Movement within the scene-specific space could be used to generate new scenes. Finally, we remark that the pixelNeRF architecture as outlined in the paper can be impenetrable, though this can be supplemented by the open-sourced implementation [59].

## 2.4 CodeNeRF

Like pixelNeRF, CodeNeRF [16] addresses NeRF's single-scene limitation. pixelNeRF extracts scene specifics from the intermediate features obtained from a pre-trained convolutional encoder [60]. By contrast, CodeNeRF optimises a pair of latent codes for each scene. While CodeNeRF performs worse than pixelNeRF [16], it offers several benefits:

- **Generation of novel instances.** Choosing new points within the respective embedding spaces to which the shape and texture codes learnt by CodeNeRF belong results in the generation of novel scenes (Figure 2.8).
- **Inference-time camera pose estimation.** CodeNeRF supports camera pose optimisation at test-time. This relaxes the supervision requirements compared to related work, such as pixelNeRF[60].
- **Architectural simplicity.** CodeNeRF's architecture is a minimal extension of the underlying NeRF architecture, making it accessible and readily extensible.

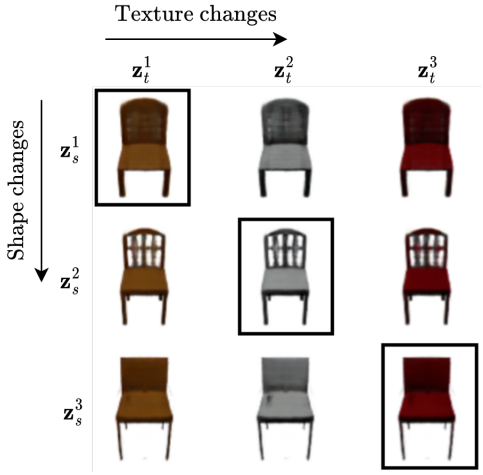



Figure 2.8: CodeNeRF captures scene specifics using a shape and texture latent code. Movement within the shape or texture embedding space results in the generation of novel shapes. The diagonal entries show observed scenes. Off-diagonal entries are obtained by replacing the shape or texture code (e.g. last entry in row 1 uses the original shape code $\mathbf{z}_s^1$ but borrows its texture code $\mathbf{z}_t^3$ from the third model). (Adapted from [16].)

### 2.4.1 Notation

Since CodeNeRF constitutes the backbone of our exploration, we now offer a more formal treatment. CodeNeRF considers a dataset $\mathcal{D}$ of $M$ models, where each model consists of camera intrinsics $K$ and $V$ viewpoints containing the image $I$ and camera pose $T$:

$$\mathcal{D} = \left\{ (I^{m,v},\ T^{m,v},\ K^m) \ \middle|\ m \in 1,\ldots,M;\ v \in 1,\ldots,V \right\}$$

CodeNeRF optimises a shape and texture latent code for each model $m$, denoted $\mathbf{z}_s^m$ and $\mathbf{z}_t^m$, respectively. Each code is shared by all $|V|$ viewpoints.

Similar to NeRF, CodeNeRF is split into a shape and texture subnetwork, $F^S_{\Theta_s}$ and $F^T_{\Theta_t}$. The shape subnetwork takes as additional input the shape latent code; the texture subnetwork works analogously:

$$
\begin{aligned}
F^S_{\Theta_s} &: (\mathbf{x};\, \mathbf{z}^m_s) \to (\mathbf{v},\, \sigma) \\
F^T_{\Theta_t} &: (\mathbf{v},\, \mathbf{d};\, \mathbf{z}^m_t) \to \mathbf{c} \\
F_{\Theta} &: (\mathbf{x},\, \mathbf{d};\, \mathbf{z}^m_s,\, \mathbf{z}^m_t) \to (\mathbf{c},\, \sigma)
\end{aligned}
$$

### 2.4.2 Architecture

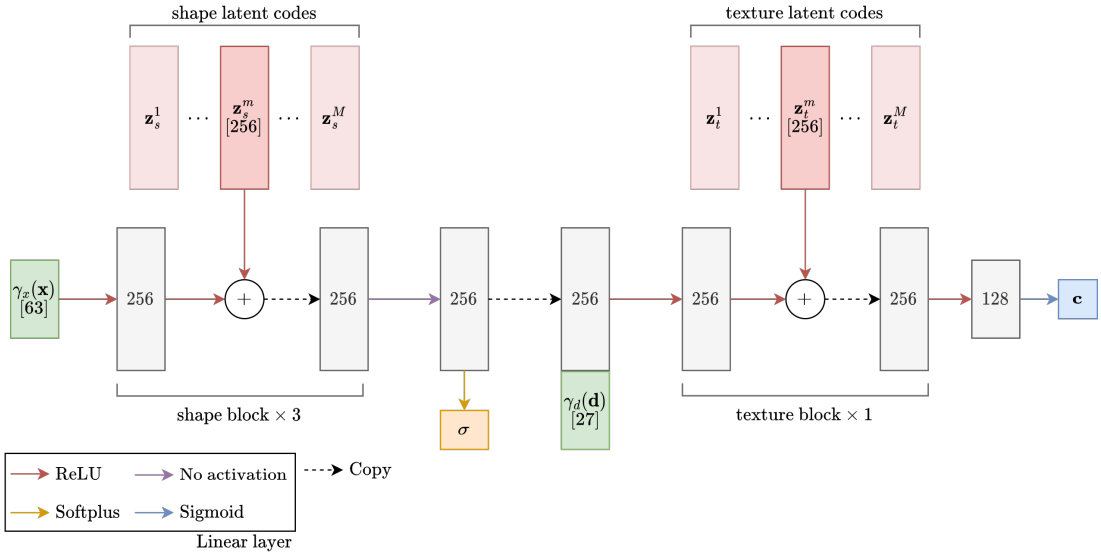



Figure 2.9: CodeNeRF builds on the NeRF (Figure 2.6) architecture. For each of the $M$ models, it keeps a shape and texture latent code ($\mathbf{z}^m_s$, $\mathbf{z}^m_t$). The codes are randomly initialised, combined with the intermediate features of the main NeRF network via linear projection layers with ReLU nonlinearity, and optimised during backpropagation.

CodeNeRF views NeRF as a combination of two subnetworks: a *shape* subnetwork that outputs volume density, and a *texture* subnetwork that outputs colour (Figure 2.6). CodeNeRF augments either with a scene-specific latent code (Figure 2.9) [16].

### 2.4.3 Loss

While NeRF and pixelNeRF use L2 loss [23, 60], CodeNeRF uses the mean-squared error (MSE) loss. For $n$-dimensional sequences $Y$, $\hat{Y}$, these are related via:

$$
\mathrm{MSE}(Y,\hat{Y}) = \frac{1}{n}\,\mathrm{L2}(Y,\hat{Y}) = \frac{1}{n}\,(Y-\hat{Y})^\top (Y-\hat{Y})
$$

We hypothesise that CodeNeRF uses MSE loss due to a cropping procedure, wherein epochs $1,\ldots,250$ use images cropped at the centre to 25% of their original size, and

subsequent epochs use full images. This is equivalent to changing sequence length $n$ after epoch 250, which would result in a loss increase in the case of L2.

### 2.4.4 Limitations

#### 2.4.4.1 Shape-texture entanglement

The authors claim that CodeNeRF decouples the shape and texture codes. While the two codes may be separated by design, we show in Section B.3.2 that perfect decoupling is theoretically impossible.

#### 2.4.4.2 Unclear architecture diagram

The architecture diagram presented in the original CodeNeRF paper [16] is easy to misinterpret due to a lack of notation and visual monotony. We provide an adapted diagram in Figure 2.9.

#### 2.4.4.3 Reproducibility

The official code release [15] does not include the implementation for one of its main contributions, inference-time camera pose and latent code optimisation. This has been a repeated discussion point [36, 62]. We contacted the authors and have been informed that they are working on a code release. Furthermore, the publication does not specify the training environment, including information such as the number of epochs or hardware used. Similar to pixelNeRF, it claims to follow the SRN evaluation protocol but does not state it. Finally, as discussed previously, CodeNeRF uses a cropping procedure and MSE loss, though this is only mentioned in the implementation.

#### 2.4.4.4 Omission of metrics

While CodeNeRF offers a quantitative comparison to related work, it focuses on few-shot estimation without reporting performance on novel view synthesis. Furthermore, it does not report the LPIPS metric common in NeRF-based literature, including CodeNeRF's main contender pixelNeRF [60]. Since pre-trained weights are not made accessible [15], this reduces CodeNeRF's comparability with related work.

#### 2.4.4.5 Redundant regularisation

CodeNeRF's optimisation problem in full is:

$$\min_{\Theta,\ ((\mathbf{z}_s^m,\ \mathbf{z}_t^m))_{m=1}^M} \sum_{\mathbf{r}\in\mathcal{R}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2 + \underbrace{\gamma\left(\|\mathbf{z}_s^m\|_2^2 + \|\mathbf{z}_t^m\|_2^2\right)}_{\text{Regularisation}} \tag{2.6}$$

where the regularisation term encourages latent codes to have low norms (and hence cluster around the origin). However, the AdamW optimiser [20] provides weight regularisation that is 2 orders of magnitude higher than $\gamma$, making the regularisation term redundant.

## 2.5 Summary of neural rendering methods

Table 2.2 offers a summary of the neural scene representations discussed in Sections 2.2, 2.3, and 2.4.

| | NeRF [23] | pixelNeRF [60] | CodeNeRF [16] |
|---|---|---|---|
| Novel-view synthesis | ✓ | ✓ | ✓ |
| Few-shot estimation | ✗ | ✓ | ✓ |
| Multi-scene | ✗ | ✓ | ✓ |
| Multi-category | ✗ | ✓ | ✗* |
| Camera pose supervision | ✗ | ✗ | ✓ |
| Scene interpolation | ✗ | ✗* | ✓ |

Table 2.2: NeRF [23] serves as a framework for a rich body of research. partCodeNeRF [60] and CodeNeRF [16] eliminate NeRF's limitation to single-scene. While partCodeNeRF is applicable to diverse scenes, CodeNeRF allows one to generate novel scenes via scene interpolation. ($*$ implies this facet of the model has not been explored.)

## 2.6 EfficientNet

EfficientNet is a novel family of convolutional encoders [44, 45]. It achieved state-of-the-art results on the well-known ImageNet classification benchmark [34] using nearly one order of magnitude fewer parameters and floating-point operations (FLOPs) [44, 45] than the well-established ResNet [13] and the concurrent Vision Transformer [10]. In the context of NeRF, ViewNeRF [21] has recently deployed EfficientNet on the task of pose estimation.

# Chapter 3

# Datasets

We first introduce existing datasets relevant to our work and explain the relationships between them. Then, we propose a new dataset, Part-SRN, which builds on these datasets. This dataset drives our exploration in Section 6.

## 3.1 ShapeNet

The ShapeNet dataset consists of over 3,000,000 3D object models. 220,000 of these models are organised into 3,135 categories (e.g. "chair", "car") called "synsets" [4].

In our experience, related works use ShapeNetCore. ShapeNetCore is a subset of ShapeNet consisting of $\sim 51,000$ 3D models, covering 55 common synsets [37]. To the best of our knowledge, ShapeNetCore is the largest publicly-available 3D model dataset. Therefore, we select ShapeNetCore as the backbone of our dataset Part-SRN.

We now introduce SRN, a derivative of ShapeNetCore often used in recent literature on implicit neural representations [16, 41, 60].

## 3.2 SRN

The SRN dataset consists of 2D renders of ShapeNetCore (version 2) 3D models. Each render is augmented with a camera pose [41]. Thus, the SRN dataset lends itself to inverse graphics tasks such as novel view synthesis and few-shot estimation. SRN is used in recent literature focussed on inverse graphics tasks [16, 41, 60]. To remain backwards-compatible, Part-SRN follows the structure of SRN.

### 3.2.1 SRN evaluation protocol

Several publications allude to the SRN evaluation protocol [16, 41, 60] without stating it. We formalise the SRN evaluation protocol for the Chair synset (Table 3.1), with Car in the Appendix (Section A.1.2). The values were determined by manual inspection of the SRN dataset [40]. We believe this summary will increase the accessibility of

the SRN dataset and facilitate the generation of derivatives using the open-sourced generation code [39].

| Subset name | Parent | Application | # Renders | Sampling method |
|---|---|---|---|---|
| `train` | $\mathcal{D}_{\text{train}}$ | Training | 250 | $\mathcal{R}$ |
| `train_val` | $\mathcal{D}_{\text{train}}$ | Novel view synthesis | 10 | $\mathcal{R}$ |
| `train_test` | $\mathcal{D}_{\text{train}}$ | Novel view synthesis | 251 | $\mathcal{S}$ |
| `val` | $\mathcal{D}_{\text{val}}$ | Few-shot reconstruction | 251 | $\mathcal{S}$ |
| `test` | $\mathcal{D}_{\text{test}}$ | Few-shot reconstruction | 251 | $\mathcal{S}$ |

Table 3.1: The SRN evaluation protocol for the Chair synset.

