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

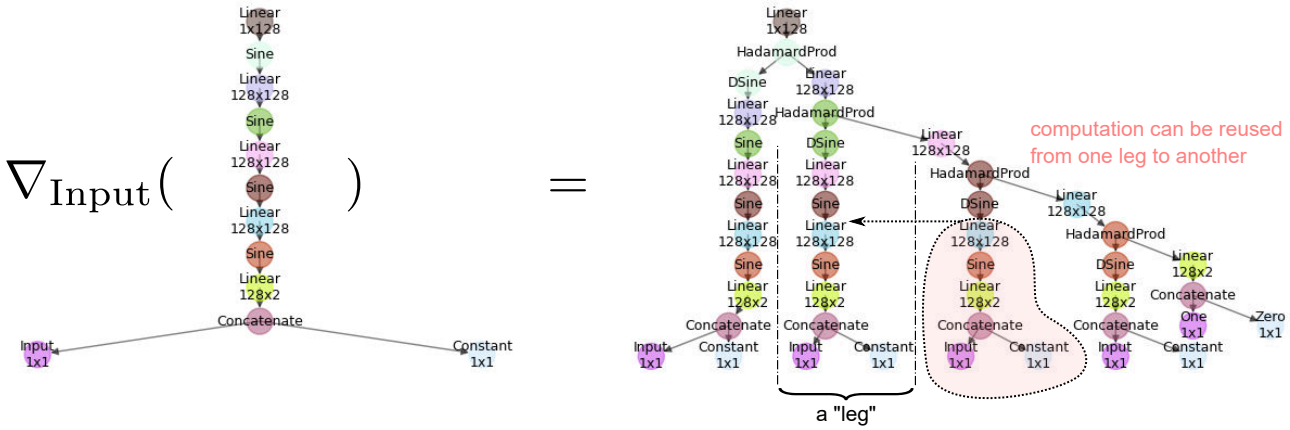

Figure 1. Visualizations of integral and grad networks generated with the AutoInt graph implementation. Directed acyclic graphs corresponding to the integral network (left) and grad network (right) consist of computational nodes and their dependencies. Here, arrows point from a node to each of its dependencies, 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

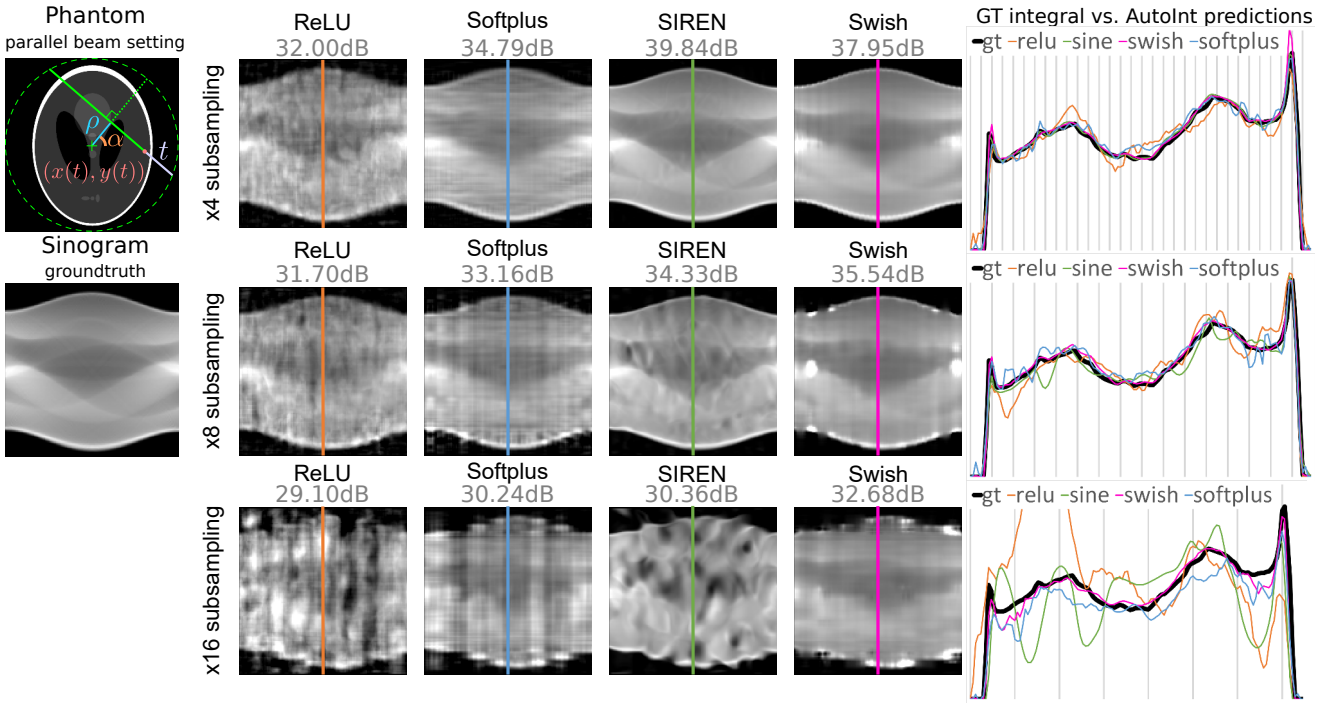

Figure 2. Supplemental results of AutoInt for computed tomography. Left: illustration of the parameterization. Center: sinograms computed with the integral networks using different nonlinear activation functions. The ground truth (GT) sinogram is subsampled in angle by 4× (top), 8× (middle), and 16× (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).

