Adaptive Multi-Grid Scene Representation Networks for Large-Scale Data Visualization

Category: Research

Paper Type: algorithm/technique

Abstract—Scene representation networks (SRNs) have been recently proposed for compression and visualization of scientific data. However, state-of-the-art SRNs do not adapt the allocation of available network parameters to the complex features found in scientific data. Additionally, they take hours to fit on large-scale data, and no open-source software for rendering the rapidly developing PyTorch-based SRNs exist. We address these three shortcomings with an adaptive multi-grid SRN (AMGSRN), an ensemble training and inference technique, and an open-source neural volume rendering application that allows plug-and-play rendering with any PyTorch-based SRN. Our proposed AMGSRN architecture uses spatially adaptive feature grids to dynamically allocate more neural network resources where error is high in the volume, improving state-of-the-art reconstruction accuracy of SRNs for scientific data without requiring expensive octree refining, pruning, and traversal like previous adaptive models. In our ensemble modeling approach for large-scale data, we train an ensemble of these networks in parallel to reduce training time while avoiding overhead necessary for an out-of-core solution for volumes too large to fit in GPU memory during network training or rendering, reducing training time from the previously required 2 hours to 7 minutes. After training, the lightweight SRNs are used for realtime neural volume rendering in our open-source Python-based renderer, where arbitrary view angles and transfer functions can be explored in realtime due to our image hierarchy-based progressive rendering. A copy of this paper, all code, all models used in our experiments, and all supplemental materials and videos are available at `https://github.com/anonymousvisuser1036/AMGSRN`.

Index Terms—Scene representation network, deep learning, scientific visualization, volume rendering

1 Introduction

Scene representation networks (SRNs) are compact neural networks that map input coordinates to output scalar field values [13, 16, 21–23]. SRNs use a small footprint on disk (data reduction between $10-10000\times$) while being efficient to evaluate with random access. With these benefits, SRNs have been used for compression [14, 26] and volume rendering [26, 27] for scientific data up to sizes of 1TB.

Despite SNRs' popularity, there are three shortcomings of current state-of-the-art SRNs for scientific data visualization. First, the SRN architectures used in recent approaches are not designed to adapt network resources to more important regions, and make the inherit assumption that the volume it will represent has uniform complexity, leading to inefficient use of the network parameters in homogeneous regions and limiting model performance. Second, the current state-of-the-art for fitting an SRN on a large-scale volume ($10240 \times 1568 \times 7680$, 450 GB) takes nearly two hours with an out-of-core sampling strategy [27], limiting practicality. Lastly, the neural volume renderers developed for recent works are either unreleased [14, 27] or are developed in custom C++/CUDA code [26], making it incompatible with the rapid advancements for SRNs that are coded in Python/PyTorch [19].

In this work, we address the three limitations mentioned above with a novel adaptive SRN model, ensemble training routine for fitting large-scale data, and an open-source neural volume renderer. First, we address the lack of adaptivity in state-of-the-art SRNs for scientific data with a novel SRN architecture called adaptive multi-grid SRN (AMGSRN). Recent SRN models have incorporated adaptive parameter allocation with spatially adaptive quadtrees and octrees that refine on the scene's geometry or features. However, these approaches require significant extra storage and computation to store and search the tree structure during training [13, 15, 23, 28], taking hours or days to fit on images with fewer degrees of freedom than most 3D scientific data. In addition to this, the tree structure itself is not directly learnable, so the trees are updated every set number of iterations with an ad-hoc method while training. The tree structure also scales poorly with dimensionality, increasing storage and computation requirements exponentially as the number of dimensions or tree depth increases. Instead of using trees as our adaptive data model in our network, AMGSRN uses a set of spatially adaptive feature grids, shown in Figure 1, whose extents are defined by learned transformation matrices which transform global space into local feature space, where the grid is defined local space as the unit cube $[-1,1]^3$. To guide the grids to cover spatial regions where the model has relatively higher error, we develop a custom *feature density loss*, that calculates the relative entropy from the current feature grid density to a target feature density, where the assumption is higher feature density improves reconstruction (shown by other feature grid networks [3, 23, 26, 28]). This alone isn't enough to train the feature grids though, as the density of a grid is a step function that has a gradient of 0 everywhere. Therefore, we approximate the feature density with a differentiable function that closely matches the step function called a *flat-top gaussian*. We also follow a specific training routine for the feature grid's transformation matrices with delayed start and early stopping to improve accuracy and converge quicker. Our adaptive grid architecture does not require expensive tree searching or pruning like other adaptive models, and dynamically allocates more neural network resources to regions of higher error for any volume, improving state-of-the-art SRN performance by using network parameters efficiently.

Using our AMGSRN architecture as a building block, we reduce state-of-the-art SRN training time on large-scale data from 2 hours [27] to 7 minutes with an ensemble training strategy that fits a large-scale volume in a model-parallel fashion. Our ensemble training approach divides data in the volume on a grid of bricks, and trains one SRN per block of data. The blocks of data fit in GPU memory, so there are no inefficiencies from an out-of-core data sampling method. We also assume that the large-scale data being fit are generated by machines equipped with multiple GPUs per node, so we use all available GPUs to train multiple networks in parallel, reducing total training time. Inference in this ensemble of models is more complicated now, since a search is necessary to find which model was trained on the spatial domain for each point being queried. To accelerate ensemble inference, we use a hash function that maps spatial coordinates to the hashtable entries for the correct model to use for inference in parallel. Not only do ensemble models allow fitting large-scale data in under 7 minutes, but they also increase reconstruction accuracy over a single model at the same number of total model parameters.

Lastly, we develop and release an open-source PyTorch-based neural volume renderer to allow plug-and-play realtime neural volume rendering with trained SRNs. The primary difficulty in a Python-based neural volume renderer is maintaining an interactive framerate. For interactive viewing, we develop an image hierarchy-based progressive rendering

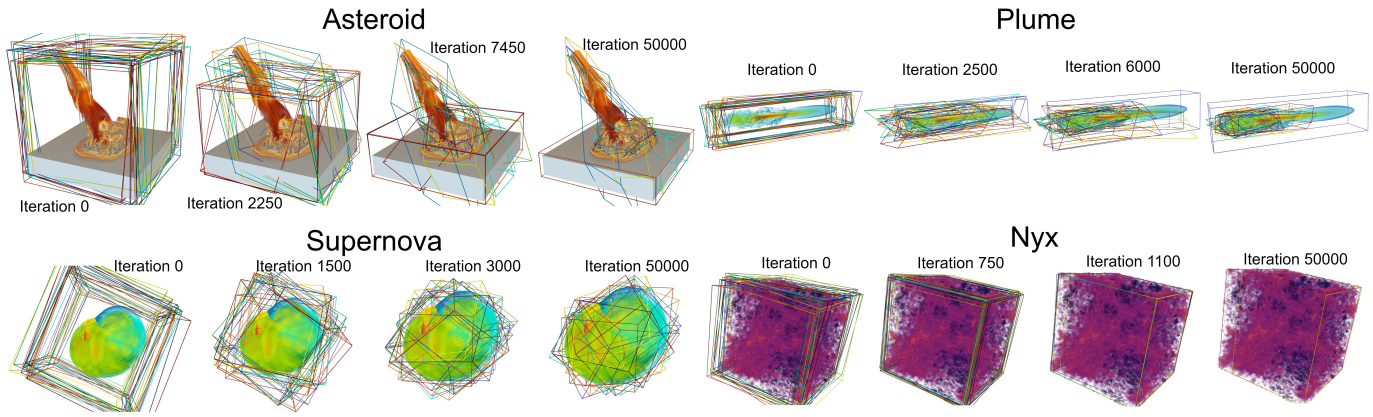

Fig. 1: Examples of our adaptive feature grids fitting to different scientific data during training. Regardless of the features in the data, the feature grids fit the domain well. Video of evolution of grids during training available in supplemental materials.

scheme that evaluates batches of pixels in an order that fills pixels of an image hierarchy from coarsest to finest, facilitating a high-quality approximation quickly regardless of the model's inference speed.

