AutoInt: Automatic Integration for Fast Neural Volume Rendering

David B. Lindell* Julien N. P. Martel* Gordon Wetzstein
Stanford University
{lindell, jnmartel, gordon.wetzstein}@stanford.edu

Abstract

Numerical integration is a foundational technique in scientific computing and is at the core of many computer vision applications. Among these applications, neural volume rendering has recently been proposed as a new paradigm for view synthesis, achieving photorealistic image quality. However, a fundamental obstacle to making these methods practical is the extreme computational and memory requirements caused by the required volume integrations along the rendered rays during training and inference. Millions of rays, each requiring hundreds of forward passes through a neural network are needed to approximate those integrations with Monte Carlo sampling. Here, we propose automatic integration, a new framework for learning efficient, closed-form solutions to integrals using coordinate-based neural networks. For training, we instantiate the computational graph corresponding to the derivative of the coordinate-based network. The graph is fitted to the signal to integrate. After optimization, we reassemble the graph to obtain a network that represents the antiderivative. By the fundamental theorem of calculus, this enables the calculation of any definite integral in two evaluations of the network. Applying this approach to neural rendering, we improve a tradeoff between rendering speed and image quality: improving render times by greater than 10× with a tradeoff of slightly reduced image quality.

1. Introduction

Image-based rendering and novel view synthesis are fundamental problems in computer vision and graphics (e.g., [5, 56]). The ability to interpolate and extrapolate a sparse set of images depicting a 3D scene has broad applications in entertainment, virtual and augmented reality, and many other applications. Emerging neural rendering techniques have recently enabled photorealistic image quality for these tasks (see Sec. 2).

*Equal contribution.
http://www.computationalimaging.org/publications/automatic-integration/

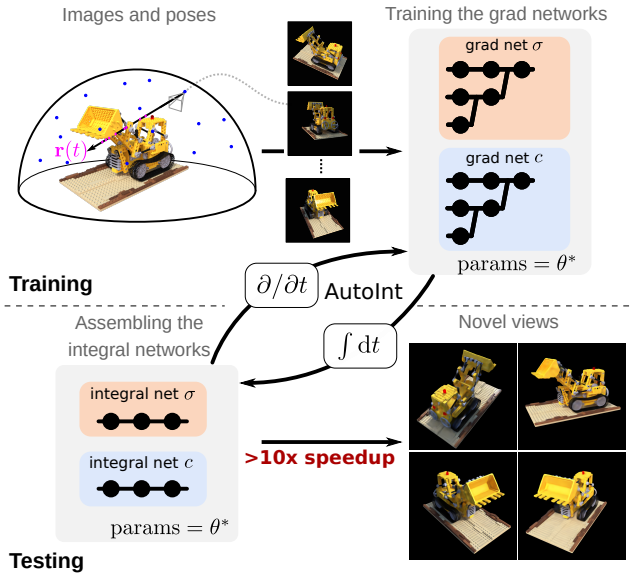

Figure 1. Automatic integration for neural volume rendering. During training, a grad network is optimized to represent multi-view images. At test time, we instantiate a corresponding integral network to rapidly evaluate per-ray integrals through the volume.

Although state-of-the-art neural volume rendering techniques offer unprecedented image quality, they are also extremely slow and memory inefficient [38]. This is a fundamental obstacle to making these methods practical. The primary computational bottleneck for neural volume rendering is the evaluation of integrals along the rendered rays during training and inference required by the volume rendering equation [34]. Approximate integration using Monte Carlo sampling is typically used for this purpose, requiring hundreds of forward passes through the neural network representing the volume for each of the millions of rays that need to be rendered for a single frame. Here, we develop a general and efficient framework for approximate integration. Applied to the specific problem of neural volume rendering, our framework improves a tradeoff between rendering speed and image quality, allowing a greater than 10× speedup in the rendering process, though with some reduction in image quality.

Our integration framework builds on previous work demonstrating that coordinate-based networks (sometimes also referred to as implicit neural representations) can represent signals (e.g., images, audio waveforms, or 3D shapes) and their derivatives. That is, taking the derivative of the coordinate-based network accurately models the derivative of the original signal. This property has recently been shown for coordinate-based networks with periodic activation functions [53], but we show that it also extends to a family of networks with different nonlinear activation functions (Sec 3.4 and supplemental).

We observe that taking the derivative of a coordinate-based network results in a new computational graph, a "grad network", which shares the parameters of the original network. Now, consider that we use as our network a multilayer perceptron (MLP). Taking its derivative results in a grad network which can be trained on a signal that we wish to integrate. By reassembling the grad network parameters back into the original MLP, we construct a neural network that represents the antiderivative of the signal to integrate.

This procedure results in a closed-form solution for the antiderivative, which, by the fundamental theorem of calculus, enables the calculation of any definite integral in two evaluations of the MLP. Inspired by techniques for automatic differentiation (AutoDiff), we call this procedure *automatic integration* or AutoInt. Although the mechanisms of AutoInt and AutoDiff are very different, both approaches enable the calculation of integrals or derivatives in an automated manner that does not rely on traditional numerical techniques, such as sampling or finite differences.

The primary benefit of AutoInt is that it allows evaluating arbitrary definite integrals quickly by querying the network representing the antiderivative. This concept could have important applications across science and engineering; here, we focus on the specific application of neural volume rendering. For this application, efficiently evaluating integrals amounts to accelerating rendering (i.e., inference) times, which is crucial for making these techniques more competitive with traditional real-time graphics pipelines. However, our framework still requires a slow training process to optimize a network for a given set of posed 2D images.

Specifically, our contributions include the following.

- We introduce a framework for automatic integration that learns closed-form integral solutions. To this end, we explore new network architectures and training strategies.
- Using automatic integration, we propose a new model and parameterization for neural volume rendering that is efficient in computation and memory.
- We improve a tradeoff between neural rendering speed and image quality, demonstrating rendering rates that are an order of magnitude faster than previous implementations [38], though with a slight reduction in image quality.

2. Related Work

Neural Rendering. Over the last few years, end-to-end differentiable computer vision pipelines have emerged as a powerful paradigm wherein a differentiable or neural scene representation is optimized via differentiable rendering with posed 2D images (see e.g., [58] for a survey). Neural scene representations often use an explicit 3D proxy geometry, such as multi-plane [15, 37, 64] or multi-sphere [2, 4] images or a voxel grid of features [32, 54]. Explicit neural scene representations can be rendered quickly, but they are fundamentally limited by the large amount of memory they consume and thus may not scale well.

As an alternative, coordinate-based networks, or implicit neural representations, have been proposed as a continuous and memory-efficient approach. Here, the scene is parameterized using neural networks, and 3D awareness is often enforced through inductive biases. The ability to represent details in a scene is limited by the capacity of the network architecture rather than the resolution of a voxel grid, for example. Such representations have been explored for modeling shape parts [16, 17], objects [3, 6, 10, 19, 25, 27, 31, 35, 36, 41, 43, 44, 51, 55, 62], or scenes [14, 21, 30, 38, 47, 53]. Coordinate-based networks have also been explored in the context of generative frameworks [7, 9, 20, 39, 40, 52].

The method closest to our application is neural radiance fields (NeRF) [38]. NeRF is a neural rendering framework that combines a volume represented by a coordinate-based network with a neural volume renderer to achieve state-of-the-art image quality for view synthesis tasks. Specifically, NeRF uses ReLU-based multilayer perceptrons (MLPs) with a positional encoding strategy to represent 3D scenes. Rendering an image from such a representation is done by evaluating the volume rendering equation [34], which requires integrating along rays passing through the neural volume parameterized by the MLP. This integration is performed using Monte Carlo sampling, which requires hundreds of forward passes through the MLP for each ray. However, this procedure is extremely slow, requiring days to train a representation of a single scene from multi-view images. Rendering a frame from a pre-optimized representation requires tens of seconds to minutes.

Here, we leverage automatic integration, or AutoInt, to significantly speed up the evaluation of integrals along rays. AutoInt reduces the number of network queries required to evaluate integrals (e.g., using Monte Carlo sampling) from hundreds to just two, greatly speeding up inference for neural rendering.

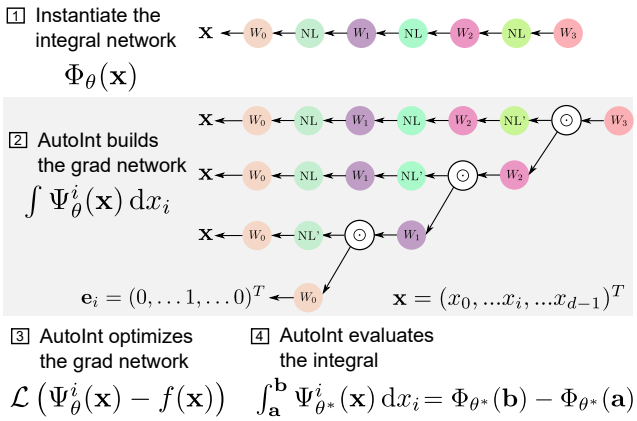

Figure 2. AutoInt pipeline. After (1) defining an integral network architecture, (2) AutoInt builds the corresponding grad network, which is (3) optimized to represent a function. (4) Definite integrals can then be computed by evaluating the integral network, which shares parameters with its grad network.

Integration Techniques. In general, integration is much more challenging than differentiation. Whereas automatic differentiation primarily builds on the chain rule, there are many different strategies for integration, including variable substitution, integration by parts, partial fractions, etc. Heuristics can be used to choose one or a combination of these strategies for any specific problem. Closed-form solutions to finding general antiderivatives exist only for a relatively small class of functions and, when possible, involve a rather complex algorithm, such as the Risch or Risch-Norman algorithm [42, 49, 50]. Perhaps the most common approach to computing integrals in practice is numerical integration, for example using Riemann sums, quadratures, or Monte-Carlo methods [11]. In these sampling-based methods, the number of samples trades off accuracy for runtime.

Since neural networks are universal function approximators, and are themselves functions, they can also be integrated analytically. Previous work has explored theory and connections between shallow neural networks and integral formulations for function approximation [12, 22]. Other work has derived closed-form solutions for integrals of simple single-layer or two-layer neural networks [57, 60]. As we shall demonstrate, our work is not limited to a fixed number of layers or a specific architecture. Instead, we directly train a grad network architecture for which the integral network is known by construction.

