# Variable Bitrate Neural Fields

TOWAKI TAKIKAWA, NVIDIA, University of Toronto, Canada
ALEX EVANS, NVIDIA, United Kingdom
JONATHAN TREMBLAY, NVIDIA, United States of America
THOMAS MÜLLER, NVIDIA, Switzerland
MORGAN MCGUIRE, ROBLOX, University of Waterloo, Canada
ALEC JACOBSON, Adobe Research, University of Toronto, Canada
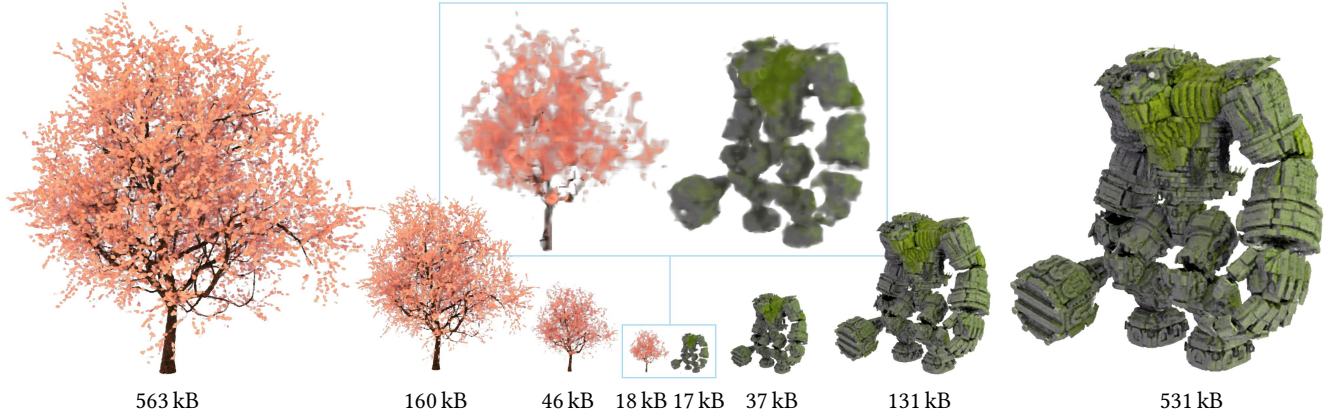SANJA FIDLER, NVIDIA, University of Toronto, Canada

Fig. 1. **Compressed streaming level of detail.** Using our *vector-quantized auto-decoder* (VQ-AD) method, we compactly encode a 3D signal in a hierarchical representation which can be used for progressive streaming and level of detail (LOD). Two example neural radiance fields are shown after streaming from 5 to 8 levels of their underlying octrees. The sizes shown are the total bytes streamed; that is, the finer LODs include the cost of the coarser ones. Prior work such as NeRF [Mildenhall et al. 2020] requires ≈ 2.5 MB to be transferred before anything can be drawn.

Neural approximations of scalar- and vector fields, such as signed distance functions and radiance fields, have emerged as accurate, high-quality representations. State-of-the-art results are obtained by conditioning a neural approximation with a lookup from *trainable feature grids* [Liu et al. 2020; Martel et al. 2021; Müller et al. 2022; Takikawa et al. 2021] that take on part of the learning task and allow for smaller, more efficient neural networks. Unfortunately, these feature grids usually come at the cost of significantly increased memory consumption compared to stand-alone neural network models. We present a dictionary method for compressing such feature grids, reducing their memory consumption by up to 100× and permitting a multiresolution representation which can be useful for out-of-core streaming. We formulate the dictionary optimization as a vector-quantized auto-decoder problem which lets us learn end-to-end discrete neural representations in a space where no direct supervision is available and with dynamic topology and structure. Our source code is available at https://github.com/nv-tlabs/vqad.

## 1 INTRODUCTION

Coordinate-based multi-layer perceptrons (MLPs) have emerged as a promising tool for computer graphics for tasks such as view synthesis [Mildenhall et al. 2020], radiance caching [Müller et al. 2021; Ren et al. 2013], geometry representations [Davies et al. 2020; Park et al. 2019], and more [Xie et al. 2021]. Whereas discrete signal representations like pixel images or voxels approximate continuous signals with regularly spaced samples of the signal, these *neural fields* approximate the continuous signal directly with a continuous, parametric function, *i.e.*, a MLP which takes in coordinates as input and outputs a vector (such as color or occupancy).

*Feature grid methods* [Chan et al. 2021; Liu et al. 2020; Martel et al. 2021; Müller et al. 2022; Takikawa et al. 2021] are a special class of neural fields which have enabled state-of-the-art signal reconstruction quality whilst being able to render [Takikawa et al. 2021] and train at interactive rates [Müller et al. 2021]. These methods embed coordinates into a high dimensional space with a lookup from a
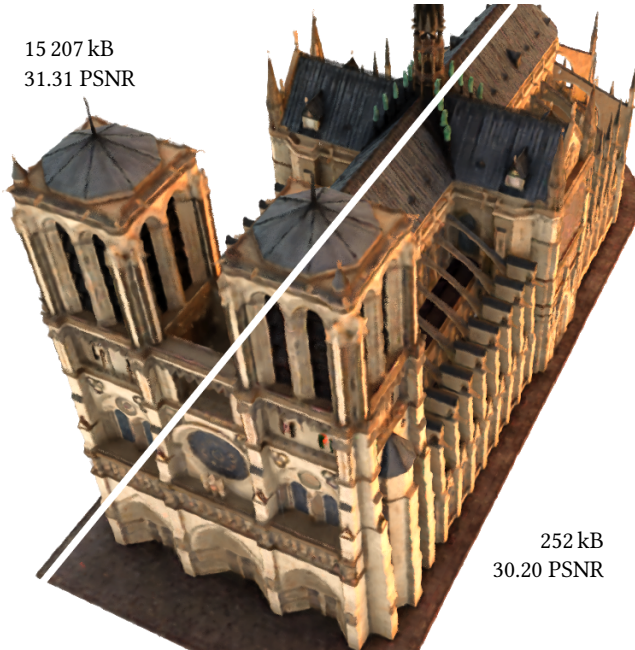
15 207 kB
31.31 PSNR

252 kB
30.20 PSNR

Fig. 2. **Feature Grid Compression**. Top-left shows a baseline neural radiance field whose *uncompressed* feature grid weighs 15 207 kB. Our method, shown bottom right, compresses this by a factor of 60x, with minimal visual impact (PSNR shown relative to training images). In a streaming setting, a coarse LOD can be displayed after receiving only the first 10 kB of data. All sizes are without any additional entropy encoding of the bit-stream.

parametric embedding (the feature grid), in contrast to non-feature grid methods which embed coordinates with a fixed function such as positional Fourier embeddings [Tancik et al. 2020]. This allows them to move the complexity of the signal representation away from the MLP and into the feature grid (a spatial data structure such as a sparse grid [Liu et al. 2020; Takikawa et al. 2021] or a hash table [Müller et al. 2022]). These methods require high-resolution feature grids to achieve good quality. This makes them less practical for graphics systems which must operate within tight memory, storage, and bandwidth budgets. Beyond compactness, it is also desirable for a shape representation to dynamically adapt to the spatially varying complexity of the data, the available bandwidth, and desired level of detail.