Informed by the requirements of the original publication [41], SRN is split into five subsets [40]: one for training, two for novel view synthesis, and two for few-shot estimation. Each subset sources its 3D models from one of three parent datasets (Table 3.2) and uses one of two sampling methods (Figure 3.1).

| Parent name | Abbreviation | % Models |
|---|---|---|
| Training | $\mathcal{D}_{\text{train}}$ | 70% |
| Validation | $\mathcal{D}_{\text{val}}$ | 10% |
| Test | $\mathcal{D}_{\text{test}}$ | 20% |

Table 3.2: The SRN evaluation protocol splits the ShapeNetCore dataset $\mathcal{D}$ into three datasets of 3D models. These are the basis of the rendering process in Table 3.1.

(a) Random points on a sphere ($\mathcal{R}$)

(b) Spherical spiral ($\mathcal{S}$)

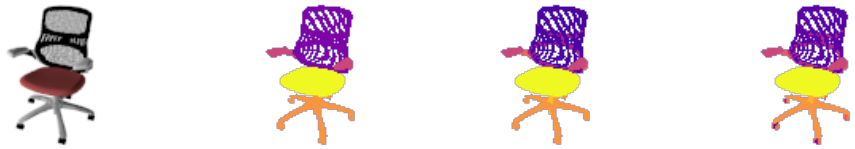
Figure 3.1: The SRN rendering pipeline uses one of two sampling methods, depending on the dataset. In $\mathcal{R}$, camera poses are sampled uniformly at random on the surface of a sphere with a fixed radius and the model at the centre [39]. On the `train` subset, this allows subsampling by taking the first $x$ out of 250 renders without biasing the sample. In $\mathcal{S}$, camera poses are sampled along a spherical spiral centred at the object [39]. This ensures thorough coverage of all possible viewpoints.

## 3.3 PartNet

We have discussed ShapeNetCore (Section 3.1), a large-scale dataset that provides 3D models across spanning 55 categories ("synsets"), and SRN (Section 3.2), a dataset of 2D renders based on ShapeNetCore. We now discuss an extension of ShapeNetCore called PartNet.

PartNet provides part segmentations for over 26,000 3D models across 24 synsets. PartNet augments a subset of ShapeNetCore with part segmentations. However, it does not include ShapeNetCore's texture information. Therefore, graphics tasks that require object appearance must use both PartNet and ShapeNetCore.

### 3.3.1 Granularity and hierarchy



(a) Original model (b) Low granularity (c) Medium granularity (d) High granularity

Figure 3.2: PartNet augments 3D models from ShapeNetCore (3.2a) with part segmentations. Segmentations support up to granularity levels: low, medium, and high. Each offers increasingly more detail: Medium granularity splits the armrest and backrest, and High separates wheels from the rest of the leg set.

In PartNet, a model's constituent parts are organised into a hierarchy. The degree of expansion of the hierarchy depends on the granularity level [27]. Each synset provides up to three granularity levels (Figure 3.2).

### 3.3.2 Motivation

To the best of our knowledge, PartNet is the largest publicly-available dataset with hierarchical, part-level annotations. Further, PartNet's use of ShapeNetCore models ensures compatibility with ShapeNetCore and, by extension, SRN. Therefore, we choose PartNet to complement ShapeNetCore as the backbone of Part-SRN.

## 3.4 Part-SRN

To the best of our knowledge, no publicly-available dataset augments ShapeNetCore renders with part-level information from PartNet. Yet, a large-scale dataset of 3D model renders with part-level information could constitute additional supervision to inverse graphics tasks, or serve as a segmentation benchmark. To this end, we propose a new dataset called Part-SRN.

(a) Original image (b) Low granularity (c) Medium granularity (d) High granularity

Figure 3.3: Part-SRN is a novel dataset that augments 2D renders from the popular SRN dataset with part segmentations from PartNet. The segmentations span up to three granularity levels.

### 3.4.1 Related work

Segmentations of 3D scenes are well-resourced, particularly in the field of autonomous driving [2] and indoor scene reconstruction [3, 8, 43]. However, these datasets work with outward-facing scenes, to which NeRF-based methods cannot be applied [23]. In addition, they segment at the object- rather than part-level. This lack of resources sometimes motivates the generation of in-house datasets. For instance, [53] developed three datasets for their study of NeRF-based object segmentation.

ShapeNetCore models have been segmented into constituent parts on multiple occasions [27, 58]. Datasets such as the popular SRN [40] offer 2D renders of ShapeNetCore models. The use of large-scale, standardised datasets promotes paper comparability even if source code is not made publicly available [60]. Yet, no dataset offers 2D renders of ShapeNetCore models with part-level segmentation. Admittedly, the advancements in neural rendering, which relaxed the need for 3D supervision in inverse rendering, are relatively recent; the NeRF paper [23] was published in 2020. To fill in the gap, we propose Part-SRN.

Part-SRN augments the SRN dataset with part-level segmentations from PartNet, across all available granularity levels. The pipeline used to generate Part-SRN readily applies to any synset present in ShapeNetCore and PartNet.

Our efforts abstract away the complexity of drawing correspondences between PartNet and ShapeNetCore both in terms of metadata and 3D structure and extracting semantic masks from 3D parts using 3D engines. Thus, we believe Part-SRN will result in significant time savings for those interested in combining ShapeNetCore and PartNet. For the sake of transparency, we now discuss the steps taken to generate Part-SRN.

### 3.4.2 Part-SRN generation pipeline

Despite being related, PartNet and ShapeNetCore models bear several differences. PartNet models are not aligned to ShapeNetCore, use different model identifiers, and are not based on the latest version of ShapeNetCore. Discussion about PartNet-ShapeNetCore deviations is dispersed throughout related code repositories [26, 39] or conveyed implicitly by dataset structure. This presents an obstacle to usability.

We first explain how the ShapeNetCore version used in Part-SRN was selected. Then, we explain the steps taken to produce Part-SRN. The code and steps to recreate the dataset will be made publicly available upon submission.

#### 3.4.2.1 Choose ShapeNetCore version

There are two ShapeNetCore versions: v1 and v2. Each uses a different normalisation method [37]. Thus, the same model will not be aligned across ShapeNetCore versions.

PartNet models are based on ShapeNetCore v1 [31]. PartNet models are transformed with respect to their ShapeNetCore v1 counterparts. The authors provide transformation matrices that align PartNet models to ShapeNetCore v1 [31]. However, ShapeNetCore v1 models are themselves transformed with respect to ShapeNetCore v2. To the best of our ability, we were unable to align PartNet models with ShapeNetCore v2. Therefore, we decided to use ShapeNetCore v1 as the backbone of Part-SRN.

#### 3.4.2.2 Remove invalid ShapeNetCore and PartNet models

PartNet is a subset of ShapeNetCore. We require that every 3D model from ShapeNetCore have corresponding part-level supervision. To this end, we discard certain models. We explain the reasons below. Figure 3.4 summarises the impact of the filtering steps on the number of models used in Part-SRN.

1. **Missing PartNet model.** PartNet consists of a subset of ShapeNetCore models. We filter out ShapeNetCore models not in PartNet.
2. **Missing transformation.** For rendering purposes, each PartNet model must be aligned to ShapeNetCore. We filter out ShapeNetCore models for which there is no PartNet-ShapeNetCore transformation matrix. Missing alignments can be generated using author-provided code [31]. This affects only 50 ($< 1\%$) models, so we do not implement this.
3. **Invalid PartNet hierarchy.** For each synset and granularity, the authors provide a list of parts [25]. Each part may or may not be in a model's hierarchy (e.g. a chair may or may not have an armrest). We noticed that some models do not support all granularities for a part they contain (Figure 3.5). We refer to these models as having an invalid hierarchy, and we filter them out.

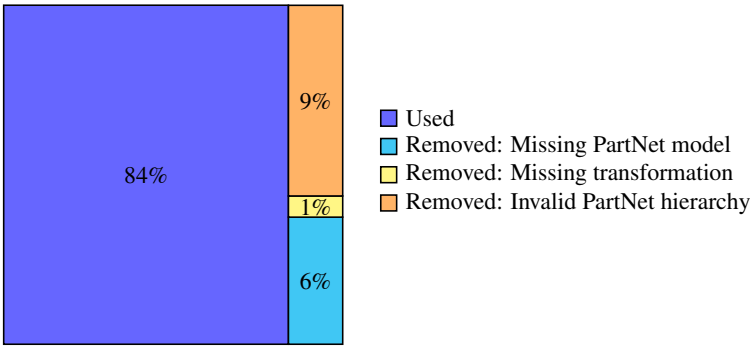
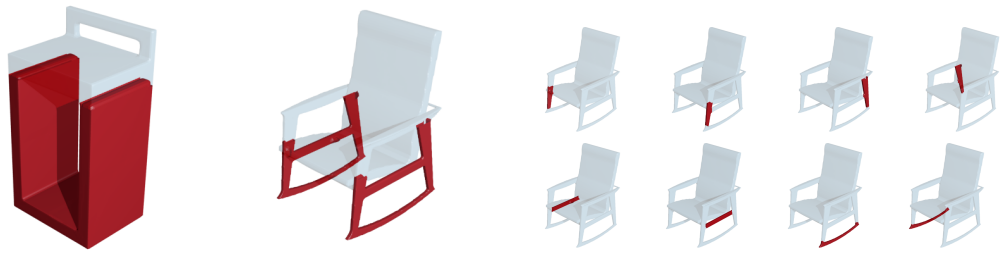



Figure 3.4: 16% of ShapeNetCore v1 models could not be included in Part-SRN.



(a) A, Low granularity (b) B, Low granularity (c) B, Medium and High granularity

Figure 3.5: Some PartNet models do not support all granularities for a part they contain. The part `chair_base` (highlighted red) is contained in Chair A and B alike. Chair B includes the part for every granularity (Low, Medium, High). Chair A only includes the part for Low granularity. Images retrieved from PartNet [27].

#### 3.4.2.3 Render SRN

The SRN dataset is based on ShapeNetCore v2 [41]. We rerender SRN using models from ShapeNetCore v1 to ensure alignability with PartNet (Section 3.4.2.1).

We implement code to split the ShapeNet dataset according to the SRN evaluation protocol (Section A.1). This was not included in the SRN code repository [38, 39].

#### 3.4.2.4 Render segmentation masks

We augment every SRN render with part segmentations across all three granularity levels provided by PartNet. To this end, we developed mask generation code based on the open-source library bpycv [57]. This part of the rendering pipeline abstracts away several complexities.

1. **Understanding PartNet.** PartNet splits ShapeNetCore v1 objects into a hierarchy with base parts at the leaves. Base parts are joined to form parts based on granularity level. To understand the joining procedure, one must cross-reference official PartNet part segmentations [25] with per-model hierarchies.
2. **Understanding Blender API.** PartNet base parts must be loaded into Blender and transformed to ensure alignment with ShapeNetCore v1. The parts must then be labelled according to the part hierarchy and granularity level.
3. **Parallelisation.** Given the scale of the SRN dataset, parallelising the rendering pipeline becomes desirable. Since the Blender API is exposed via the command line, built-in parallelisation features cannot be fully used. In our experience, calling Blender binaries can result in pitfalls such as deadlocks caused by child processes halting due to a lack of buffer flushing by the parent process.

### 3.4.3 Deviations from SRN

Part-SRN is $\sim 13\%$ smaller than SRN (Table 3.3). The SRN dataset contains both Chair and Car synset [41]. Part-SRN only contains the Chair synset, since PartNet does not include part segmentations for cars [27].

| | SRN | Part-SRN |
|---|---|---|
| Train | 4,612 | 4,033 |
| Validation | 662 | 576 |
| Test | 1,317 | 1,152 |
| Total | 6,591 | 5,761 |

Table 3.3: Number of instances in SRN and Part-SRN. Part-SRN contains $\sim 13\%$ fewer data points than SRN. The relative train/val/test sizes are the same.

### 3.4.4 Limitations

Part-SRN only contains models that exist in both ShapeNetCore and PartNet. Secondly, some models in SRN are improperly scaled [41]. Our pipeline çannot detect improper scaling. Thirdly, Part-SRN uses ShapeNetCore v1 to ensure compatibility with PartNet. However, some v1 models lack texture and have lower-quality geometry than ShapeNetCore v2 [37].

### 3.4.5 Summary

The Part-SRN rendering pipeline abstracts away the intricacies of the ShapeNetCore and PartNet datasets, including the Blender Python API. The pipeline can be applied to any synset in PartNet. The pipeline readily extends to datasets beyond the ShapeNet family. The pipeline development took over 400 developer hours, and we believe it will present researchers with non-negligible time savings.

