Learning Neural Light Fields with Ray-Space Embedding

Supplementary Material

Benjamin Attal* Carnegie Mellon University
Jia-Bin Huang Meta
Michael Zollhöfer Reality Labs Research
Johannes Kopf Meta
Changil Kim Meta

https://neural-light-fields.github.io

Abstract

In this document, we provide additional implementation details, analysis on our embedding network, and experimental results. See the accompanying web page for more visual results.

A. Implementation Details

Our code is written entirely in python, using the PyTorch Lightning framework [2]. The design of our codebase was inspired by [5]. Below we include additional implementation details for our method for both the dense dataset (Stanford Light Field [7]) and sparse forward facing datasets (Real Forward-Facing [4] and Shiny [8]). We also include information about the NeRF [4], NeX [8], and AutoInt [3] baselines.

A.1. Stanford Light Field Dataset

Calibration information is not provided with the Stanford Light Field dataset, apart from the (x, y) positions of all images on the camera plane π^{xy}. All Stanford Light Fields are parameterized with respect to a plane π^{uv} that approximately cuts through the center of each scene. Thus, a pixel coordinate in an image corresponds the location that a ray intersects π^{uv}. We heuristically set the location of π^{xy} to $z = -1$, and the location of the object plane π^{uv} to $z = 0$. We scale the camera positions so that they lie between $[-0.25, 0.25]$ in both x and y, and the pixel coordinates on π^{uv} so they lie between $[-1, 1]$. The camera positions, and the vector from the camera origin to the location of the pixel on the object plane then comprise our ray origins and directions.

For our method, we take camera coordinates and object plane coordinates, which correspond to intersections of the rays with π^{xy} and π^{uv}, as our initial two-plane ray parameterization.

For NeRF/NeX, we set the near distance to 0.5 and the far distance to 2 for all scenes, except for the Knights scene, where we set the near distance to 0.25. Note that, as defined, the scene coordinates may differ from the true (metric) world space coordinates by a projective transform. However, multi-view constraints still hold (intersecting rays remain intersecting, epipolar lines remain epipolar lines), and thus NeRF is able to learn an accurate volume.

A.2. Real Forward-Facing Dataset

For the Real Forward-Facing Dataset, we perform all experiments in NDC space. For our subdivided model, we re-parameterize each ray $\mathbf{r}$ within a voxel v by first transforming space so that the voxel center lies at the origin. We then intersect the transformed ray with the voxel's front and back planes, and take the xy coordinates of these intersections, which lie within the range $[-\textit{voxel width}, +\textit{voxel width}]$ in all dimensions as the parameterization.

A.3. Shiny Dataset

Our procedure for evaluating on Shiny [8] is identical to the Real Forward-Facing dataset. We perform experiments in NDC space and use a 32^3 voxel grid that covers all of NDC space on all scenes *except* CD/Lab where we use a coarser 4^3 grid.

A.4. NeRF and AutoInt Baselines

For the Stanford Dataset, we train NeRF for 400k iterations with a batch size of 1,024 for all scenes. For the Real Forward-Facing dataset, AutoInt does not provide pretrained models and training with their reported parameters on a V100 GPU with 16GB of memory leads to out-of-memory errors. They also do not provide multi-GPU training code. As such, we report the quantitative metrics for their method and for NeRF published in their paper, which come from models trained at the same resolution (504×378), and using the same heldout views as ours on the Real Forward-Facing Dataset.

A.5. NeX Baseline

We train NeX on all datasets (Stanford, Undistorted RFF, Shiny), using their public codebase, for 4000 epochs on all scenes (the default number of epochs in their training script),

or about 36 hours each. We use their multi-GPU training code to split training over 2 16GB V100 GPUs. We note that NeX can perform real-time rendering after discretizing their view dependent basis functions into 400×400 textures. However the NeX codebase does not include evaluation code for their real-time renderable MPIs, and thus the numbers for NeX in Table 1 of the main paper are reported *before* baking. While this presumably leads to an increase in quality, it takes a longer time to render images, hence the smaller FPS scores in Table 1.