In this paper, we propose the *vector-quantized auto-decoder* (VQ-AD) method to directly learn compressed feature-grids for signals without direct supervision. Our representation enables progressive, *variable bitrate* streaming of data by being able to scale the quality according to the available bandwidth or desired level of detail, see Figure 1. Our method enables end-to-end *compression-aware* optimization which results in significantly better results than typical vector quantization methods for discrete signal compression. We evaluate our method by compressing feature-grids which represent neural radiance fields (NeRF) [Mildenhall et al. 2020] and show that our method is able to reduce the storage required by two orders of

magnitude with relatively little visual quality loss without entropy encoding (see Figure 2).

## 2 RELATED WORKS

### 2.1 Compression for Computer Graphics

The ability to dynamically compress and filter data is of great importance in computer graphics. Nanite [Karis et al. 2021] uses mesh levels of detail [Luebke et al. 2003] for out-of-core streaming of assets, which adapts to the image-space footprint to decouple the complexity of assets from the complexity of the render. Balsa Rodríguez et al. [2014] survey techniques to render massive volume datasets, which make extensive use of streaming and compression. Maglo et al. [2015] survey mesh compression, which include progressive streaming methods. Prefiltered representations like mipmaps [Williams 1983] are heavily utilized in real-time graphics systems.

Most relevant to our paper are the works on the compression and streaming of voxels, which approximate volumes with regularly spaced samples. The primary challenge for voxel-based systems is their high memory utilization. Crassin et al. [2009] adopt a block-based $N^3$-tree [Lefebvre et al. 2005], which stores dense $N^3$ bricks in a sparse tree. They take advantage of the efficient GPU texture unit and leverage lazy out-of-core streaming with cone-tracing [Crassin et al. 2011]. These approaches require extra care to handle interpolation at block boundaries, which can either be handled through neighbour pointers [Ljung et al. 2006] or by allocating extra storage for redundant border elements [Crassin et al. 2009]. Filtering data in voxel structures has also been well studied [Heitz et al. 2015; Heitz and Neyret 2012]. The nodes of such $N^3$-trees can be compressed with transform coding. Tang et al. [2018] compress with a block-wise Karhunen-Loève transform (KLT). Wang et al. [2019] and Tang et al. [2020] compress with auto-encoders. Zhang et al. [2014] use a tree whose nodes are sparse blocks encoded with a graph Laplacian transform. De Queiroz and Chou [2016] compress with a global transform on the sparse tree structure using an adaptive Haar wavelet transform. Efficient software libraries like OpenVDB [Museth 2021; Museth et al. 2019] exist to work with sparse brick structures. Our work is similar in that we want to compress a feature grid, which consists of sparse blocks of features that condition a neural network; however this comes with additional difficulties of having to optimize with respect to loss functions on downstream tasks. We take inspiration from these works and propose an end-to-end trainable compression scheme for feature grids. By storing data at *all* levels of a multi-resolution octree as in [Crassin et al. 2009; Takikawa et al. 2021], we can stream the resulting data structure in a breadth-first fashion. This allows coarse levels of detail to be rendered almost immediately, with features from the subsequent tree levels progressively refining the model.

### 2.2 Compression for Neural Fields

Compression is one of the often mentioned benefits of using neural fields to represent signals, however there are still relatively few works which evaluate this property. We review works which evaluate compression for both global methods which use standalone neural networks, as well as feature-grid methods.

*2.2.1 Global methods.* Many works [Davies et al. 2020; Dupont et al. 2021; Zhang et al. 2021] formulate compression as an architecture search problem where a hyperparameter sweep is used to find the optimal architecture with the desired rate-distortion tradeoff, and variable bitrate is achieved by storing multiple models. Bird et al. [2021] directly minimize the rate-distortion tradeoff with a differentiable approximation of entropy, and variable bitrate is achieved by tuning for different tradeoffs. Lu et al. [2021] use vector quantization of MLP parameters alongside quantization at different bitwidths for variable rate compression. These are all global-methods and hence reformulate the problem as neural network model compression. Although not for compression, Lindell et al. [2021] learn a series of bandlimited signals with multiplicative filter networks [Fathony et al. 2020], Barron et al. [2021] propose a special positional encoding function which can represent the expanding spatial footprint of cone-tracing samples, and Baatz et al. [2021] uses the same spatial footprint as an input to the neural network to filter. The resulting filterable representations have fixed size (static bitrate), in contrast to our work where we aim to achieve variable bitrate via streaming level of detail.

*2.2.2 Feature-grid methods.* Takikawa et al. [2021] learn a multiresolution tree of feature vectors which can be truncated at any depth; this achieves an adaptive bitrate through streaming of a breadth-first prefix of the tree. Isik et al. [2021] directly learn the transform coefficients of an adaptive Haar wavelet transform for a feature grid. Müller et al. [2022] use a hash table to learn compact but fixed-size feature grids; large tables are required for good quality. We see these methods as complementary to our contributions. In all of these works, neural signal compression is generally treated as a separate problem from discrete signal compression. In this paper, we show-case how the auto-decoder framework [Park et al. 2019] can directly bridge the gap between discrete signal compression and neural signal compression.

## 3 BACKGROUND

We will first give a background on signal compression and neural fields to provide an overview of the terminology and concepts used throughout the paper. We define a signal $u(x) : \mathbb{R} \to \mathbb{R}$ as a continuous function which maps a coordinate $x$ to a value. In discrete signal processing, signals are typically approximated by a sequence of values of length $n$, representing evenly spaced samples of the continuous signal:

$$u_x = [u_1, ..., u_n] \tag{1}$$

where $u_i$ denotes the $i$-th sample in the sequence. If the continuous signal $u(x)$ is bandlimited and the spacing of the samples $u_x$ exceeds the Nyquist rate [Shannon 1984], the continuous signal can be reconstructed exactly with sinc interpolation. Computer graphics deals with multi-dimensional, multi-channel signals where the coordinates are often 2-dimensional, 3-dimensional, or higher and the output dimension is $d > 1$. The multi-dimensional axis can often be flattened into 1D and the channels can be dealt as separate signals.

A neural field [Xie et al. 2021] is a parametric function $\psi_\theta(x) \approx u(x)$ which approximates a continuous signal with a continuous function with parameters $\theta$, fitted through stochastic optimization.

The parameters of *Global* methods, which includes NeRF [Mildenhall et al. 2020], consist entirely of the MLP's weights and biases.

Conversely, *Feature-grid* methods augment the MLP with feature-grid parameters $Z$. The feature-grid is typically a regularly spaced grid, and the function interp is used to interpolate the local feature vectors $z = \text{interp}(x, Z)$ for a given coordinate $x$.
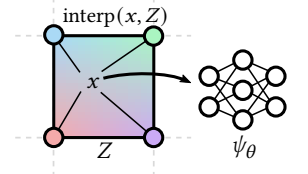


Since $\psi_\theta(x, \text{interp}(x, Z)) \approx u(x)$ is a non-linear function, this approach has the potential to reconstruct signals with frequencies above the usual Nyquist limit. Thus coarser grids can be used, motivating their use in signal compression.

