Sample-based Monte Carlo Denoising using a Kernel-Splatting Network

MICHAËL GHARBI, Adobe and MIT CSAIL
TZU-MAO LI, MIT CSAIL
MIIKA AITTALA, MIT CSAIL
JAAKKO LEHTINEN, Aalto University and NVIDIA
FRÉDO DURAND, MIT CSAIL

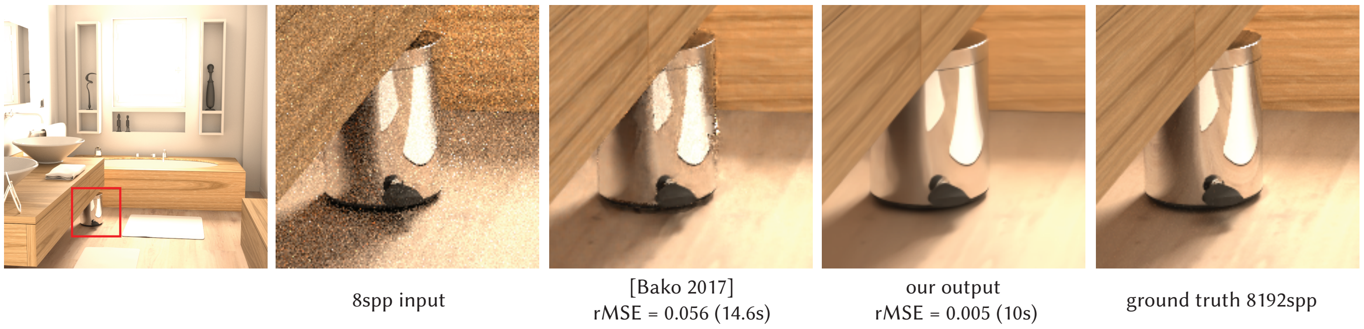

Fig. 1. State-of-the-art pixel-based Monte Carlo denoising algorithms (right) struggle with very noisy inputs rendered with a low sample count (left). Our method (middle) works with the *samples* directly, it uses a *splatting* approach, and is trained using deep learning. This makes it possible to appropriately handle various components of the illumination (indirect lighting, specular reflection, motion blur, depth of field, etc) more effectively.

Denoising has proven to be useful to efficiently generate high-quality Monte Carlo renderings. Traditional pixel-based denoisers exploit summary statistics of a pixel's sample distributions, which discards much of the samples' information and limits their denoising power. On the other hand, sample-based techniques tend to be slow and have difficulties handling general transport scenarios. We present the first convolutional network that can learn to denoise Monte Carlo renderings directly from the samples. Learning the mapping between samples and images creates new challenges for the network architecture design: the order of the samples is arbitrary, and they should be treated in a permutation invariant manner. To address these challenges, we develop a novel kernel-predicting architecture that *splats* individual samples onto nearby pixels. Splatting is a natural solution to situations such as motion blur, depth-of-field and many light transport paths, where it is easier to predict which pixels a sample contributes to, rather than a *gather* approach that needs to figure out, for each pixel, which samples (or nearby pixels) are relevant. Compared to previous state-of-the-art methods, ours is robust to the severe noise of low-sample count images (e.g. 8 samples per pixel) and yields higher-quality results both visually and numerically. Our approach retains the generality and efficiency of pixel-space methods while enjoying the expressiveness and accuracy of the more complex sample-based approaches.

Authors' addresses: Michaël Gharbi, Adobe and MIT CSAIL, mgharbi@adobe.com; Tzu-Mao Li, MIT CSAIL, tzumao@mit.edu; Miika Aittala, MIT CSAIL, miika@mit.edu; Jaakko Lehtinen, Aalto University and NVIDIA, jaakko.lehtinen@aalto.fi; Frédo Durand, MIT CSAIL, fredo@mit.edu.

CCS Concepts: • **Computing methodologies → Neural networks**; **Ray tracing**; *Image processing*.

Additional Key Words and Phrases: Monte Carlo denoising, deep learning, data-driven methods, convolutional neural networks

ACM Reference Format:
Michaël Gharbi, Tzu-Mao Li, Miika Aittala, Jaakko Lehtinen, and Frédo Durand. 2019. Sample-based Monte Carlo Denoising using a Kernel-Splatting Network. *ACM Trans. Graph.* 38, 4, Article 125 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322954

1 INTRODUCTION

Because Monte Carlo methods rely on stochastic point samples of an intricate integrand, they often suffer from noise. This motivates Monte Carlo denoising techniques, which broadly fall into two categories. *Sample-based* techniques keep track of the individual samples, while *pixel-based* methods work directly on the rendered image. Most methods operate in pixel space (e.g. [Bako et al. 2017; Bitterli et al. 2016]). In addition to the noisy input radiance, they usually exploit first and second order statistics of auxiliary buffers (like depth, normal, albedo, etc) [McCool 1999]. We argue that, in many lighting configurations, simple per-pixel aggregates can under-represent the complexity of the local light transport phenomena, in particular because the distribution is often multimodal.