## 3.5 Dataset used for experiments

We use the Chair synset from Part-SRN for experiments (Section 3.4.4). CodeNeRF performs worse on Chair than on the Car synset [16], so we expect this to pose a greater challenge.

pixelNeRF [60] and CodeNeRF [16] were trained on the full SRN training set 3.3, which comprises 4,612 models. We estimate that even in a multi-accelerator setting, it would take 144 days to train CodeNeRF on this data (Section B.3.1). To reduce the expected computational cost to $\sim 3$ days, we curate a training set comprising 32 models (Section A.2). The validation and test sets are downsampled accordingly. This corresponds to $\sim 0.7\%$ of the original training set.

# Chapter 4

# Experimental setup

## 4.1 Metrics

We use the PSNR, SSIM and LPIPS metrics, common to NeRF-based literature [16, 23, 60] for evaluation. We use the TorchMetrics implementation [49, 50, 51].

### 4.1.1 PSNR

Adapting the notation to images, peak signal-to-noise-ratio (PSNR) is defined as:

$$\mathrm{PSNR}(I,\ \hat{I}\ ) = 10 \cdot \log_{10}\left(\frac{\max(I)}{\mathrm{MSE}\left(I,\ \hat{I}\ \right)}\right)$$

where $\mathrm{MSE}(I,\ \hat{I}\ ) = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(I_{i,j} - \hat{I}_{i,j}\right)$ [14]. Intuitively, PSNR is high if there is little between the reference image $I$ and predicted image $\hat{I}$, and vice versa. However, PSNR does not match perceived visual quality well [55]. We include PSNR to remain comparable but also report the more advanced SSIM and LPIPS.

### 4.1.2 SSIM

Structural similarity index (SSIM) postulates that the human visual system is adapted to extracting *structural* information [55]. It obtains a similarity metric by comparing luminance, contrast and structure between the reference and predicted image. For conciseness, we do not state the formulae, which can be found in [55].

### 4.1.3 LPIPS

Despite their popularity, PSNR and SSIM fail to account for nuances of human perception [61]. The Learned Perceptual Image Patch Similarity (LPIPS) metric exploits features of deep neural models and achieves state-of-the-art agreement with human judgement. We use LPIPS with the VGG backbone to remain comparable with pixelNeRF [59].

## 4.2 Implementation

We identified several shortcomings in the official implementation of CodeNeRF [15], open-sourced by the authors.

- **Missing activation.** The NeRF architecture produces RGB values between 0 and 1. This is achieved by applying the sigmoid activation function to the RGB head [23]. The RGB head in the CodeNeRF architecture works identically [16]. However, the implementation does not include the nonlinearity, producing off-range RGB values. We evaluate the impact empirically in Figure 4.1. The predictions in the original publication resemble Figure 4.1c more closely than Figure 4.1b [16], suggesting that the open-sourced implementation deviates from the original code. This has already impacted follow-up work, e.g. [54].
- **Code quality.** The repository violates the KISS principle by re-implementing framework-specific solutions (e.g. the repository uses the `getattr`, `setattr` API to register variable numbers of modules instead of PyTorch's `ModuleList`). In addition, the repository does not follow best language-specific practices (e.g. due to the use of index-based iteration and laconic variable names).
- **Lack of maintenance.** The authors participated in discussions as late as July 2022 [15]. However, several issues remain unresolved, including the following:
- **Missing parts of inference pipeline.** One of the main contributions of CodeNeRF, test-time latent code and camera pose optimisation, is not implemented.
- **Missing validation and test-set.** The repository does not include code to obtain validation- and test-set results.
- **Code efficiency.** The repository creates a duplicate data loader every epoch, resulting in unnecessary overhead. Alarmingly, the implementation discards 98% of gradients due to a misplaced call to the optimisation procedure. Effectively, the source code only uses the last viewpoint for training.

(a) Reference (b) Prediction (no activation) (c) Prediction (activation)

Figure 4.1: The open-sourced CodeNeRF implementation [15] omits the sigmoid activation function at the RGB head, producing off-range RGB outputs. The logging facility maps these to a valid range, skewing pixel colours towards grey (4.1b).

### 4.2.1 Overwrite

The CodeNeRF repository has shortcomings that make it an unreliable basis for further research. To address these issues, we completely rewrote the official implementa-

tion [15]. We approximate the CodeNeRF training procedure as closely as possible. Relevant values are available in Section B.3.3. Our rewritten version improves upon the original in several ways:

- **Readability.** We follow idiomatic principles, use high-level convenience functions (e.g. `torch.split` instead of direct indexing) and appropriate data structures (e.g. `ModuleList` for variable numbers of modules, `ParameterDict` to collect parameters registered at training time). Finally, we use explicit typing to eliminate ambiguity.
- **Extensiblility.** We let the NeRF model depend on a `LatentCodeProvider` interface. Thus, novel approaches to generating latent codes can be implemented in a plug-and-play fashion. Secondly, we eliminate the original repository's coupling to the TensorBoard logger thanks to PyTorch Lightning's logging interface.
- **Scalability.** We base our implementation on TorchMetrics and the PyTorch Lightning machine learning framework. This makes the model device-agnostic and ready to scale to multiple types of accelerators.

The implementation process took $\sim$ 500 junior developer hours. It will be open-sourced in its entirety upon submission. For a fair comparison, we use our *fixed* implementation of CodeNeRF throughout our experiments.

# Chapter 5

# Idea 1: Latent code augmentation

CodeNeRF can generalise across scenes thanks to its use of latent codes. The codes are initialised using a normal distribution: $\mathbf{z}_s,\ \mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, 2/d\ \mathbf{I}_d)$, where $d$ is the code dimensionality, and optimised during backpropagation. Keeping a pair of latent codes per instance allows for interpolations within the embedding space, yielding unseen instances (Figure 2.8), while keeping the architecture simple (Figure 2.9).

While pixelNeRF outperforms CodeNeRF across all reported metrics (Table 5.1), its architecture is more complex. It obtains scene-specific information from a pre-trained encoder before combining them with NeRF [60]. Inspired by pixelNeRF's performance, we contemplate a CodeNeRF-based architecture wherein latent codes are conditioned on a pre-trained encoder. We refer to this architecture as EnCodeNeRF.

## 5.1 Methodology

### 5.1.1 High-level considerations

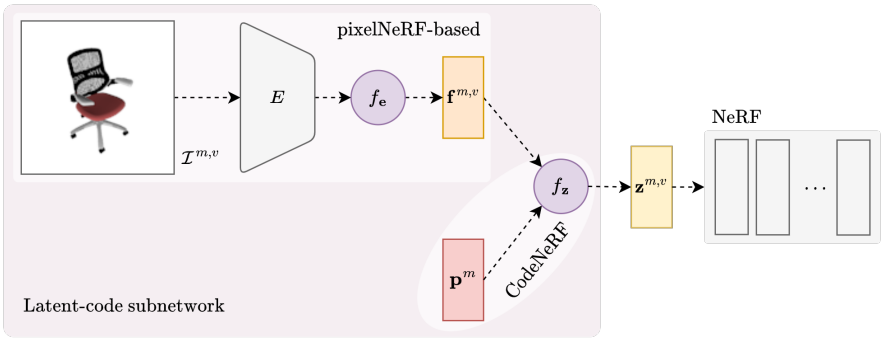



Figure 5.1: We consider a procedure $f_{\mathbf{z}}$ that combines the learnable parameter $\mathbf{p}^m$ with a feature $\mathbf{f}^{m,v}$ to produce the latent code $\mathbf{z}^{m,v}$. The feature $\mathbf{f}^{m,v}$ is obtained by applying a transformation $f_{\mathbf{e}}$ to the output of a pre-trained encoder $E$.

First, to keep CodeNeRF's ability to interpolate in shape and texture space, we restrict modifications to the latent-code part of the network, which outputs a single vector $\mathbf{z}$

(Figure 5.1). Second, to reduce search space, we consider the same modification for the shape and texture latent-code subnetwork alike. Hence, we refer to either latent code using $\mathbf{z}$. Finally, to maintain CodeNeRF's architectural simplicity, we minimise modifications. We now outline the selection of $E$ and the incremental design of $f_{\mathbf{e}}$ and $f_{\mathbf{z}}$, respectively.

### 5.1.2 Choosing $E$

We select EfficientNet B0 from the EfficentNet family (Section 2.6), as it is the most compact, and it outperformed ResNet50 while using 4.9x fewer parameters and 11x fewer FLOPs [44]. pixelNeRF uses the less-performant ResNet34 [60] to extract scene specifics, which is 4x larger than EfficientNet B0 [7].

### 5.1.3 Designing $f_{\mathbf{e}}$

We now discuss the choices we made in designing $f_{\mathbf{e}}$, the part of the latent code subnetwork that transforms EfficientNet outputs to a feature vector to be combined with the learnable parameter using $f_{\mathbf{z}}$ and input into NeRF (Figure 5.1).

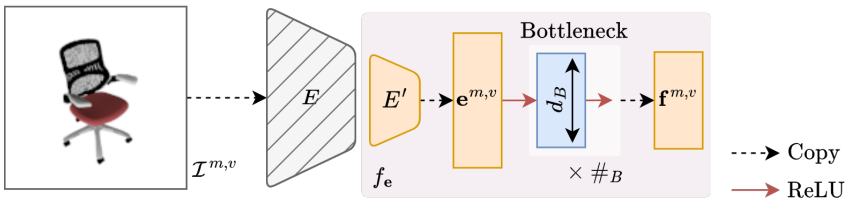



Figure 5.2: $f_{\mathbf{e}}$ bridges the EfficientNet-based encoder $E$ and CodeNeRF. Given a viewpoint $I^{m,v}$, it extracts a feature $\mathbf{e}^{\mathbf{m},\mathbf{v}}$ from $E$, and passes it through through $\#_B$ bottleneck layers of dimensionality $d_B$. The output feature $\mathbf{f}^{m,v}$ is then combined with the parameter $\mathbf{p}^m$ (Figure 5.1).

#### 5.1.3.1 Encoder

EfficientNet B0 consists of a series of convolutional layers, yielding a 1280-dimensional feature vector $\mathbf{e}$. B0 projects $\mathbf{e}$ to ImageNet categories using a classification head consisting of a fully-connected layer and a softmax [44]. We do not perform classification, so we discard the classification head and operate on $\mathbf{e}$.

We do not fine-tune $E$'s convolutional layers to save compute. However, we remark that EfficientNet was trained on ImageNet classification, wherein our inputs would correspond to a $\sim$4 categories (Section B.1.2). Therefore, we suspect that EfficientNet might produce less-informative features. In the case of underperformance, we will consider fine-tuning the final convolutional layer. We choose only the final convolutional layer to minimise the increase in parameter count.

We remark that EfficientNet uses an initial learning rate that is two orders of magnitude higher than that used for CodeNeRF parameters, with a more rapid learning rate decay. Thus, the weight spaces of EfficientNet and CodeNeRF are conditioned differently

(Section B.1.1). We will consider the learning procedure used by EfficientNet authors for fine-tuning.

#### 5.1.3.2 New components

We consider fully-connected projections, which are well-suited to transformations in feature spaces. $\mathbf{e}$ is high-dimensional relative to the 256-dimensional features used in NeRF. To avoid rapid parameter increase, we project $\mathbf{e}$ onto a bottleneck dimension $d_b$. The recent success of neural methods is partially owed to network depth [48], so we also vary the number of bottleneck layers $\#_B$. To learn nonlinear projections, we use the popular ReLU non-linearity for $\#_B > 1$. Finally, similar to CodeNeRF, we consider distinct learning rates for NeRF and latent codes [16].

#### 5.1.3.3 Feature aggregation

In CodeNeRF, a model $m$ has a single shape and texture latent code used for *all* viewpoints. The output feature $\mathbf{f}^{m,v}$, depends on the *current* viewpoint $v$. We suspect this might confuse the NeRF part of the network. To prevent variation in a model's latent codes, we consider an aggregation procedure wherein the outputs of $f_\mathbf{e}$ are averaged:

$$\mathbf{f}^{m,\cdot} = \frac{1}{V}\sum_{v=1}^{V} f_\mathbf{e}(I^{m,v})$$

pixelNeRF does not aggregate encoder features [60], so we also consider an aggregation-free feature extraction procedure $\mathbf{f}^{m,v} = f_\mathbf{e}(I^{m,v})$. We summarise the hyperparameters considered in Table B.2.

### 5.1.4 Designing $f_\mathbf{z}$

$f_\mathbf{z}$ is the part of the network that combines the transformed encoder output $\mathbf{f}^{m,v}$ and the learnable parameter $\mathbf{p}^m$ (Figure 5.1). This is a generalisation of the CodeNeRF architecture if we define $\mathbf{z}^{m,v} = f_\mathbf{z}(\cdot,\ \mathbf{p}^m) = \mathbf{p}^m$. We now outline the candidate designs for $f_\mathbf{z}$. To minimise architectural parameter increase, we start with the simplest modifications and only gradually increase the complexity of interactions between $\mathbf{f}^{m,v}$ and $\mathbf{p}^m$.