The feature grid can be represented as a matrix $Z \in \mathbb{R}^{m \times k}$ where $m$ is the number of grid points, and $k$ is the dimension of the feature vector at each grid point. Since $m \times k$ may be quite large compared to the size of the MLP, the feature vectors are by far the most memory hungry component. For an example, Müller et al. [2022] utilize ten thousand MLP weights and *12.6 million* feature grid parameters to represent radiance fields. We therefore wish to compress the feature grids and look to discrete signal compression for inspiration.

A standard method for compressing *discrete* signals is known as transform coding [Goyal 2001], where a function transforms the discrete signal $u_x$ to a representation $v_x$:

$$
\begin{aligned}
v_x &= f(u_x) \\
u_x &= f^{-1}(v_x)
\end{aligned}
\tag{2}
$$

We refer to the transformed representation $v_x$ as the *transform coefficients*. The role of this transform is to decorrelate the signal such that quantization or truncation can be applied on the coefficients to effectively compress them.

***Linear*** transform coding uses a linear transform $A \in \mathbb{R}^{n \times n}$ to produce the transform coefficients $v_x = Au_x$; the signal $u_x$ can then be reconstructed with the inverse $A^{-1}$. These transform matrices can be fixed (based on an efficient transform such as DFT and DCT [Ahmed et al. 1974]) or constructed from data. The Karhunen-Loève transform (KLT), for example, is a data-driven transform which can optimally decorrelate Gaussian distributed data.

***Non-linear*** transform coding [Ballé et al. 2020] uses a parametric function $f_\gamma : \mathbb{R}^n \to \mathbb{R}^n$ with parameters $\gamma$ along with its inverse $f_\gamma^{-1}$ to encode and decode discrete signals. The transform $f_\gamma$ is often a neural network. In comparison to the similar setup known as the *auto-encoder* [Kingma and Welling 2013; Kramer 1991], non-linear transform coding has the additional goal of compressing the transform coefficients. On the other hand, *auto-decoder* [Park et al. 2019] refers to only explicitly defining the inverse transform $f_\gamma^{-1}$ and performing the forward transform via stochastic optimization on the transform parameters $\gamma$ and the transform coefficients $v_x$. That is, the forward transform is $f(u_x) = \arg\max_{v_x, \gamma} \|f_\gamma^{-1}(v_x) - u_x\|$. This could also be seen as a form of stochastic variational inference [Hoffman et al. 2013]. Similar to non-linear transform coding for auto-encoders, we define the *compressed auto-decoder* as the compressive equivalent of the auto-decoder.
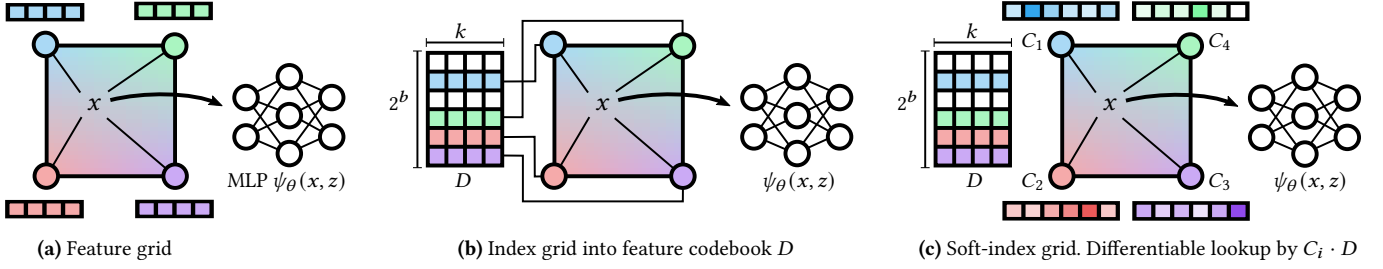
**(a)** Feature grid     **(b)** Index grid into feature codebook $D$     **(c)** Soft-index grid. Differentiable lookup by $C_i \cdot D$

Fig. 3. **(a)** shows the baseline uncompressed version of our data structure, in which we store the bulky feature vectors at every grid vertex, of which there may be millions. In **(b)**, we store a compact $b$-bit code per vertex, which indexes into a small codebook of feature vectors. This reduces the total storage size, and this representation is directly used at inference time. This indexing operation is not differentiable; at training time **(c)**, we replace the indices with vectors $C_i$ of width $2^b$, to which softmax $\sigma$ is applied before multiplying with the entire codebook. This 'soft-indexing' operation *is* differentiable, and can be converted back to 'hard' indices used in **(b)** through an *argmax* operation.

Computing the transform with respect to the entire sequence can be computationally expensive for large sequences. Block-based transform coding divides the signal $u_x$ into fixed size chunks of size $k$. Instead of computing global transform coefficients $v_x = Au_x$ with a large $n \times n$ matrix, we can reshape $u_x \in \mathbb{R}^n$ into a matrix $U \in \mathbb{R}^{\frac{n}{k} \times k}$. The smaller, *block-wise* transform $\hat{A} \in \mathbb{R}^{k \times k}$ is applied to get block transform coefficients:

$$V = U\hat{A}$$
$$U = V\hat{A}^{-1} \tag{3}$$

In the non-linear case, we can use a function $f_\gamma : \mathbb{R}^k \to \mathbb{R}^k$ to code individual blocks (rows of the matrix $U$). For further compression, the rows of $U$ can be clustered via vector quantization [Gray 1984]. We also compress our feature-grids using methods inspired by block-based compression, specifically vector quantization.

## 4 METHOD

We propose the *vector-quantized auto-decoder* method which uses the auto-decoder framework with an extra focus on learning compressed representations. The key idea is to replace bulky feature-vectors with indices into a learned codebook[1]. These indices, the codebook, and a decoder MLP network are all trained jointly. See Fig. 3 for an overview of the method.

By eschewing the encoder function typically used in transform coding, we are able to learn compressed representations with respect to arbitrary domains, such as the continuous signal that a coordinate network MLP encodes, even under indirect supervision (such as training a neural radiance field from images with a volumetric renderer). We give an overview of the compressed auto-decoder framework in Section 4.1, show how feature-grid compression fits into the framework in Section 4.2, and discuss our specific implementation of this framework in Section 4.3.