We present a new Monte Carlo denoising technique that leverages the power of deep learning in the following key manners compared to previous denoising methods:

- Rather than work on pixel-based representations, our input is the raw set of Monte Carlo samples, which we argue allows our method to appropriately handle information of different nature. In particular, depth of field and motion blur generate

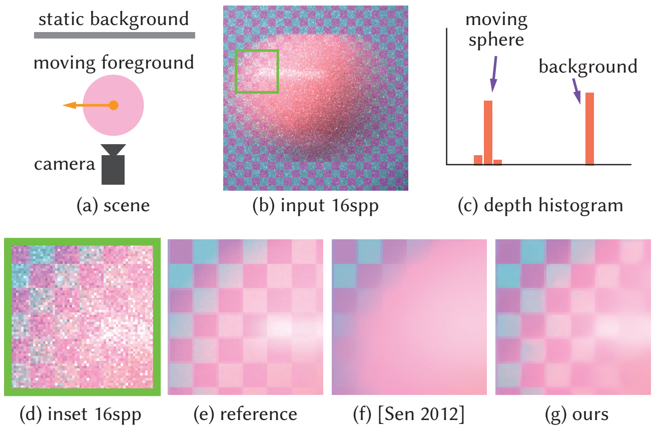

Fig. 2. The distribution of sample features within a given pixel is often multimodal. Take the example of a motion-blurred sphere, moving in front of a static background (a). The depth values in the region highlighted have a bimodal distribution (c). The correct output color for a pixel in this region is an average of the foreground and background radiance (b), integrated over the camera's shutter time. Because the background is static, it should remain sharp even though it is partially occluded by the moving sphere, but the foreground should appear blurred along the direction of motion (e). In this example, the depth of each sample is a good predictor of which object it belongs to, and therefore a useful cue for a denoiser. Most image-space denoisers use per-pixel first and second order statistics of the sample features (like depth). This gives them an erroneous view of the world, which can lead to overblurring or other artifacts (f). Our model works directly from the samples and can exploit the richer information they provide (g).

per-pixel sample distributions that tend to be poorly encoded by low-order statistics (Figure 2), and difficult light transport phenomena such as specular effects generate outliers ("firefly" artifacts) that are best handled at the sample level.

- We can handle variable-sized and unordered sets of input samples, building on recent permutation-invariant ideas [Aittala and Durand 2018; Zaheer et al. 2017].
- We use a splatting (scatter) approach, rather than a gather, because we argue it is more natural to assess where a sample contributes in the image rather than figuring out where *all* the contributions to a pixel are coming from. Consider the example of motion blur, where one can easily splat a sample along a trajectory, while figuring out all the parts of the scene that overlap a pixel is harder.

We hypothesize that processing individual samples instead of summaries has a fundamentally better outlook in complex transport scenarios. Unfortunately, prior sample-based methods tend to be costly and have difficulty handling general combination of lighting phenomena. This is either due to the curse of dimensionality [Hachisuka et al. 2008], or because they are derived analytically with specific effects in mind (e.g. [Lehtinen et al. 2011, 2012]).

To achieve full generality, we use deep learning to construct a mapping between Monte Carlo samples and final image. We develop a novel kernel splatting architecture that can take arbitrary number of input samples and is permutation invariant. Our method is inspired by recent pixel-based kernel predicting architecture [Bako et al. 2017]. However, the original *pixel-centric* architecture, which gathers from nearby samples, does not treat each sample individually and hence is not permutation invariant. Therefore, we center the kernels around the samples and *splat* the samples onto pixels. In addition, to facilitate communication between samples, we pool across the sample dimension to obtain per-pixel features.

We trained our network using a random scene generator and extensively tested our method on a variety of scenes, including various lighting phenomena such as distribution effects and diffuse and specular global illumination. Our method significantly reduces the error comparing to other methods, especially in the case where the sample count is low (e.g. 8 or less).