B. Importance of Light Field Parameterization

Here, we expand on the discussion in Section 4 of the main paper and describe why our light field parameterization is crucial for enabling good view interpolation. Let the two planes in the initial two plane parameterization be π^{xy} and π^{uv}, with local coordinates (x,y) and (u,v). In addition, let us denote the plane of the textured square as π^{st} with local coordinates (s,t), and assume that it is between π^{xy} and π^{uv}. Assume, without loss of generality, that the depth of π^{xy} is 0, and suppose the depths of π^{st} and π^{uv} are z_{st} and z_{uv} respectively.

For a ray originating at $(\hat{x},\hat{y})$ on π^{xy} and passing through $(\hat{s},\hat{t})$ on π^{st}, we can write (by similar triangles):

$$\hat{s} - \hat{x} = (\hat{u} - \hat{x})\frac{z_{st}}{z_{uv}}, \tag{1}$$

$$\hat{t} - \hat{y} = (\hat{v} - \hat{y})\frac{z_{st}}{z_{uv}}, \tag{2}$$

which gives

$$\hat{u} = \frac{\hat{x}(z_{st} - z_{uv}) + \hat{s}}{z_{st}}, \tag{3}$$

$$\hat{v} = \frac{\hat{y}(z_{st} - z_{uv}) + \hat{t}}{z_{st}}. \tag{4}$$

Recall that the positional encoding of the 4D input parameterization $\gamma(\hat{x},\hat{y},\hat{u},\hat{v})$ will be fed into the light field network. Thus, the network will produce interpolation kernels aligned with $\hat{u}$ and $\hat{v}$. However for perfect interpolation, we would like the output of the light field network to only depend on $(\hat{s},\hat{t})$, or for the interpolation kernels to be aligned with $(\hat{s},\hat{t})$. It can be observed in equations (3) and (4) that the greater the distance $(z_{st} - z_{uv})$, the larger the difference between $(\hat{u},\hat{v})$ and $(\hat{s},\hat{t})$, and the *less aligned* the interpolation kernels become with $(\hat{s},\hat{t})$.

On the other hand, by learning a re-parameterization of the light field, such that π^{uv} is moved towards π^{st} (i.e. reducing the distance $(z_{st} - z_{uv})$), we align the color network's interpolation kernels with $(\hat{s},\hat{t})$. As in the feature-embedding approach, the finite capacity of the light field MLP will drive the embedding network to learn to map rays intersecting the same point on the textured square to the same point in the latent space—and thus will drive learning of an optimal re-parameterization that leads to good interpolation.

Table B.1. **Empirical validation of parameterization.** We calculate PSNR, SSIM, and LPIPS for our model without embedding trained on different initial parameterizations.

π^{uv} location	PSNR↑	SSIM↑	LPIPS↓
$z = 0$	**29.631**	**0.937**	**0.059**
$z = 1$	<u>24.411</u>	<u>0.848</u>	<u>0.096</u>
$z = 3$	21.491	0.790	0.146

B.1. Empirical Validation

We claim that quality of a *baseline* neural light field is correlated with $(z_{st} - z_{uv})$. In order to support this claim, we perform a set of simple experiments with parameterization. In particular, we choose the *Amethyst* scene from the Stanford Light Field dataset [7], which has very little depth variation. As described in Section A.1, each Stanford light field has the object plane π^{st} at $z = 0$. We re-parameterize the input light field for π^{uv} at $z = 0, 1, 3$, train our model *without* an embedding network, and report validation metrics, as well as showing reconstructed epipolar images. See Figure B.1 and Table B.1.

As hypothesized, the models trained with the object plane closer to $z = 0$ perform better qualitatively and quantitatively. While this is, perhaps, an obvious result, we believe that it is an important one. In particular, it means that light fields with worse initial parameterization are more difficult to learn, and supports the result that *learning* re-parameterization via local affine transforms vastly improves neural light field interpolation quality.