## 4.1 Compressed Auto-decoder

In order to effectively apply discrete signal compression to feature-grids, we leverage the auto-decoder [Park et al. 2019] framework where only the decoder $f_\gamma^{-1}$ is explicitly constructed; performing

[1]In prior work such as [Takikawa et al. 2021], the feature vectors consumed 512 bits each; the codebook indices that replace them in this work may be as small as 4 bits.

the forward transform involves solving the following optimization problem through stochastic gradient descent:

$$\underset{v_x, \gamma}{\arg\min} \|f_\gamma^{-1}(v_x) - u_x\|. \tag{4}$$

A strength of the auto-decoder is that it can reconstruct transform coefficients with respect to supervision in a domain different from the signal we wish to reconstruct. We define a differentiable forward map [Xie et al. 2021] as an operator $F$ which lifts a signal onto another domain. Now, we must solve the following problem:

$$\underset{v_x, \gamma}{\arg\min} \|F(f_\gamma^{-1}(v_x)) - F(u_x)\|. \tag{5}$$

For radiance field reconstruction, the signal of interest $u_x$ is volumetric density and plenoptic color, while the supervision is over 2D images. In this case, $F$ represents a differentiable renderer.

## 4.2 Feature-Grid Compression

The feature grid is a matrix $Z \in \mathbb{R}^{m \times k}$ where $m$ is the size of the grid and $k$ is the feature vector dimension. Local embeddings are queried from the feature grid with interpolation at a coordinate $x$ and fed to a MLP $\psi$ to reconstruct continuous signals. The feature grid is learned by optimizing the following equation:

$$\underset{Z, \theta}{\arg\min} \, \mathbb{E}_{x,y} \|F(\psi_\theta(x, \mathrm{interp}(x, Z))) - y\| \tag{6}$$

where interp represents trilinear interpolation of the 8 feature grid points surrounding $x$. The forward map $F$ is applied to the output of the MLP $\psi$; in our experiments, it is a differentiable renderer [Mildenhall et al. 2020] and $y$ are the training image pixels.

The feature grid $Z$ can be treated as a block-based decomposition of the signal where each row vector (block) of size $k$ controls the local spatial region. Hence, we consider block-based inverse transforms $f_\gamma^{-1}$ with block coefficients $V$. Since we want to learn the compressed features $Z = f_\gamma^{-1}(V)$, we substitute $Z$:

$$\underset{V, \theta, \gamma}{\arg\min} \, \mathbb{E}_{x,y} \|F(\psi_\theta(x, \mathrm{interp}(x, f_\gamma^{-1}(V)))) - y\|. \tag{7}$$

Considering the $F(\psi(x, \theta, \mathrm{interp}(x, Z)))$ as a map which lifts the discrete signal $Z$ to a continuous signal where the supervision (and other operations) are applied, we can see that this is equivalent to a block-based compressed auto-decoder. This allows us to work only

with the discrete signal $Z$ to design a compressive inverse transform $f_\gamma^{-1}$ for the feature-grid $Z$, in our case the vector-quantized inverse transform to directly learn compressed representations.

## 4.3 Vector-Quantization

We show how vector quantization can be incorporated into the compressed auto-decoder framework. We define our compressed representation $V$ as an integer vector $V \in \mathbb{Z}^m$ with the range $[0, 2^b - 1]$. This is used as an *index* into a codebook matrix $D \in \mathbb{R}^{2^b \times k}$ where $m$ is the number of grid points, $k$ is the feature vector dimension, and $b$ is the bitwidth. Concretely, we define our decoder function $f_D^{-1}(V) = D[V]$ where $[\cdot]$ is the indexing operation. See Fig. 3(**b**). The optimization problem is:

$$\underset{D,V,\theta}{\arg\min}\mathbb{E}_{x,y}\|\psi_\theta(x, \text{interp}(x, D[V])) - y\| \qquad (8)$$

Solving this optimization problem is difficult because indexing is a non-differentiable operation with respect to the integer index $V$.

As a solution, in training we propose to represent the integer index with a *softened* matrix $C \in \mathbb{R}^{m \times 2^b}$ from which the index vector $V = \arg\max_i C[i]$ can be obtained from a row-wise argmax. We can then replace our index lookup with a simple matrix product and obtain the following optimization problem:

$$\underset{D,C,\theta}{\arg\min}\mathbb{E}_{x,y}\|\psi_\theta(x, \text{interp}(x, \sigma(C)D)) - y\| \qquad (9)$$

where the softmax function $\sigma$ is applied row-wise on the matrix $C$. This optimization problem is now differentiable. (See Fig. 3(**c**))

In practice, we adopt a straight-through estimator [Bengio et al. 2013] approach to make the loss be aware of the hard indexing during training. That is, we use Equation 8 in the forward pass and Equation 9 in the backward pass. Other choices of approximations like the Gumbel softmax [Jang et al. 2016] exist, but we empirically find that straight-through softmax works well.

At storage and inference, we discard the softened matrix $C$ and only store the integer vector $V$. Even without entropy coding, this gives us a compression ratio of $16mk/(mb + k2^b)$ which can be orders of magnitude when $b$ is small and $m$ is large. We generally observe $m$ to be in the order of millions, and evaluate $b \in \{4, 6\}$ for our experiments. In contrast to using a hash function [Müller et al. 2022] for indexing, we need to store $b$-bit integers in the feature grid but we are able to use a much smaller codebook (table) due to the learned adaptivity of the indices.

Rather than a single resolution feature-grid, we arrange $V$ in a multi-resolution sparse octree as in NGLOD [Takikawa et al. 2021], to facilitate streaming level of detail. Thus, for a given coordinate, multiple feature vectors $z$ are obtained - one from each tree level - which can then be summed (i.e. in a Laplacian pyramid fashion) or concatenated before being passed to the MLP. We train a separate codebook for each level of the tree. Similarly to NGLOD [Takikawa et al. 2021], we also train multiple levels of details jointly.

## 5 EXPERIMENTS

### 5.1 Baseline and Implementation Details

As an *uncompressed* baseline for our experiments, we implement the sparse multi-resolution feature grid architecture in NGLOD [Takikawa

| Method | PSNR ↑ | SSIM ↑ | LPIPS ↓ | Storage [fp16] |
|--------|--------|--------|---------|----------------|
| NeRF | 28.28 | 0.9398 | 0.0410 | 2.5 MB |
| mip-NeRF | 31.61 | 0.9582 | 0.0214 | 1.2 MB |
| Plenoxels | 31.38 | 0.9617 | 0.0431 | ≈ 168 MB |
| NGLOD-NeRF | 32.72 | 0.9700 | 0.0379 | ≈ 20 MB |

Table 1. **Baseline References**. This table shows the baseline feature-grid method (NGLOD-NeRF) in comparison to NeRF and mip-NeRF which are state-of-the-art global-methods, and Plenoxels which is also a feature-grid method. We see from the results that NGLOD-NeRF is a strong baseline with similar quality to both. All floats are half precision.

| Method | PSNR ↑ | SSIM ↑ | Bitrate (CR) |
|--------|--------|--------|--------------|
| NGLOD-NeRF | 32.72 | 0.9700 | 20 MB (1.0×) |
| + LRA (8f) | 29.09 | 0.9546 | 10.2 MB (2.0×) |
| + LRA (4f) | 26.98 | 0.9387 | 5.1 MB (4.0×) |
| + kmVQ (6 bw) | 27.25 | 0.9322 | 0.49 MB (40.9×) |
| + kmVQ (4 bw) | 25.02 | 0.9112 | 0.33 MB (61.3×) |
| Ours (6 bw) | 30.76 | 0.9567 | 0.49 MB (40.9×) |
| Ours (4 bw) | 30.09 | 0.9482 | 0.33 MB (61.3×) |

Table 2. **LRA, VQ vs loss-aware VQ (ours)**. This table shows the comparison between low-rank approximation (LRA), vector quantization (kmVQ) and learned vector quantization (ours) at different truncation sizes (for LRA) and different quantization bitwidths (for kmVQ and ours). We see that across all metrics we see a significant improvement by learning vector quantization. The bitrate is data dependent, so we report average bitrate.