2 RELATED WORK

2.1 Denoising for Monte Carlo Rendering

Zwicker et al. [2015] present a recent survey of denoising techniques for Monte Carlo rendering. *A priori* methods characterize the structure of the integrand to derive analytical sampling rates and filters (e.g. [Belcour et al. 2013; Durand et al. 2005; Egan et al. 2011, 2009]). They are often limited to a specific combination of light transport scenarios (e.g. motion blur, global illumination, etc). In contrast, our approach falls in the *a posteriori* class of algorithms, that estimate smoothness and drive path reuse using auxiliary features to break the limitation to specific transport scenarios and to remain minimally intrusive with respect to the underlying rendering algorithm. As opposed to most *a posteriori* methods, ours works directly with the samples rather than per-pixel aggregates of the features.

Pixel-space adaptive sampling and reconstruction. A fruitful line of research adapted well-known image-processing filters [Kalantari and Sen 2013; Overbeck et al. 2009; Rousselle et al. 2012]. Unlike typical image-processing, Monte Carlo rendering noise has strong variations across pixels. These methods often set the parameters of the filter according to a statistical estimate of the error obtained from the noisy render buffers. The most successful pixel-space techniques employ auxiliary feature buffers such as depth, normals or albedo. These extra buffers have been used to drive anisotropic diffusion [McCool 1999] or cross-bilateral filters [Li et al. 2012; Rousselle et al. 2013]. Another line of work propose several local regression schemes between feature buffers and final radiance (e.g. [Bauszat et al. 2011; Bitterli et al. 2016; Moon et al. 2014]), inspired by the guided filter [He et al. 2010]. Some methods decompose the radiance contributions into multiple buffers to adapt the pixel filters [Mara et al. 2017; Mehta et al. 2014; Zimmer et al. 2015]. Our method also utilizes auxiliary features for filtering. However, instead of working with per-pixel averages, we operate directly on the individual samples.

Laplacian Kernel Splatting [Leimkühler et al. 2018] accelerates large splats by exploiting sparsity in the Laplacian domain. It can be used for denoising, but in contrast to our model, it requires analytical expressions for the kernels.

Sample-based reconstruction. Methods that work directly on the raw radiance samples often first reconstruct the radiance function using existing samples, then integrate the reconstructed function to generate images. Hachisuka et al. [2008] use the nearest neighbors of each sample for reconstruction, which suffers from the curse of dimensionality when the integrand dimension is more than five. Lehtinen et al. [2011; 2012] reproject existing samples using analytically derived trajectories, which also hinders generality. Rushmeier and Ward [1994] use variable-width splatting kernels. The kernel support is determined from simple local statistics. Their method preserves energy while ours does not. This allows us to more effectively suppress outliers. Unlike previous sample-based techniques, we *learn* splatting kernels from samples to final pixel color for full generality.

Sen and Darabi [2012]'s method stands between the pixel-based and sample-based approach. They deal with noise in features by analyzing their statistical dependency on the random parameters produced by the renderer (e.g. sub-pixel, lens, time coordinates). However, they still rely on heuristics based on statistical aggregates, and therefore cannot handle multi-modal cases like the one in Figure 2. Delbracio et al. [2014] use 1D color histograms to match pixels that should be denoised jointly. The histograms capture multi-modal effects and are invariant to permutations of the samples. However, their filter still operates on pixels and the histogram parameterization needs to be carefully set beforehand to cover the expected dynamic range. Bauszat et al. [2015] extend adaptive manifolds [Gastal and Oliveira 2012] to work directly with individual samples. They focus on limited global illumination and depth-of-field effects (e.g. they do not handle motion blur).

Learning for Monte Carlo denoising. Kalantari et al. [2015] predict the parameters of a fixed-function filter (e.g. cross-bilateral filter) from pixel-space features using a multilayer neural network. Chaitanya et al. [2017] use a recurrent autoencoder to denoise undersampled videos renderings. They target interactive setting and handle more restricted lighting conditions than our method. Our approach is similar in its use of 1-sample images, but we pay particular attention to permutation invariance. Bako et al. [2017] and Vogels et al. [2018] propose a kernel-predicting convolutional network for offline denoising. We adopt the kernel-predicting paradigm with two key differences: (1) we operate on samples, instead of pixel aggregates, and (2) we enforce permutation invariance through splatting. We also do not factorize the diffuse vs. specular illumination nor do we un-premultiply the albedo to obtain a pseudo-irradiance.

2.2 Permutation Invariance in Neural Networks

Independent Monte Carlo samples at a given pixel have no meaningful ordering. A learning method that remains aware of the ordering may counterproductively attempt to assign meaning to it. Recent work has demonstrated significant benefits from using models that are *by construction* invariant to permutations of the inputs via the use of symmetric *pooling* operations such as mean or max [Zaheer et al. 2017]. Examples of recent applications include e.g. point cloud processing [Qi et al. 2017] and burst deblurring [Aittala and Durand 2018]. These architectures also naturally accept an arbitrary number of inputs at test time.