3. AutoInt for Neural Integration

In this section, we introduce a fundamentally new approach to compute and evaluate antiderivatives and definite integrals of coordinate-based neural networks.

3.1. Principles

We consider a coordinate-based network, i.e., a neural network with parameters θ mapping low-dimensional input coordinates to a low-dimensional output $\Phi_\theta : \mathbb{R}^{d_{\text{in}}} \mapsto \mathbb{R}^{d_{\text{out}}}$. We assume this network admits a (sub-)gradient with respect to its input $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$, and we denote by $\Psi^i_\theta = \partial\Phi_\theta / \partial x_i$ its derivative with respect to the coordinate x_i. Then, by the fundamental theorem of calculus we have that

$$\Phi_\theta(\mathbf{x}) = \int \frac{\partial \Phi_\theta}{\partial x_i}(\mathbf{x})\,\mathrm{d}x_i = \int \Psi^i_\theta(\mathbf{x})\,\mathrm{d}x_i. \tag{1}$$

This equation relates the coordinate-based network Φ_θ to its partial derivative Ψ^i_θ and, hence, Φ_θ is an antiderivative of Ψ^i_θ.

A key idea is that the partial derivative Ψ^i_θ is itself a coordinate-based network, mapping the same low-dimensional input coordinates $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ to the same low-dimensional output space $R^{d_{\text{out}}}$. In other words, Ψ^i_θ is a different neural network that shares its parameters θ with Φ_θ while also satisfying Equation 1. Now, rather than optimizing the coordinate-based network Φ_θ, we optimize Ψ^i_θ to represent a target signal, and we reassemble the optimized parameters (i.e., weights and biases) θ to form Φ_θ.

As a result, Φ_θ is a network that represents an antiderivative of Ψ^i_θ. We call this procedure of training Ψ^i_θ and reassembling θ to construct the antiderivative *automatic integration*. How to reassemble θ depends on the network architecture used for Φ_θ, and is addressed in the next section.

3.2. The Integral and Grad networks

Coordinate-based neural networks are usually formed from multilayer perceptron (MLP), or fully connected, architectures:

$$\Phi_\theta(\mathbf{x}) = \mathbf{W}_n(\phi_{n-1} \circ \phi_{n-2} \circ \cdots \circ \phi_0)(\mathbf{x}), \tag{2}$$

with $\phi_k : \mathbb{R}^{M_k} \mapsto \mathbb{R}^{N_k}$ being the k-th layer of the neural network defined as $\phi_k(\mathbf{y}) = \text{NL}_k(\mathbf{W}_k\mathbf{y} + \mathbf{b}_k)$ using the parameters $\theta = \{\mathbf{W}_k \in \mathbb{R}^{N_k \times M_k}, \mathbf{b}_k \in \mathbb{R}^{M_k}, \forall k\}$ and the nonlinearity NL, which is a function applied point-wise to all the elements of a vector.

The computational graph of a 3-hidden-layer MLP representing Φ_θ is shown in Figure 2. Operations are indicated as nodes and dependencies as directed edges. Here, the arrows of the directed edges point towards nodes that must be computed first.

For this MLP, the form of the network $\Psi^i_\theta = \partial\Phi_\theta/\partial x_i$ can be found using the chain rule

$$\Psi^i_\theta(\mathbf{x}) = \hat{\phi}_{n-1} \circ (\phi_{n-2} \circ \ldots \phi_0)(\mathbf{x}) \odot \ldots \\ \cdots \odot \hat{\phi}_1 \circ \phi_0(\mathbf{x}) \odot \mathbf{W}_0\,\mathbf{e}_i, \tag{3}$$

where $\odot$ indicates the Hadamard product, $\hat{\phi}_k(\mathbf{y}) = \mathbf{W}_k^T \text{NL}'_{k-1}(\mathbf{W}_{k-1}\mathbf{y} + \mathbf{b}_{k-1})$ and $\mathbf{e}_i \in \mathbb{R}^{d_{\text{in}}}$ is the unit vector that has 0's everywhere but at the i-th component. The

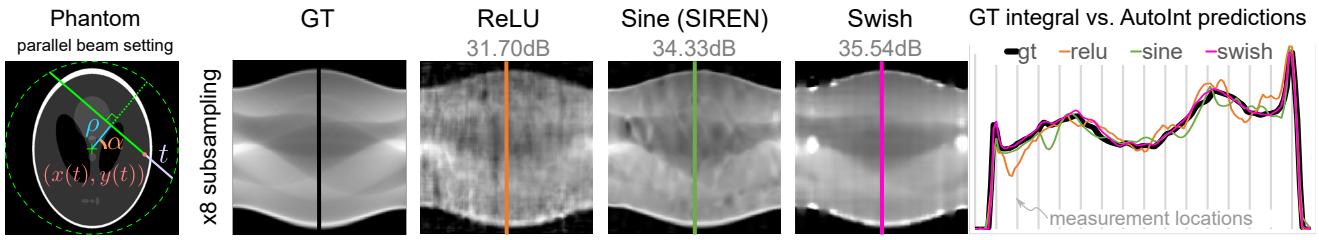

Figure 3. AutoInt for computed tomography. Left: illustration of the parameterization. Center: sinograms computed for integral networks using different activation functions. In all cases, the ground truth (GT) sinogram is subsampled 8× and the optimized integral network is sampled to interpolate missing measurements. The Swish activation performs best in terms of peak signal-to-noise ratio (PSNR). Right: a 1D scanline of the sinogram shows that Swish interpolates missing data best while sine activations [53] tend to overfit the measurements.

corresponding computational graph is shown in Figure 2. As we noted, despite having a different architecture (and vastly different number of nodes) the two networks share the same parameters. We refer to the network associated with Φ_θ as the *integral network* and the neural network associated with Ψ^i_θ as the *grad network*. Homologous nodes in their graphs are shown in the same color. This color scheme explicitly shows how the grad network parameters are reassembled to create the integral network after training.

3.3. Evaluating Antiderivatives & Definite Integrals

To compute the antiderivative and definite integrals of a function f in the AutoInt framework, one first chooses the specifics of the MLP architecture (number of layers, number of features, type of nonlinearities) for an integral network Φ_θ. The grad network Ψ^i_θ is then instantiated from this integral network based on AutoDiff. In practice, we developed a custom AutoDiff framework that traces the integral network and explicitly instantiates the corresponding grad network while maintaining the shared parameters (additional details in the supplemental). Once instantiated, parameters of the grad network are optimized to fit a signal of interest using conventional AutoDiff and optimization tools [45]. Specifically we optimize a loss of the form

$$\theta^* = \arg\min_\theta \mathcal{L}\left(\Psi^i_\theta(\mathbf{x}), f(\mathbf{x})\right). \tag{4}$$

Here, $\mathcal{L}$ is a cost function that aims at penalizing discrepancies between the target signal $f(\mathbf{x})$ we wish to integrate and the coordinate-based network Ψ^i_θ.

Once trained, the grad network approximates the signal, that is $\Psi^i_{\theta^*} \approx f(\mathbf{x})$, $\forall \mathbf{x}$. Therefore, the antiderivative of f can be calculated as

$$\int f(\mathbf{x})\,\mathrm{d}x_i \approx \int \Psi^i_{\theta^*}(\mathbf{x})\,\mathrm{d}x_i = \Phi_{\theta^*}(\mathbf{x}). \tag{5}$$

This corresponds to evaluating the integral network at $\mathbf{x}$ using weights θ^*. Furthermore, any definite integral of the signal f can be calculated using *only* two evaluations of Φ_θ, according to the Newton–Leibniz formula

$$\int_{\mathbf{a}}^{\mathbf{b}} f(\mathbf{x})\,\mathrm{d}x_i = \Phi_\theta(\mathbf{b}) - \Phi_\theta(\mathbf{a}). \tag{6}$$

We also note that AutoInt extends to integrating high-dimensional signals using a generalized fundamental theorem of calculus, which we describe in the supplemental.

3.4. Example in Computed Tomography

In tomography, integrals are at the core of the imaging model: measurements are line integrals of the absorption of a medium along rays that go through it. In particular, in a parallel beam setup, assuming a 2D medium of absorption $f(x, y) \in \mathbb{R}_+$, measurements can be modeled as

$$s(\rho, \alpha) = \int_{t_n}^{t_f} f\left(x(t), y(t)\right)\,\mathrm{d}t, \tag{7}$$

and (x, y) is on the ray $(\rho, \alpha) \in [-1, 1]\times[0, \pi]$ by satisfying $x(t)\cos(\alpha) + y(t)\sin(\alpha) = \rho$ with α being the orientation of the ray and ρ its eccentricity with respect to the origin as shown in Figure 3. The measurement s is called a sinogram, and this particular integral is referred to as the Radon transform of f [24].

The inverse problem of computed tomography involves recovering the absorption f given a sinogram. Here, for illustrative purposes, we will look at a tomography problem in which a grad network is trained on a sparse set of measurements and the integral network is evaluated to produce unseen ones. Sparse-view tomography is a standard reconstruction problem [18, 46], and this setup is analogous to the novel view synthesis problem we solve in Section 4.

We consider a dataset of measurements $\mathcal{D} = \{(\rho_i, \alpha_i, s(\rho_i, \alpha_i)\}_{i<D}$ corresponding to D sparsely sampled rays. We train a grad network using the AutoInt framework. For this purpose, we instantiate a grad network Ψ_θ whose input is a tuple (ρ, α, t). It is trained to match the dataset of measurements

$$\theta^* = \arg\min_\theta \sum_{i<D} \left\| \left(\frac{1}{T} \sum_{t_j<T} \Psi^t_\theta(\rho_i, \alpha_i, t_j) \right) - s(\rho_i, \alpha_i) \right\|_2^2. \tag{8}$$

Thus, at training time, the grad network is evaluated T times in a Monte Carlo fashion with $t_j \sim \mathcal{U}([t_n, t_f])$. At inference, just two evaluations of Φ_{θ^*} yield the integral

$$s(\rho, \alpha) = \Phi_{\theta^*}(\rho, \alpha, t_f) - \Phi_{\theta^*}(\rho, \alpha, t_n). \tag{9}$$

Results in Figure 3 show that the two evaluations of the integral network Φ_{θ^*} can faithfully reproduce supervised measurements and generalize to unseen data. Generalization, however depends on the type of nonlinearity used. We show that Swish [48] with normalized positional encoding (details in Sec. 5) generalizes well, and SIREN [53] fits the measurements better but fails to generalize to unseen views.