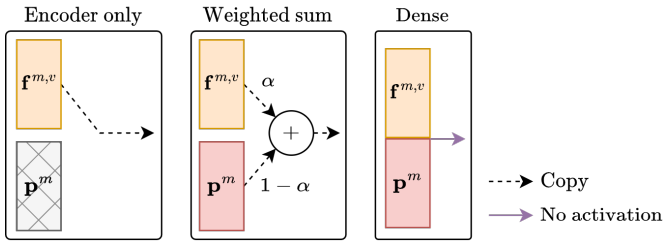



Figure 5.3: In keeping with pixelNeRF and CodeNeRF, we use an encoder-extracted feature $\mathbf{f}^{m,v}$ and a learnable parameter $\mathbf{p}^m$ for a model $m$ and viewpoint $v$. We combine them to produce the latent code $\mathbf{z}^{m,v}$. We consider three combining methods of increasing complexity.

#### 5.1.4.1 Encoder-only

Unlike CodeNeRF, pixelNeRF does not use model-specific learnable parameters [16, 60]. Thus, we disregard $\mathbf{p}^m$ and consider the encoder output, i.e. $\mathbf{z}^{m,v} = f_{\mathbf{z}}(\mathbf{f}^{m,v}, \cdot) = \mathbf{f}^{m,v}$ (Figure 5.3 left).

#### 5.1.4.2 Weighted sum

We suspect that the architectural differences between CodeNeRF (Figure 2.9) and pixelNeRF (Section 2.3) might necessitate the incorporation of learnable parameters. We consider two approaches to allow some interaction between $\mathbf{f}^{m,v}$ and $\mathbf{p}^m$.

We first consider the weighted sum, wherein $\mathbf{f}^{m,v}$ is multiplied by $\alpha$ and $\mathbf{p}^m$ by $1-\alpha$ (Figure 5.3 centre). By the nature of $E$, we expect $\mathbf{f}^{m,v}$ to encapsulate crude model features. Since $\mathbf{p}^m$ is initialised randomly, we expect it to learn any remaining detail.

#### 5.1.4.3 Dense

We finally consider projecting the concatenation of $\mathbf{f}^{m,v}$ and $\mathbf{p}^m$ using a fully-connected layer (Figure 5.3 right). This is a generalisation of the weighted sum. It makes the weights learnable, and it allows any component in $\mathbf{f}^{m,v}$ to interact with any component in $\mathbf{f}^{m,v}$ and $\mathbf{p}^m$, while allowing either vector to have different dimensionality. This significantly relaxes the constraints enforced by weighted sum, albeit at the cost of more parameters. We outline the considered hyperparameters in Section B.4.2.

## 5.2 Results

A single model takes $\sim$ 3 days to train. To reduce the computational cost, we run configurations for $\sim$ 20% of the maximum 300 epochs and discard runs with poor validation-set performance immediately afterwards. In total, we try over 100 configurations. Since our environment's automated hyperparameter search does not work on the University's clusters (Section C.1), we evaluate hyperparameter configurations manually. We now summarise our findings and present the best-performing model.

### 5.2.1 Observations and Best-performing model

**Combiner $f_{\mathbf{z}}$.** We find that *Encoder only* leads to poor generalisation, achieving no convergence on the validation set. We suspect this is because the encoder output fails to capture instance specifics. *Weighted sum* achieves performance on par with CodeNeRF using $\alpha = 30\%$. We have found that reducing the dimensionality of the learnable parameter on CodeNeRF inhibits performance. This implies that the model must be using the full extent of $\mathbf{z}^m$, and the encoder-extracted feature $\mathbf{f}^{m,v}$ provides useful information. Finally, *Dense* outperforms CodeNeRF on validation-set loss, albeit by a slim margin of 1%. We now outline the remaining details of the corresponding model.

**Encoder feature extractor $f_{\mathbf{e}}$.** The best performance is offered by a single, 128-dimensional bottleneck layer. Wider bottlenecks do not improve performance, whereas

narrower reduce it. We find that using no viewpoint feature aggregation performs better than averaging. Fine-tuning the final convolutional layer of the encoder $E$ is the change that eventually outperformed CodeNeRF on the validation set. We use the original training procedure (Section B.1.1) for the final convolutional layer and a learning rate of $10^{-4}$ for the rest of the network. As is common in fine-tuning, we tried reducing the original learning rate. Specifically, we tried halving the learning rate for the final convolutional layer, but this harmed performance.

### 5.2.2 Outputs

Table 5.1 captures automated metrics. We report the performance on four models curated to capture the structural and textural diversity of ShapeNetCore in Figure 5.4. We adjusted the training procedure of pixelNeRF to ensure convergence and reasonable training time (Section B.2.1). Finally, we stress that the dataset under consideration corresponds to $\sim 0.7\%$ of the original dataset (Section 3.5) used in pixelneRF [60] and CodeNeRF [16].

| Method | GPUs | $\lvert V \rvert$ | PSNR↑ | SSIM↑ | LPIPS ↓ |
|---|---|---|---|---|---|
| pixelNeRF [60] | 1 | 1 | 20.753 | 0.815 | 0.271 |
| pixelNeRF | 1 | 2 | 21.702 | 0.847 | 0.202 |
| CodeNeRF [16] | 1 | 50 | **21.943** | **0.872** | **0.165** |
| EnCodeNeRF | 1 | 50 | 21.798 | 0.867 | 0.171 |
| CodeNeRF | 8 | 50 | **21.546** | **0.850** | **0.203** |
| EnCodeNeRF | 8 | 50 | 21.329 | 0.845 | 0.212 |

Table 5.1: (Bold implies best performance on a metric. $\lvert V \rvert$ denotes the number of views used to extract scene specifics.) Out of the tested models, CodeNeRF performs best across all metrics in both single- and multi-GPU settings. EnCodeNeRF comes second, outperforming pixelNeRF. Similar to CodeNeRF [16], we test pixelNeRF with $\lvert V \rvert$ set to 1 and 2; this is significantly fewer than the 50 views (i.e. one for each input viewpoint) used in CodeNeRF and EnCodeNeRF.

### 5.2.3 Discussion

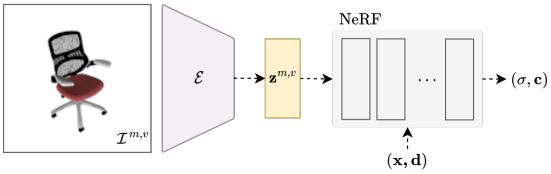
The outputs of CodeNeRF shown in Figure 5.4 are on par with those reported in the original work [16], indicating the success of our overwrite (Section 4.2.1). Our proposed method achieves worse results than CodeNeRF. However, it still outperforms pixelNeRF (Table 5.1).


Figure 5.4: ("Ref", "PNR", "CNR", "ENR" stand for Reference, pixelNeRF, CodeNeRF and EnCodeNeRF (ours), respectively.) All methods perform well, with 8-GPU methods performing considerably worse than 1-GPU. pixelNeRF outputs appear blurry in comparison to CodeNeRF (e.g. Chair 2, 3), despite outperforming it on the *full* dataset [16]. On Chair 1, pixelNeRF captures the backrest in the most detail. CodeNeRF outputs preserve more detail than EnCodeNeRF (see e.g. leg sets of Chair 3, 4), while EnCodeNeRF sometimes better captures parts in the context of complex structure (e.g. front leg of Chair 2). On 8 GPUs, CodeNeRF produces fewer background artefacts than CodeNeRF and again shows more detail.

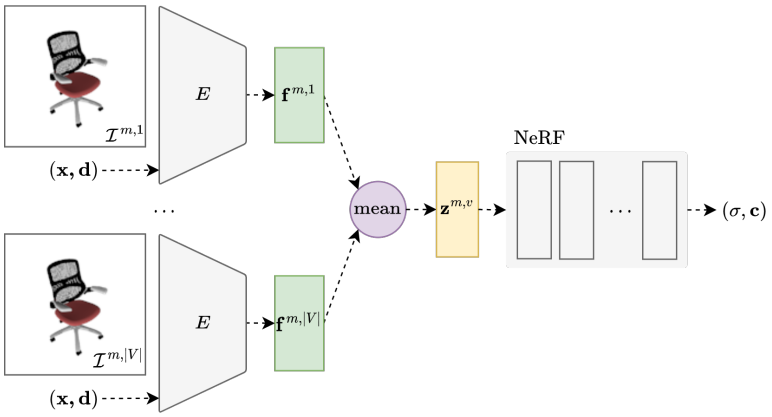
We stress that our architecture (Figure 5.2) and the accompanying implementation (Section 4.2.1) allow new feature-providing modules to be incorporated into the network in a plug-and-play fashion. Our implementation of the combiner $f_{\mathbf{z}}$ accommodates a dynamic number of latent code providers, allowing interactions beyond just a pair of features ($\mathbf{f}^{m,v}$ and $\mathbf{p}^{m}$).

We now discuss the reasons behind the underperformance of our model. The main non-trivial modifications in our architecture over CodeNeRF are the incorporation of the encoder $E$ and the feature-generating module $f_{\mathbf{e}}$ (Figure 5.1). We inspect each of these in turn.

The chosen EfficientNet is trained on classification (Section 2.6), which may produce less informative features (Section 5.1.3.1). However, pixelNeRF also uses an encoder pre-trained on classification, achieving state-of-the-art results [13, 60]. Therefore, we rule out encoder choice and instead inspect feature generation.




(a) EnCodeNeRF. (ε encapsulates $f_{\mathbf{e}}$ and $f_{\mathbf{z}}$.)




(b) pixelNeRF [60].

Figure 5.5: At a high level, our architecture and pixelNeRF are similar. Both use an encoder to extract a feature $\mathbf{z}^{m,v}$ that is combined with the NeRF part of the network. pixelNeRF averages features across all viewpoints $V$, whereas EnCodeNeRF only considers the *current* viewpoint $v$. EnCodeNeRF inputs viewing direction and position in space $(\mathbf{x}, \mathbf{d})$ into NeRF, while pixelNeRF inputs them into the encoder. (Incidentally, pixelNeRF's coupling of positional variables to appearance features prevents it from performing interpolations in feature space like CodeNeRF (Figure 2.8)). (5.5b adapted from [60])

In Section 5.2.1, we ascertained that the encoder feature $\mathbf{f}^{m,v}$ contains *relevant* information. However, since EfficientNet was pre-trained on image classification, where most instances would belong to the same category (Section 5.1.3.1), this information may not be *useful*. To this end, we fine-tune the *final* convolutional block out of 9 blocks. preceding features may still correlate with the original classification domain. Indeed, pixelNeRF uses only the first 4/5 encoder blocks, all of which are fine-tuned [59, 60]. In a similar vein, pixelNeRF combines all intermediate features [60], whereas EnCodeNeRF only considers the final output feature. At a high level, our proposed architecture and pixelNeRF differ predominantly in the feature-generating procedure (Figure 5.5). Therefore, we suspect inappropriate feature generation to be the cause of our model's underperformance.

In summary, our proposed method did not outperform CodeNeRF. Due to time constraints, we do not address the feature-extraction issues discussed earlier.

# Chapter 6

# Idea 2: Part-level latent code augmentation

EnCodeNeRF and CodeNeRF perform similarly (Table 5.1). Upon closer inspection, both architectures struggle to model rare parts (Figure 6.1). We contemplate a method to improve the performance of CodeNeRF and EnCodeNeRF when modelling such parts since better performance on constituent parts should improve prediction quality. Thanks to Part-SRN (Section 3.4) and the extensibility of our CodeNeRF implementation (Section 4.2.1), we can segment each viewpoint into constituent parts (Figure 6.2). We now inspect how part-level segmentations could be used to improve predictions.

(a) Ref (b) CNR (c) ENR (a) Ref (b) CNR (c) ENR

(a) Common parts (b) Rare parts

Figure 6.1: ("Ref", "CNR", "ENR" stand for Reference, CodeNeRF and EnCodeNeRF, respectively.) CodeNeRF and EnCodeNeRF achieve reasonable performance on common instances (Figure 6.1a). However, both produce blurry outputs on rare parts, such as sofa pillows (Figure 6.1b).

| Characteristic | Method | PSNR↑ | SSIM↑ | LPIPS ↓ |
|---|---|---|---|---|
| Common | CodeNeRF [16] | 20.812 | 0.877 | 0.137 |
| | EnCodeNeRF 5 | **21.423** | **0.891** | **0.123** |
| Rare | CodeNeRF | **17.629** | **0.764** | 0.239 |
| | EnCodeNeRF | 16.765 | 0.750 | **0.233** |

Table 6.1: The performance of CodeNeRF and EnCodeNeRF reduces significantly on the chair instance that contains rare parts (Figure 6.1b). This motivates us to look for ways to improve the models' part-level predictions.