In summary, our contributions are threefold:

- A novel SRN architecture called adaptive multi-grid scene representation network (AMGSRN) with adaptive feature grids that localize trainable network parameters on regions of the volume with high error during training
- An ensemble SRN training strategy that trains multiple SRNs in parallel for fitting large-scale data within minutes
- An open-source neural volume rendering application for interactive exploration of the volume data on a local workstation

Our AMGSRN architecture and ensemble modelling technique are evaluated on several scientific datasets ranging from volumes of size $128^2 \times 512$ (32MB to store) up to $10240 \times 1536 \times 7680$ (450GB to store). We compare the reconstruction quality of our proposed AMGSRN architecture (single model, not ensemble) with other state-of-the-art models, and demonstrate that our adaptive feature grids improve reconstruction quality over state-of-the-art at similar model sizes. We further evaluate the performance of our ensemble framework by fitting two datasets that are 250 GB and 450 GB in less than 7 minutes each. Lastly, we evaluate the performance of our neural rendering application with different state-of-the-art models. We provide all code for our model, training, and neural volume renderer on GitHub: `https://github.com/anonymousvisuser1036/AMGSRN`, along with installation instructions and supplemental videos.

2 Related Works

As our method is primarily related to scene representation networks, we review related literature covering SRN architectures and training, examining both applications in the computer vision domain as well as sci-vis domain.

2.1 Scene representation networks

Scene representation networks, also called implicit neural representations (INRs) or coordinate networks, are networks that encode a given scene with the weights of the neural network, such that input coordinates are mapped to output values. In the context of computer vision, SRNs can learn to represent images [15, 22], signed distance functions [15, 22], or radiance fields [16, 21, 28]. In the context of scientific data and visualization, SRNs have been used to model 3D scalar fields and time-varying scalar fields [14, 26, 27]. We broadly classify the architecture of SRN models into two categories: fully connected and grid-based encoding.

Fully connected SRNs. Fully connected SRNs use only linear layers with activations between them to map an input coordinate to output value, making them slow to train and perform inference on. SIREN [22]

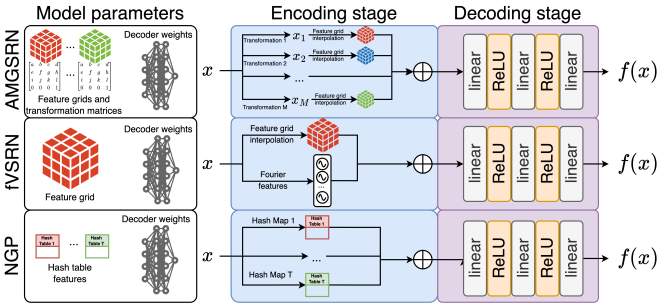

Fig. 2: Comparison of our AMGSRN with other state-of-the-art models fVSRN [26] and NGP [18]. The $\oplus$ operator represents concatenation.

is an example of an fully connected SRN which uses Fourier features [24] to encode input coordinates to a higher dimension before going through a fully connected network with sinusoidal activation functions. NeRF [16] is a fully connected SRN for modelling a radiance field, and trains using a set of input images taken from different points in the real-world or 3D modelling scene. NeRF models take a coordinate and viewing direction as input and return the color and volume density for that spatial location. After training, novel viewpoints can be rendered by volume rendering using the model. AutoInt [12] requires far less model inferences during neural volume rendering compared to NeRF by exploiting the fact that the gradient of a SIREN network is another SIREN network that shares the same weights as the original network. Then the "gradient" network can be trained as a NeRF, resulting in the original "integral" network learning the integration of colors and density through rays in space. Lu et al. [14] use a SIREN-based architecture with residual connections to model 3D volumetric data in a compressed format. Han and Wang [5] propose CoordNet, a single implicit model based on the residual siren architecture of Lu et al. [14] that is used for spatiotemporal super resolution and novel view synthesis.

Grid-based encoding SRNs. In contrast to fully connected SRNs, grid-based encoding SRNs move a large majority of the network parameters to an efficient encoding scheme that transforms the input coordinates to a high-dimensional feature space. SRNs with grid-based encoding schemes train and infer faster than their fully connected counterparts due to the efficient encoding and limited fully connected operations. Weiss et al. [26] create fVSRN, an SRN that models 3D scientific data using a feature-grid encoder which places a low-resolution feature grid over the data domain and interpolates within the feature grid to obtain a feature vector for decoding. Yu et al. [28] accelerate a pre-trained NeRF model with spherical harmonics and an octree data structure for empty-space skipping, enabling realtime rendering of NeRF models. Liu et al. [13] propose neural sparse voxel fields, which

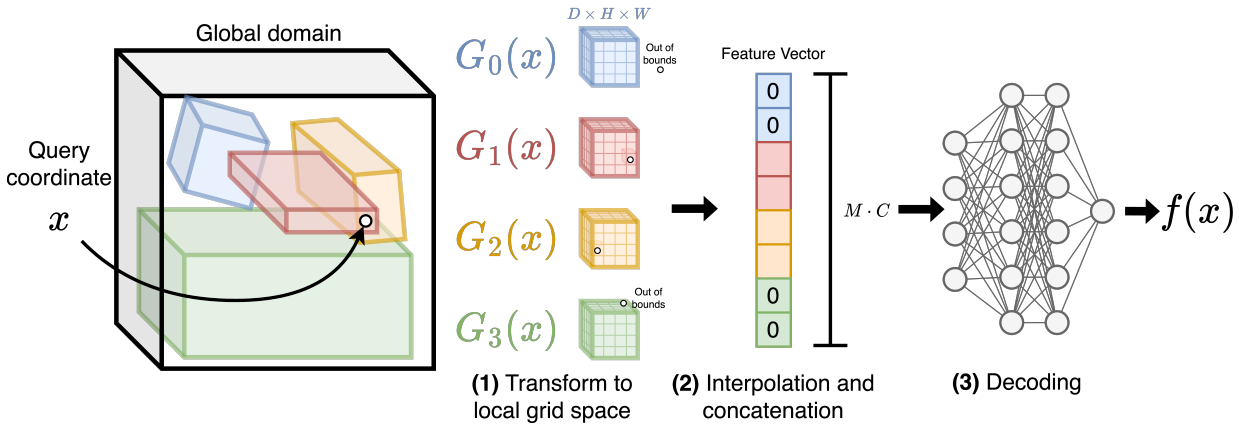

Fig. 3: An overview of the AMGSRN architecture. In **(1)**, a query coordinate is transformed into each of the M grid's local coordinate systems learned by the transformation matrices, where grid local extents are assumed to be $[-1,1]^3$. **(2)**, the local coordinates are used to trilinearly interpolate within each feature grid (of resolution $D \times H \times W$) to obtain the corresponding feature for each grid. If the coordinate is out of bounds, zeros are returned instead. The resulting $M \cdot C$ features are concatenated into a feature vector. Steps (1) and (2) are considered the encoding process. In step **(3)**, the feature vector is decoded in a shallow MLP for the final output value $f(x)$.

uses a coarse 3D grid of voxels that are pruned and split at checkpoints during training to reduce the training time of NeRF with improved accuracy. Takikawa et al. [23] propose NGLOD, which learns an octree data structure for realtime rendering of an SRN fitting a signed-distance function. Martel et al. [15] create an adaptive coordinate network that uses quadtrees/octrees during training, updated by solving an integer linear programming optimization problem every set number of iterations. Chen et al. [3] use tensor (de)composition to reduce the space complexity of 3D feature grids with their VM decomposition of a tensor. Muller et al. [18] introduce hash grid encoding for neural graphics primitives (NGP), which uses random hashing to map input coordinates to features in a hash table at multiple levels of detail. Wu et al. [27] use the hash grid architecture [18] to model scientific data up to 1TB in size.

Existing adaptive models such as ACORN [15], NSVF [13], and NGLOD [23] are expensive to train, store, and require ad-hoc steps for pruning/refining the tree data structure. Additionally, the tree-based adaptivity only helps for scenes with static surfaces. Volumetric scientific data do not generally have static surfaces, since different surfaces arise from visualizing different isovalues. Thus, the space-skipping and tree structures used by other methods ignore relevant regions of the scientific data. Our approach's adaptive feature grids do not require surfaces to refine to, and is driven only by reconstruction error. AMGSRN is conceptually an adaptive version of fVSRN that splits the single highly-parameterized feature grid into many less-parameterized feature grids, where each feature grid now has the ability to adjust its transformation within the domain to focus on high-error regions. A visual comparison of our model and two state-of-the-art models compared with in this paper, fVSRN and NGP, is provided in Figure 2.

3 Overview

Our approach is composed of three components: (1) a novel SRN architecture called AMGSRN, (2) an ensemble training and inference method to train on large scale data, (3) an open-source neural volume renderer for visualizing the trained models. In section 4, we detail the AMGSRN architecture, a SRN model that adaptively learns where multiple feature grids should be spatially located while training. In section 5, we explain the ensemble framework, which dissects a volume into a 3D grid of networks that are each trained on their own local region. Lastly, we discuss the volume rendering application in section 6.