B.2. Non-Axis-Aligned Positional Encoding

Positional encoding with non-axis-aligned interpolation kernels (e.g. Gaussian positional encoding [6]) could be seen as a potential way around the issues discussed above. However, it is important to note that positional encoding with *any fixed set* of interpolation kernels will break down for certain scenes with different depths/depth ranges. For example, this is the case when the interpolation kernels do not align with the light field's color level sets, or there are not enough interpolation kernels to represent high frequencies for particular directions in the light field. In other words, no single-set of interpolation kernels works for all scenes, and ray-space embedding effectively tunes the interpolation per-scene/per-region in ray space,

C. Embedding Visualization

In Figure C.2 we visualize predicted views, predicted EPIs, and the embedded ray-space given as input to the color network for:

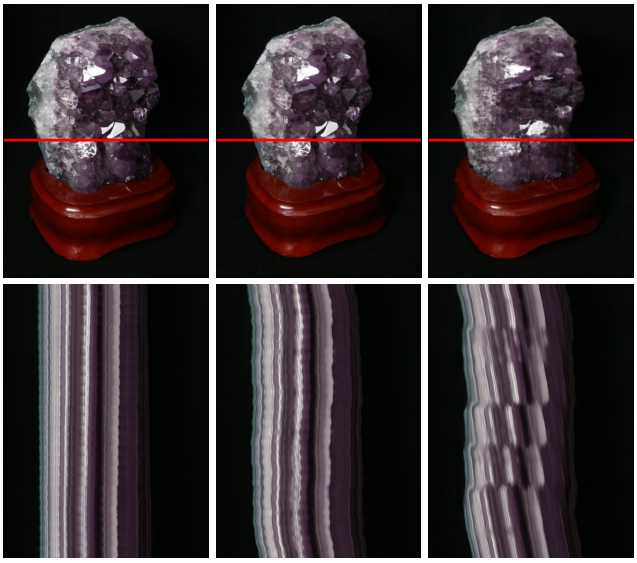

Figure B.1. **Effect of initial parameterization.** The *top* row shows predicted images, *bottom* predicted EPIs. Reconstruction becomes progressively worse for a worse initial parameterization (π^{uv} moving further and further from the object plane at $z = 0$).

1. The baseline approach
2. Feature space embedding
3. Local affine transform embedding

For the embedding visualization, RGB colors denote the first three principal components of embedded ray-space.

Note the wiggling artifacts in the EPI predicted without embedding (*left*) . As discussed above, the baseline approach produces axis-aligned interpolation kernels which are ill-suited for interpolating slanted color level sets in the EPI.

The feature embedding network (*center*) learns what is essentially a set of texture coordinates for the object. It registers disparate rays that hit the same 3D points, and interpolates views fairly well, but does not guarantee multi-view consistency as the embedding network is still under-constrained for unobserved views.

Although the transforms themselves are not visualized (it is ray-space *after* applying the transforms that is visualized), the local affine transform network (*right*) predicts a set of transforms that are *largely constant*. This is because the depth range of the scene is limited, and a small set of re-parameterizations works well for all of ray-space. As the network learns a simpler output signal, it naturally interpolates more effectively, even to unobserved views. While the results for feature embedding and local affine transform embedding look similar in Figure C.2, we encourage readers to visit our web-page. It is far easier to see artifacts of the feature embedding approach in video form.

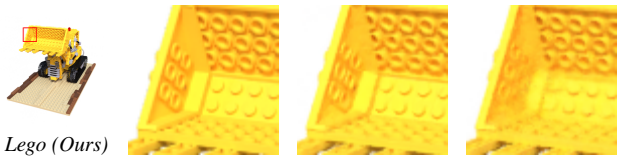

Figure C.1. We achieve 29.14 dB on the *Lego* sequence from the NeRF Synthetic dataset [4] with a 32^3 voxel grid at 800×800 pixel resolution, compared to 27.26 dB for AutoInt [3] with 32 sections, and 32.54 dB for NeRF.

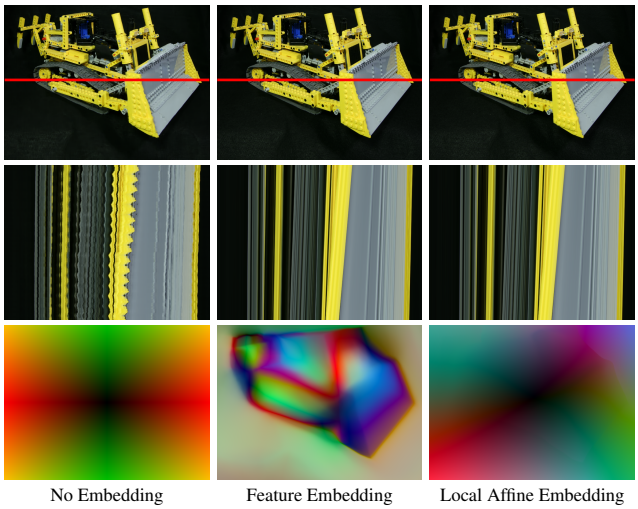

Figure C.2. **Effect of different embeddings**. The *top* row shows predicted images, *middle* predicted EPIs, and *bottom* a visualization of the embedding space for each method.