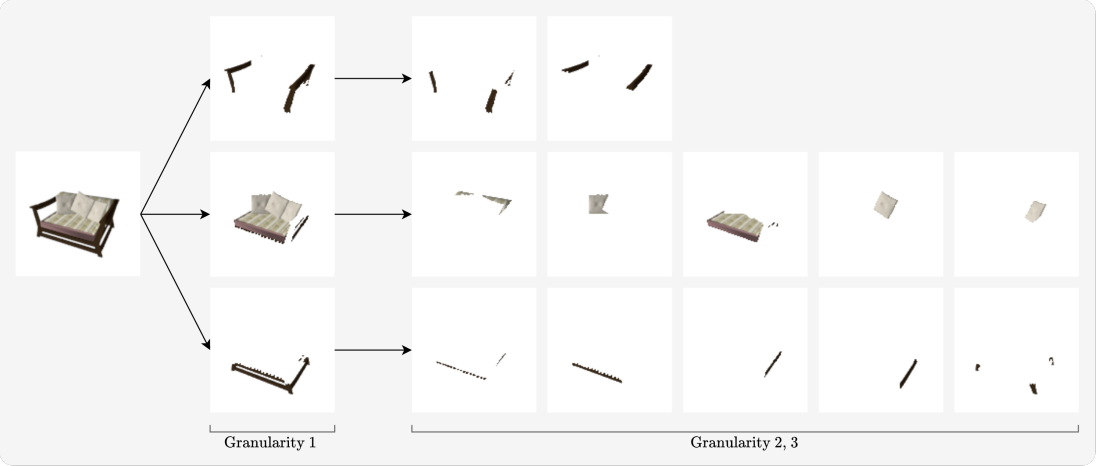



Figure 6.2: Part-SRN segments each viewpoint into constituent parts. For the Chair synset, segmentations come in three granularities: low, medium, and high. We will exploit this part-level information to obtain better latent codes.

## 6.1 Methodology

Our goal is to improve part-level predictions for CodeNeRF or EnCodeNeRF. We first consider incorporating part-level supervision at the loss level. This approach is the least invasive, as it does not require modifying the underlying model. Specifically, we consider masked MSE loss. We mask out all but one part and apply MSE loss to it; we then average the per-part MSE losses. Initial experiments proved unfruitful, resulting in poor convergence and significant artefacts (i.e. smudges on the white background). Therefore, we consider incorporating part-level supervision into the model.

Our method proposed in Section 5 performs subjectively better than CodeNeRF on rare parts (Figure 6.1). This is corroborated by the LPIPS metric (Table 6.1), which should reflect human perception. We suspect that this is due to our use of an encoder. The encoder is pre-trained on a significantly larger dataset of images, so it may be able to extract useful features even for images not common to our dataset (e.g. the pillows in Figure 6.1b). We thus look for a way to extend our method to part-level segmentations. Due to time constraints, we use the best-performing model from (Chapter 5) without further grid search over $f_{\mathbf{e}}$ and $f_{\mathbf{z}}$. Figure 6.3 summarises the high-level idea. We expand our notation and then consider two methods inspired by our previous work (Figure 6.4).

### 6.1.1 Notation

We now leverage the part-level information in our dataset, Part-SRN. To this end, we extend the notation introduced in Section 2.4.1. We consider a dataset $\mathcal{D}_{\mathcal{P}}$ of $M$ models, where each model consists of camera intrinsics $K$ and $V$ viewpoints containing image $I$ and camera pose $T$. Let us denote the set of parts for a given granularity level by $\mathcal{P}_G$.

$$\mathcal{D}_{\mathcal{P}} = \left\{ \left( \left(I^{m,v,p}\right)_{p\in\mathcal{P}},\, T^{m,v},\, K^{m} \right) \;\middle|\; m \in 1,\ldots,M;\; v \in 1,\ldots,V \right\}$$

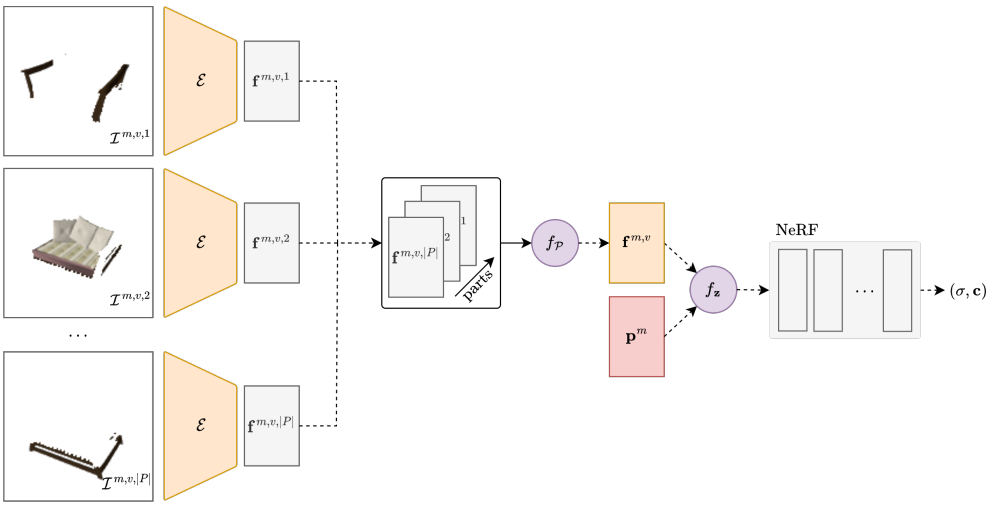



Figure 6.3: We contemplate a procedure $f_P$ that combines per-part features $\mathbf{f}^{m,v,\cdot}$ into a single feature vector $\mathbf{f}^{m,v}$. (ε encapsulates encoder $E$ and $f_\mathbf{e}$ discussed in Chapter 5.)

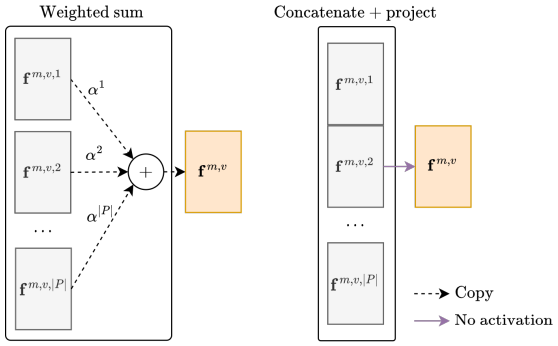



Figure 6.4: Inspired by our work on combining feature vectors (Section 5.1.4.2, 5.1.4.3), we consider two implementations of $f_\mathcal{P}$: Learnable weighted sum (left), which increases parameter count minimally but allows for simple interactions, and Concatenate and project (right), which allows for more complex interactions.

### 6.1.2 Weighted sum

Since a feature $\mathbf{f}^{m,v,p}$ must be extracted for each part, our feature-extraction method now has $O(|\mathcal{P}|)$ time complexity. To minimise parameter increase, we first consider a simple combination procedure $f_\mathcal{P}$ that assigns to each part-level feature $\mathbf{f}^{m,v,p}$ a learnable weight $\alpha^p$. The prediction is then simply $\mathbf{f}^{m,v} = \sum_{p\in\mathcal{P}} \alpha^p \cdot \mathbf{f}^{m,v,p}$.

### 6.1.3 Concatenate and project

EnCodeNeRF benefitted from allowing complex interactions between the encoder-extracted feature $\mathbf{f}^{m,v}$ and the learnable parameter $\mathbf{p}^m$ (Section 5.1.4.3). We consider an equivalent approach in the context of part-level features. For simplicity, we consider a single layer and we do not experiment with nonlinearities.

### 6.1.4 Hyperparameter search

Initial experiments on weighted sum showed poor generalisation. Therefore, we only consider Concatenate and project. Owing to its use of part-level segmentations, we call this new architecture SegCodeNeRF.

Due to time constraints, we do not test many hyperparameters. Since $f_{\mathbf{e}}$ performed best with the lower learning of $10^{-4}$ (Section 5.2.1), we apply the same to $f_{\mathcal{P}}$. For simplicity, $f_{\mathcal{P}}$ again uses the CodeNeRF learning rate scheduler. We only vary the dimensionality of the part-level features $\mathbf{f}^{m,v,\cdot}$ and the aggregated feature $\mathbf{f}^{m,v}$, with an aim to mitigate hyperparameter increase. We find that a 64-dimensional part-level feature vector $\mathbf{f}^{m,v,\cdot}$ and a 128-dimensional $\mathbf{f}^{m,v}$ instance-level feature vector perform best.

We remark that the relatively high feature dimensionality incurs a considerable computational cost at higher granularity levels. Therefore, we consider a shortened in addition to the full training procedure. The shortened procedure trained SegCodeNeRFs for 100 epochs (equivalent to $\sim$80 hours for Granularity 2 and 3). We now list the results for either, and then discuss them.

## 6.2 Results

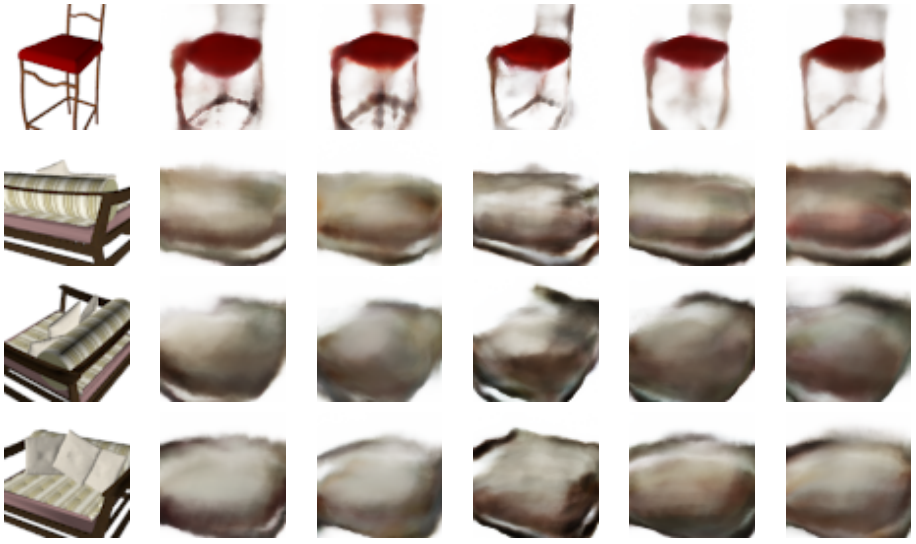



Figure 6.5: "Ref", "CNR", "ENR", and "SNR" stand for Reference, CodeNeRF, EnCodeNeRF, and SegCodeNeRF, respectively. The subscript indicates the Granularity level. The models were trained for 100 epochs to save compute. All methods perform reasonably well on the chair model with common parts (first row). When faced with rare parts, they produce blurry images. Our SegCodeNeRF$_1$ captures the underlying shape, producing the most realistic output. (The images are cropped due to CodeNeRF's cropping procedure).

### 6.2.1 Shortened training

We summarise the results on automated metrics in Table 6.2. The corresponding predictions are shown in Figure 6.5.

| Model | Granularity | PSNR↑ | SSIM↑ | LPIPS ↓ |
|---|---|---|---|---|
| CodeNeRF [16] | Ø | 15.974 | 0.537 | 0.397 |
| EnCodeNeRF | Ø | 15.821 | 0.549 | 0.387 |
| SegCodeNeRF (ours) | 1 | **17.506** | **0.633** | **0.328** |
| | 2 | 15.787 | 0.552 | 0.386 |
| | 3 | 16.097 | 0.546 | 0.391 |

Table 6.2: Due to the high computational cost of Granularity 2 and 3, we run every model for 100 epochs, i.e. 1/3 of the previously-used epoch count. The incorporation of coarse part-level information (Granularity 1) improves performance with respect to both CodeNeRF and EnCodeNeRF. Increasing Granularity results in performance deterioration, likely because the segmented parts become too small to be correctly identified by the encoder (Figure 6.2), resulting in uninformative or wrong part-level features.

### 6.2.2 Full training

We ran the least-compute-intensive run, SegCodeNeRF$_1$, for up to 300 epochs. Despite limited model search, it achieved on-par performance with CodeNeRF (Table 6.3). The corresponding outputs are discussed in Figure 6.6.