4 Adaptive Multi-Grid Scene Representation Network

Recent state-of-the-art SRNs have found that using explicitly defined feature grids within the model reduces training and inference times [3, 18, 26]. In these feature grid-based models, a feature grid is interpolated at an input spatial coordinate to retrieve a feature vector, which is fed through a shallow multi-layer perceptron (MLP) to obtain the final output value. Tree structures have been added to SRN models to support adaptive allocation of network resources for specific regions [13, 15, 28], but the approaches are ad-hoc, require hours to days to train, and have significant training/storage overhead to manage the tree structure. Additionally, the cost of the tree structures scale exponentially as the number of dimensions or tree depth increases.

Our method improves adaptivity within SRNs with a novel adaptive multi-grid scene representation network (AMGSRN), depicted in Figure 3. Instead of a tree structure, AMGSRN uses a set of multiple spatially adaptive feature grids (shown in Figure 1), each described by a learnable transformation matrix, for coordinate encoding before being decoded by a shallow MLP, as described in section 4.1. Since a basic reconstruction loss is not enough to learn the transformation matrices properly, section 4.2 describes our method to learn grid positions with a feature density-based loss function. Finally, we discuss the network training in section 4.3. Our model gives strong reconstruction quality even with few model parameters thanks to the ability to transform the feature grids to fit the complexities in the data. Since our model does not rely on a tree structure, inference remains quick and storage costs remain low.

4.1 AMGSRN architecture

An AMGSRN is a function $f(x)$ that maps input spatial coordinates $x \in [-1,1]^3$ to the scalar value at that location in space. The model architecture is composed of an encoder e and decoder d such that $f(x) = d(e(x))$, depicted in Figure 2 and Figure 3 Our encoder, described in section 4.1.1, is our adaptive multi-grid encoder, and our decoder, described in section 4.1.2, is a small fully-connected network that decodes the encoded feature vector to an output scalar value.

4.1.1 Adaptive multi-grid encoder

The encoder in AMGSRN contains M learnable feature grids of resolution $D \times H \times W$ with C channels, represented as a tensor F with shape $[M,C,D,H,W]$.

Transformation to local space. In our model, the spatial extents of the i-th feature grid is defined by a 4×4 transformation matrix G_i, which transforms global-space coordinates to local-space coordinates according to the following:

$$\begin{bmatrix} p_x^l \\ p_y^l \\ p_z^l \\ 1 \end{bmatrix} = \begin{bmatrix} G_{i,0,0} & G_{i,0,1} & G_{i,0,2} & G_{i,0,3} \\ G_{i,1,0} & G_{i,1,1} & G_{i,1,2} & G_{i,1,3} \\ G_{i,2,0} & G_{i,2,1} & G_{i,2,2} & G_{i,2,3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x^g \\ p_y^g \\ p_z^g \\ 1 \end{bmatrix}, \quad (1)$$

where $G_{i,r,c}$ is the r-th row and c-th column of the transformation matrix, p_x^g, p_y^g, p_z^g is the global coordinate, and p_x^l, p_y^l, p_z^l is the resulting local coordinate. Conceptually, the first 3 rows and columns of G_i are responsible for scale, rotation, and shearing of the grid, while the 4th column is responsible for translation. The grids local extents are assumed to be $[-1,1]^3$, and global coordinates for the 8 corners of a grid can be determined by calculating the inverse of G_i and transforming the coordinates of the local extents to global space. For shorthand, we consider $G_i(x_g)$ to be a function mapping a 3D global coordinate x_g to the coordinate in grid i's local space according to Equation 1.

Encoding. To encode an input global coordinate, the coordinate is transformed into each grid's local coordinate system via the defining transformation matrices. Then, each 3D local coordinate x_l is used to perform trilinear interpolation within each feature grid:

$$e_i(x_l) = \begin{cases} \texttt{interp}(F_i, x_l) & \text{if } -1 \leq x_l \leq 1 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where F_i is the feature grid for the i-th grid and $\texttt{interp}(F_i, x_l)$ performs trilinear interpolation within F_i at point x_l assuming the grid has vertices corner-aligned at the extents of $[-1,1]^3$. When the input coordinate is not within the feature grid, the feature returned is 0 for each of the expected C channels. The encoding step for a single grid as described in Equation 2 is performed for each grid, with results concatenated together. From global space, the encoding is calculated as:

$$e(x_g) = e_0(G_0(x_g)) \oplus e_1(G_1(x_g)) \oplus \ldots \oplus e_{M-1}(G_{M-1}(x_g)) \tag{3}$$

where $\oplus$ is the concatenation operator. The result is a feature vector $y = e(x_g)$ with $C \cdot M$ features, where C is the number of channels and M is the number of grids.

4.1.2 Decoding

Our decoder d is a shallow 2-layer MLP m with no bias terms used and 64 neurons per layer, which is lightweight and efficient. After each layer besides the last layer, ReLU activation is used as a nonlinearity. We use the Tiny-CUDA-NN package [17] for tensor-core accelerated decoding, which reduces training time by about 10% compared to a pure PyTorch decoder.

Scaling. Unlike other data formats that SRNs represent such as images, radiance fields, and distance fields, scientific data values are generally not bounded, posing a challenge for the decoder to accurately represent the unknown data range. We tackle this with a preprocessing step before training that identifies the minimum and maximum values for the volume the network will represent and saving those along with the network. Then, the decoding is calculated as $d(y) = m(y) \cdot (\texttt{max} - \texttt{min}) + \texttt{min}$ where m is the MLP in the decoder, $\texttt{min}$ is the saved minimum value, and $\texttt{max}$ is the saved maximum value. This formulation allows us to avoid numerical issues by scaling the network output, which results in the MLP learning to output values between 0 and 1. More specific scaling for specific data types, such as z-score-, log-, and exponential-scaling, is not tested in this paper and is left to future work.

4.2 Learning feature grid transformation matrices

The adaptivity of our model stems from the learnable transformations for each feature grid in the encoder. The goal during training is to localize feature grids on/around regions that have high error. We focus more network parameters to those spatial regions with high error in order to improve reconstruction quality in that region.

While the idea is intuitive, naively attempting to learn the transformation matrices defining each grid is not possible using only a reconstruction loss (such as L1 or MSE). This is a direct consequence of our encoding (Equation 2), which returns a 0 feature if a query point is outside of a grid. This means the only gradients that will update a transformation matrix are from points that reside within that grid, so high error regions outside of a feature grid will not "pull" the feature grid toward it as desired.

To properly train the feature grid's transformation matrices, we introduce a feature density loss, which is a quantifiable and differentiable metric for the difference between the current feature density ρ, and a derived *target* feature density ρ^*. We use the term *feature density* to describe the average number of features (from the feature grids) that exist per unit volume at some point in space. In this section, we will discuss how we calculate the current feature density in a differentiable manner, how we derive a *target feature density* that warps the current feature density to increase where the error is relatively higher and decrease where error is relatively lower, and a loss function to quantify the error between the current and target feature density which is ultimately used to update the transformation matrices during training.

Feature density. The simplest way to represent feature density is by the number of grids that overlap a spatial position x. The feature density using this formulation at a global location $\rho(x)$ can be described by the following equation:

$$\rho(x) = \sum_{i=0}^{M-1} \begin{cases} 1 & \text{if } -1 \leq G_i(x) \leq 1 \\ 0, & \text{otherwise} \end{cases}, \tag{4}$$

where $G_i(x)$ is the function that transforms global coordinate x into feature grid i's local coordinate frame. The if statement is true when it holds for each of the three dimensions of $G_i(x)$. However, this approach does not accurately represent the *density* of features at that point. Each feature grid has the same resolution, but a feature grid with small extents will have more dense features than a feature grid with near global extents. Instead, a better feature grid formulation is:

$$\rho(x) = \sum_{i=0}^{M-1} \begin{cases} \det(G_{i,0:3,0:3}) & \text{if } -1 \leq G_i(x) \leq 1 \\ 0, & \text{otherwise} \end{cases}, \tag{5}$$

where $G_{i,0:3,0:3}$ is the top-left 3×3 matrix of G_i representing scale, shear, and rotation for the grid, and $\det(\cdot)$ is the determinant of a matrix. As the grid becomes more dense (smaller in the global domain), the determinant above increases, representing the true feature density of the grid.

Though an accurate description of the feature density of the encoder, this formulation suffers from a large drawback in that there is no gradient to change the transformation matrices G when a point is outside of a grid, as the gradient of ρ will be 0 everywhere. This is a critical component for learning the transformation matrices in our method, and without it, the grid's scales, translations, etc., cannot update based on the error of points outside of their local domain.