Note that both the nonlinearity NL and its derivative NL$'$ appear in the grad network architectures (Eq. (3) and Figure 2). This implies that integral networks with ReLU nonlinearities have step functions appearing in the grad network, possibly making training Ψ_θ difficult because of nodes with (constant) zero-valued derivatives. We explore several other nonlinearities here (with additional details in the supplemental), and show that Swish heuristically performs best in the grad networks used in our application. Yet, we believe the study of nonlinearities in grad networks to be an important avenue for future work.

4. Neural Volume Rendering

Combining volume rendering techniques with coordinate-based networks has proved to be a powerful technique for neural rendering and view synthesis [38]. Here, we briefly overview volume rendering and describe an approximate volume rendering model that enables our efficient rendering approach using AutoInt.

4.1. Volume Rendering

Classical volume rendering techniques are derived from the radiative transfer equation [8] with an assumption of minimal scattering in an absorptive and emissive medium [13, 34]. We adopt a rendering model based on tracing rays through the volume [23, 38], where the emission and absorption along camera rays produce color values that are assigned to rendered pixels.

The volume itself is represented as a high-dimensional function parameterized by position, $\mathbf{x} \in \mathbb{R}^3$, and viewing direction $\mathbf{d}$. We also define the camera rays that traverse the volume from an origin point $\mathbf{o}$ to a ray position $\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d}$. At each position in the volume, an absorption coefficient, $\sigma \in \mathbb{R}_+$, gives the probability per differential unit length that a ray is absorbed (i.e., terminates) upon interaction with an infinitesimal particle. Finally, an emissive radiance field $\mathbf{c} = (r, g, b) \in [0, 1]^3$, describes the color of emitted light at each point in space in all directions.

Rendering from the volume requires integrating the emissive radiance along the ray while also accounting for absorption. The transmittance T describes the net reduction from absorption from the ray origin to the ray position $\mathbf{r}(t)$, and is given as

$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma\big(\mathbf{r}(s)\big)\,\mathrm{d}s\right), \tag{10}$$

where t_n indicates a near bound along the ray. With this expression, we can define the volume rendering equation (VRE), which describes the color $\mathbf{C}$ of a rendered camera ray.

$$\mathbf{C}(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\,\sigma(\mathbf{r}(t))\,\mathbf{c}(\mathbf{r}(t), \mathbf{d})\,\mathrm{d}t. \tag{11}$$

Conventionally, the VRE is computed numerically by Riemann sums, quadratures, or Monte-Carlo methods [11], whose accuracy thus largely depends on the number of samples taken along the ray.

4.2. Approximate Volume Rendering for Automatic Integration

Automatic integration allows us to efficiently evaluate definite integrals using a closed-form solution for the antiderivative. However, the VRE cannot be directly evaluated with AutoInt because it consists of multiple nested integrations: the integration of radiance along the ray weighted by integrals of cumulative transmittance. We therefore choose to approximate this integral in piecewise sections that can each be efficiently evaluated using AutoInt. For N piecewise sections along a ray, we give the approximate VRE and transmittance as

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \bar{\sigma}_i\,\bar{\mathbf{c}}_i \bar{T}_i\,\delta_i, \quad \bar{T}_i = \exp\left(-\sum_{j=1}^{i-1} \bar{\sigma}_j \delta_j\right), \tag{12}$$

where

$$\bar{\sigma}_i = \delta_i^{-1} \int_{t_{i-1}}^{t_i} \sigma(t)\,\mathrm{d}t \quad \text{and} \quad \bar{\mathbf{c}}_i = \delta_i^{-1} \int_{t_{i-1}}^{t_i} \mathbf{c}(t)\,\mathrm{d}t,$$

and $\delta_i = t_i - t_{i-1}$ is the length of each piecewise interval along the ray. Equation 12 can also be viewed as a repeated alpha compositing operation with alpha values of $\bar{\sigma}_i\delta_i$. After some simplification and substitution into Equation 12 (see supplemental), we have the following expression for the piecewise VRE:

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \delta_i^{-1} \int_{t_{i-1}}^{t_i} \sigma(t)\,\mathrm{d}t \cdot \int_{t_{i-1}}^{t_i} \mathbf{c}(t)\,\mathrm{d}t \cdot \prod_{j=1}^{i-1} \exp\left(-\int_{t_{j-1}}^{t_j} \sigma(s)\,\mathrm{d}s\right). \tag{13}$$

While this piecewise expression is only an approximation to the full VRE, it enables us to use AutoInt to efficiently evaluate each piecewise integral over absorption and radiance. In practice, there is a tradeoff between improved computational efficiency and degraded accuracy of the approximation as the value of N decreases. We evaluate this tradeoff in the context of volume rendering and learned novel view synthesis in Sec. 6.

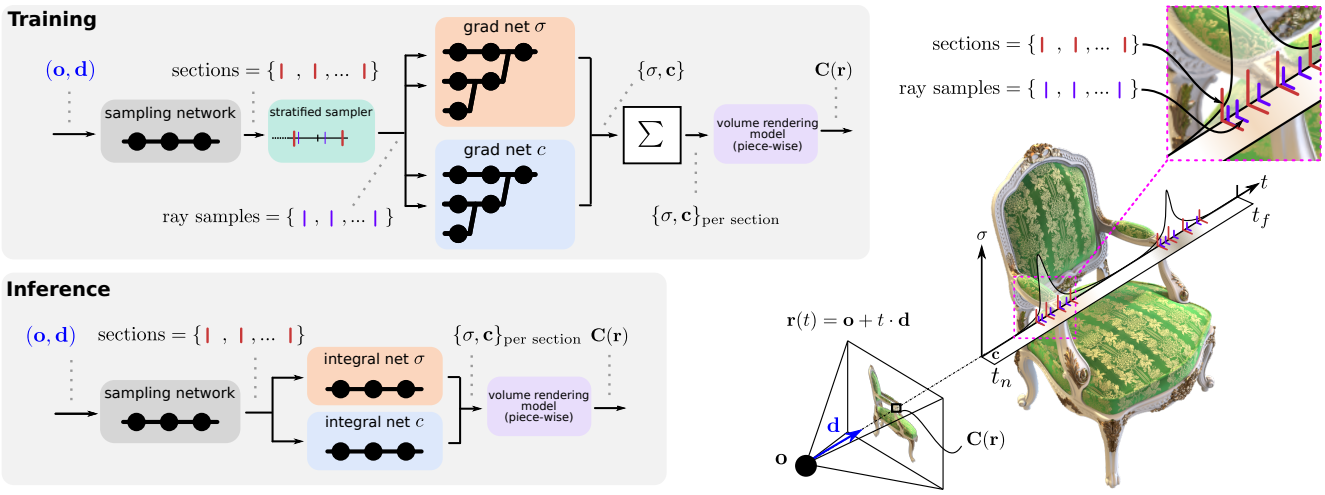

Figure 4. Volume rendering pipeline. During training, the grad networks representing volume density σ and color $\mathbf{c}$ are optimized for a given set of multi-view images (top left). For inference, the grad networks' parameters are reassembled to form the integral networks, which represent antiderivatives that can be efficiently evaluated to calculate ray integrals through the volume (bottom left). A sampling network predicts the locations of piecewise sections used for evaluating the definite integrals (right).

5. Optimization Framework

We evaluate the piecewise VRE introduced in the previous section using an optimization framework overviewed in Figure 4. At the core of the framework are two MLPs that are used to compute integrals over values of σ and $\mathbf{c}$ as we detail in the following.

Network Parameterization. Rendering an image from the high-dimensional volume represented by the MLPs requires evaluating integrals along each ray $\mathbf{r}(t)$ in the direction of t. Thus, the grad network should represent $\partial\Phi_\theta/\partial t$, the partial derivative of the integral network with respect to the ray parameter. In practice, the networks take as input the values that define each ray: $\mathbf{o}$, t, and $\mathbf{d}$. Then, positions along the ray are calculated as $\mathbf{x} = \mathbf{o} + t\,\mathbf{d}$ and passed to the initial layers of the networks together with $\mathbf{d}$. With this dependency on t, we use our custom AutoDiff implementation to trace computation through the integral network, define the computational graph that computes the partial derivative with respect to t, and instantiate the grad network.

Grad Network Positional Encoding. As demonstrated by Mildenhall et al. [38], a positional encoding on the input coordinates to the network can significantly improve the ability of a network to render fine details. We adopt a similar scheme, where each input coordinate is mapped into a higher dimensional space as using a function $\gamma(p) : \mathbb{R} \mapsto \mathbb{R}^{2L}$ defined as

$$\gamma(p) = (\sin(\omega_0 p), \cos(\omega_0 p), \cdots, \sin(\omega_{L-1} p), \cos(\omega_{L-1} p)), \tag{14}$$

where $\omega_i = 2^i \pi$ and L controls the number of frequencies used to encode each input. We find that using this scheme directly in the grad network produces poor results because it introduces an exponentially increasing amplitude scaling into the coordinate encoding. This can easily be seen by calculating the derivative $\partial\gamma/\partial p = (\cdots \omega_i \cos(\omega_i p), -\omega_i \sin(\omega_i p) \cdots)$. Instead, we use a normalized version of the positional encoding for the integral network, which improves performance when training the grad network:

$$\bar{\gamma}(p) = \left(\cdots, \omega_i^{-1} \sin(\omega_i p), \omega_i^{-1} \cos(\omega_i p), \cdots\right). \tag{15}$$

Predictive Sampling. While AutoInt is used at inference time, at training time, the grad network is optimized by evaluating the piecewise integrals of Equation 13 using a quadrature rule discussed by Max [34]:

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \bar{T}_i \left(1 - \exp(-\bar{\sigma}_i \delta_i)\right) \bar{\mathbf{c}}_i \,. \tag{16}$$

We use Monte Carlo sampling to evaluate the integrals $\bar{\sigma}_i$ and $\bar{\mathbf{c}}_i$ by querying the networks at many positions within each interval δ_i.

However, some intervals δ_i along the ray contribute more to a rendered pixel than others. Thus, assuming we use the same number of samples per interval, we can improve sample efficiency by strategically adjusting the length of these intervals to place more samples in positions with large variations in σ and $\mathbf{c}$. This idea is similar in spirit to accelerated volume rendering techniques using hierarchical sampling or adaptive ray termination [29].