We extend this line of work by an architecture that applies permutation invariance not only on the level of input buffers (e.g. the ordering of input images), but on the level of samples at individual pixels – that is, the output of our network remains unchanged if the samples at any given pixel are shuffled among each other.

3 SAMPLE-BASED DENOISING NETWORK

We cast Monte Carlo denoising as a supervised learning problem. Given an unordered set of input radiance samples s for pixel (x, y) with contribution L_{xys} and associated auxiliary features $\boldsymbol{f}_{xys}$ (specified in Appendix A) drawn from the entire image, we seek to reconstruct a noise-free, high dynamic range output. Typically, the samples are light paths that come from a path tracer [Kajiya 1986].

Our goal of supporting per-sample processing and built-in invariance to sample ordering, requires a novel network architecture: standard single-pass feedforward neural networks do not support varying numbers of inputs, and recurrent networks (RNN) are not permutation-invariant. Indeed, unordered input sets are fundamentally different from time series. We design a novel network architecture that achieves these goals by tightly integrating multiple steps of per-sample non-linear processing with spatial information sharing through standard convolutional neural networks on per-pixel summary statistics (Section 3.1).

We train our model on a large-scale dataset of input/output pairs rendered by path-tracing many randomly-generated scenes. Our scenes exhibit a variety of complex light transport. The dataset and training procedure of the network are described in Section 4.

3.1 Network Architecture Overview

Inspired by earlier work [Bako et al. 2017; Vogels et al. 2018], our model outputs, for each input sample, *a kernel* indicating how much the radiance contribution of the sample should affect the final intensity of nearby pixels, instead of directly predicting the output intensity.

Our overall strategy is to focus on samples instead of pixels, and build our architecture around the idea of each sample being able to answer *"how should I contribute to nearby pixels, given all the other samples around me?"* instead of every pixel asking "how should nearby samples influence me?". This at first seemingly trivial difference is in fact crucial; our results show superior performance for the scattering approach. We study the reasons for this using a didactic example in Section 3.4.

Therefore, for each sample s at pixel (x, y), we want to generate a kernel that would let it splats to some other pixels (u, v). The final image I is reconstructed as:

$$I_{uv} = \frac{\sum_{x,y,s} K_{xyuvs} \, L_{xys}}{\sum_{x,y,s} K_{xyuvs}}, \tag{1}$$

where the kernel K_{xyuvs} encodes the contribution of the sth sample in pixel (x, y) to the final intensity of the pixel at (u, v). The main difference with respect to previous gathering approaches is that the kernel is associated with the *sample* at (x, y), instead of the *pixel* at (u, v). These kernels are complex, nonlinear functions of *all* input samples and features parameterized by a neural network. The division is required for normalization. Note that we normalize *per-pixel*, not *per-sample*. This formulation is not energy-preserving: it can selectively suppress outliers, at the cost of some energy loss. We use 21×21 kernels.