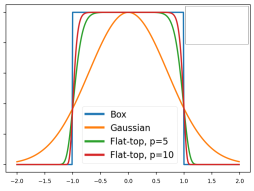

Fig. 4: Visualizing how flat-top gaussians (defined in Equation 6) with increasing flat-top power p generates a curve that is closer to a box, useful for approximating feature density in a differentiable manner.

Therefore, we use a gaussian approximation of the feature density, which is differentiable everywhere. Specifically, we use a class of gaussian functions called *flat-top* gaussians or *super-gaussians*. A flat-top gaussian is the same as the normal gaussian equation with a p term in the exponent:

$$g(x,p) = A \exp\left(-\left(\frac{(x-\mu)^{2p}}{2\sigma^2}\right)\right), \tag{6}$$

where A is a normalizing coefficient, x is the location, μ is the center of the distribution, σ is the standard deviation of the distribution, and p is the strength of the flat-top. As p increases, the gaussian shape become more box-shaped, depicted in Figure 4.

Using the flat-top approximation for feature density box function, we give the final form of $\rho(x)$ as:

$$\rho(x) = \sum_{i=0}^{M-1} \left(\det\left(G_{i,0:3,0:3}\right) \cdot \exp\left(-\left(\sum_{d=0}^{2} \left(G_i(x)_d\right)^{2p}\right)\right)\right), \tag{7}$$

where $G_i(x)_d$ is the transformed coordinate in the x-, y-, and z-axis for $d = 0, 1, 2$. Since the center of the local coordinate space is 0 and the

standard deviation is 1, the gaussian equation does not require the μ or σ terms in the transformed space. This formulation is differentiable everywhere, facilitating proper training. Experimentally, we find $p = 10$ to be strong enough to match the box shape of the feature grid without being too strong so as to have exploding gradients on the ramps to and from the flat-top.

Target feature density. With a way to calculate our current feature density at an arbitrary point, we now require a target feature density to steer toward during training. Our goal is to increase $\rho(x)$ where the error is relatively high and decrease $\rho(x)$ where error is relatively low. We formulate a target feature density for a coordinate $\rho^*(x)$ according to the following:

$$\rho^*(x) = \exp\left(\frac{\overline{h}}{h(x)+\varepsilon}\log\left(\rho_{\text{scaled}}(x)+\varepsilon\right)\right) \tag{8}$$

where $h(x)$ is the model's error at location x, $\overline{h}$ is the average model error (in practice, the average error over a batch), $\rho_{\text{scaled}}(x)$ is $\rho(x)$ divided by its sum over the set (to rescale the density between 0 and 1), and ε is a small number to avoid dividing by zero and other numerical issues. In language, Equation 8 will logarithmically scale the current (scaled) feature density according to the *relative error* at each position. If the error is exactly the average, then the target feature density is unchanged with respect to ρ_{scaled}. This logarithmic scaling assures that more importance is placed increasing feature density in regions with very large relative error, and decreasing feature density in regions with very small relative error.

Feature density loss function. To quantify the difference between the current feature density and the target feature density, we view this as a distribution-matching problem. We calculate our feature density loss over a set of coordinates X as

$$\mathscr{L}_{\text{density}} = \frac{1}{|X|}\sum_{x\in X}\rho_{\text{scaled}}(x)\log\left(\frac{\rho_{\text{scaled}}(x)}{\rho^*(x)}\right), \tag{9}$$

where $|X|$ is the number of coordinates in X. The loss function measures the relative entropy from the target feature density to the current feature density, and is also known as the Kullback–Leibler divergence (KL divergence). This loss function provides gradients that can effectively update the parameters of each feature grid to match the calculated target feature density.

4.3 AMGSRN training

In this section, we cover the losses, training routine, and initialization we use for AMGSRN.

4.3.1 Loss functions.

We use two loss functions while training - a reconstruction loss $\mathscr{L}_{\text{rec}}$ and the density loss $\mathscr{L}_{\text{density}}$ described in Equation 9. The reconstruction loss is the mean-squared error (MSE) between the model output and the ground truth data $\mathscr{L}_{\text{rec}} = \frac{1}{|X|}\sum_{x\in X}\left(f(x)-v(x)\right)^2$, where X is a batch of coordinates, $|X|$ is the number of coordinates, f is the SRN, and $v(x)$ returns the ground truth data value at spatial position x via trilinear interpolation.

4.3.2 Training routine.

Each training step has two parts: (1) update the feature grid and decoder parameters using $\mathscr{L}_{\text{rec}}$, (2) update the transformation matrices using $\mathscr{L}_{\text{density}}$. As long as the model is training, step (1) will happen every iteration. However, performing step (2) during each iteration would slow down training and potentially negatively impact reconstruction quality for reasons discussed in the following paragraphs. Therefore, we use a few techniques to minimize the number of step (2) updates.

Delayed start. Since $\mathscr{L}_{\text{density}}$ is dependent on the error for the current training iteration $\mathscr{L}_{\text{rec}}$, the update to the transformation matrices will be more useful when the model has already seen enough to know where the low- and high-error regions will be. When the model begins training, the error distribution may be random based on the network initialization, grid initialization, and data. Therefore, we delay the training for the transformation matrices until the rest of the network has been briefly trained on the data so as to remove most of the noise from initialization. We find that 500 iterations (with our learning rate and network sizes) is enough to provide clear details about what regions of the volume will be hard to learn, so the transformation matrices are frozen for the first 500 iterations, and are updated according to $\mathscr{L}_{\text{density}}$ after that.

Early stopping. Our formulation presents a challenge for the neural network parameters in our AMGSRN in that if both training step (1) and (2) occur simultaneously, the model is training to chase a moving target. The parameters are updated from $\mathscr{L}_{\text{rec}}$ with the assumption that the feature grids are static, but the grids are moving with each update step (2). Fixing the feature grids as early as possible is essential for the network to fine-tune the feature grid parameters to their location in space, as well as to reduce training time spent updating the transformation matrices when they may have already converged.

While training the transformation matrices, we keep a running track of $\mathscr{L}_{\text{density}}$ each iteration, and set an early stopping flag when the 1000-iteration moving average of $\mathscr{L}_{\text{density}}$ has not reduced by 0.01%. Alternatively, if this early stopping criteria is not met after 80% of the total training steps, we stop updating the transformation matrices in order to allow the other network parameters to learn given fixed feature grid positions.

When the grids do not converge before 80% of the training iterations, a single AMGSRN model takes at most 4 minutes to train for 50k iterations with the hyperparameters experimented with in this paper. Often, the grids converge quickly, reducing training time to as short as 40 seconds.

4.3.3 Initialization

With feature grid positions being crucial to the performance of our model, the initialization of the transformation matrices is a relevant factor in training speed and model accuracy. We initialize our transformation matrices to cover a near global domain with small random shears, rotations, and translations. The diagonals of the matrices (representing the scale of the grid) are sampled from $\mathscr{N}(1, 0.05)$, while the remaining entries in the first three rows of the transformation matrix are sampled from $\mathscr{N}(0, 0.05)$. This initialization assures that each (relevant) entry in the transformation matrix is non-zero so that it may contribute to the final output, guaranteeing gradients can be used to update each entry of the matrix. Additionally, the domains created with this initialization scheme are all nearly equal to the global extents, and so while fixed at the beginning of training (see section 4.3.2), each grid is initially helping learn the global domain, providing a better approximation of which regions will be challenging to learn.

Besides our transformation matrices, we initialize our feature grids from $\mathscr{U}(-0.0001, 0.0001)$, encouraging near-zero initial guesses with small randomness as following Müller et al. [18]. Our decoder network weights are initialized following Glorot and Bengio [4].

5 Ensemble Scene Representation Network

It may not always be feasible to train a single model for some large-scale data for two reasons. From one end, as training data become more complex with higher resolutions, larger neural networks are necessary to obtain adequate reconstruction accuracy, which means longer training times and a large model footprint on disk. At some point with increasing model complexity, a user may run into limitations such as running out of GPU memory during training or unacceptably long training time due to large network sizes. From the other end, the large-scale data we wish to model with an SRN may not fit within GPU memory for efficient query in the training loop. As a back-up, the main CPU memory can be used to host data and support

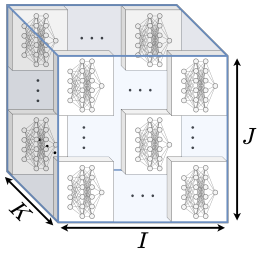

Fig. 5: A depiction of an ensemble of networks, which cover the domain in a grid pattern.

on-demand data transfer to the GPU for training, which can still be costly due to the random sampling of data points during training. If the CPU memory also cannot support hosting the data, then an out-of-core solution is the only option available. Current state-of-the-art out-of-core SRN training for scientific data is highly inefficient [27], taking nearly 2 hours while our method takes less than 7 minutes on the same size data.