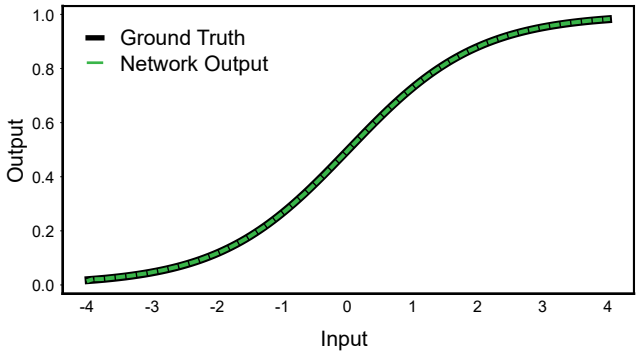

Figure 3. Supervising the integral network with definite integrals. We set an antiderivative F to be equal to a sigmoid function and supervise an integral network with values of definite integrals as described by Equation 16. The network recovers the antiderivative F up to a scalar constant and is compared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

We show extended results that evaluate the effect of the sampling network for a range of piecewise sections in the approximate VRE in Figure 6. Additional qualitative comparisons on the synthetic Blender datasets are provided in Figure 7, where we compare our method to Neural Volumes [2] and NeRF [5]. Finally, 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. 5, 8, 9

[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, 9

[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].

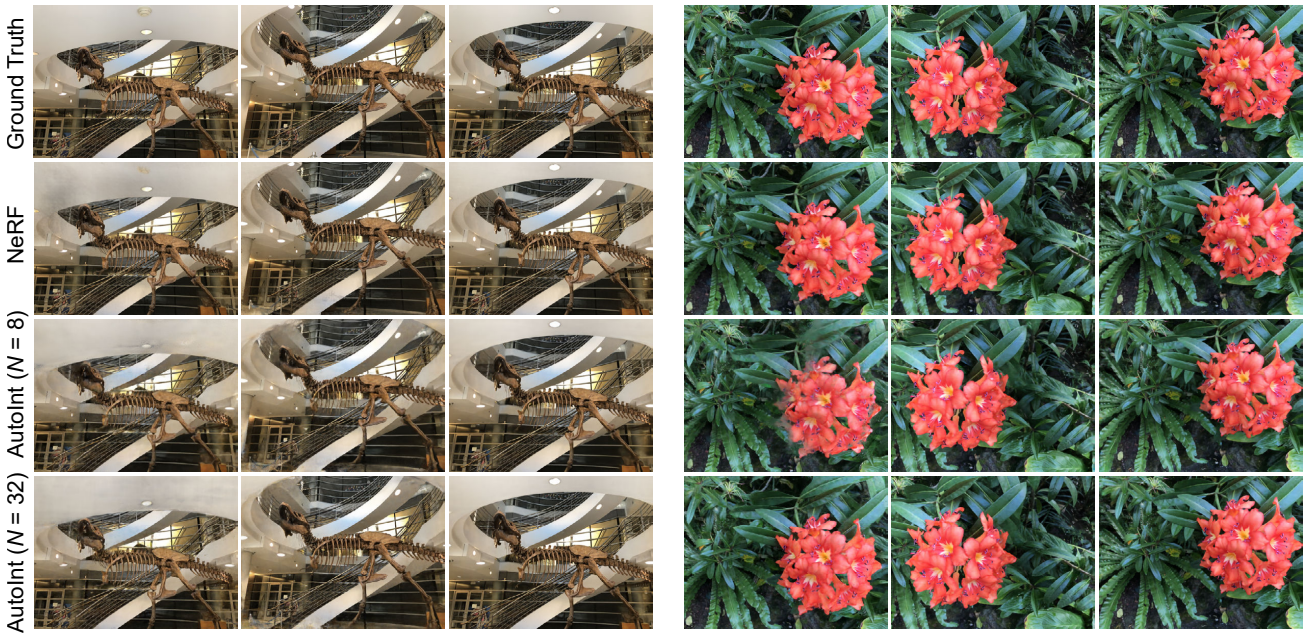

Figure 5. Supplemental results of AutoInt for real captures from the Local Light Field Fusion datasets [4]. We show qualitative results of AutoInt with 8 and 32 sections on the *T-Rex* and *Flower* scenes.