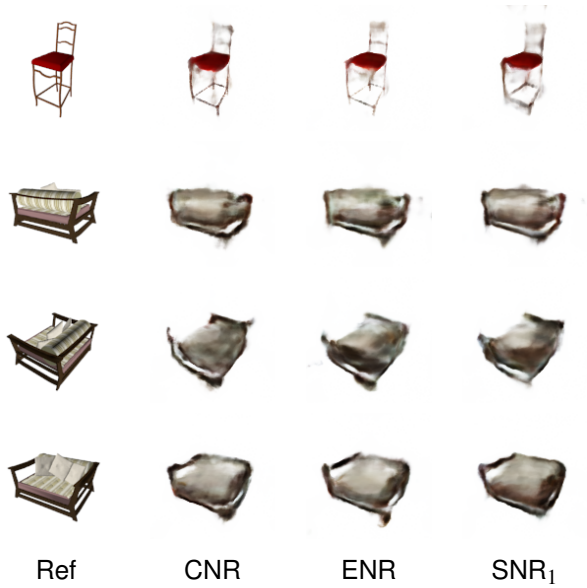



Figure 6.6: "CNR" (CodeNeRF), "ENR" (EnCodeNeRF), and SNR$_1$ (SegCodeNeRF$_1$), trained for up to 300 epochs. While SegCodeNeRF$_1$ significantly outperformed CodeNeRF and EnCodeNeRF over the first 100 epochs, its performance gain shrunk after 300 epochs, and it achieved results that are on par with CodeNeRF (Table 6.3).

| Model | PSNR↑ | SSIM↑ | LPIPS ↓ |
|---|---|---|---|
| CodeNeRF [16] | 21.943 | **0.872** | **0.165** |
| EnCodeNeRF | 21.798 | 0.867 | 0.171 |
| SegCodeNeRF$_1$ (ours) | **22.040** | **0.872** | 0.166 |

Table 6.3: We continued the least compute-intensive SegCodeNeRF model for up to 300 epochs to remain comparable with Chapter 5. SegCodeNeRF$_1$ outperforms EnCodeNeRF, highlighting the potential of using part-level information. Yet, its performance is only on par with CodeNeRF, as discussed in Section 6.2.3

### 6.2.3 Discussion

We now evaluate the results, starting with the shortened training procedure. The performance gain of SegCodeNeRF when using coarse granularity (Table 6.2) shows that part-level information is useful for model predictions. It also indicates that our incorporation of an encoder is reasonable. We notice that higher granularities not only result in slower training due to using more parameters but also reduce performance across all metrics. We hypothesise that this is linked to the fact that part size diminishes with granularity (Figure 6.2). Since the encoder was trained on full images, it struggles to identify the objects and produces unreliable features. We pursue this in Table 6.4.

| Granularity | Encoder confidence | | | |
|---|---|---|---|---|
| | Mean | Median | Min | Max |
| 1 | **0.199** | **0.191** | **0.089** | **0.411** |
| 2 | 0.160 | 0.149 | 0.064 | 0.369 |
| 3 | 0.157 | 0.144 | 0.072 | 0.369 |

Table 6.4: We view encoder prediction confidence as a proxy for feature quality: if the encoder is confident about its prediction, the intermediate features must capture sufficient information. We compute the mean confidence for all 32 models and compute aggregate metrics at all granularity levels. The encoder is most confident on the coarsest granularity. The confidence on Granularity 2 and 3 is similar, likely because these contain similar part segmentations: for the Chair synset, we have $|\mathcal{P}_1| = 5 < |\mathcal{P}_2| = 29,\ |\mathcal{P}_3| = 38$ [25].

On the full training procedure, SegCodeNeRF loses its performance gain (Table 6.1). First, this may be linked to a lack of hyperparameter search. When experimenting with EnCodeNeRF, we noticed its sensitivity to hyperparameter values. Admittedly, we did not search for hyperparameters, and we suspect a better configuration exists. Secondly, since the encoder has *not* been modified following our discussion in Chapter 5, it still applies that using earlier features, all of which would be fine-tuned, could improve the features extracted from the encoder. Finally, the images in our dataset are $128 \times 128$, while EfficientNet supports up to four times more pixels [6][1]. Using higher-resolution pictures could improve the quality of features, in particular on small parts.

[1]To maintain comparability to previous sections, we do not re-render the dataset in a higher resolution. The process takes ∼4 days on all 4,000 models. Although we fixed the random seed, downsampling to 32 models would alter the order of renders, producing different viewpoints.

# Chapter 7

# Conclusions

In Chapter 3, we introduced Part-SRN, the first publicly-available dataset that combines ShapeNetCore [4] and PartNet [27], to augment the popular SRN dataset [41]. We reimplemented CodeNeRF in an extensible manner, addressing bugs in the unmaintained open-sourced repository [15] (Section 4.2.1). In an attempt to improve CodeNeRF's performance, we leveraged our implementation's extensibility to develop two new architectural modifications, EnCodeNeRF and SegCodeNeRF, (Chapters 5, 6). While EnCodeNeRF does not outperform CodeNeRF (Section 5.2.2), it serves as a springboard for SegCodeNeRF. SegCodeNeRF outperformed CodeNeRF and pixelNeRF at early training stages (Section 6.2.1).

## 7.1 Limitations

First, our dataset was restricted to $\sim$0.7% of the data used to train CodeNeRF and pixelNeRF (Section 3.5). During initial experiments, we found that the models react to dataset size: 10 models caused overfitting, and 100 improved performance on 8 GPUs. More modest hyperparameter sweeps over a larger dataset may have offered better performance. Second, we consider complex modifications without allowing for equivalent complexity in CodeNeRF by e.g. increasing the number of shape layers.

The time investment into Part-SRN and the CodeNeRF overwrite was significant[1]. While each constitutes an open-source contribution, this left less time for iterating on architecture designs. A more performant architecture may be a few modifications away.

## 7.2 Future work

To obtain more comparable results (and likely improve either model's performance), we first suggest using a larger dataset. Using 100 models could be a reasonable starting point (Section 7.1).

[1]We summarise advice related to the development environment in Chapter C.

### 7.2.1 EnCodeNeRF

To improve EnCodeNeRF, we suggest removing more of the final encoder blocks and fine-tuning all remaining ones, similar to pixelNeRF [60] (Section 5.2.3). Inspired by later discussion, we also suggest increasing input image resolution (Section 6.2.3).

### 7.2.2 SegCodeNeRF

To improve SegCodeNeRF, we first suggest filtering out parts that do not meet a size threshold in a pre-processing step over the entire dataset, since small parts are incorrectly classified by the encoder producing irrelevant features (Section 6.2.3).

Second, since SegCodeNeRF incurs a considerable computational cost, we suggest a way to reduce the parameter count. A plausible modification is to replace the fully-connected linear projection with a few convolutional layers. To avoid concatenating all $|\mathcal{P}_G|$ features, one could only concatenate the features in the input image and instead convey part-feature correspondences using e.g. a positional encoding of the part number. Indeed, [52] famously used positional encoding of positions within an input sequence to convey ordering information.

Third, to relax supervision requirements, one could pre-train a model that performs part segmentation, so that part segmentation inputs are not necessary at inference time. Finally, we suggest hyperparameter search. In our experience, the models are sensitive to changes in the learning rate and learning rate scheduler. As discussed earlier, the need for hyperparameter tuning may be mitigated by working with a larger dataset.

# Bibliography


[1] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields, 2021. arXiv:2103.13415.

[2] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[3] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3d: Learning from rgb-d data in indoor environments. *arXiv preprint arXiv:1709.06158*, 2017.

[4] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[5] PyTorch Contributors. Distributeddataparallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html, 2022.

[6] PyTorch Contributors. Efficientnet_b0. https://pytorch.org/vision/stable/models/generated/torchvision.models.efficientnet_b0.html, 2023.

[7] PyTorch Contributors. Resnet34. https://pytorch.org/vision/stable/models/generated/torchvision.models.resnet34.html, 2023.

[8] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5828–5839, 2017.

[9] Tianyu Dai. Is pixelnerf sensitive to initialization? https://github.com/sxyu/pixel-nerf/issues/45#issuecomment-1183529207, 2022.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg

Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[11] Kyle Gao, Yina Gao, Hongjie He, Denning Lu, Linlin Xu, and Jonathan Li. Nerf: Neural radiance field in 3d vision, a comprehensive review. *arXiv preprint arXiv:2210.00379*, 2022.

[12] Mengran Gao, Ningjun Ruan, Junpeng Shi, and Wanli Zhou. Deep neural network for 3d shape classification based on mesh feature. *Sensors*, 22(18), 2022. URL: https://www.mdpi.com/1424-8220/22/18/7040, doi:10.3390/s22187040.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Alain Horé and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369, 2010. doi:10.1109/ICPR.2010.579.

[15] Wonbong Jang. code-nerf. https://github.com/wbjang/code-nerf, 2021.

[16] Wonbong Jang and Lourdes Agapito. Codenerf: Disentangled neural radiance fields for object categories. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12949–12958, 2021.

[17] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey, 2020. URL: https://arxiv.org/abs/2006.12057, doi:10.48550/ARXIV.2006.12057.

[18] Yen-Chen Lin, Alex Zhang, Andy Baker, Michal Tarnowski, and Yuliang Zou. awesome-nerf. https://github.com/awesome-NeRF/awesome-NeRF, 2023.

[19] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7708–7717, 2019.

[20] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[21] Octave Mariotti, Oisin Mac Aodha, and Hakan Bilen. Viewnerf: Unsupervised viewpoint estimation using category-level neural radiance fields. In *33rd British Machine Vision Conference 2022, BMVC 2022, London, UK, November 21-24, 2022*. BMVA Press, 2022. URL: https://bmvc2022.mpi-inf.mpg.de/0740.pdf.

[22] Stephen Robert Marschner. *Inverse rendering for computer graphics*. Cornell University, 1998.

[23] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[24] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Deep dive into the volumetric rendering function.

https://sites.google.com/berkeley.edu/nerf-tutorial/home#h.aoizq6lqleed_l, October 2022.

[25] Kaichun Mo. partnet_seg_exps. https://github.com/daerduoCarey/partnet_seg_exps, 2020.

[26] Kaichun Mo. partnet_dataset. https://github.com/daerduoCarey/partnet_dataset, 2021.

[27] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[28] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (ToG)*, 41(4):1–15, 2022.

[29] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3504–3515, 2020.

[30] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

[31] PartNet. Partnet. https://partnet.cs.stanford.edu/, 2020. Accessed on 2023-01-31.

[32] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310. PMLR, 2019.

[33] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14335–14345, 2021.

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[35] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[36] seasandwpy. How to jointly optimize the pose? https://github.com/wbjang/code-nerf/issues/5, 2022.

[37] Shapenet.org. Shapenetcore. https://shapenet.org/download/shapenetcore, 2022. Accessed on 2023-01-31.

[38] Vincent Sitzmann. scene_representation_networks. https://github.com/vsitzmann/scene-representation-networks, 2020.

[39] Vincent Sitzmann. shapenet_renderer. https://github.com/vsitzmann/shapenet_renderer, 2020.

[40] Vincent Sitzmann. SRN datasets. https://drive.google.com/drive/folders/1OkYgeRcIcLOFu1ft5mRODWNQaPJ0ps90, 2021. Data retrieved on 2023-01-25.

[41] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *Advances in Neural Information Processing Systems*, 2019.

[42] Artsiom Skarakhod. 100% offline sweep. https://community.wandb.ai/t/100-offline-sweep/3482, 2022.

[43] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, et al. The replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019.

[44] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[45] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*, pages 10096–10106. PMLR, 2021.

[46] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *NeurIPS*, 2020.

[47] thanos-wandb. Stop W&B from creating tmp files. https://community.wandb.ai/t/how-to-stop-weights-biases-wandb-from-creating-random-tmp-files/3460, 2022.

[48] Amirhosein Toosi, Andrea G Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A brief history of ai: how to prevent another winter (a critical review). *PET clinics*, 16(4):449–469, 2021.

[49] TorchMetrics. LEARNED PERCEPTUAL IMAGE PATCH SIMILARITY (LPIPS). https://torchmetrics.readthedocs.io/en/stable/image/learned_perceptual_image_patch_similarity.html, 2023.

[50] TorchMetrics. PEAK SIGNAL-TO-NOISE RATIO (PSNR). https://torchmetrics.readthedocs.io/en/stable/image/peak_signal_noise_ratio.html, 2023.

[51] TorchMetrics. STRUCTURAL SIMILARITY INDEX MEASURE (SSIM). https://torchmetrics.readthedocs.io/en/stable/image/structural_similarity.html, 2023.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[53] Suhani Vora, Noha Radwan, Klaus Greff, Henning Meyer, Kyle Genova, Mehdi SM Sajjadi, Etienne Pot, Andrea Tagliasacchi, and Daniel Duckworth. Nesf: Neural semantic fields for generalizable semantic segmentation of 3d scenes. *arXiv preprint arXiv:2111.13260*, 2021.

[54] Thomas Walker. Disentangling neural implicit representations for 3d assets, 2022.

[55] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi:10.1109/TIP.2003.819861.

[56] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. Neural fields in visual computing and beyond. In *Computer Graphics Forum*, volume 41, pages 641–676. Wiley Online Library, 2022.

[57] Lei Yang. bpycv. https://github.com/DIYer22/bpycv, 2022.

[58] Li Yi, Vladimir G Kim, Duygu Ceylan, I-Chao Shen, Mengyan Yan, Hao Su, Cewu Lu, Qixing Huang, Alla Sheffer, and Leonidas Guibas. A scalable active framework for region annotation in 3d shape collections. *ACM Transactions on Graphics (ToG)*, 35(6):1–12, 2016.

[59] Alex Yu. pixel-nerf. https://github.com/sxyu/pixel-nerf, 2022.