Instead of supporting a complicated and inefficient out-of-core sampling method, we take a model-parallel approach to modelling a large-scale volume. Instead of a single network representing the volume, we use an ensemble of models, where each model has learned a separate brick of the volume. The models are arranged in a grid, and do not require communication during training or inference. We discuss the data partitioning in section 5.1, the training procedure in section 5.2, and querying an ensemble model during inference in section 5.3.

5.1 Data partitioning

To partition data into bricks for one network to be trained per brick, we use a grid of networks as shown in Figure 5. Ahead of training, grid resolution I,J,K must be chosen, representing the number of networks for the width, height, and depth of the volume, respectively. We recommend picking an I,J,K such that the resolution of the grid assigned to each network is roughly the same aspect ratio as the feature grids in the model. For instance, if the feature grids are 32^3, choose I,J,K such that each model is assigned a roughly cube-shaped volume.

Ghost cells. In order to mitigate boundary artifacts along adjacent network boundaries during visualization, we experiment with the use of a ghost layer of cells, which are cells that overlap between networks. During data partitioning, this means that the extents of a network (that is part of the ensemble) will get extended by some number of ghost cells along each axis. This solution does not guarantee that seams will be removed, but tends to reduce the effect of them, as the networks should have less of a difference along boundaries if they are learning beyond the actual extent. We discuss a more in-depth solution in our future work, but do not go beyond ghost cells in this paper.

5.2 Ensemble training

To reduce the training time needed for the $I \cdot J \cdot K$ models, we dissect the domain and train the models in parallel across available GPUs. Since large-scale data are often generated by powerful machines, we assume a user is likely training on compute node with multiple GPUs. Our ensemble training routine pre-calculates the extents of each network within the ensemble (including ghost cells) and generates a list of jobs that are assigned to the available GPUs. When a GPU finishes training, the GPU returns to the available GPUs list, where it waits to be issued the next model to train. When a GPU begins training, an AMGSRN model is initialized, and only the data within its assigned extents are loaded from disk and moved to the GPU memory. This reduces I/O overhead and is reasonably efficient with data formats such as NetCDF supporting parallel file access and arbitrary cropping from disk.

Early stopping. Just as we employ early stopping to stop learning the transformation matrices earlier during training to converge faster, we also use early stopping on models within an ensemble while training so the GPU is freed for the next model to train quicker. We use a plateau learning rate scheduler that detects when $\mathscr{L}_{\text{rec}}$ has not decreased by 0.01% over 500 iterations. When this is triggered, the learning rate of the model's parameters is reduced by a factor of 10. When this is triggered 3 times, we finish training the model. This can reduce training times dramatically in regions where there may be very sparse data (down to 20 seconds per model in our experiments).

5.3 Ensemble inference

After training (presumably on a remote server), the neural networks can be moved to a local workstation for visualization. Once on a local machine, all models are loaded into GPU memory. Since each network in the ensemble represents a certain spatial extent, each global query location needs to be mapped to the correct model for inference, so we develop a spatial hashing function for efficient inference.

Spatial hashing function. To efficiently determine what model a query location belongs to, we implement a spatial hashing function to hash input 3D global coordinates to the correct index in the array of models. We load models into an array in x-,y-,z-dimension order. With this order in mind, we can hash input global coordinates directly to array index. Assuming a global spatial coordinate p is in the domain $[-1,1]^3$, we calculate the grid index for p with $i = \left\lfloor I\frac{p_x+1}{2} \right\rfloor, j = \left\lfloor J\frac{p_y+1}{2} \right\rfloor, k = \left\lfloor K\frac{p_z+1}{2} \right\rfloor$, where $\lfloor\cdot\rfloor$ is the integer floor operation. If any dimension of the coordinate has exactly 1.0 for an axis, then the result is an out of bound location, so these values need to be clamped such that $i < I, j < J, k < K$. Since the networks are stored in a list, the 3D i,j,k index for the network is flattened in C-order, giving the final hash as $i + Ij + IJk$. With the correct network determined to perform inference with, the global coordinate can be scaled to the networks local domain for proper inference.

Our implementation loops over all ensemble networks to do evaluations for each, but a significant speedup could be achieved with further engineering effort, as shown by Reiser et al. [21], who develop custom CUDA code for inference within their ensemble of networks for radiance fields. We leave this to future work, but expect the inference time of large ensemble models to be much quicker when ensemble inference code is written in custom CUDA code that parallelizes network evaluation.

6 Neural volume rendering

We develop a Python/PyTorch [19]-based neural volume renderer (included with our code on GitHub) that supports our own AMGSRN as well as state-of-the-art models fVSRN [26] and neural graphics primitives (NGP) using a hash-grid encoding [18], on top of raw data volume rendering. Any PyTorch model that can support mapping 3D coordinates to an output density can also be plugged in with minimal reconfiguration. Our renderer supports arbitrary transfer function and view direction, and uses progressive rendering to support 60+ fps rendering for immediate feedback while panning, rotating, and zooming. We support transfer functions exported from ParaView directly to our renderer, and interacting with the scene follows the typical 3D viewer paradigm of clicking and dragging rotating around the scene, the scrollwheel zooms, and middle-mouse clicking and dragging pans the camera. We use 3D rendering helper functions from the nerfacc Python package [8], which provides CUDA-accelerated raymarching and compositing.

Optimizations. Since rendering a single 1080p or larger image with a large number of samples per pixel (spp) can take up to a few seconds depending on the scene, transfer function, and network complexity, we implement a progressive rendering scheme that renders the image in a checkerboard pattern, evaluating the pixels in order of an image hierarchy. After each progressive rendering pass, the in-progress image displayed on the screen is an bilinearly-upascaled version of the current finest-available image from the image hierarchy masked with the fully evaluated pixels. We also offer a trade-off between interactivity and full render time with a user choice for batch size, with large batch sizes slowing down interactivity but speeding up full-frame render times, and vice versa for small batch sizes. Our progressive rendering keeps GPU memory minimized and quickly generates good approximations of the final image while offering interactive framerates (30-400 fps).

7 Experiments

We perform experiments to test the training and reconstruction metrics of our proposed AMGSRN model in section 7.1. Our ensemble training approach is evaluated in section 7.2. Lastly, we discuss our renderer for visualizing trained models in section 7.3. Our datasets and brief descriptions are shown in Table 1.

Hyperparameters and evaluation hardware. To save page space, model hyperparameters for each size SRN (small/medium/large/ensemble) are available in GitHub in the "BatchRunSettings" folder, as there are many hyperparameters to enumerate for each architecture. For our model specifically, the number of grids we use is either 16 or 32, the grid resolution varies from 16^3 to

Table 1: Datasets, their size on disk (in raw, single-precision floating point representation), and their descriptions, as well as quantitative results for SRN models trained to represent each dataset. Training type is either single (for single model) or an ensemble grid resolution for our ensemble training approach. For ensemble model, average training time per model is reported since models are trained in parallel. Best performing models in one size category are bold and underlined. Peak signal-to-noise ratio (PSNR) is given in the data space.