To this end, we introduce a small sampling network (illustrated in Figure 4), which is implemented as an MLP $\mathcal{S}(\mathbf{o}, \mathbf{d})$ that predicts interval lengths $\boldsymbol{\delta} \in \mathbb{R}^N$. Then, we calculate stratified samples along the

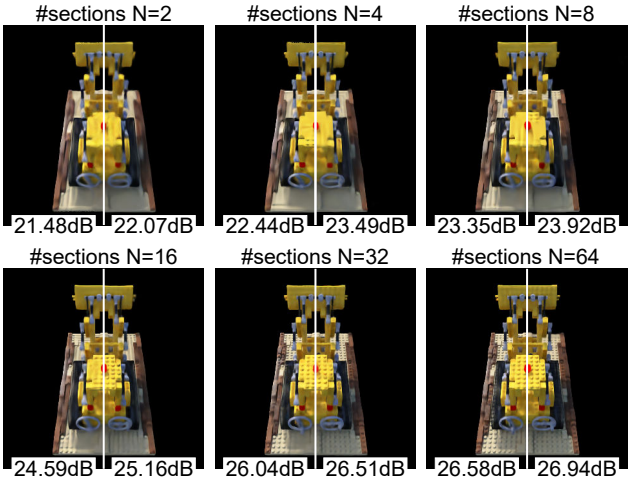

Figure 5. Ablation studies. A view of the *Lego* scene is shown with a varying number of intervals ($N = \{2, 4, 8, 16, 32, 64\}$) without (left half of the images) and with (right half) the sampling network. PSNR is computed on the 200 test set views.

ray by subdividing each interval δ_i into M bins and calculating samples $t_{i,j}, j = 1, \ldots, M$ as $t_{i,j} \sim \mathcal{U}\left(t_{i-1} + \frac{j-1}{M}\delta_i, t_{i-1} + \frac{j}{M}\delta_i\right)$.

Fast Grad Network Evaluation. AutoInt can be implemented directly in popular optimization frameworks (e.g., PyTorch [45], Tensorflow [1]); however, training the grad network is generally computationally slow and memory inefficient. These inefficiencies stem from the two step procedure required to compute the grad network output at each training iteration: (1) a forward pass through the integral network is computed and then (2) AutoDiff calculates the derivative of the output with respect to the input variable of integration. Instead, we implemented a custom AutoDiff framework on top of PyTorch that parses a given integral network and explicitly instantiates the grad network modules with weight sharing (see Figure 2). Then, we evaluate and train the grad network directly, without the overhead of the additional per-iteration forward pass and derivative computation. Compared to the two-step procedure outlined above, our custom framework improves per-iteration training speed by a factor of 1.8 and reduces memory consumption by 15% for the volume rendering application. More details about our AutoInt implementation can be found in the supplemental, and our code is publicly available[1].

Implementation Details. In our framework, a volume representation is optimized separately for each rendered scene. To optimize the grad networks, we require a collection of RGB images taken of the scene from varying camera positions, and we assume that the camera poses and intrinsic parameters are known. At training time, we randomly sample images from the training dataset, and from each image

[1] https://github.com/computational-imaging/automatic-integration

	NeRF	Neural Volumes	AutoInt (N=#sections)		
			$N = 8$	$N = 16$	$N = 32$
PSNR (dB)	31.0	26.1	25.6	26.0	26.8
Memory (GB)	15.6	10.4	15.5	15.0	15.5
Runtime (s/frame)	30	0.3	2.6	4.8	9.3

Table 1. NeRF [38] achieves the best image quality measured by average peak signal-to-noise ratio (PSNR). Neural Volumes [32] is faster and slightly more memory efficient, but suffers from lower image quality. AutoInt allows us to approximate the NeRF solution with a tradeoff between image quality and runtime defined by the number of intervals used by our sampling network. Results are aggregated over the 8 Blender scenes of the NeRF dataset.

VRE	Network Type	Samples/Forward Passes	PSNR (dB) ↑	SSIM ↑	LPIPS ↓
Piecewise (approx.)	Grad MLP (proposed)	9, $N = 8$	25.09	0.900	0.175
		17, $N = 16$	25.48	0.905	0.171
		33, $N = 32$	27.26	0.929	0.135
Piecewise (approx.)	Standard MLP	128, $N = 8$	29.21	0.952	0.052
		128, $N = 16$	29.97	0.960	0.047
		128, $N = 32$	29.68	0.959	0.049
Full (exact)	Grad MLP	128	27.95	0.936	0.082
NeRF		128	30.68	0.968	0.045
		32	23.30	0.920	0.093
		8	14.62	0.761	0.258

Table 2. Comparison of performance on the *Lego* scene for different network configurations. We report PSNR/SSIM [61] and LPIPS [63]. AutoInt uses the piecewise VRE and a grad network (top rows), and the number of forward passes required at inference depends on the number of piecewise sections (N). We also evaluate using a fixed number of samples with the piecewise VRE and a standard MLP (i.e., Monte Carlo sampling, no grad network), as well as using the full VRE with a grad network. Finally we compare to NeRF [38] using varying samples at inference, which reduces computational requirements.

we randomly sample a number of rays. Then, we optimize the network to minimize the loss function

$$\mathcal{L} = \sum_{\mathbf{r} \in \mathcal{R}} \|\tilde{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2, \tag{17}$$

where $\mathbf{C}$ is the ground truth pixel value for the selected ray.

In our implementation, we train the networks using PyTorch and the Adam optimizer [26] with a learning rate of 5×10^{-4}. The networks representing volume density and color each have 8 hidden layers with 256 hidden units, we use a batch size of 4 with 1024 rays sampled from each image, and we decay the learning rate by a factor of 0.2 every 10^5 iterations. Training and inference are performed using NVIDIA V100 GPUs. For the sampling network, we evaluate using $M = 128/N$ samples within each piecewise interval for $N \in \{2, 4, 8, 16, 32, 64\}$ (see Figure 5) and find that using 8, 16, or 32 piecewise intervals produces acceptable results while achieving a significant computational acceleration with AutoInt. Finally, for the positional encoding, we use $L = 10$ and $L = 4$ for $\mathbf{x}$ and $\mathbf{d}$, respectively.

6. Results

We evaluate AutoInt for volume rendering on a synthetic dataset of scenes with challenging geometries and

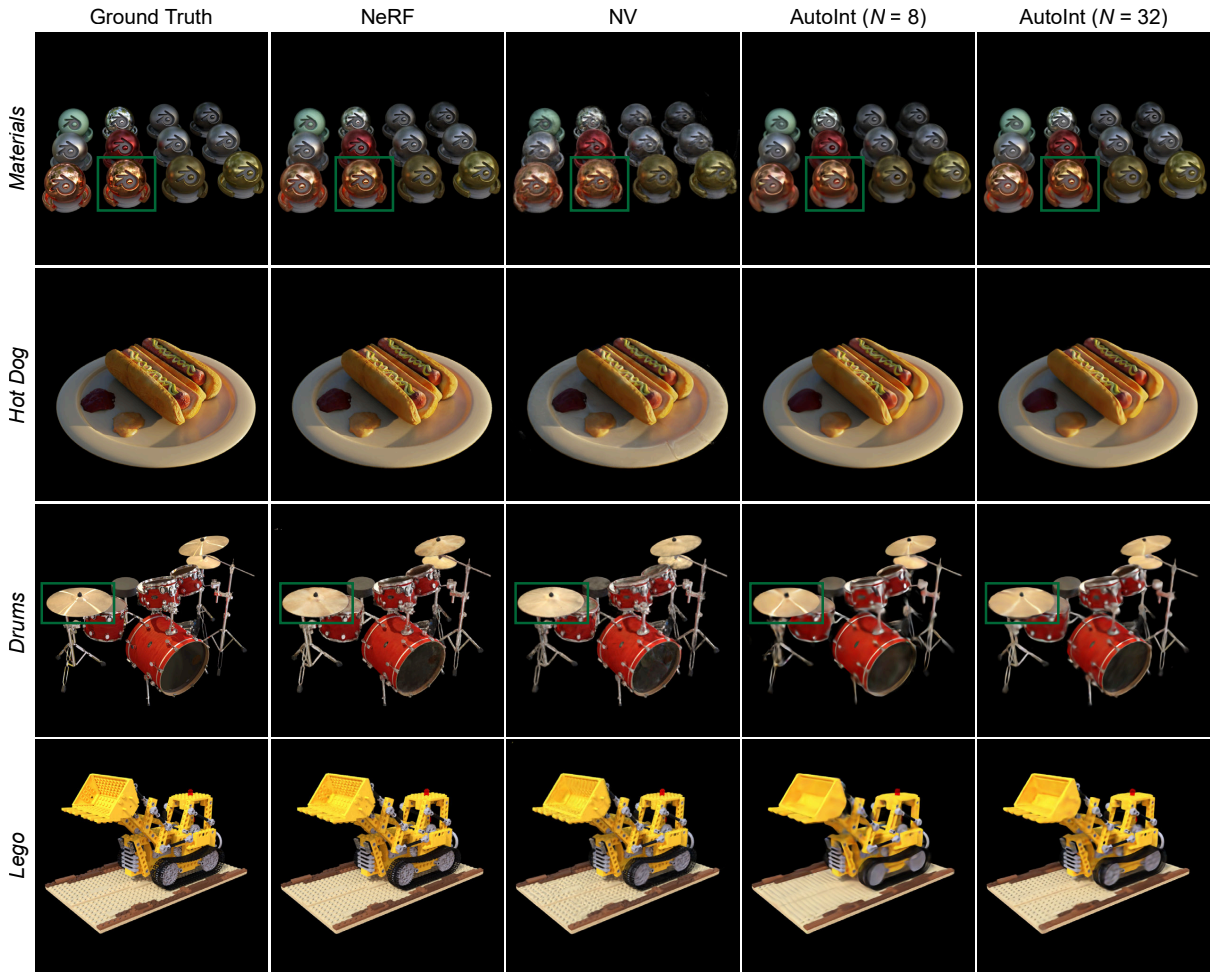

Figure 6. Qualitative results. We compare the performance of Neural Volumes [32] and NeRF [38] to AutoInt using $N = 8$ and $N = 32$ in our approximate volume rendering equation. AutoInt accurately captures view-dependent effects like specular reflections (green boxes) and reduces render times by greater than $10\times$ relative to NeRF, though with some reduction in overall image quality.

reflectance properties. We demonstrate that the approach allows an improved tradeoff between image quality and rendering speed for neural volume rendering. Rendering times are improved by greater than $10\times$ compared to the state-of-the-art [38], though at slightly reduced image quality.