[60] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelNeRF: Neural radiance fields from one or few images. In *CVPR*, 2021.

[61] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[62] Henry Cheng Zhao. How to optimize and estimate the camera pose. https://github.com/wbjang/code-nerf/issues/8, 2022.

# Appendix A

# Datasets

## A.1 Inferring the SRN evaluation protocol

We used the publicly-available SRN dataset [40] for inspection. We inspect both available synsets, Car and Chair.

**Application** This was inferred from the `README.txt` file in the dataset repository [40].

**Size** For each synset, we counted the number of model IDs in the `train`, `val` and `test` subset, respectively. Note that the model IDs across `train`, `train_val` and `train_test` are identical, the difference being in the number of renders and sampling method used.

**# views** For each synset, we hand-picked a model ID from the `train`, `val` and `test` subset, respectively. We counted the number of renders in the `/rgb` subdirectory.

**Sampling method** For each synset, we hand-picked a model ID from the `train`, `val` and `test` subset, respectively. We manually inspected a sequence of the first 5 renders in the `/rgb` subdirectory. If camera poses changed slightly from instance to instance and seemed to trace out a path on a sphere, we inferred the sampling method to be "Spherical spiral" (Figure 3.1b) and Random points on a sphere otherwise 3.1a.

### A.1.1 Chair synset

| Subset | Hand-picked model ID |
|---|---|
| `train`, `train_val`, `train_test` | `7035d480f6bda22938b39a90ee80e328` |
| `val` | `d28423569bfd84708336a02debb9923b` |
| `test` | `1f8e18d42ddded6a4b3c42e318f3affc` |

Table A.1: Hand-picked 3D model IDs for the Chair synset.

| Subset | Absolute size | Relative size |
|---|---|---|
| `train`, `train_val`, `train_test` | 4612 | $\frac{4612}{4612+662+1317} \approx 70\%$ |
| `val` | 662 | $\frac{662}{4612+662+1317} \approx 10\%$ |
| `test` | 1317 | $\frac{1317}{4612+662+1317} \approx 20\%$ |

Table A.2: Absolute and relative subset sizes for the Chair synset.

### A.1.2 Car synset

| Subset | Hand-picked model ID |
|---|---|
| `train`, `train_val`, `train_test` | `952160c39258af7616abce8cb03e7794` |
| `val` | `49f3932e6fe0828951cc889a6330ab15` |
| `test` | `2c6b14bcd5a5546d6a2992e9465c023b` |

Table A.3: Hand-picked 3D model IDs for the Car synset.

| Subset | Absolute size | Relative size |
|---|---|---|
| `train`, `train_val`, `train_test` | 4612 | $\frac{2151}{2151+352+704} \approx 67\%$ |
| `val` | 352 | $\frac{352}{2151+352+704} \approx 11\%$ |
| `test` | 704 | $\frac{704}{2151+352+704} \approx 22\%$ |

Table A.4: Absolute and relative subset sizes for the Car synset.

## A.2 Selected models

We select a subset of the training set to reduce computational cost. We manually curate a training set of 32 images, wherein models are chosen to maximise the variation in shape and texture. Figure A.1 shows the chosen models, Table A.5 the corresponding ShapeNetCore IDs.

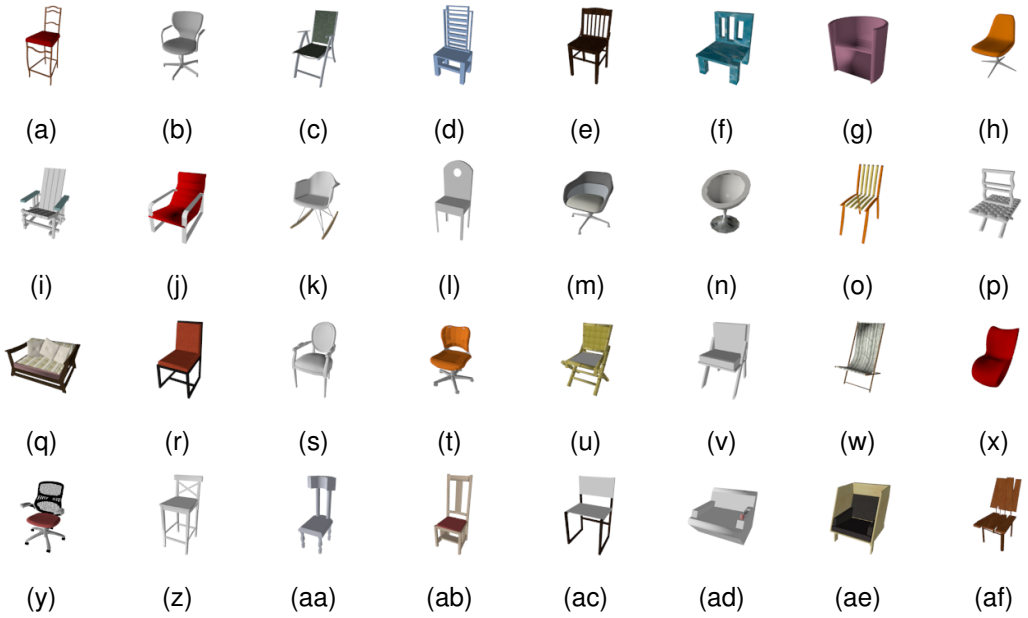



Figure A.1: The 32 models selected for training, as seen from the same viewpoint.

| Model | ShapeNetCore model ID |
|---|---|
| (a) | ae02a5d77184ae2638449598167b268b |
| (b) | 7a9969fac794484c327289c00b6dc9ca |
| (c) | bbef67b2c3d3864e8adc2f75cf0a8389 |
| (d) | 95ac07c8c517929be06a9b687d35bd76 |
| (e) | 96b2bf512fcb51b2af7a8f97983e7906 |
| (f) | 997b0aaad2301a44b31fb46b2e6304f4 |
| (g) | f551bf7431e0fd7cf937a9747c26991f |
| (h) | 197ae965385b8187ae663e348bd216d3 |
| (i) | 11f1511799d033ff7962150cab9888d6 |
| (j) | 7d3b7916dc5325a9c862eec8232fff1e |
| (k) | 712415ce3f126dd921bdbc0445d9f748 |
| (l) | 28fad854838ac444e9920dbaf13176cb |
| (m) | bff224175aed2816597976c675750537 |
| (n) | 76389d102e3fb729f51f77a6d7299806 |
| (o) | 1d7fdf837564523dc89a28b5e6678e0 |
| (p) | c8daa8e9496580667b9c6deef486a7d8 |
| (q) | d2af105ee87bc66dae981a300c94a911 |
| (r) | 63b84cdf260ab81b14b86d5282eb8301 |
| (s) | 8a232028c2b2cfad43649af30eba8304 |
| (t) | 4275718494dd309bc7d25fde6b97816 |
| (u) | 65e770a8307a9332e68b0e385524ba82 |
| (v) | 7eb842de7ad4cbad3e329950ec40f6dd |
| (w) | d64a812bb87a822b8380de241b5e0725 |
| (x) | e5b8d52826245f3937b2bb75885cfc44 |
| (y) | cf88ae03d8cc2fabfcce6278f5ffb13a |
| (z) | 35053caa62eea36c116cc4e115d5fd2 |
| (aa) | f6810de4042cc5ce57bd4bc6eae9b341 |
| (ab) | 2ab159f83754a93ea6c03a53cf0a14c9 |
| (ac) | 21f2927b04e2f22830ddb6ead95f49cc |
| (ad) | cb5f7944ec02defcc6a2b7fc00a47507 |
| (ae) | 9d2cf09ddd9a05fa1f8b303c0da5108d |
| (af) | c1a0882e6e8f8b082b722fc42ccb4c6a |

Table A.5: The ShapeNetCore IDs corresponding to the chosen models displayed in Figure A.1.

# Appendix B

# Models

We outline model-specific details such as hyperparameter sweeps and commands to run experiments, as appropriate. All single-GPU models were trained on a GeForce RTX 2080 Ti GPU. All multi-GPU models used a GTX 1060 GPU. The number of workers matched the number of GPUs used.

## B.1 EfficientNet

We use the EfficientNet implementation provided by torchvision with the default `IMAGENET1K_V1` weights [6].

### B.1.1 Training hyperparameters

Table B.1 summarises the main differences between the hyperparameters used by CodeNeRF [16] and EfficientNet [44].

| | CodeNeRF [16] | EfficientNet [44] |
|---|---|---|
| Initial lr | $10^{-4}$ (NeRF), $10^{-3}$ (latent codes) | $2.56 \times 10^{-1}$ |
| Scheduler | Step-wise, decay by 0.5 every 125 epochs | Step-wise, decay by 0.97 every 2.4 epochs |
| Optimiser | AdamW | RMSProp |
| Weight decay | $10^{-2}$ | $10^{-5}$ |

Table B.1: CodeNeRF [16] and EfficientNet use considerably different hyperparameter values. EfficientNet uses a more rapid learning rate scheduler, orders of magnitude higher initial learning, and a more modest weight decay [44]. This indicates that EfficientNet's weight space is conditioned differently than that of CodeNeRF.

### B.1.2 Inspection of pre-trained weights

Before training, we inspect the pre-trained weights. We are interested in which categories could be loosely mapped onto the "Chair" synset. To this end, we run an

interactive session in Python where we query the set of target categories in the classification task using various synonyms for the term "chair". The chair-like categories are "barber chair", "folding chair", "rocking chair", and "park bench". We outline the session below:

```
>>> from torchvision.models import EfficientNet_B0_Weights
>>> categories = EfficientNet_B0_Weights.IMAGENET1K_V1\
    .value.meta['categories']
>>> find = lambda query:\
    [c for c in categories if query in c]
>>> find('chair')
['barber chair', 'folding chair', 'rocking chair']
>>> find('seat')
['seat belt', 'toilet seat']
>>> find('arm')
['ptarmigan', 'marmot', 'armadillo', 'marmoset', 'harmonica']
>>> find('rock')
['rock python', 'rock crab', 'rock beauty', 'Crock Pot',\
    'rocking chair']
>>> find('bench')
['park bench']
>>> find('recline')
[]
>>> find('sofa')
[]
>>> find('sofa')
[]
>>> find('stool')
[]
>>> find('tabourette')
[]
```

## B.2 pixelNeRF

We use the pixelNeRF [60] code made available by the authors at [59]. We use default parameters wherever possible. **The learning rate and hyperparameters controlling experiment duration had to be adjusted to ensure convergence on our dataset.** We explain this below:

### B.2.1 Training

The original pixelNeRF is trained for 400,000 iterations when trained on individual SRN categories. 400,000 over $\sim 4,500$ models corresponds to $\sim 100$ epochs. In addition, for the first 75% iterations, the authors use a tight bounding box [60]. In our case, this translates to $75\% \times 32\text{models/epoch} \times 100\text{epochs} = 2,400$ iterations.

We notice that pixelNeRF would produce black outputs. As per [9]'s advice, we reduce

the learning rate. We find that reducing it to $5 \times 10^{-5}$, half the original $10^{-4}$ [60], prevents this problem. Admittedly, we did not try other learning rates.

We notice that pixelNeRF's performance is poor. We observe significant cross-epoch improvement on the validation set. We suspect the underperfromance is due to training for too few epochs. We increase the number of epochs 40-fold. This choice is arbitrary, though it extends pixelNeRF experiment duration to $\sim 2$ days, which matches the run time of CodeNeRF experiments.

The command we used to train pixelNeRF on our dataset is:

```
conda run -n pixelnerf python train/train.py \
    -n <experiment_name> \
    -D <datset_directory>/chair \
    -c conf/exp/srn.conf \
    --epochs 4000 \
    --no_bbox_step 96000 \
    --lr 0.00005 \
    --gpu_id 0
```

### B.2.2 Testing

To obtain test-set metrics, we ran the following commands in sequence:

```
conda run -n pixelnerf python eval/eval.py \
    -n <experiment_name> \
    -D <datset_directory>/chair \
    -O eval_out/srn_chair \
    -F srn \
    -c conf/exp/srn.conf \
    --gpu_id=0

conda run -n pixelnerf python eval/calc_metrics.py \
    -D <datset_directory>/chair_test \
    -O eval_out/srn_chair \
    -F srn \
    --gpu_id=0 \
    --overwrite
```

All commands were run using the `pixelnerf` environment provided in the official code publication [59].

## B.3 CodeNeRF

We use the CodeNeRF [16] code made available by the authors at [15] as the basis for our implementation.

### B.3.1 Training time

#### B.3.1.1 Observation


Figure B.1: The fixed implementation of CodeNeRF executes 300 epochs in 12 hours when trained on 14 models. The difference in gradient is due CodeNeRF's cropping routine. The first 250 epochs are trained on 64 $\times$ 64 centercrops, the rest on the full 128 $\times$ 128 images. For each model, 50 out of 250 viewpoints were used, in accordance with CodeNeRF.