Dataset	Model storage size	Model	Training type	PSNR (dB) ↑	Training time
Plume 512×128^2 32 MB Velocity magnitude of particles in a solar plume simulation. Somewhat sparse.	Small - 280 KB	fVSRN	single	44.775	1m 12s
		NGP	single	46.713	1m 32s
		AMGSRN	single	**48.954**	52s
	Medium - 4 MB	fVSRN	single	52.580	1m 9s
		NGP	single	50.847	1m 7s
		AMGSRN	single	**56.327**	1m 20s
	Large - 64 MB	fVSRN	single	55.064	2m 21s
		NGP	single	53.282	2m 10s
		AMGSRN	single	58.134	2m 25s
		AMGSRN	$4 \times 1 \times 1$	**58.363**	56s
Nyx [1] 256^3 64 MB Log density of dark matter in cosmological simulation. Somewhat sparse.	Small - 280 KB	fVSRN	single	29.203	1m 11s
		NGP	single	28.391	1m 33s
		AMGSRN	single	**29.424**	51s
	Medium - 4 MB	fVSRN	single	37.761	1m 14s
		NGP	single	35.142	1m 8s
		AMGSRN	single	**37.807**	1m 34s
	Large - 64 MB	fVSRN	single	43.889	2m 21s
		NGP	single	42.750	2m 10s
		AMGSRN	single	44.075	3m 7s
		AMGSRN	$2 \times 2 \times 2$	**46.118**	1m 35s
Supernova 432^3 403 MB Entropy in supernova simulation. Somewhat sparse.	Small - 280 KB	fVSRN	single	39.497	1m 11s
		NGP	single	41.523	1m 29s
		AMGSRN	single	**41.891**	1m 16s
	Medium - 4 MB	fVSRN	single	43.513	1m 11s
		NGP	single	44.811	1m 8s
		AMGSRN	single	**46.831**	1m 19s
	Large - 64 MB	fVSRN	single	47.480	2m 22s
		NGP	single	47.806	2m 32s
		AMGSRN	single	49.880	2m 41s
		AMGSRN	$2 \times 2 \times 2$	**50.644**	35s
Asteroid 1000^3 3.73 GB Volume fraction of asteroid plus water fields from simulation of an asteroid hitting the ocean. SciVis contest 2018 data. Very sparse.	Small - 280 KB	fVSRN	single	31.283	1m 12s
		NGP	single	**33.713**	1m 32s
		AMGSRN	single	30.130	1m 43s
	Medium - 4 MB	fVSRN	single	33.934	1m 10s
		NGP	single	**39.012**	1m 6s
		AMGSRN	single	35.799	1m 49s
	Large - 64 MB	fVSRN	single	38.823	2m 22s
		NGP	single	42.928	2m 24s
		AMGSRN	single	39.455	2m 49s
		AMGSRN	$4 \times 4 \times 4$	**43.188**	45s
Isotropic [9, 20] 1024^3 4 GB Velocity magnitude field of forced isotropic turbulence. Not sparse.	Small - 280 KB	fVSRN	single	27.427	1m 11s
		NGP	single	27.007	1m 32s
		AMGSRN	single	**27.622**	55s
	Medium - 4 MB	fVSRN	single	31.781	1m 13s
		NGP	single	30.786	1m 8s
		AMGSRN	single	**31.825**	1m 14s
	Large - 64 MB	fVSRN	single	38.464	2m 21s
		NGP	single	37.198	2m 10s
		AMGSRN	single	38.120	3m 23s
		AMGSRN	$4 \times 4 \times 4$	**41.086**	57s
Rotstrat [9, 20] 4096^3 250 GB Velocity magnitude of rotating stratified turbulence, t=4. Not sparse.	Small - 55 MB	AMGSRN	$3 \times 3 \times 3$	41.322	1m 3s
	Medium - 183 MB	AMGSRN	$3 \times 3 \times 3$	44.294	1m 15s
	Large - 864 MB	AMGSRN	$3 \times 3 \times 3$	49.074	1m 58s
Channel [7, 9, 20] $7680 \times 1568 \times 10240$ 450GB Channel flow at Reynolds number $\approx$ 5200, t=10. Not sparse.	97 MB	AMGSRN	$4 \times 1 \times 6$	40.055	35s

64^3, and the number of features per feature grid vertex is either 1 or 2. Grid resolution has the largest impact on both network performance and network storage size.

Training for all models, compression, and throughput tests are run on NVidia DGX system with 8 NVidia 40GB A100 GPUs, a 64-core AMD EPYC 7742 processor, and 1TB of system memory connected to a Lustre-based file system. Rendering evaluation is performed on a Windows 11 workstation machine (within Windows subsystem for Linux) with a NVidia 2080 Ti, a 12-core AMD Ryzen 9 3900X, and 32GB of system memory.

Baseline SRNs. Neural graphics primitives (NGP) [18] uses a hierarchical hash-grid encoding scheme that uses a random hash function to map spatial coordinates to hash table entries which hold feature values. The encoding scheme allows hash collisions and copes with them by letting the gradients average, which puts more emphasis on the higher-error regions. Therefore, NGP benefits from very high sparsity, where hash collisions are frequent, but a majority of the space is empty anyway. The second model we compare with is fast volumetric scene representation network (fVSRN) [26], which is a standard feature-grid based SRN. The feature grid is dense and lines up with the extents of the data. For both comparison models, we use initialization and training routines as suggested by the publication they were introduced in.

Training. Python 3.9 with PyTorch [19] are used for training all networks. All models use the same fully-connected decoder from Tiny-CUDA-NN [17] with 2 layers of 64 neurons, accelerated with custom CUDA code. The custom CUDA implementation of NGP's hash grid encoding [17] is used, but a 32-bit PyTorch implementation is used for fVSRN's encoding as well as our own multi-grid encoding. All networks are trained for 50,000 iterations or until an early stopping criteria, as discussed in section 4.3.2. For all models, the Adam [6] optimizer is used with $\beta_1 = 0.9, \beta_2 = 0.99$ with a learning rate of 0.01. Our transformation matrices use a learning rate of 0.001. Each iteration, a batch of 100,000 coordinates are sampled uniformly from $[-1, 1]^3$, and the ground truth values are trilinearly interpolated from the volume.

7.1 Scene representation network comparison

We compare our model against two state-of-the-art scene representation networks, neural graphics primitives (NGP) [18] and fVSRN [26], for data reconstruction quality and training times at fixed model sizes.

Single network performance. Across results from each dataset and network size shown in Table 1, AMGSRN is first for reconstruction peak signal-to-noise ration (PSNR - higher is better) in all but 4 settings. AMGSRN's adaptivity seems most dominant in somewhat sparse data, such as plume, Nyx, and supernova, where it outperforms other models by 2+ dB at the largest model size. Our model underperforms against NGP only for the asteroid dataset because of the high degree of sparsity, which NGP excels at. Examples of learned grid positions are shown in Figure 1, and verifies that the grids learn to migrate to regions with complex features regardless of dataset.

Training times across models are similar, with our model being the least performant compared to NGP and fVSRN. This is a consequence of using a second loss function for updating our transformation matrices, and more matrix operations per model forward pass due to the transformations required before feature grid interpolation. Our multi-grid encoding is a pure PyTorch implementation, but custom CUDA code like that used by NGP, could improve model performance.

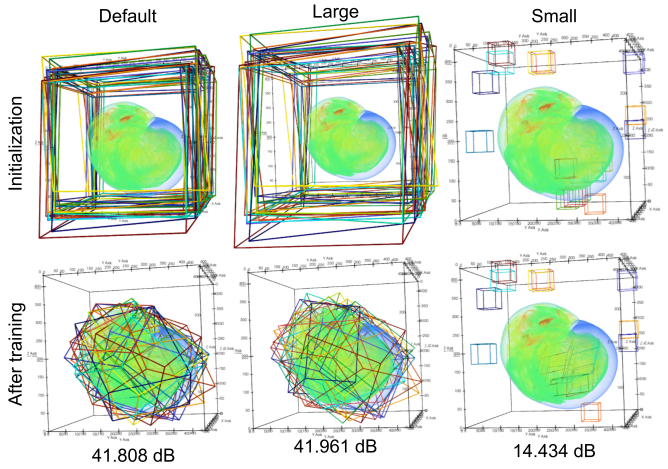

Fig. 6: Examples of 3 grid initializations and the effect on final grid locations and data reconstruction PSNR. Underlying data volume rendered depicts ground truth data for clarity of grid locations with respect to the raw data.

Importance of grid initialization and grid visualization. As feature grid location in the volume is the only means of encoding coordinates to a high-dimensional feature space for learning, the initialization of the grids may affect learning quality. We experiment with grid initialization with a small sized model trained on the supernova dataset. We use three initialization techniques for the grids: (1) our recommended initialization as described in section 4.3.3, (2) our recommended initialization, with the grid scales increased by 20% so as to cover extra empty space outside of the volume, (3) initializing grids at a small scale, randomly distributed within the domain. The results of the ex-

periment are shown in Figure 6. We can see that the final grids and reconstruction quality of the first two initialization schemes are similar, but initializing the grids to random spots with a small scale performed much worse. The grids that do not cover any data sitting in empty space have near-zero gradient for changing any of the transformation matrix parameters. A larger learning rate or a less powerful flat-top gaussian can make up for this for some cases, but that may cause larger-scale grids to see unstable updates to their parameters. For best results, we recommend our default initialization of global-scale grids with small random perturbations, but we believe there may be room for improved performance with different differentiable approximations of grid density for smoother learning. We recommend readers view our supplemental material for videos of feature grids during training on the smaller datasets experimented with, as it gives a strong intuition for what the network is learning and how it is learning it.