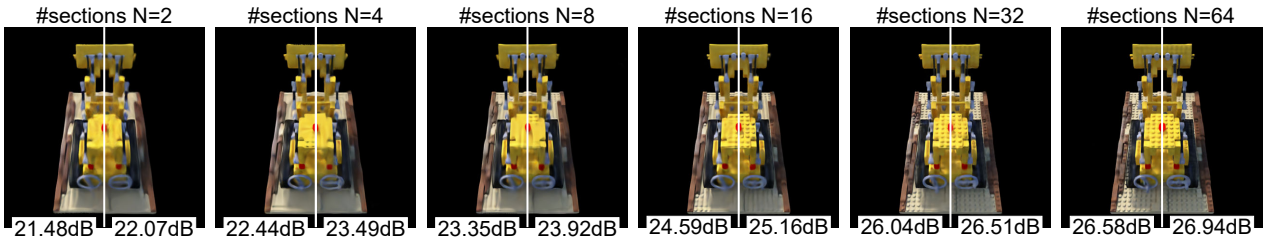

Figure 6. Ablation studies. A view of the *Lego* scene is shown with a varying number of intervals ($N = \{2, 4, 8, 16, 32, 64\}$) without (left half of the images) and with (right half) the sampling network. PSNR is computed on the 200 test set views.

	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].

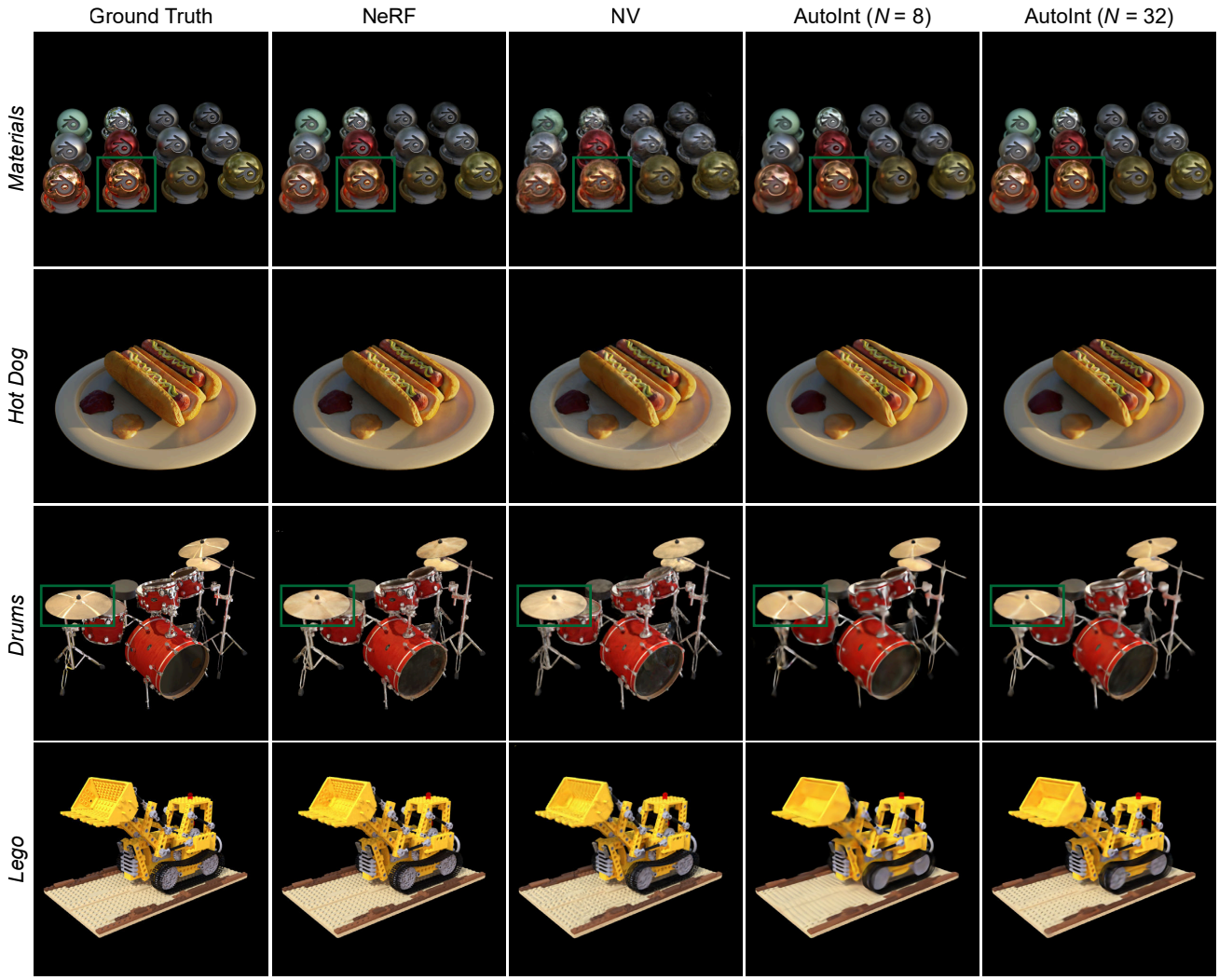

Figure 7. Qualitative results. We compare the performance of Neural Volumes [2] and NeRF [5] to AutoInt using $N = 8$ and $N = 32$ in our approximate volume rendering equation. AutoInt accurately captures view-dependent effects like specular reflections (green boxes) and reduces render times by greater than $10\times$ relative to NeRF, though with some reduction in overall image quality.

	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.