Our training dataset consists of eight objects, each rendered from 100 different camera positions using the Blender Cycles engine [38]. For the test set, we evaluate on an additional 200 images. We compare AutoInt to two other baselines: Neural Radiance Fields (NeRF) [38] and Neural Volumes [32]. NeRF uses a similar architecture and Monte Carlo sampling with the full volume rendering model, rather than our piecewise approximation and AutoInt. Neural Volumes is a voxel-based method that encodes a deep voxel grid representation of a scene using a convolutional neural network. Novel views are rendered by applying a learned warping operator to the voxel grid and sampling voxel values by marching rays from the camera position.

In Table 1 we report the peak signal-to-noise ratio (PSNR) averaged across all scenes and test images. AutoInt outperforms Neural Volumes quantitatively, while achieving a greater than $10\times$ improvement in render time relative to NeRF, though with a tradeoff in image quality. Increasing the number of piecewise sections in the approximate VRE improves render quality at the cost of computation.

We evaluate the effect of the sampling network and the number of sections in the approximate VRE in Figure 5 for the *Lego* scene. Using the sampling network improves performance and sample efficiency by allocating more sections in regions with large variations in the volume density.

In Table 2 we show the effect of the VRE approximation and grad network architecture on render quality of the *Lego* scene. Using the full VRE achieves similar performance to the approximate, piecewise VRE with 32 sections. We attribute most of the difference in performance between our method and NeRF to the regularized, tree-like structure of the grad network, which is constrained by weight sharing between the branches (see Figure 2). While evaluating

NeRF with fewer samples along each ray reduces computation, rendering quality degrades significantly compared to using AutoInt with the same number of samples.

We also show qualitative results in Figure 6 for the *Materials*, *Hot Dog*, *Drums*, and *Lego* scenes. Again, the quality of the rendered images improves as the number of sections increases. In the *Materials* scene (Figure 6), the proposed technique exhibits fewer artifacts compared to Neural Volumes. AutoInt also shows improved modeling of view-dependent effects in the *Drums* scene relative to Neural Volumes and NeRF (e.g., specular highlights on the symbols). We show additional results on captured scenes from the Local Light Field Fusion and DeepVoxels datasets [37, 54] in the supplemental.

7. Discussion

In this work, we introduce a new framework for numerical integration in the context of coordinate-based neural networks. Applied to neural volume rendering, AutoInt enables improvements to computational efficiency by learning closed-form solutions to integrals. Although these computational speedups currently come with a tradeoff to image quality, the method takes steps towards efficient learned integration using deep network architectures.

Our approach is analogous to conventional methods for fast evaluation of the VRE; for example, methods based on shear-warping [28] and the Fourier projection-slice theorem [33, 59]. Similar to our method, these techniques use approximations (e.g., with sampling and interpolation) that trade off image quality with computationally efficient rendering. Additionally, we believe our approach is compatible with recent work that aims to speed up volume rendering by pruning areas of the volume that do not contain the rendered object [30].

A key idea of AutoInt is that an integral network can be automatically created after training a corresponding grad network. We note that grad networks represent a fundamentally different and relatively unexplored type of network architecture compared to conventional fully-connected networks. While using grad networks with certain classes of non-linearities (e.g., Swish) can improve image quality, exploring improved training strategies and more expressive grad network architectures is an important and promising direction for future work. Finally, we believe that AutoInt will be of interest to a wide array of application areas beyond computer vision, especially for problems related to inverse rendering, sparse-view tomography, and compressive sensing.

Acknowledgments. Julien N. P. Martel was supported by a Swiss National Foundation (SNF) Fellowship (P2EZP2 181817). Gordon Wetzstein was supported by an NSF CAREER Award (IIS 1553333) and a PECASE from the ARO.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Benjamin Attal, Selena Ling, Aaron Gokaslan, Christian Richardt, and James Tompkin. MatryODShka: Real-time 6DoF video view synthesis using multi-sphere images. In *Proc. ECCV*, 2020.

[3] Matan Atzmon and Yaron Lipman. SAL: Sign agnostic learning of shapes from raw data. In *Proc. CVPR*, 2020.

[4] Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew Duvall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. Immersive light field video with a layered mesh representation. *ACM Trans. Graph. (SIGGRAPH)*, 39(4), 2020.

[5] Joel Carranza, Christian Theobalt, Marcus A. Magnor, and Hans-Peter Seidel. Free-viewpoint video of human actors. *ACM Trans. Graph. (SIGGRAPH)*, 22(3), 2003.

[6] Rohan Chabra, Jan Eric Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard Newcombe. Deep local shapes: Learning local SDF priors for detailed 3D reconstruction. In *Proc. ECCV*, 2020.

[7] Eric R Chan, Marco Monteiro, Petr Kellnhofer, Jiajun Wu, and Gordon Wetzstein. pi-GAN: Periodic implicit generative adversarial networks for 3D-aware image synthesis. In *Proc. CVPR*, 2021.

[8] Subrahmanyan Chandrasekhar. *Radiative transfer*. Courier Corporation, 2013.

[9] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proc. CVPR*, 2019.

[10] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. Overfit neural networks as a compact shape representation. *arXiv preprint arXiv:2009.09808*, 2020.

[11] Philip J. Davis and Philip Rabinowitz. *Methods of numerical integration*. Courier Corporation, 2007.

[12] Anton Dereventsov, Armenak Petrosyan, and Clayton Webster. Neural network integral representations with the ReLU activation function. *arXiv preprint arXiv:1910.02743*, 2019.

[13] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *ACM SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[14] S. M. Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S. Morcos, Marta Garnelo, Avraham Ruderman, Andrei A. Rusu, Ivo Danihelka, Karol Gregor, et al. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018.

[15] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. Deepview: View synthesis with learned gradient descent. In *Proc. CVPR*, 2019.

[16] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas Funkhouser. Local deep implicit functions for 3D shape. In *Proc. CVPR*, 2020.

[17] Kyle Genova, Forrester Cole, Daniel Vlasic, Aaron Sarna, William T. Freeman, and Thomas Funkhouser. Learning

shape templates with structured implicit functions. In *Proc. ICCV*, 2019.

[18] Richard Gordon, Robert Bender, and Gabor T. Herman. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. *J. Theor. Bio.*, 29(3):471–481, 1970.

[19] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. Implicit geometric regularization for learning shapes. In *Proc. ICML*, 2020.

[20] Philipp Henzler, Niloy J. Mitra, and Tobias Ritschel. Escaping Plato's cave: 3D shape from adversarial rendering. In *Proc. ICCV*, 2019.

[21] Chiyu Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Nießner, and Thomas Funkhouser. Local implicit grid representations for 3d scenes. In *Proc. CVPR*, 2020.

[22] Paul C. Kainen, Vera Kurková, and Andrew Vogt. An integral formula for heaviside neural networks. *Neural Network World*, 10:313–319, 2000.

[23] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics*, 18(3):165–174, 1984.

[24] A. C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. IEEE Press, 2002.

[25] Petr Kellnhofer, Lars Jebe, Andrew Jones, Ryan Spicer, Kari Pulli, and Gordon Wetzstein. Neural lumigraph rendering. In *Proc. CVPR*, 2021.

[26] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proc. ICLR*, 2014.

[27] Amit Kohli, Vincent Sitzmann, and Gordon Wetzstein. Semantic implicit neural scene representations with semi-supervised training. *Proc. 3DV*, 2020.

[28] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proc. SIGGRAPH*, 1994.

[29] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[30] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. In *NeurIPS*, 2020.

[31] Shaohui Liu, Yinda Zhang, Songyou Peng, Boxin Shi, Marc Pollefeys, and Zhaopeng Cui. DIST: Rendering deep implicit signed distance function with differentiable sphere tracing. In *Proc. CVPR*, 2020.

[32] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph. (SIGGRAPH)*, 38(4), 2019.

[33] Tom Malzbender. Fourier volume rendering. *ACM Trans. Graph.*, 12(3):233–250, 1993.

[34] Nelson Max. Optical models for direct volume rendering. *IEEE Trans. Vis. Comput. Graph*, 1(2):99–108, 1995.

[35] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3D reconstruction in function space. In *Proc. CVPR*, 2019.

[36] Mateusz Michalkiewicz, Jhony K. Pontes, Dominic Jack, Mahsa Baktashmotlagh, and Anders Eriksson. Implicit surface representations as layers in neural networks. In *Proc. ICCV*, 2019.

[37] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Trans. Graph. (SIGGRAPH)*, 38(4), 2019.

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

[39] Thu Nguyen-Phuoc, Chuan Li, Lucas Theis, Christian Richardt, and Yong-Liang Yang. HoloGAN: Unsupervised learning of 3D representations from natural images. In *Proc. ICCV*, 2019.

[40] Thu Nguyen-Phuoc, Christian Richardt, Long Mai, Yong-Liang Yang, and Niloy Mitra. BlockGAN: Learning 3D object-aware scene representations from unlabelled images. In *Proc. NeurIPS*, 2020.

[41] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3D representations without 3D supervision. In *Proc. CVPR*, 2020.

[42] Arthur C. Norman and P. M. A. Moore. Implementing the new Risch integration algorithm. In *Proc. 4th. Int. Colloquium on Advanced Computing Methods in Theoretical Physics*, 1977.

[43] Michael Oechsle, Lars Mescheder, Michael Niemeyer, Thilo Strauss, and Andreas Geiger. Texture fields: Learning texture representations in function space. In *Proc. ICCV*, 2019.

[44] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *Proc. CVPR*, 2019.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.

[46] Daniël M Pelt, Kees Joost Batenburg, and James A Sethian. Improving tomographic reconstruction from limited data using mixed-scale dense convolutional neural networks. *J. Imaging*, 4(11):128, 2018.

[47] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. In *Proc. ECCV*, 2020.

[48] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[49] Robert H. Risch. The problem of integration in finite terms. *Trans. Am. Math. Soc*, 139:167–189, 1969.

[50] Robert H. Risch. The solution of the problem of integration in finite terms. *Bull. Am. Math. Soc.*, 76(3):605–608, 1970.

[51] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa, and Hao Li. PIFu: Pixel-aligned implicit function for high-resolution clothed human digitization. In *Proc. ICCV*, 2019.