VQ (bw-4)          Ours (bw-4)

Fig. 4. **Post-Process vs. Learned Vector Quantization**. We compare applying k-means vector quantization on the feature grid as a post-processing after training, vs. learning vector quantization end-to-end with the same number of codebook entries. We see the k-means quantization has visible discoloration, whereas ours preserves the visual quality.

et al. 2021] for the task of learning radiance fields from RGB-D input, with minor modifications to make NGLOD more suitable for this task (see supplemental materials for more details). Although we choose NGLOD as our baseline, our method is agnostic to the choice of data structure in which to store the feature grid.

We initialize NGLOD's octree with depth maps associated with each training image. To train without such calibrated depth information, we could in principle adopt a coarse-to-fine approach [Liu et al. 2020; Yu et al. 2021], and refine the octree structure during training.
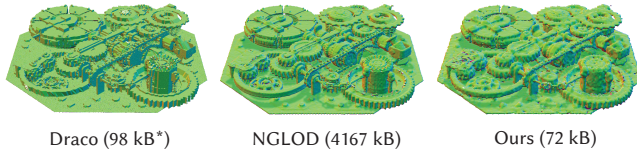
| Draco (98 kB*) | NGLOD (4167 kB) | Ours (72 kB) |

Fig. 5. **Compressing geometry**. We show how VQ-AD can compress signed distance functions as in NGLOD. Our method introduces visible artifacts in the normals, however it does result in a significant bitrate reduction. We also compare against a quantized Draco mesh which has similar bitrates when entropy coded (2 MB as the decompressed binary .ply mesh).

However we do not evaluate this, choosing instead to focus on our proposed contributions in compression and streaming. Note that the references we compare against did not use depth during training, and could benefit from depth supervision [Deng et al. 2021]. These methods are mainly used only as reference points of quality.

For all evaluations, we use the 10 brick scenes from RTMV dataset [Tremblay et al. 2022] which has challenging high-complexity assets. We use the same evaluation scheme and evaluate the results at a $400 \times 400$ resolution with LPIPS [Zhang et al. 2018] (with VGG), SSIM [Wang et al. 2004], and PSNR. We list detailed architectural hyperparameters in the supplemental material. For some figures, we use our own custom dataset of assets collected from TurboSquid which we renderered in NViSII [Morrical et al. 2021].

Our reference comparisons with NeRF [Mildenhall et al. 2020], mip-NeRF [Barron et al. 2021], and Plenoxels [Yu et al. 2021] were produced using the author's code using default hyperparameters and with the same evaluation setting as described previously. Table 1 shows a comparison of the reconstruction quality of these prior methods to our own baseline, uncompressed NGLOD-NeRF. We are able to achieve comparable quality despite being orders of magnitudes faster than the global methods. The storage impact for the feature-grid methods is much higher than the global methods, motivating our compression technique which we evaluate next.

### 5.2 Feature Grid Compression

To evaluate the efficacy of the vector-quantized auto-decoder (VQ-AD) method, we evaluate several different baselines which perform compression as *postprocessing* on a trained model. Our first baseline is low-rank approximation (LRA) through the KLT transform on individual feature vectors at different truncation sizes (f). Our second baseline is vector quantization (kmVQ) through k-means clustering with different numbers of clusters at different quantization bitwidths (bw). We report their average bitrate (see Table 2) along with their compression rate with respect to the baseline. All results assume half-precision (16 bit) floating point numbers and do not use entropy coding. LRA is not competitive in quality nor compression ratio compared to the other methods. Figure 4 shows qualitative results between kmVQ and our method, where kmVQ causes noticeable discoloration. This is corroborated by the quantitative results which show kmVQ to be at a significant quality disadvantage, at equal size; this shows the efficacy of learning the indices during training.

Our VQ-AD method can also be applied in contexts other than fitting radiance fields. Figure 5 shows the original results from NGLOD
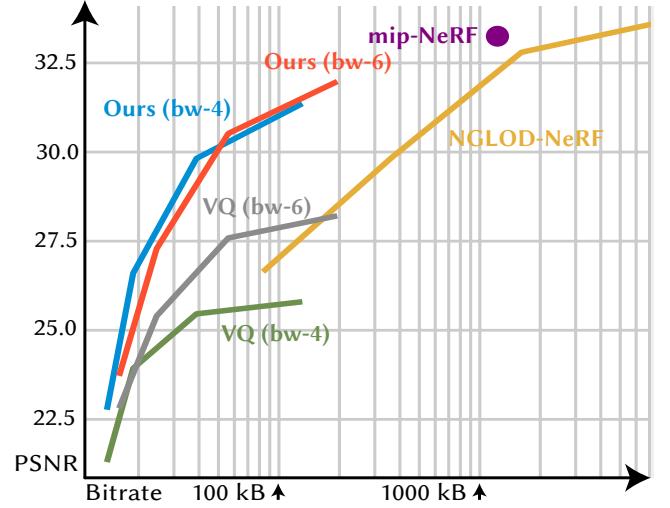


Fig. 6. **Rate Distortion Curve**. This graph shows the rate-distortion trade-offs of different methods on the 'Night Fury' RTMV scene, where the y-axis is PSNR and the x-axis is bitrate (in log-scale). Single-bitrate architectures are represented with a dot. For Mip-NeRF (purple), the filtering mechanism can move the dot vertically, but not horizontally. Our compressed architecture (red and blue) has variable-bitrate and is able to dynamically scale the bitrate to different levels of details. Our architecture is more compact than feature-grid methods like NGLOD (yellow) and achieves better quality than postprocessing methods like k-means VQ (gray and green).

which fits truncated signed distance functions (TSDF) alongside the VQ-AD version. We also compare against Draco [Galligan et al. 2018], which compresses meshes through entropy coding and heavy quantization of the vertex positions. Our method does introduce visible high frequency artifacts for TSDFs, but we are able to nonetheless reduce the bitrate significantly without relying on entropy coding.

### 5.3 Streaming Level of Detail

We also showcase our ability to learn compressed multi-resolution representations which can be used for progressive streaming. Figure 7 shows a visual comparison between the Fourier encoding-based filtering mechanism from mip-NeRF [Barron et al. 2021] at different cone radii, in comparison to our multiresolution representation. Both are able to produce different levels of detail, but our method is able to also reduce the bitrate accordingly at lower resolutions thus enabling progressive streaming.