We run the fixed implementation of CodeNeRF in distributed data-parallel (DDP) mode on a machine with 8 CPU cores and 8 NVIDIA GeForce GTX 1060 6GB GPUs. It takes 12 hours to execute 300 epochs on 14 models (Figure B.1). The duration would be the same for 16 models[1].

#### B.3.1.2 Extrapolation

If it takes 12 hours to execute 300 epochs on 16 models, then by linear extrapolation, it would take $4,612/16 \times 12h = 3,459h \approx 144$ days to do the same on 4,612 models, and $\sim 1$ day on 32 models.

### B.3.2 Non-decouplability of shape and texture codes

**Claim.** *In NeRF-based architectures, the gradient of the loss with respect to volume density* $\sigma_i$ *at sample* $i$ *depends on the colour output* $\mathbf{c}$ *at samples* $i' \geq i$*. Formally,*

$$\frac{\delta \mathcal{L}}{\delta \sigma_i} \propto \mathbf{c}_i,\ \mathbf{c}_{i+1},\ \cdots,\ \mathbf{c}_n$$

[1] In DDP, data is split along the batch dimension [5]. Here, a batch is a single model. Thus, two GPUs will be assigned a single model, whereas the rest will be assigned two. In other words, the duration would be the same using 16 models.

*Discussion.* The intuition behind the proof is as follows. For a given sample $i$ the volume density $\sigma_i$ and colour $\mathbf{c}_i$ output from NeRF are independent. However, in NeRF's rendering procedure, multiple network outputs along a ray $\mathbf{r}$ are combined and projected to produce the pixel colour $\hat{\mathbf{C}}(\mathbf{r})$. During optimisation, there will be interactions between the volume density $\sigma_i$ and other colour outputs $\mathbf{c}_j$ ($j \neq i$). We now argue formally.

*Proof.* Direct proof. Recall Equations 2.1, 2.2, 2.3 from Section 2.2:

$$\hat{\mathbf{C}}(\mathbf{r};\, \Theta) = \sum_{i=1}^{N} T_i \alpha_i \mathbf{c}_i \tag{B.1}$$

$$\alpha_i = (1 - \exp(-\sigma_i \delta_i)), \text{ where } \delta_i = t_{i+1} - t_i \tag{B.2}$$

$$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{B.3}$$

Recall also that NeRF uses L2 loss (Equation 2.5), applied to a batch of rays $\mathcal{R}$:

$$\sum_{\mathbf{r} \in \mathcal{R}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2$$

where $\hat{\mathbf{C}}(\mathbf{r})$ is the prediction and $\mathbf{C}(\mathbf{r})$ the ground truth. Let us denote the ray that contains sample $i$ using $\mathbf{r}_i$. We used the relaxed notation $\mathbf{y} = f(\mathbf{x})$ to indicate that $\mathbf{y}$ is equal to some function $f$ of $\mathbf{x}$. Repeated use of $f$ does not imply that the same function is assumed. We have:

$$\begin{aligned}
\frac{\delta \mathcal{L}}{\delta \sigma_i} &= \frac{\delta}{\delta \sigma_i} \sum_{\mathbf{r} \in \mathcal{R}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2 && \text{// B.3.2} \\
&= \sum_{\mathbf{r} \in \mathcal{R}} \frac{\delta}{\delta \sigma_i} \underbrace{\|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2}_{f(\hat{\mathbf{C}}(\mathbf{r}))} && \text{// Sum rule} \\
&= \sum_{\mathbf{r} \in \mathcal{R}} \frac{\delta}{\delta \hat{\mathbf{C}}(\mathbf{r})} f(\hat{\mathbf{C}}(\mathbf{r})) \frac{\delta}{\delta \sigma_i} \hat{\mathbf{C}}(\mathbf{r}) && \text{// Chain rule} \\
&= \sum_{\mathbf{r} \in \mathcal{R}} 2 \left( \hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r}) \right) \frac{\delta}{\delta \sigma_i} \mathbf{C}(\mathbf{r}) && \text{// Power rule} \\
&= 2 \left( \hat{\mathbf{C}}(\mathbf{r}_i) - \hat{\mathbf{C}}(\mathbf{r}_i) \right) \underbrace{\frac{\delta}{\delta \sigma_i} \hat{\mathbf{C}}(\mathbf{r}_i)}_{A} && \text{// } \mathcal{R} \setminus \{\mathbf{r}_i\} \text{ constant w.r.t. } \sigma_i
\end{aligned} \tag{B.4}$$

We simplify $A$ accordingly:

$$\begin{aligned}
A &= \frac{\delta}{\delta\sigma_i}\hat{\mathbf{C}}(\mathbf{r}_i) \\
&= \frac{\delta}{\delta\sigma_i}\sum_{k=1}^{N} T_k\alpha_k\mathbf{c}_k && \text{// Eq. B.1} \\
&= \sum_{k=1}^{N}\frac{\delta}{\delta\sigma_i}T_k\alpha_k\mathbf{c}_k && \text{// Sum rule} \\
&= \sum_{k=1}^{N}\mathbf{c}_k\frac{\delta}{\delta\sigma_i}T_k\alpha_k && \text{// } \mathbf{c}_k \text{ constant w.r.t } \sigma_i \\
&= \sum_{k=i}^{N}\mathbf{c}_k\frac{\delta}{\delta\sigma_i}T_k\alpha_k && \text{// Change sum bounds, see } (*) \\
&= \mathbf{c}_i\underbrace{\frac{\delta}{\delta\sigma_i}T_i\alpha_i}_{B} + \underbrace{\sum_{k=i+1}^{N}\mathbf{c}_k\frac{\delta}{\delta\sigma_i}T_k\alpha_k}_{C} && \text{// Split sum}
\end{aligned}$$

where $(*)$ follows from the fact that $T_k = f(\sigma_1, \cdots, \sigma_{k-1})$ and $\alpha_k = f(\sigma_k)$, so that $i < k \implies T_k$ and $\alpha_k$ constant w.r.t. $\sigma_i$. We now treat $B$ and $C$, respectively:

$$\begin{aligned}
B &= \frac{\delta}{\delta\sigma_i}T_i\alpha_i \\
&= T_i\frac{\delta}{\delta\sigma_i}\alpha_i && \text{// } T_i = f(\sigma_1, \cdots, \sigma_{i-1}) \text{ constant w.r.t } \sigma_i \\
&= T_i\frac{\delta}{\delta\sigma_i}\left(1-\exp(-\sigma_i\delta_i)\right) && \text{// Eq. B.2} \\
&= T_i\exp(-\sigma_i\delta_i)\delta_i && \text{// Sum and chain rule} \qquad \text{(B.5)}
\end{aligned}$$

$$
\begin{aligned}
C &= \sum_{k=i+1}^{N} \mathbf{c}_k \frac{\delta}{\delta\sigma_i} T_k \alpha_k \\
&= \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \frac{\delta}{\delta\sigma_i} T_k && \text{// } \alpha_k = f(\sigma_k) \text{ const. w.r.t. } \sigma_i \; \forall k \neq i \\
&= \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \frac{\delta}{\delta\sigma_i} \prod_{j=1}^{k-1} (1-\alpha_j) && \text{// Eq. B.3} \\
&= \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \frac{\delta}{\delta\sigma_i} (1-\alpha_i) \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) && \text{// Isolate } \alpha_i \\
&= \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) \; \frac{\delta}{\delta\sigma_i} (1-\alpha_i) && \text{// Proudct constant w.r.t. } \sigma_i \\
&= -\sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) \; \exp(-\sigma_i \delta_i) \delta_i && \text{// Sum rule and Eq. B.5 } \sigma_i \qquad \text{(B.6)}
\end{aligned}
$$

(B.7)

Plugging $B$ (Eq. B.5) and $C$ (Eq. B.6) back into $A$, we obtain:

$$
\begin{aligned}
A &= \mathbf{c}_i B + C \\
&= \mathbf{c}_i T_i \exp(-\sigma_i \delta_i) \delta_i - \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) \; \exp(-\sigma_i \delta_i) \delta_i \\
&= \left( \mathbf{c}_i T_i - \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) \right) \exp(-\sigma_i \delta_i) \delta_i \qquad \text{(B.8)}
\end{aligned}
$$

(B.9)

Finally, we plug $A$ (Eq. B.8) into Equation B.4, obtaining:

$$
\begin{aligned}
\frac{\delta \mathcal{L}}{\delta \sigma_i} &= 2 \left( \hat{\mathbf{C}}(\mathbf{r}_i) - \hat{\mathbf{C}}(\mathbf{r}_i) \right) A \\
&= 2 \left( \hat{\mathbf{C}}(\mathbf{r}_i) - \hat{\mathbf{C}}(\mathbf{r}_i) \right) \left( \mathbf{c}_i T_i - \sum_{k=i+1}^{N} \mathbf{c}_k \alpha_k \prod_{\substack{j=1, \\ j\neq i}}^{k-1} (1-\alpha_j) \right) \exp(-\sigma_i \delta_i) \delta_i \\
&\propto \mathbf{c}_i, \; \mathbf{c}_{i+1}, \; \cdots, \; \mathbf{c}_n
\end{aligned}
$$

□

### B.3.3 Training procedure

We train CodeNeRF for 300 epochs, as implied by (albeit not formally stated in) the figures in the original work [16]. We use the epoch that achieved the lowest validation loss to evaluate out-of-sample performance. To reduce the computational cost, we run the validation step every ten epochs. We use a learning rate of $10^{-3}$ for latent codes, and $10^{-4}$ for the rest of the model. For optimisation, we use AdamW [20] with a weight decay of 0.01 and $\beta_1 = 0.9, \beta_2 = 0.999$.

As gleaned from the implementation, we follow a cropping procedure wherein the middle 25% of an image is used for the first 250 epochs. We employ a step decay learning rate schedule whereby the learning rate is halved every 125 epochs. This value was expressed in terms of iterations in the original implementation. We converted it to epochs since this is dataset size-agnostic.

## B.4 EnCodeNeRF

### B.4.1 Hyperparameter search for Encoder feature extractor

We summarise the hyperparameter values in consideration in Table B.2. The values were obtained by systematically changing the original values used in CodeNeRF.

| Scope | Hyperparameter | Values |
|---|---|---|
| $f_{\mathbf{e}}$ | Learning rate ($\eta$) | $10^{-2}, 10^{-3}, 10^{-4}$ |
| $f_{\mathbf{e}}$ | Number of bottleneck layers ($\#_B$) | 0, 1, 2 |
| $f_{\mathbf{e}}$ | Bottleneck layer dimensionality ($d_B$) | 16, 32, 64, 128, 256 |
| $f_{\mathbf{e}}$ | Aggregation | Mean, Ø |
| $E$ | Fine-tune final layer | $\perp$, $\top$ |
| $E$ | Learning rate$^*$ | Ours ($\eta$), Theirs (Section B.1.1) |

Table B.2: $f_{\mathbf{e}}$ bridges two models: an EfficientNet encoder $E$ and CodeNeRF (Figure 5.2). We make simplifying architectural assumptions to reduce search space, e.g. we consider the same modification for the shape and texture latent code part of the network. However, we carry out extensive *hyperparameter* search to identify the optimal configuration. ($*$ applicable if fine-tuning the final encoder layer.).

### B.4.2 Hyperparameter search for Encoder feature - Parameter combiner

We remark that hyperparameter values under consideration were extended or pruned according to intermediate results. In Encoder-only, there are no hyperparameters. In Weighted sum, we make the weight $\alpha$ a hyperparameter. We let $\alpha \in \{10\%, 30\%, 50\%\}$. For Dense, we consider adding a ReLU nonlinearity after the projection. We use a single layer for simplicity and parameter efficiency. Finally, note that this search complements the one in Section B.4.1. All parameters use the CodeNeRF learning rate decay procedure (Section B.1.1), except where noted otherwise.

# Appendix C

# Advice

## C.1 Weights & Biases

We curate advice related to Weights & Biases (W&B). W&B is a logging framework that interfaces with deep learning frameworks, such as PyTorch Lightning.

- **The W&B API does not support IPv6 traffic.** The machine must connect to the W&B API to synchronise logs with the online logging dashboard. In cases where the machine uses an IPv6 IP address, SSH tunnelling via an IPv4 proxy must be set up to connect to the API. For instance, in the University's SLURM cluster, compute nodes and the head node use IPv6 and IPv4 addresses, respectively. Thus, all traffic to the W&B API must be tunnelled through the head node.
- **W&B can run a machine out of memory if media files are being logged.** W&B process logs media files in the `/tmp` directory [47]. If multiple experiments run on the same node, the node may run out of memory. This will crash all experiments on that node. To remedy this, a script such as

  ```
  find /tmp -type d -name "*wandb*" -delete -mmin <threshold>
  ```

  should be run periodically on the node.
- **W&B sweeps cannot be run offline.** W&B offers several automated methods (called "sweeps") for hyperparameter search. However, these methods require connection to the W&B API [42]. As per previous points, this is impossible on the University's SLURM cluster.