[52] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. GRAF: Generative radiance fields for 3D-aware image synthesis. In *Proc. NeurIPS*, 2020.

[53] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *Proc. NeurIPS*, 2020.

[54] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. DeepVoxels: Learning persistent 3D feature embeddings. In *Proc. CVPR*, 2019.

[55] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3D-structure-aware neural scene representations. In *Proc. NeurIPS*, 2019.

[56] Richard Szeliski. *Computer vision: algorithms and applications*. Springer, 2011 edition, 2010.

[57] Gregory H. Teichert, A. R. Natarajan, A. Van der Ven, and Krishna Garikipati. Machine learning materials physics: Integrable deep neural networks enable scale bridging by learning free energy functions. *Comput. Methods Appl. Mech. Eng.*, 353:201–216, 2019.

[58] Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason Saragih, Matthias Nießner, et al. State of the art on neural rendering. *Proc. Eurographics*, 2020.

[59] Takashi Totsuka and Marc Levoy. Frequency domain volume rendering. In *Proc. SIGGRAPH*, 1993.

[60] Paul Turner and John Guiver. Introducing the bounded derivative network—superceding the application of neural networks in control. *J. Process Control*, 15(4):407–415, 2005.

[61] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. Image Process.*, 13(4):600–612, 2004.

[62] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Ronen Basri, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. In *Proc. NeurIPS*, 2020.

[63] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proc. CVPR*, 2018.

[64] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *ACM Trans. Graph. (SIGGRAPH)*, 37(4), 2018.

Supplemental Material
AutoInt: Automatic Integration for Fast Neural Volumetric Rendering

David B. Lindell* Julien N. P. Martel* Gordon Wetzstein
Stanford University
{lindell, jnmartel, gordon.wetzstein}@stanford.edu

1. Multivariable Integration with AutoInt

We consider an implicit neural representation realized by a neural network with parameters θ. The network maps low-dimensional input coordinates to a low-dimensional output $\Phi_\theta : \mathbb{R}^{d_{\text{in}}} \mapsto \mathbb{R}^{d_{\text{out}}}$, and we assume that the network admits a (sub-)gradient with respect to its input $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$. We denote by $\Psi_\theta^i = \partial \Phi_\theta / \partial x_i$ the derivative of the network output with respect to the input coordinate x_i, and, as described in the main text, we call Ψ_θ^i the grad network and Φ_θ the integral network.

By the fundamental theorem of calculus, the grad network and integral network are related as

$$\Phi_\theta(\mathbf{x}) = \int \frac{\partial \Phi_\theta}{\partial x_i}(\mathbf{x})\, \mathrm{d}x_i = \int \Psi_\theta^i(\mathbf{x})\, \mathrm{d}x_i. \tag{1}$$

As a corollary we have that definite integrals can be computed by two evaluations of the integral network:

$$\int_{a_i}^{b_i} \Psi_\theta^i(\mathbf{x})\, \mathrm{d}x_i = \Phi_\theta(\mathbf{x})\Big|_{x_i=b_i} - \Phi_\theta(\mathbf{x})\Big|_{x_i=a_i}. \tag{2}$$

Now, we will extend this result to multiple integrations. First, we let $\Psi_\theta^{i,j} = \partial \Psi_\theta^i / \partial x_j$ be the partial derivative with respect to x_j such that

$$\Psi_\theta^i = \int \frac{\partial \Psi_\theta^i}{\partial x_j}(\mathbf{x})\, \mathrm{d}x_j = \int \Psi_\theta^{i,j}(\mathbf{x})\, \mathrm{d}x_j. \tag{3}$$

*Equal contribution.

http://www.computationalimaging.org/publications/automatic-integration/

Then we can express the double integral as

$$\begin{aligned}
&\int_{a_i}^{b_i} \int_{a_j}^{b_j} \Psi_\theta^{i,j}(\mathbf{x})\, \mathrm{d}x_j\, \mathrm{d}x_i \\
&\quad = \int_{a_i}^{b_i} \Psi_\theta^i(\mathbf{x})\Big|_{x_j=b_j} - \Psi_\theta^i(\mathbf{x})\Big|_{x_j=a_j} \mathrm{d}x_i \\
&\quad = \left(\Phi_\theta(\mathbf{x})\Big|_{x_j=b_j} - \Phi_\theta(\mathbf{x})\Big|_{x_j=a_j}\right)\Big|_{x_i=b_i} \\
&\qquad - \left(\Phi_\theta(\mathbf{x})\Big|_{x_j=b_j} - \Phi_\theta(\mathbf{x})\Big|_{x_j=a_j}\right)\Big|_{x_i=a_i}.
\end{aligned} \tag{4}$$

Equation 4 can be further simplified with a slight abuse of notation by letting $\Phi_\theta(\mathbf{x})\Big|_{x_i=a_i, x_j=a_j} = \Phi_\theta(a_i, a_j)$, resulting in

$$\begin{aligned}
\int_{a_i}^{b_i} \int_{a_j}^{b_j} \Psi_\theta^{i,j}(\mathbf{x})\, \mathrm{d}x_j\, \mathrm{d}x_i = \Phi_\theta(b_i, b_j) - \Phi_\theta(b_i, a_j) \\
- \Phi_\theta(a_i, b_j) + \Phi_\theta(a_i, a_j).
\end{aligned} \tag{5}$$

Stated otherwise, a definite integral over two dimensions can be computed with four evaluations of the integral network at the given bounds.

This result can be extended to n dimensions with the following formula [6].

$$\begin{aligned}
&\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} \Psi_\theta^{1,\ldots,n}(\mathbf{x})\, \mathrm{d}x_n \cdots \mathrm{d}x_1 \\
&= \sum_{\epsilon_1,\ldots,\epsilon_n=0}^{1} (-1)^{\epsilon_1+\cdots+\epsilon_n} \Phi_\theta(\epsilon_1 a_1 + \bar{\epsilon}_1 b_1, \ldots, \epsilon_n a_n + \bar{\epsilon}_n b_n),
\end{aligned} \tag{6}$$

with $\bar{\epsilon}_i = 1 - \epsilon_i$. Thus for $n = 3$, we would have

$$\begin{aligned}
&\Phi_\theta(b_i, b_j, b_k) - \Phi_\theta(b_i, a_j, b_k) - \Phi_\theta(a_i, b_j, b_k) \\
&+ \Phi_\theta(a_i, a_j, b_k) - \Phi_\theta(b_i, b_j, a_k) + \Phi_\theta(b_i, a_j, a_k) \\
&+ \Phi_\theta(a_i, b_j, a_k) - \Phi_\theta(a_i, a_j, a_k).
\end{aligned} \tag{7}$$

Overall, using AutoInt for multivariable integration follows a similar procedure to evaluating a single integral as described in the main text. First, one constructs the grad network by taking partial derivatives of the integral network with respect to each of the variables of integration. Then, after training, the integral network is reassembled from the parameters θ and evaluated at the bounds of the domain as described by Equation 6.

2. Deriving the VRE Approximation and Quadrature

2.1. Piecewise Constant VRE

Our approximation of the volume rendering equation (VRE), can be viewed as a Riemann integral using N piecewise constant sections:

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \bar{\sigma}_i \, \bar{\mathbf{c}}_i \bar{T}_i \, \delta_i, \tag{8}$$

where $\delta_i = t_i - t_{i-1}$ is the length of the section i along the ray. The density $\bar{\sigma}_i$ and radiance $\bar{c}_i$ of each section are defined as:

$$\bar{\sigma}_i = \delta_i^{-1} \int_{t_{i-1}}^{t_i} \sigma(t) \, \mathrm{d}t, \tag{9}$$

and

$$\bar{\mathbf{c}}_i = \delta_i^{-1} \int_{t_{i-1}}^{t_i} \mathbf{c}(t) \, \mathrm{d}t, \tag{10}$$

and the transmittance as

$$\bar{T}_i = \exp\left(- \sum_{j=1}^{i-1} \bar{\sigma}_j \delta_j \right). \tag{11}$$

After substituting the terms defined in Equations (9),(10) and (11) into Equation (8) and simplifying, we obtain the following expression for the piecewise volume rendering equation:

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \delta_i^{-1} \int_{t_{i-1}}^{t_i} \sigma(t) \, \mathrm{d}t \cdot \int_{t_{i-1}}^{t_i} \mathbf{c}(t) \, \mathrm{d}t \cdot \prod_{j=1}^{i-1} \exp\left(- \int_{t_{j-1}}^{t_j} \sigma(s) \, \mathrm{d}s \right). \tag{12}$$

2.2. Calculation via Quadrature

To obtain the quadrature rule proposed by Max [3], used in the NeRF model, we first note that in the limit of $\bar{\sigma}_i \delta_i \to 0$ the following Taylor expansion holds

$$\exp(\bar{\sigma}_i \delta_i) \overset{\bar{\sigma}_i \delta_i \to 0}{=} 1 + \bar{\sigma}_i \delta_i + O(\bar{\sigma}_i \delta_i). \tag{13}$$

That is, using the definition of $\bar{\sigma}_i$ from Equation (9) we have

$$1 - \exp(-\bar{\sigma}_i \delta_i) \approx \bar{\sigma}_i \delta_i = \int_{t_{i-1}}^{t_i} \sigma(t) \, \mathrm{d}t, \tag{14}$$

yielding the quadrature

$$\tilde{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} \bar{T}_i \left(1 - \exp(-\bar{\sigma}_i \delta_i)\right) \bar{\mathbf{c}}_i \tag{15}$$

3. AutoInt Implementation

3.1. Overview

In the AutoInt framework, we start by specifying the architecture of the integral network: the number of layers, features, the type of non-linearities, and the input parameterization. Our implementation of AutoInt relies on a evaluating computational graphs, where dependencies are modeled using directed acyclic graphs (DAGs). With this graph-based representation, we create an automated pipeline for instantiating grad networks from integral networks, and we develop an efficient procedure for evaluating grad networks during training.

DAGs to represent computational graphs. Our AutoInt implementation internally maintains the computational graph of neural networks as a Directed Acyclic Graphs (DAG). Most nodes in the graph represent computational operators and there are two kinds of leaf nodes: (1) an input node with respect to which we can take the derivative of the graph and (2) a constant input node. Directed edges represent dependencies between nodes and point towards dependencies (i.e., other nodes to be computed first). Hence, nodes of in-degree zero are final results of the computation graph.