Figure 6 shows the rate-distortion curves for different methods, including our compressed multi-resolution architecture. The graph shows that our VQ-AD can achieve orders of magnitudes smaller bitrates, without significantly sacrificing quality like post-processing methods, *e.g.,* kmVQ. The graph highlights that our representation has variable-bitrate and encodes multiple different resolutions which can be progressively streamed at different levels of detail. The memory overhead of our method prevents us from evaluating higher bitrates, we hope to explore this frontier in future work.

mip-NeRF

1224 kB  1224 kB  1224 kB  1224 kB

Ours
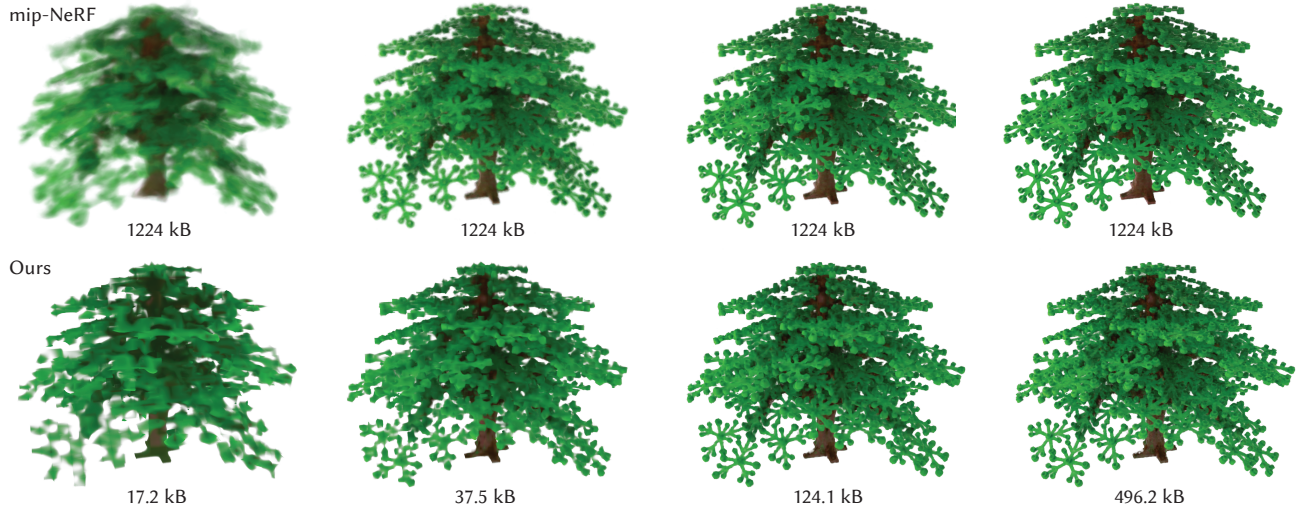
17.2 kB  37.5 kB  124.1 kB  496.2 kB

Fig. 7. **Compressed levels of detail.** From left to right: the different mip levels. Top row: mip-NeRF at different cone widths. Although mip-NeRF produces filtered results, they are constant bitrate. Bottom row: Our multiresolution and vector quantized representation. We are able to simultaneously filter and compress the representation, making it suitable for progressive streaming and level of detail.

## 5.4 Random vs. Learned Indices

Our codebook learning method can be seen as a special form of the hash encoding method from Muller et al. [Müller et al. 2022] where instead of using a static hash function to find an index into the codebook, we learn the indices and bake them into the grid. This in turn allows the use of much smaller codebook sizes at the cost of having to store indices.

To evaluate this tradeoff, we implement an equivalent method to the hash encodings in our NGLOD-NeRF baseline implementations by simply allocating random indices of range $[0, 2^b]$ with a corresponding codebook size. We train this for 120 epochs at several different bitwidths (bw), and show that learning the indices can use a much smaller bitwidth than in the random case (see Table 3) at approximately equal quality. Figure 8 evaluates the visual quality of the random and learned approaches at roughly equal storage cost; we see that the random indices based approach has much more visible noise.

## 6 CONCLUSION

Simultaneous filtering and compression is an important feature for real-life graphics systems. We believe that neural rendering [Tewari et al. 2020, 2021] and neural fields [Xie et al. 2021] will become more integrated into next generation graphics pipelines, and as such it is important to design neural representations that are able to perform the same signal processing operations currently possible with other representations like meshes and voxels. We believe that our method, the vector-quantized auto-decoder, is a step forward in that direction as we demonstrated our method can learn a streamable, compressive representation with minimal visual quality loss.

One of the major drawbacks of our presented method is its training footprint in terms of memory and compute at training time,
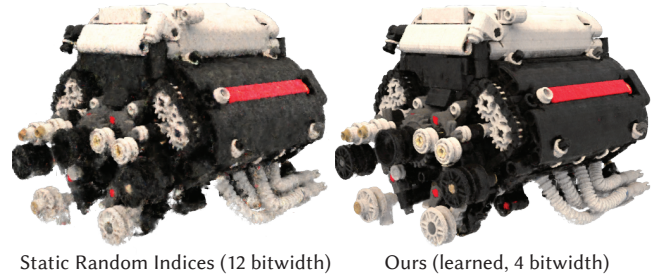


Static Random Indices (12 bitwidth)          Ours (learned, 4 bitwidth)

Fig. 8. **Qualitative comparison of static and learned indices**. We qualitatively compare a hash approach with 12 bitwidth codebooks and our learned indices with 4 bitwidth codebooks which have similar compression rates. We see that our learned indices are able to reconstruct with less noise.

which requires the allocation of a matrix of size $m \times 2^b$ to hold the softmax coefficients before they are converted into indices at inference and storage. We believe that this could be addressed via a hybrid approach between random and learned indices, where instead of storing softened version of indices, we learn a parametric function with respect to coordinates which can predict softened indices on-the-fly. Our approach is also directly compatible with highly efficient frameworks like instant neural graphics primitives [Müller et al. 2022] and we believe that the synthesis of these techniques is a very exciting research direction.

| Method | PSNR ↑ | ‖V‖ | ‖D‖ | Total BR (CR) |
|---|---|---|---|---|
| Hash (16 bw) | 29.75 | 0 kB | 8388 kB | 8400 kB (2.42×) |
| Hash (14 bw) | 28.48 | 0 kB | 2097 kB | 2109 kB (9.65×) |
| Hash (12 bw) | 26.66 | 0 kB | 524 kB | 536 kB (37.9×) |
| Hash (10 bw) | 23.70 | 0 kB | 131 kB | 143 kB (141.9×) |
| Ours (6 bw) | 29.92 | 477 kB | 8 kB | 497 kB (40.9×) |
| Ours (4 bw) | 29.60 | 318 kB | 2 kB | 332 kB (61.3×) |
| Ours (2 bw) | 27.59 | 159 kB | 0.5 kB | 172 kB (118.5×) |
| Ours (1 bw) | 25.57 | 79 kB | 0.3 kB | 92 kB (221.2×) |

Table 3. **Comparison between random indices and learned indices**. This table shows the effects of learning codebook indices with VQAD at 120 epochs with different quantization bitwidths (bw). To highlight the tradeoff, we list the size of the indices $V$ and codebook $D$ separately. We see that even when storing indices, we are able to achieve higher quality than the hash-based approach.