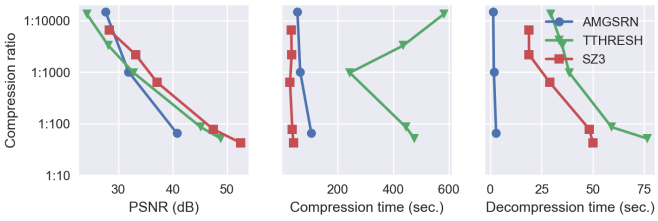

Fig. 7: Comparison of AMGSRN with TTHRESH and SZ for compressing the isotropic volume (1024^3). Our "compression" time is the time it takes to train the model, and our decompression time is the time it takes to query each point in the volume.

Compression. Though data reduction is not the only intended use case for our approach, nor is it something we design for specifically (such as with network weight quantization for further data reduction [14]), we believe it is useful to the community to compare the compression ability of state-of-the-art SRNs with state-of-the-art compressors. We compare compression results from TTHRESH [2] and SZ3 [10, 11, 29] with our approach in Figure 7. We do not compare to a rendering-focused compressor such as cudaCompress [25] or a bricked version of TTHRESH [2], as this has been done by recent work [26] showing that its decoding is significantly slower, compression rates smaller, memory use is higher, and image quality is worse than state-of-the-art SNRs.

As shown in Figure 7, our approach only provides a higher compression ratio than TTHRESH or SZ3 at a 10000× data reduction rate with a reconstruction quality of under 30 dB PSNR, and is otherwise less compressive than state-of-the-art compressors. The throughput (decompression) from AMGSRN is much quicker regardless of compression level, decoding on the order of 500 million points per second, but keep in mind that AMGSRN is decoding on the GPU whereas SZ3 and TTHRESH are on CPU. The largest difference between AMGSRN and these compressors is that our approach can efficiently perform arbitrary point evaluations, whereas SZ3 and TTHRESH both require decompressing to the original data size before trilinear interpolation.

We also test compression on our two large datasets, rotstrat and channel. The tested compressors take significantly longer to compress our large-scale data, or fail to do so at all. In fact, TTHRESH and SZ3 both run out of memory on our machine with 1TB of memory when trying to compress the channel dataset (450GB), and TTHRESH also runs out of memory when trying to compress the rotstrat dataset (250GB). SZ3 successfully compresses the 250GB rotstrat data in 38 minutes, resulting in a compression ratio of 509× and a reconstruction quality of 20.60 dB PSNR after another 14 minutes of decompression. Both our small and medium sized (ensemble) models for rotstrat (see Table 1) achieve higher PSNRs (41.32 dB and 44.29 dB) with higher compression rates (4791× and 1432×) while only needing to train for 34 minutes if all models are trained sequentially on 1 GPU, or 5 minutes in parallel on 8 GPUs. Additionally, our decoding takes 3 minutes for the whole 4096^3 volume.

Table 2: Single frame rendering results for a 1024^2 image with a ray step size of 1 voxel and no empty space skipping. Each model is the "large" size, 64 MB to store, trained with the supernova data. A batch size of 2^{23} is used during forward passes for each network. The reference metrics are from our renderer with the raw data (432^3).

	Reference	fVSRN	NGP	AMGSRN	AMGSRN ensemble (8)
Render time	0.70s	3.23s	**0.98s**	1.77s	1.43s
Memory use	0.74 GB	3.09 GB	**0.68 GB**	5.56 GB	0.90 GB
PSNR ↑	–	30.43 dB	29.31 dB	32.72 dB	**33.52 dB**
SSIM ↑	–	0.877	0.875	0.916	**0.929**

7.2 Ensemble network evaluation

We evaluate our ensemble training approach on each dataset, and show results in Table 1. Even though the smaller datasets do not require an ensemble of models to train without memory limitations, the ensemble model outperforms a single model at the same storage size for each dataset. This is expected, as each of the ensemble models will have their own decoder and feature grid fitting that may create and advantage over the single large model. The downside of ensemble models is that training may take longer if you only have 1 GPU.

For data that cannot fit in a single GPU, such as rotstrat and channel, our ensemble model provides an efficient alternative to an out-of-core training routine. In Table 1, we list the *average* training time per model for the ensemble models. Across the 27 models for the rotstrat data, the training would take just over 27 minutes if done serially with a single GPU. Our training machine had 8 GPUs training in parallel, so the total training time was less than 4 minutes total, excluding data IO, which took roughly 13 seconds per network.

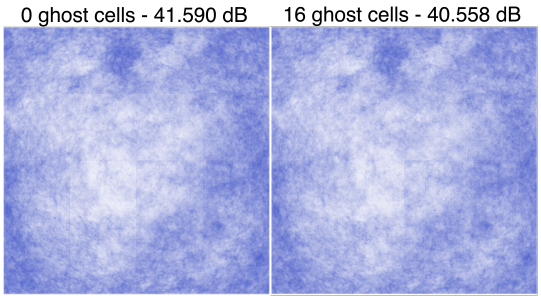

Fig. 8: Orthographic volume render from one axis (to accentuate boundary artifacts) of error volume from identical $4\times4\times4$ ensemble network setups trained with either 0 or 16 ghost cells. Zooming in reveals the boundary artifacts on each in a 4×4 grid.

Effect of ghost cells. By default, our ensemble model has 0 ghost cells, but 1 vertex overlap for corner alignment across models. We show the effect of increasing ghost cells within our ensemble models in Figure 8, which is a volume rendering of the squared error between the raw data and the learned neural representation. The boundary artifacts are visible in both renders, though less prominent in the render from the model with 16 ghost cells. However, a significant PSNR drop of -1 dB is noticed with this increase in ghost cells, attributed to the fact that a single network in the ensemble learning a volume of size $288^3 = (256+2\cdot16)^3$ is a 42% larger volume than the original 256^3 volume learned on in the version without ghost cells. The redundancy of learning a significant portion of the full volume in multiple networks reduces the learning capacity of the ensemble. Since the artifacts are still present with a large number of ghost cells and quality decreases significantly, we recommend using none or few ghost cells. We consider future work for reducing the boundary effect between networks with a network communication scheme during training.

7.3 Neural volume rendering

In this section, we evaluate the single frame performance of neural volume rendering. For a fair comparison, our renderer abstracts the volume point query to used either trilinear interpolation in a volume of raw

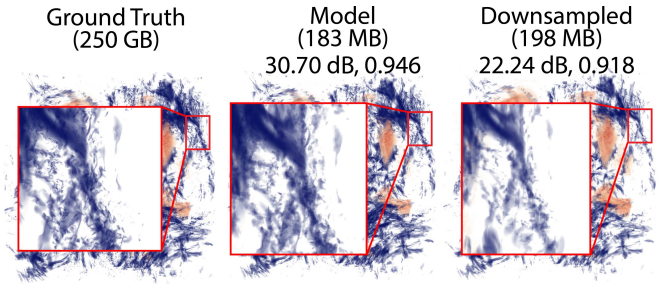

Fig. 9: Volume renders of the original rotstrat data, our "medium" size model representing the data, and a low resolution version of the original data, which has been subsampled by 11 in each direction to have a similar storage footprint as the model. Each image is rendered at 8000^2 with the same visual mapping parameters. Metrics listed are PSNR/SSIM for the entire image (not just zoomed portion).

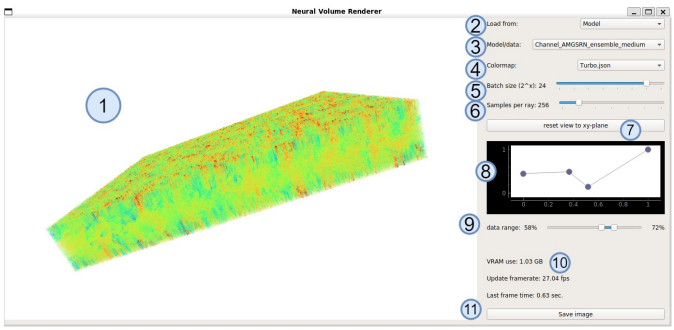

Fig. 10: The interface for our volume renderer with a volume render of an ensemble model trained on the 450GB channel dataset. **(1)** The viewing area for interaction. **(2)** Dropdown to choose to load a model or data (NetCDF). **(3)** Which model/data to load from. **(4)** Colormap to use. **(5)** Batch size when querying the network. Larger reduces total frame time, but requires more GPU memory and slows down interactivity. **(6)** Samples per ray. **(7)** Resets camera view on data. **(8)** Opacity transfer function editor. **(9)** Relative min/max for transfer function. **(10)** Render statistics. **(11)** Save current frame button.

data or inference in a trained SRN. The raymarching and compositing are done with the same code, so the timing and memory use difference is accounted for solely by using a SRN to represent the volume.