Building the grad network. To instantiate a *grad network*, AutoInt performs auto differentiation on the DAG of the integral network. Each node is called in a topological order and provides its own derivative. This recursive chain of calls builds the computation graph of the grad network.

Evaluating the grad network. Once the grad network is built it can be evaluated using a reverse topological ordering of its nodes: starting from the leaves and tracing computation back to the root(s). Note that this procedure is different than backpropagation, where intermediate results from the forward pass are stored in order to evaluate the graph associated with the backward pass. In AutoInt, given the grad network is a separate entity from the integral network, the intermediate results from the forward pass are unavailable. However, the grad network can still be computed efficiently by reusing computations from the "legs" of the network, since these nodes share weights and perform the same computations (see Figure 1). This is done by maintaining a lexicographic ordering between nodes of same in-degree in the

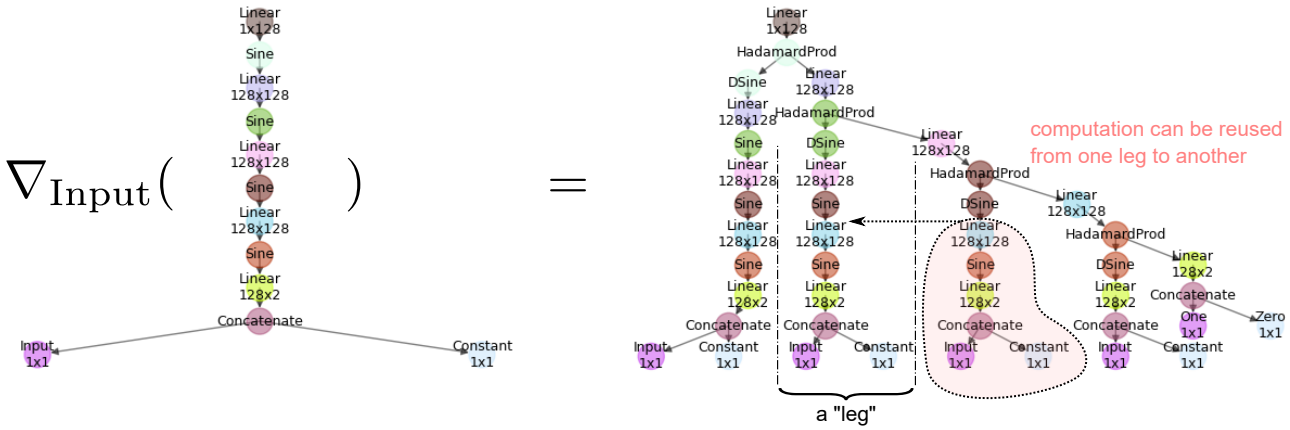

Figure 1. Visualizations of integral and grad networks generated with the AutoInt graph implementation. Directed acyclic graphs corresponding to the integral network (left) and grad network (right) consist of computational nodes and their dependencies. Here, arrows point from a node to each of its dependencies, indicating which node should be computed first. The grad network has a tree-like structure, and computes the partial derivative of the integral network with respect to one of its inputs. Nodes within the "legs" of the grad network appear multiple times, thus these computations can be re-used during a forward pass in order to improve performance.

topological ordering. The lexicographic ordering is defined by the order in which nodes were created during differentiation of the integral network: nodes created last appear first in the ordering. Thus the evaluation of the grad network in forward mode proceeds by calling each node in this lexicographic-topological ordering. Nodes save their computation in a cumulative fashion; as the legs of the network are computed, the last results are kept to be reused in other legs.

Training the grad network. The weights of the grad network are trained with backpropagation. During the evaluation of the grad network, computations are also saved for the backward pass performed by the backpropagation algorithm. The weights of the grad network are then updated with Adam [1], a variant of SGD.

3.2. Implementation and Results

We implemented AutoInt in Python and PyTorch [7], and we used the Python Networkx library to maintain the data structures forming the backend of our computational graphs. At the core of our AutoInt implementation is a custom AutoDiff tool that acts on computational nodes defined by the framework. After an integral network is built using the computational nodes, it is parsed by our AutoDiff tool, and the grad network is created. Importantly, each of our computational nodes wraps a Pytorch module, enabling parameter sharing within and between the networks. After the networks are instantiated, it is convenient to use the Pytorch AutoDiff to perform backpropagation and weight updates during training. Reassembling the integral network is trivial due to the weight sharing mechanism; the integral and grad network parameters always match as they are updated during training.

AutoInt can also be directly implemented in Pytorch, though with relatively severe performance penalties. To train the grad network, one would perform the following steps during each training iteration: (1) compute the output of the integral network, (2) use AutoDiff to calculate the derivative of the output with respect to the input variable of integration, and (3) perform backpropagation on a loss calculated using the derivative. This procedure is inefficient because, compared to our framework, it requires an extra forward pass through the integral network, and it requires re-assembling the grad network at every training iteration. Our AutoInt implementation compares favorably to this direct PyTorch implementation. We measure a savings of over 15% in GPU memory, and a more than 1.8x speedup in the number of training iterations per second for the volume rendering task described in the main paper.

4. Supplemental Results

4.1. Sparse-View Computed Tomography

We include supplemental results for the sparse-view computed tomography task described in the main paper. Here, we train a grad network on sparse angular projections (i.e., a subsampled sinogram) of a 2-dimensional phantom. After training, the integral network is evaluated for all projection angles to inpaint the missing regions of the sinogram. As shown in Figure 2, we find that using the Swish non-linearity in the integral network results in the best generalization performance in term of inpainting the unseen projections. SIREN [8], which uses the sine nonlinearity, performs well in the densely supervised case, but performs increasingly degrades for inpainting the sinograms for $8\times$ and $16\times$ angular subsampling. We also show the performance of ReLU and Softplus. Both of these non-linearities

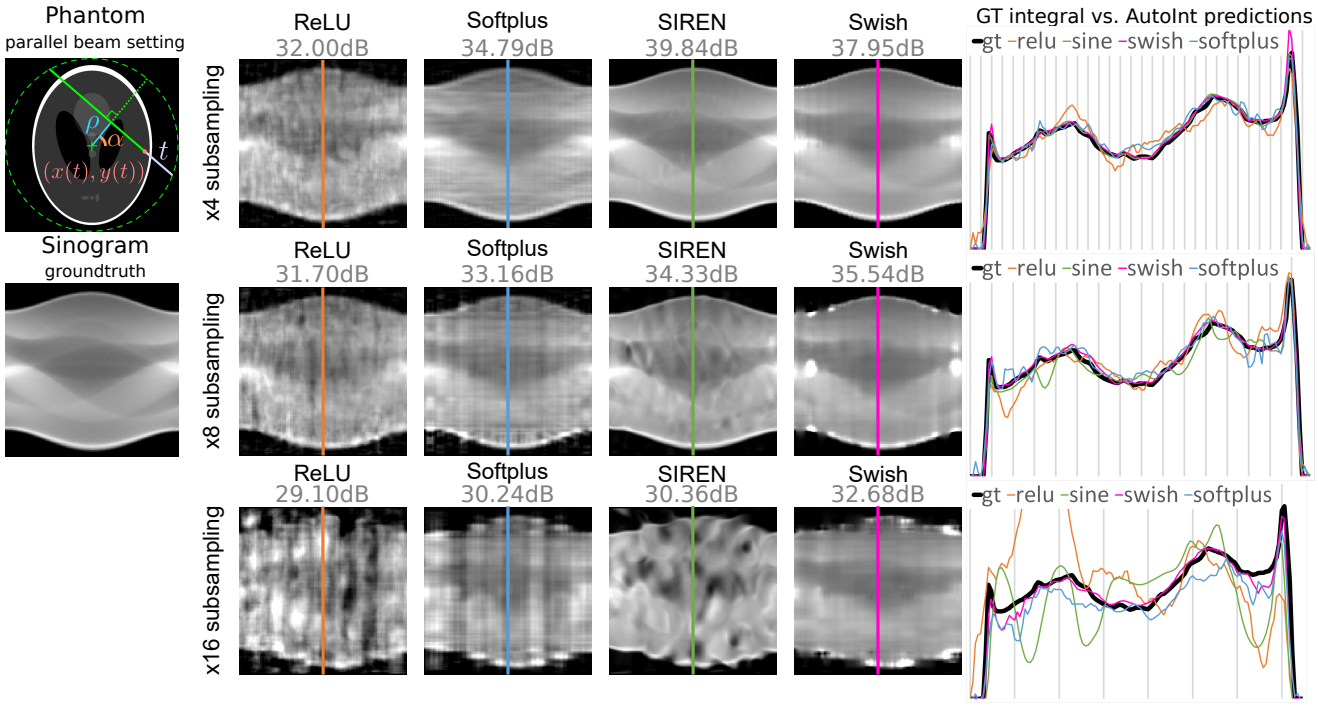

Figure 2. Supplemental results of AutoInt for computed tomography. Left: illustration of the parameterization. Center: sinograms computed with the integral networks using different nonlinear activation functions. The ground truth (GT) sinogram is subsampled in angle by 4× (top), 8× (middle), and 16× (bottom). The optimized networks are used to interpolate the missing measurements. Using the Swish activation performs best in these experiments. Right: 1D scanlines of the sinogram centers shows the interpolation behavior of each method for each subsampling level.

produce relatively noisy inpainted results. ReLU is particularly unsuited to learning this representation, as its derivative, which appears in the grad network, has a zero-valued derivative almost everywhere.

4.2. Supervising the Integral Network

In the main paper, we use AutoInt to supervise the grad network and then reconstruct the integral network. It is also possible to directly supervise the integral network, Φ_θ, to learn an antiderivative. One way to do this is with a provided dataset of definite integrals. For example, consider we wish to learn an antiderivative $F(x)$ over an interval $[a, b]$. In this case, we can supervise the integral network directly over definite integrals $F(x_2) - F(x_1) = \int_{x_1}^{x_2} f(x)\mathrm{d}x$ for $x_1, x_2 \in [a, b]$. Here, the integral network is trained to minimize a loss function of the following form.

$$\arg\min_\theta \|[\Phi_\theta(x_2) - \Phi_\theta(x_1)] - [F(x_2) - F(x_1)]\|_2^2. \quad (16)$$

We demonstrate training a neural network with Swish nonlinearities to learn a sigmoid function $F(x) = 1/(1 + e^{-x})$ from a dataset of randomly sampled definite integrals as described by Equation 16. The resulting fit is shown in Figure 3. The network accurately fits the sigmoid function up to a scalar constant (which we remove in the visualization of Figure 3).