## REFERENCES

Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao. 1974. Discrete cosine transform. *IEEE transactions on Computers* 100, 1 (1974), 90–93.

Hendrik Baatz, Jonathan Granskog, Marios Papas, Fabrice Rousselle, and Jan Novák. 2021. NeRF-Tex: Neural Reflectance Field Textures. (2021).

Johannes Ballé, Philip A Chou, David Minnen, Saurabh Singh, Nick Johnston, Eirikur Agustsson, Sung Jin Hwang, and George Toderici. 2020. Nonlinear transform coding. *IEEE Journal of Selected Topics in Signal Processing* 15, 2 (2020), 339–353.

Marcos Balsa Rodríguez, Enrico Gobbetti, Jose Antonio Iglesias Guitian, Maxim Makhinya, Fabio Marton, Renato Pajarola, and Susanne K Suter. 2014. State-of-the-art in compressed GPU-based direct volume rendering. In *Computer graphics forum*, Vol. 33. Wiley Online Library, 77–100.

Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. 2021. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. *arXiv preprint arXiv:2103.13415* (2021).

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

Thomas Bird, Johannes Ballé, Saurabh Singh, and Philip A Chou. 2021. 3D Scene Compression through Entropy Penalized Neural Representation Functions. *arXiv preprint arXiv:2104.12456* (2021).

Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. 2021. Efficient Geometry-aware 3D Generative Adversarial Networks. In *arXiv*.

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 15–22.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 1921–1930.

Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. 2020. On the effectiveness of weight-encoded neural implicit 3D shapes. *arXiv preprint arXiv:2009.09808* (2020).

Ricardo L De Queiroz and Philip A Chou. 2016. Compression of 3D point clouds using a region-adaptive hierarchical transform. *IEEE Transactions on Image Processing* 25, 8 (2016), 3947–3956.

Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan. 2021. Depth-supervised NeRF: Fewer Views and Faster Training for Free. *arXiv preprint arXiv:2107.02791* (2021).

Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. 2021. COIN: COmpression with Implicit Neural representations. *arXiv preprint arXiv:2103.03123* (2021).

Rizal Fathony, Anit Kumar Sahu, Devin Willmott, and J Zico Kolter. 2020. Multiplicative Filter Networks. In *International Conference on Learning Representations*.

Frank Galligan, Michael Hemmer, Ondrej Stava, Fan Zhang, and Jamieson Brettle. 2018. Google/draco: a library for compressing and decompressing 3d geometric meshes and point clouds. https://github.com/google/draco.

Vivek K Goyal. 2001. Theoretical foundations of transform coding. *IEEE Signal Processing Magazine* 18, 5 (2001), 9–21.

Robert Gray. 1984. Vector quantization. *IEEE Assp Magazine* 1, 2 (1984), 4–29.

Eric Heitz, Jonathan Dupuy, Cyril Crassin, and Carsten Dachsbacher. 2015. The SGGX Microflake Distribution. *ACM Trans. Graph.* 34, 4, Article 48 (jul 2015). https://doi.org/10.1145/2766988

Eric Heitz and Fabrice Neyret. 2012. Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In *High Performance Graphics 2012*. Eurographics, 125–134.

Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *Journal of Machine Learning Research* 14, 5 (2013).

Berivan Isik, Philip A Chou, Sung Jin Hwang, Nick Johnston, and George Toderici. 2021. LVAC: Learned Volumetric Attribute Compression for Point Clouds using Coordinate Based Networks. *arXiv preprint arXiv:2111.08988* (2021).

Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144* (2016).

Krishna Murthy Jatavallabhula, Edward Smith, Jean-Francois Lafleche, Clement Fuji Tsang, Artem Rozantsev, Wenzheng Chen, Tommy Xiang, Rev Lebaredian, and Sanja Fidler. 2019. Kaolin: A pytorch library for accelerating 3d deep learning research. *arXiv preprint arXiv:1911.05063* (2019).

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. Nanite: A Deep Dive. (2021).

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

Mark A Kramer. 1991. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal* 37, 2 (1991), 233–243.

Sylvain Lefebvre, Samuel Hornus, Fabrice Neyret, et al. 2005. Octree textures on the GPU. *GPU gems* 2 (2005), 595–613.

David B Lindell, Dave Van Veen, Jeong Joon Park, and Gordon Wetzstein. 2021. BACON: Band-limited Coordinate Networks for Multiscale Scene Representation. *arXiv preprint arXiv:2112.04645* (2021).

Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *arXiv preprint arXiv:2007.11571* (2020).

Patric Ljung, Claes Lundström, and Anders Ynnerman. 2006. Multiresolution interblock interpolation in direct volume rendering. (2006).

Yuzhe Lu, Kairong Jiang, Joshua A Levine, and Matthew Berger. 2021. Compressive Neural Representations of Volumetric Scalar Fields. *arXiv preprint arXiv:2104.04523* (2021).

David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. 2003. *Level of detail for 3D graphics*. Morgan Kaufmann.

Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 2015. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–41.

Julien NP Martel, David B Lindell, Connor Z Lin, Eric R Chan, Marco Monteiro, and Gordon Wetzstein. 2021. ACORN: Adaptive Coordinate Networks for Neural Scene Representation. *arXiv preprint arXiv:2105.02788* (2021).

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2020. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*. Springer, 405–421.

Nathan Morrical, Jonathan Tremblay, Yunzhi Lin, Stephen Tyree, Stan Birchfield, Valerio Pascucci, and Ingo Wald. 2021. NViSII: A Scriptable Tool for Photorealistic Image Generation. arXiv:2105.13962 [cs.CV]

Thomas Müller, Alex Evans, Christopher Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. (2022).

Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. 2021. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372* (2021).

Ken Museth. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*. 1–2.

Ken Museth, Nick Avramoussis, and Dan Bailey. 2019. OpenVDB. In *ACM SIGGRAPH 2019 Courses*. 1–56.

Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. 2019. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 165–174.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

Peiran Ren, Jiaping Wang, Minmin Gong, Stephen Lin, Xin Tong, and Baining Guo. 2013. Global illumination with radiance regression functions. *ACM TOG* 32, 4 (2013), 1–12.

Claude E Shannon. 1984. Communication in the presence of noise. *Proc. IEEE* 72, 9 (1984), 1192–1201.

Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. In *CVPR*. 11358–11367.

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. 2020. Fourier Features Let Networks Learn High Frequency Functions in Low

Dimensional Domains. (2020). arXiv:2006.10739 [cs.CV]

Danhang Tang, Mingsong Dou, Peter Lincoln, Philip Davidson, Kaiwen Guo, Jonathan Taylor, Sean Fanello, Cem Keskin, Adarsh Kowdle, Sofien Bouaziz, et al. 2018. Real-time compression and streaming of 4d performances. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–11.

Danhang Tang, Saurabh Singh, Philip A Chou, Christian Hane, Mingsong Dou, Sean Fanello, Jonathan Taylor, Philip Davidson, Onur G Guleryuz, Yinda Zhang, et al. 2020. Deep implicit volume compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 1293–1303.

Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason Saragih, Matthias Nießner, et al. 2020. State of the art on neural rendering. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 701–727.

Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Yifan Wang, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, et al. 2021. Advances in neural rendering. *arXiv preprint arXiv:2111.05849* (2021).

Jonathan Tremblay, Moustafa Meshry, Alex Evans, Jan Kautz, Alexander Keller, Sameh Khamis, Charles Loop, Nathan Morrical, Thomas Müller, Koki Nagano, Towaki Takikawa, and Stan Birchfield. 2022. RTMV: A Ray-Traced Multi-View Synthetic Dataset for Novel View Synthesis. http://www.cs.umd.edu/~mmeshry/projects/rtmv/.

Jianqiang Wang, Hao Zhu, Zhan Ma, Tong Chen, Haojie Liu, and Qiu Shen. 2019. Learned point cloud geometry compression. *arXiv preprint arXiv:1909.12037* (2019).

Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

Lance Williams. 1983. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques.* 1–11.

Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. 2021. Neural Fields in Visual Computing and Beyond. (2021).

Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2021. Plenoxels: Radiance Fields without Neural Networks. *arXiv preprint arXiv:2112.05131* (2021).

Cha Zhang, Dinei Florencio, and Charles Loop. 2014. Point cloud attribute compression with graph transform. In *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2066–2070.

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

Yunfan Zhang, Ties van Rozendaal, Johann Brehmer, Markus Nagel, and Taco Cohen. 2021. Implicit Neural Video Compression. *arXiv preprint arXiv:2112.11312* (2021).

# 7 IMPLEMENTATION DETAILS

## 7.1 Minor Modifications to NGLOD

We follow the open source implementations NGLOD [Takikawa et al. 2021] architecture available at https://github.com/nv-tlabs/nglod. Since the original NGLOD architecture was designed for learning and rendering signed distance functions, we make minor modifications to make the architecture more suitable for learning neural radiance fields. First, in NGLOD, the features can only be queried in the regions where the sparse voxels are allocated for the given level. This can be an issue for radiance fields, because this leads to perturbations in the viewpoint causing rapid change in the voxels being traced, which can cause instability in training and rendering. Instead, we modify the feature lookup function such that any location where sparse voxels are allocated for the *coarsest* level in the multiresolution hierarchy can be sampled. If while traversing the tree for a location $x$ and the location is no longer allocated in the tree for finer resolutions, we simply return a vector of zeros for those levels. In practice, the NGLOD architecture is a sparse Laplacian pyramid which sums the feature vectors from multiple levels, so the zeros end up being a no-op, allowing for an efficient implementation. We also use a single unified MLP which is shared across all levels instead of using a separate MLP per level as in the original NGLOD implementation.

## 7.2 Architectural Hyperparameters

We use a feature vector size of 16, concatenated with the 3-dimensional view direction which is positionally encoded to produce an embedded view direction vector of size 27. The concatenated feature vector and embedded view direction create a vector of size 43, which is the input to the neural network. The neural network is a 2-layer network with a single hidden layer of size 128 and an output dimension of 4 (density and RGB color). We use the ReLU activation function for the hidden layer, ReLU activation on the density output, and sigmoid activation on the RGB color output. We initialize the feature grid with normally distributed samples with a standard deviation of 0.01. Our implementation for volumetric integration uses 16 samples for each voxel that was intersected by each ray, and thus our implementation could benefit (in compute) from an early termination scheme. Because this creates a variable number of voxels and samples per ray, we cannot use standard PyTorch operations to integrate them, and as such we use custom packed CUDA primitives for volumetric integration to process them. We implemented everything with PyTorch [Paszke et al. 2019], custom CUDA kernels, and the differentiable rendering primitives from the Kaolin library [Jatavallabhula et al. 2019].

## 7.3 Other training details

The point cloud to initialize the sparse NGLOD grid is generated by taking the ray origins and ray directions for the ground truth camera parameters and adding the directions multiplied by depth (pixel-wise) to the origins to produce a point cloud. We then normalize this point cloud within a normalize cube with range $[-1, 1]$. We apply this same normalization factors to the ray origins such that the cameras are aligned. We store the normalization scale and offset in the model to use consistent offsets at validation time.

All optimizations and evaluations are performed in SRGB space. We downscale the images using area-weighted bilinear interpolation implemented in OpenCV. On the ground truth images, we premultiply the alphas to remove the boundary artifacts. Training on the baselines were trained for 600 epochs (unless otherwise noted) with a batch size of 4096 with a learning rate of 0.001 with the Adam optimizer. We scale the learning rate of the feature grid by 100 which we find to be important for performance. We performed some minor experiments with TV regularizations as in Plenoxels [Yu et al. 2021] however we found the effects to be minimal and as such we did not use them.

To train multiple levels of details, we follow a similar strategy to NGLOD [Takikawa et al. 2021] where we train multiple level of details with a single model. Whereas in NGLOD they sum the loss function from all levels and train them simultaneously, we instead randomly sample a level of detail per batch. We sample levels from a distribution where each level of detail (starting from the coarsest level) has a 2x more likely chance of being sampled compared to the previous level. We also find that *only* sampling the finest level of detail also manages to learn some level of detail effects, although at compromised quality for the lower levels of detail.

## 8  OTHER EXPERIMENTAL DETAILS

### 8.1  Training and Inference Speeds

Since the timings for both training and inference depends on the model, we will report timings for the Night Fury model on an RTX 8000. We make a note that we do not utilize any optimizations like early stopping which we expect will make a large impact on the training and inference performance.

*8.1.1  Inference.* Inference runs at around 15 FPS at 720p with 8 GB memory for both the uncompressed NGLOD-NeRF baseline and our compressed version (4,6 bitwidth). We expect that the compressed version could be faster with an optimized implementation that utilizes cache better (as showcased by Instant NGP [Müller et al. 2022]). These numbers are heavily influenced by viewpoints, batching, and other implementation choices.

*8.1.2  Training.* Training for 600 epochs takes around 20 minutes for the uncompressed model and around 40 minutes for the compressed model. Both achieve reasonable PSNR (30+) within 50 epochs, which takes around 2 minutes for the uncompressed model and around 4 minutes for the compressed model. These numbers assume there is no extra logging, debug rendering, model checkpoint, etc happening.

The peak memory used during training is 8 GB for the uncompressed model, 8 GB for the 4 bitwidth compressed model, and 18 GB for the 6 bitwidth compressed model. While the memory usage in training is high (as noted in the limitation section of the paper), the memory usage for inference is not affected.

### 8.2  Entropy Coding

In the experiments in the main paper, we do not use any entropy coding. If we do use entropy coding (gzip) on the uncompressed NGLOD-NeRF weights, we get a 7% reduction in size. Using entropy coding on the compressed weights yields a 4% reduction in size. We generally find that the trained indices are somewhat uniformly distributed, leading to smaller gains made by entropy coding.

To make entropy coding more effective, we can apply entropy minimization on the softmax weights in training as a regularization. This can give up to a 56% reduction in size through entropy coding, but at the cost of a large quality drop. Entropy coding also precludes streaming.