Our single frame rendering results are shown in Table 2 where we compare the render results of the large sized (64 MB on disk) models for the supernova dataset. The reference render using the volume is the quickest, but assumes the volume can fit in GPU memory, which is not possible datasets larger than 11 GB on our test machine. NGP, with custom CUDA kernel accelerated code for efficient evaluation on TensorCores, provides the quickest render times of the networks on our 2080Ti, as well as the lowest memory use. Our ensemble network comes in second for render time and memory use to NGP, but has considerably better image reconstruction shown by the higher PSNR and structural similarity index measure (SSIM) [30]. Our approach (single model) requires the most memory in part due to the transformation to local grid spaces; the single model has 64 grids, so the input coordinates need to be transformed into 64 local spaces (in parallel), requiring 64× the memory of the global coordinates for the local space coordinates.

In Figure 9, we compare our trained "medium" sized ensemble model for the 250 GB rotstrat data with a downsampled version of the raw data with a similar storage footprint as the model. We subsample the raw data by 11 in each dimension to create a volume that takes 198 MB to store, which is only slightly larger than our network which takes 183 MB to store. We use our renderer to generate 8000^2 images of each representation of the data with the same visual mapping parameters. The image from our model achieves a significantly higher 30.70 dB PSNR compared with the render from the downsampled data which only achieves 22.24 dB. Additionally, the neural network captures features that are entirely missed by the subsampled data.

Regardless of model chosen, the render times are not at realtime speeds, which is why we develop a technique for progressive rendering in our renderer application as described section 6. We include an image of our rendering application in Figure 10, but do not discuss it in detail here, though the code release contains all information necessary for using both the offline and online renderer. Using our online progressive rendering approach, a single checkerboard and image-hierarchy update allows rendering at speeds between 50-300 frames per second, so the user has an immediate feedback while interacting and changing transfer function or viewpoint. Please see supplemental materials for a video illustrating this effect.

8 Limitations

One limitation of our model is that memory use blows up and training speeds reduce as the number of feature grids increases. This is from the global to local transformation, which needs to allocate memory equal to the memory of the query coordinates times the number of feature grids, and then perform interpolation for each of those coordinates. Further engineering effort could reduce the overhead with custom CUDA code or by limiting the number of grids operated on at once. Regardless, our max use of 64 feature grids has fit the testing data in our evaluation well, and we expect more grids to be redundant for most datasets.

Though our neural renderer is compatible with any PyTorch model coded in Python, the renderer is much slower than what is reported by (unreleased) contemporary work [18], which cites 10-60fps. This could be due to their "macrocell" optimization and empty space skipping, which we do not employ in our renderer. We will continue improving our python renderer with suggestions from the community, and expect our renderer to be useful to others using, developing, and debugging SRNs for scientific data.

9 Conclusion and Future Work

In this work, we present a novel SRN called AMGSRN, a model which excels at representing scientific data due to the adaptivity of the model. In order to train on datasets that cannot fit in-core during training, we propose an ensemble training and inference approach that divides the domain into separate chunks to be learned by multiple networks. For visualization of the trained models on a local CUDA-enabled workstation, we develop a neural volume rendering application that generates volume renderings for any PyTorch-based neural network at interactive framerates using a progressive rendering approach. In our evaluations, we find that our proposed AMGSRN architecture outperforms the reconstruction performance of other state-of-the-art models at similar model sizes in both data space and image space. Our model cannot compress as well as state-of-the-art compressors like TTHRESH, but achieves high compression rate with efficient volume rendering in the compression domain without needed to decompress to the original data's size.

In the future, we think our adaptive feature grids can be augmented in two ways: (1) the feature grids need not be trilinearly interpolated - another encoding scheme, such as a hash grid from NGP, can be used for query within each grid, (2) the feature grids could extend the time dimension, creating 4D learnable feature grids supporting temporal interpolation. Reducing boundary artifacts is another key direction to improve upon to reduce distracting seams in visualizations. On the engineering side, there are many places where custom CUDA code can improve efficiency, such as the multi-grid encoder or ensemble model inference. Our approach could also work with minimal changes to represent non-uniform grid data types, such as unstructured, point-based, or AMR, so long as the data can return a scalar value for an arbitrary point query. With the increasing popularity and usefulness of SRNs, future work to support neural volume rendering in mature open-source software such as ParaView would greatly benefit the visualization community.

References

[1] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. V. Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal*, 765(1):39–53, 2013. 7

[2] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. TTHRESH: Tensor Compression for Multidimensional Visual Data. *IEEE Transactions on Visualization and Computer Graphics*, 26(9):2891–2903, 2020. doi: 10.1109/TVCG.2019.2904063 8

[3] A. Chen, Z. Xu, A. Geiger, J. Yu, and H. Su. TensoRF: Tensorial Radiance Fields. In *Proc. European Conference on Computer Vision*, pp. 333–350, 2022. 1, 3

[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, vol. 9, pp. 249–256, 13–15 May 2010. 5

[5] J. Han and C. Wang. CoordNet: Data Generation and Visualization Generation for Time-Varying Volumes via a Coordinate-Based Neural Network. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–12, 2022. doi: 10.1109/TVCG.2022.3197203 2

[6] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *Proc. 2015 International Conference on Learning Representations*, 2015. 7

[7] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $\mathrm{Re}_\tau = 5200$. *Journal of Fluid Mechanics*, 774:395–415, 2015. 7

[8] R. Li, M. Tancik, and A. Kanazawa. NerfAcc: A General NeRF Acceleration Toolbox, 2022. doi: 10.48550/ARXIV.2210.04847 6

[9] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink. A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, 9, 2008. 7

[10] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pp. 438–447, 2018. doi: 10.1109/BigData.2018.8622520 8

[11] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello. SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors. *IEEE Transactions on Big Data*, 9(2):485–498, 2023. doi: 10.1109/TBDATA.2022.3201176 8

[12] D. B. Lindell, J. N. P. Martel, and G. Wetzstein. AutoInt: Automatic Integration for Fast Neural Volume Rendering. In *Proc. CVPR*, 2021. 2

[13] L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt. Neural sparse voxel fields. *NeurIPS*, 2020. 1, 2, 3

[14] Y. Lu, K. Jiang, J. A. Levine, and M. Berger. Compressive neural representations of volumetric scalar fields. *Computer Graphics Forum*, 40(3):135–146, 2021. doi: 10.1111/cgf.14295 1, 2, 8

[15] J. N. P. Martel, D. B. Lindell, C. Z. Lin, E. R. Chan, M. Monteiro, and G. Wetzstein. Acorn: Adaptive Coordinate Networks for Neural Scene Representation. *ACM Trans. Graph.*, 40(4), 2021. doi: 10.1145/3450626.3459785 1, 2, 3

[16] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *Proc. European Conference on Computer Vision*, 2020. 1, 2

[17] T. Müller. tiny-cuda-nn. https://github.com/NVlabs/tiny-cuda-nn, 4 2021. 4, 7

[18] T. Müller, A. Evans, C. Schied, and A. Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4), jul 2022. doi: 10.1145/3528223.3530127 2, 3, 5, 6, 7, 9

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. 2019. 1, 6, 7

[20] E. Perlman, R. Burns, Y. Li, and C. Meneveau. Data Exploration of Turbulence Simulations using a Database Cluster. In *Proc. ACM/IEEE Conference on Supercomputing*, pp. 1–11, 2007. 7

[21] C. Reiser, S. Peng, Y. Liao, and A. Geiger. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 14315–14325, 2021. 1, 2, 6

[22] V. Sitzmann, J. N. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein. Implicit Neural Representations with Periodic Activation Functions. In *Proc. NeurIPS*, 2020. 1, 2

[23] T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. *arXiv preprint arXiv:2101.10994*, 2021. 1, 3

[24] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *NeurIPS*, 2020. 2

[25] M. Treib, K. Burger, F. Reichl, C. Meneveau, A. Szalay, and R. Westermann. Turbulence Visualization at the Terascale on Desktop PCs. *IEEE transactions on visualization and computer graphics*, 18(12):2169–2177, 2012. 8

[26] S. Weiss, P. Hermüller, and R. Westermann. Fast neural representations for direct volume rendering. *Computer Graphics Forum*, 41(6):196–211, 2022. doi: 10.1111/cgf.14578 1, 2, 3, 6, 7, 8

[27] Q. Wu, D. Bauer, M. J. Doyle, and K.-L. Ma. Instant neural representation for interactive volume rendering, 2022. doi: 10.48550/ARXIV.2207.11620 1, 2, 3, 6

[28] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021. 1, 2, 3

[29] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1643–1654, 2021. doi: 10.1109/ICDE51399.2021.00145 8

[30] 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. 9