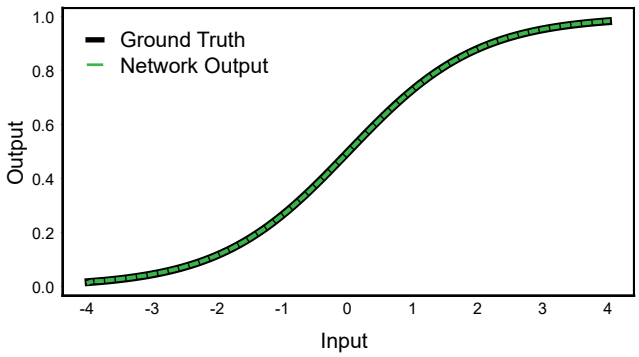

Figure 3. Supervising the integral network with definite integrals. We set an antiderivative F to be equal to a sigmoid function and supervise an integral network with values of definite integrals as described by Equation 16. The network recovers the antiderivative F up to a scalar constant and is compared to ground truth (we remove the offset for visualization).

4.3. Results on Captured Data

In Figure 4, we include supplemental results using real captured data from DeepVoxels [9]. The scenes were trained on half of the randomly sampled images of the pro-

vided RGB data and are shown here for a held out test set. Both AutoInt results using 8 and 32 sections achieve similarly high image quality on these captured scenes.

We also show results from the Local Light Field Fusion (LLFF) datasets [4] and qualitative comparisons between NeRF [5] and AutoInt in Figure 5. The LLFF datasets consist of captured, forward-facing image data, and we train and evaluate all images at a resolution of 378 by 504 pixels. We train with a batch size of one and otherwise use the same training parameters as the Blender datasets as described in the main text. Following the LLFF authors, we partition the captured images so that 1/8 of the images are used for the test set. In Table 1, we show quantitative comparisons between AutoInt and NeRF for the LLFF datasets.

4.4. Synthetic Blender Dataset Scenes

In Table 2 we provide additional quantitative evaluations using peak signal-to-noise ratio (PSNR), the structural similarity index measure (SSIM) [10], and the learned perceptual image patch similarity (LPIPS) metric [11].

References

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proc. ICLR*, 2014. 3

[2] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph. (SIGGRAPH)*, 38(4), 2019. 8

[3] Nelson Max. Optical models for direct volume rendering. *IEEE Trans. Vis. Comput. Graph*, 1(2):99–108, 1995. 2

[4] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Trans. Graph. (SIGGRAPH)*, 38(4), 2019. 5, 6, 7

[5] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proc. ECCV*, 2020. 5, 7, 8

[6] Ulrich Mutze. The fundamental theorem of calculus in $\mathbb{R}^n$. Technical report, 2010. https://web.ma.utexas.edu/mp_arc/c/04/04-165.pdf. 1

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019. 3

[8] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *Proc. NeurIPS*, 2020. 3

[9] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. DeepVoxels: Learning persistent 3D feature embeddings. In *Proc. CVPR*, 2019. 4, 6

[10] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. Image Process.*, 13(4):600–612, 2004. 5

[11] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proc. CVPR*, 2018. 5

Figure 4. Supplemental results of AutoInt for real captures. We show qualitative results of AutoInt with 8 and 32 sections on the *Statue* and the *Globe* scenes used in DeepVoxels [9].

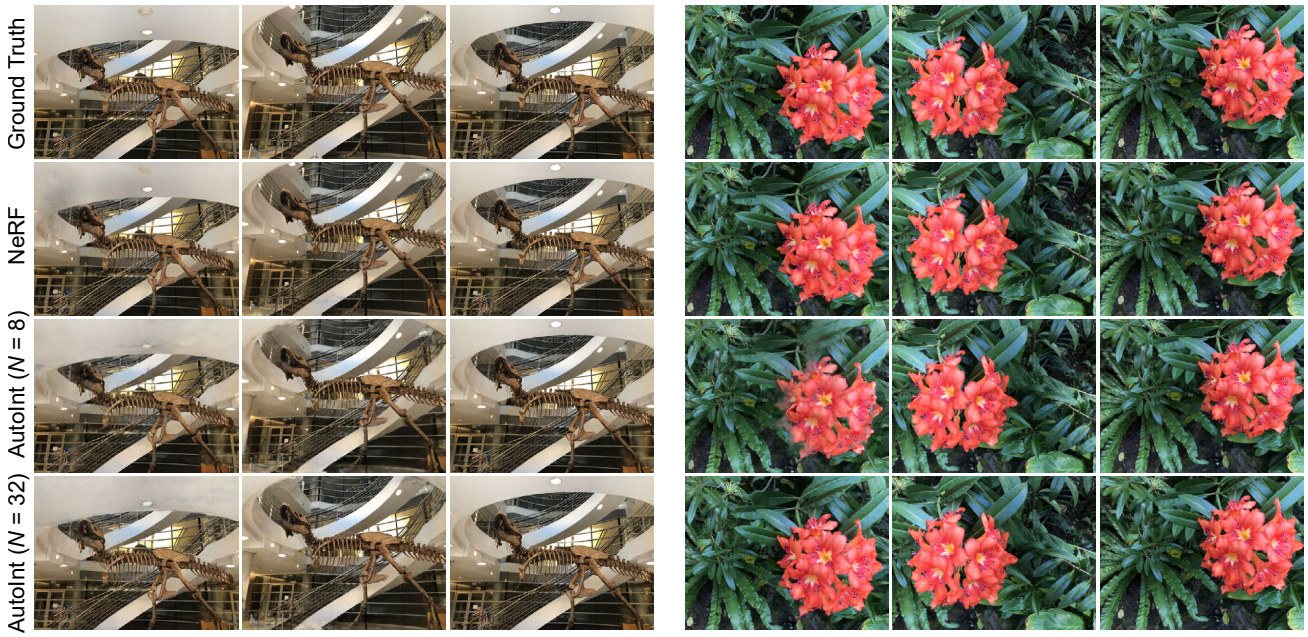

Figure 5. Supplemental results of AutoInt for real captures from the Local Light Field Fusion datasets [4]. We show qualitative results of AutoInt with 8 and 32 sections on the *T-Rex* and *Flower* scenes.

	PSNR↑							
	Room	Fern	Leaves	Fortress	Orchids	Flower	T-Rex	Horns
NeRF [5]	**33.60**	**26.92**	**22.50**	**32.94**	**21.37**	**28.57**	**28.26**	**29.26**
AutoInt (N=8)	28.33	22.11	19.61	28.63	16.85	26.65	24.90	26.01
AutoInt (N=16)	29.97	23.29	18.78	<u>29.53</u>	<u>17.71</u>	27.60	25.58	26.72
AutoInt (N=32)	<u>30.72</u>	<u>23.51</u>	<u>20.84</u>	28.95	17.30	<u>28.11</u>	<u>27.18</u>	<u>27.64</u>

	SSIM↑							
	Room	Fern	Leaves	Fortress	Orchids	Flower	T-Rex	Horns
NeRF [5]	**0.980**	**0.903**	**0.851**	**0.962**	**0.800**	**0.931**	**0.953**	**0.947**
AutoInt (N=8)	0.941	0.771	0.745	0.896	0.560	0.882	0.888	0.880
AutoInt (N=16)	0.954	0.802	0.712	<u>0.914</u>	<u>0.607</u>	0.903	0.904	0.894
AutoInt (N=32)	<u>0.966</u>	<u>0.810</u>	<u>0.795</u>	0.910	0.583	<u>0.917</u>	<u>0.931</u>	<u>0.908</u>

	LPIPS↓							
	Room	Fern	Leaves	Fortress	Orchids	Flower	T-Rex	Horns
NeRF [5]	**0.038**	**0.085**	**0.103**	**0.024**	**0.108**	**0.057**	**0.049**	**0.058**
AutoInt (N=8)	0.110	<u>0.276</u>	0.171	0.092	0.313	0.113	0.123	0.213
AutoInt (N=16)	0.102	0.283	0.218	<u>0.086</u>	<u>0.268</u>	0.090	0.107	0.202
AutoInt (N=32)	<u>0.075</u>	0.277	<u>0.156</u>	0.107	0.302	<u>0.075</u>	<u>0.080</u>	<u>0.177</u>

Table 1. Per-scene quantitative results calculated across the test sets of the Local Light Field Fusion datasets [4].

	PSNR↑							
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship
NeRF [5]	**33.00**	**25.01**	**30.13**	**36.18**	**32.54**	**29.62**	**32.91**	**28.65**
NV [2]	28.33	22.58	24.79	30.71	26.08	24.22	27.78	23.93
AutoInt (N=8)	25.60	20.78	22.47	32.33	25.09	25.90	28.10	24.15
AutoInt (N=16)	25.65	21.30	23.95	31.28	25.48	28.05	28.36	24.26
AutoInt (N=32)	25.82	22.02	25.51	31.84	27.26	28.58	28.42	25.18

	SSIM↑							
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship
NeRF [5]	**0.967**	**0.925**	**0.964**	**0.974**	**0.961**	**0.949**	**0.980**	0.856
NV [2]	0.916	0.879	0.910	0.944	0.880	0.888	0.946	0.784
AutoInt (N=8)	0.928	0.861	0.898	**0.974**	0.900	0.930	0.948	0.852
AutoInt (N=16)	0.925	0.869	0.909	0.971	0.905	0.947	0.951	0.853
AutoInt (N=32)	0.926	0.885	0.926	0.973	0.929	0.953	0.951	**0.869**

	LPIPS↓							
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship
NeRF [5]	**0.046**	**0.091**	**0.044**	0.121	**0.050**	**0.063**	**0.028**	**0.206**
NV [2]	0.109	0.214	0.162	0.109	0.175	0.130	0.107	0.276
AutoInt (N=8)	0.141	0.224	0.148	**0.080**	0.175	0.136	0.131	0.323
AutoInt (N=16)	0.149	0.221	0.139	0.095	0.171	0.110	0.130	0.320
AutoInt (N=32)	0.149	0.209	0.109	0.088	0.135	0.100	0.127	0.295

Table 2. Per-scene quantitative results calculated across the test sets of the synthetic Blender datasets. These simulated scenes contain challenging geometries and reflectance properties.