D. More Experimental Results

We show per-scene metrics in Tables D.1, D.2, D.3, and D.4. Additionally, we highly encourage readers to visit our project webpage, which contains image comparisons for every scene and video comparisons for a select few scenes.

360 ° Scenes. In Figure C.1 we show preliminary results for our 32^3 subdivided model with Plücker parameterization applied to the *Lego* scene in the NeRF Synthetic [4] dataset. We use the same evaluation protocol as in NeRF for this scene. Our PSNR is slightly worse than NeRF, but better than AutoInt for the same grid resolution. Additionally, in some regions, we are able to better recover fine-grained texture on the *Lego* model.

Student-Teacher Training. We additionally provide results in Tables D.1 for our method when the input data is augmented with a $10x10$ grid of renderings from a fully trained NeRF. We label this method as "Ours (w/t)," or our method with "student-teacher" training. With this approach, our method outperforms NeRF quantitatively in terms of PSNR, but at the cost of increased training time.

References

[1] Mojtaba Bemana, Karol Myszkowski, Hans-Peter Seidel, and Tobias Ritschel. X-fields: implicit neural view-, light- and time-image interpolation. *ACM Trans. Graph.*, 39(6):257:1–257:15, 2020. 5

[2] William Falcon and The PyTorch Lightning team. PyTorch Lightning, 3 2019. 1

[3] David B Lindell, Julien NP Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural volume rendering. In *CVPR*, 2021. 1, 3, 5

[4] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 1, 3, 5

[5] Chen Quei-An. Nerf_pl: a pytorch-lightning implementation of nerf, 2020. 1

[6] Matthew Tancik, Pratul P Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. In *NeurIPS*, 2020. 2

[7] Bennett Wilburn, Neel Joshi, Vaibhav Vaish, Eino-Ville Talvala, Emilio R. Antúnez, Adam Barth, Andrew Adams, Mark Horowitz, and Marc Levoy. High performance imaging using large camera arrays. *ACM Trans. Graph.*, 24(3):765–776, 2005. 1, 2, 5

[8] Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. NeX: Real-time view synthesis with neural basis expansion. In *CVPR*, 2021. 1, 5

Table D.1. Per-scene breakdown results from NeRF's Real Forward-Facing dataset [4].

Scene	PSNR↑				SSIM↑				LPIPS↓			
	NeRF [4]	AutoInt [3]	Ours	Ours (w/ t)	NeRF [4]	AutoInt [3]	Ours	Ours (w/ t)	NeRF [4]	AutoInt [3]	Ours	Ours (w/ t)
Fern	**26.92**	23.51	24.25	26.06	**0.903**	0.810	0.850	0.893	**0.085**	0.277	0.114	0.104
Flower	28.57	28.11	28.71	**28.90**	0.931	0.917	**0.934**	**0.934**	0.057	0.075	**0.038**	0.053
Fortress	**32.94**	28.95	31.46	32.60	**0.962**	0.910	0.954	0.961	**0.024**	0.107	0.027	0.028
Horns	29.26	27.64	**30.12**	29.76	0.947	0.908	**0.955**	0.952	0.058	0.177	**0.044**	0.062
Leaves	**22.50**	20.84	21.82	22.27	0.851	0.795	0.847	**0.855**	0.103	0.156	**0.086**	0.104
Orchids	**21.37**	17.30	20.29	21.10	**0.800**	0.583	0.766	0.794	0.108	0.302	**0.103**	0.113
Room	33.60	30.72	33.57	**34.04**	0.980	0.966	0.979	**0.981**	0.038	0.075	**0.037**	**0.037**
T-rex	28.26	27.18	**29.41**	28.80	0.953	0.931	**0.959**	**0.959**	0.049	0.080	**0.034**	0.040

Table D.2. Per-scene breakdown results from the *Undistorted* Real Forward-Facing dataset used in NeX [8]

Scene	PSNR↑		SSIM↑		LPIPS↓	
	NeX [8]	Ours	NeX [8]	Ours	NeX [8]	Ours
Fern	**26.46**	24.49	**0.913**	0.856	**0.068**	0.107
Flower	**29.39**	28.93	**0.947**	0.937	0.041	**0.033**
Fortress	**32.31**	31.32	**0.963**	0.955	**0.024**	0.026
Horns	29.81	**29.88**	**0.959**	0.952	**0.039**	0.050
Leaves	**22.66**	21.62	**0.879**	0.845	**0.082**	0.082
Orchids	**20.51**	19.93	**0.792**	0.754	**0.096**	0.109
Room	**33.40**	33.24	**0.979**	0.978	**0.033**	0.036
T-rex	29.36	**29.44**	**0.965**	0.963	0.037	**0.032**

Table D.3. Per-scene breakdown results from NeX's Shiny Dataset [8]

Scene	PSNR↑		SSIM↑		LPIPS↓	
	NeX [8]	Ours	NeX [8]	Ours	NeX [8]	Ours
CD	31.92	**35.44**	0.971	**0.980**	**0.028**	**0.014**
Crest	**24.78**	24.48	**0.870**	0.858	**0.051**	0.052
Food	**25.61**	25.21	**0.905**	0.885	**0.048**	0.053
Giants	**28.50**	27.99	**0.946**	0.930	**0.038**	0.039
Lab	31.20	**34.39**	0.965	**0.982**	0.031	**0.013**
Pasta	**23.21**	22.11	**0.915**	0.890	**0.045**	0.065
Seasoning	**31.07**	29.48	**0.970**	0.957	**0.028**	0.045
Tools	**29.86**	28.90	**0.974**	0.968	**0.018**	0.022

Table D.4. Per-scene breakdown results from the Stanford Light Field dataset [7]

Scene	PSNR↑				SSIM↑				LPIPS↓			
	NeRF [4]	X-Fields [1]	NeX [8]	Ours	NeRF [4]	X-Fields [1]	NeX [8]	Ours	NeRF [4]	X-Fields [1]	NeX [8]	Ours
Amethyst	39.746	37.232	39.062	**40.120**	0.984	0.982	0.983	**0.985**	0.026	0.032	0.023	**0.019**
Beans	**42.519**	40.911	41.776	41.659	**0.9944**	0.9931	0.9938	0.9933	0.014	0.017	0.016	**0.012**
Bracelet	36.461	34.112	34.888	**36.586**	0.9909	0.9857	0.988	**0.9913**	0.0094	0.0260	0.0152	**0.0087**
Bulldozer	38.968	37.350	38.131	**39.389**	0.983	0.986	0.985	**0.987**	0.063	0.032	0.027	**0.024**
Bunny	43.370	42.251	42.722	**43.591**	0.9892	0.9894	0.9885	**0.9900**	0.029	0.022	0.036	**0.013**
Chess	**41.146**	37.996	39.938	40.910	0.9915	0.9882	0.9910	**0.9920**	0.028	0.034	0.020	**0.016**
Flowers	37.910	37.590	36.982	**39.951**	0.977	0.981	0.978	**0.984**	0.076	0.035	0.036	**0.030**
Knights	**35.978**	31.491	35.678	34.591	**0.986**	0.974	0.986	0.982	**0.0142**	0.0501	0.0168	0.0143
Tarot (Small)	34.221	30.830	33.134	**36.046**	0.982	0.975	0.981	**0.989**	0.014	0.033	0.016	**0.006**
Tarot (Large)	**24.907**	24.154	22.487	24.904	0.910	0.893	0.833	**0.914**	0.059	0.074	0.117	**0.039**
Treasure	34.761	33.904	32.350	**37.465**	0.972	0.977	0.967	**0.982**	0.027	0.041	0.040	**0.019**
Truck	40.723	38.883	38.292	**41.440**	0.986	0.984	0.986	**0.989**	0.087	0.042	0.038	**0.033**