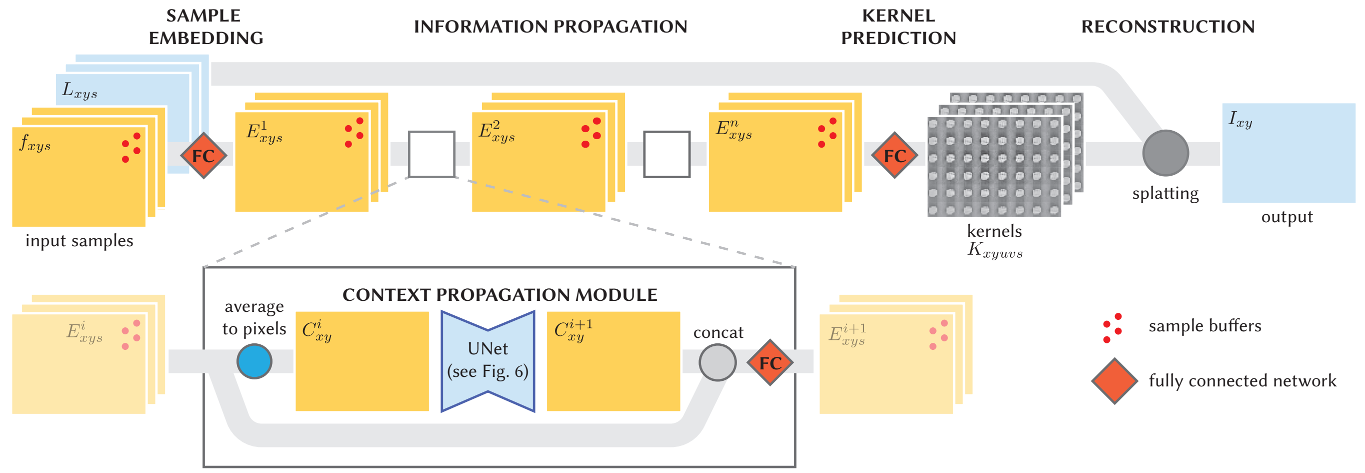

Fig. 3. We develop a novel kernel splatting architecture that maps unordered samples to an image. To support permutation invariance [Aittala and Durand 2018; Zaheer et al. 2017] and process arbitrary number of samples per pixel, we need to process each sample individually while making them aware of the neighborhood patterns. To achieve this, we generate *sample embeddings* E for individual samples and average them into *context features* C for propagating information. We repeat this process for the sample embeddings and context features to better exchange information. Finally, we produce a splatting kernel for each sample, similar to previous kernel-predicting methods [Bako et al. 2017]. This results in an architecture that does not change its outcome based on the order of samples, and is able to process arbitrary number of samples per pixel.

Our architecture is designed to be perfectly invariant to per-pixel permutations of the samples. We achieve this by splitting the per-sample feature extraction and spatial information sharing into alternating steps via the use of two key concepts: individual *sample embeddings* and per-pixel *context features*.

- The sample embedding is a non-linear feature space associated with each individual sample that is ultimately responsible for predicting the scattering kernel to nearby pixels. Importantly, the prediction can be implemented independently for each sample, independent of the number or ordering of other samples in the current or nearby pixels.
- The context features inform samples about their neighborhood; they are per-pixel averages of sample embeddings that can be processed using standard convolutional neural networks thanks to their fixed dimensionality, unlike the sample embeddings.

Figure 3 visualizes our architecture, and Algorithm 1 shows a pseudocode representation.

3.2 Sample Embeddings and Context Features

After we get the samples with radiance contribution L and auxiliary feature vector $\boldsymbol{f}$, the first sample embeddings are computed from the individual input samples by a small fully connected network with parameters θ^0_E applied to all samples *separately*:

$$E^0_{xys} = \text{FC}(L_{xys}, \boldsymbol{f}_{xys};\ \boldsymbol{\theta}^0_E) \quad \forall x, y, s. \tag{2}$$

Each sample embedding is a vector (with 128 dimensions in our implementation). Importantly, there is no interaction between samples here, so the effect of the network is invariant to sample ordering. Fully connected network to all samples can be efficiently implemented using 1×1 convolutions on the input sample buffers.

Algorithm 1 Sample-based kernel-splatting denoising.

```
notations
    $L_{xys}, f_{xys}$        input samples and auxiliary features
    $\theta$                  network weights
    $E^i_{xysc}$              sample embeddings
    $C^i_{xyc}$               context features
    $K_{xyuvs}$               kernels
    $I_{uv}$                  output image
    $W, H, N_s, D_E$          image width, height, #samples, embed dim.
procedure DENOISE
    $E^0_{xys} = \text{FC}(L_{xys}, f_{xys};\ \theta^0_E)$   ▷ Per-sample, $W \times H \times N_s \times D_E$
    for $i = 0 \dots n-1$ do
        $C^i_{xy} = \text{reduce\_mean}_s(E^i_{xys})$   ▷ Per-pixel, $W \times H \times D_E$
        $C^{i+1} = \text{UNet}(C^i;\ \theta^i_C)$   ▷ U-Net with skips
        $E^{i+1}_{xys} = \text{FC}(E^i_{xys}, C^{i+1}_{xy}; \theta^i_E)$   ▷ Per-sample, $W \times H \times N_s \times D_E$
    end for
    $K_{xyuvs} = \text{FC}(E^n_{xys}, C^n_{xy};\ \theta_K)$   ▷ Generate kernel
    $I_{uv} = \sum_{x,y,s} K_{xyuvs} L_{xys} / \sum_{x,y,s} K_{xyuvs}$   ▷ Splat on image
    return $I$
end procedure
```

To enable coherent sharing of information between samples both within and across pixels, the sample embeddings E^0 are turned into per-pixel context features C^0 by averaging across the sample axis:

$$C^0_{xy} = \underset{s}{\text{reduce\_mean}}(E^0_{xys}). \tag{3}$$

This has two key properties: first, averaging is permutation-invariant, that is, the ordering of samples within the pixels has no effect on the result apart from floating-point inaccuracies; second, the result of

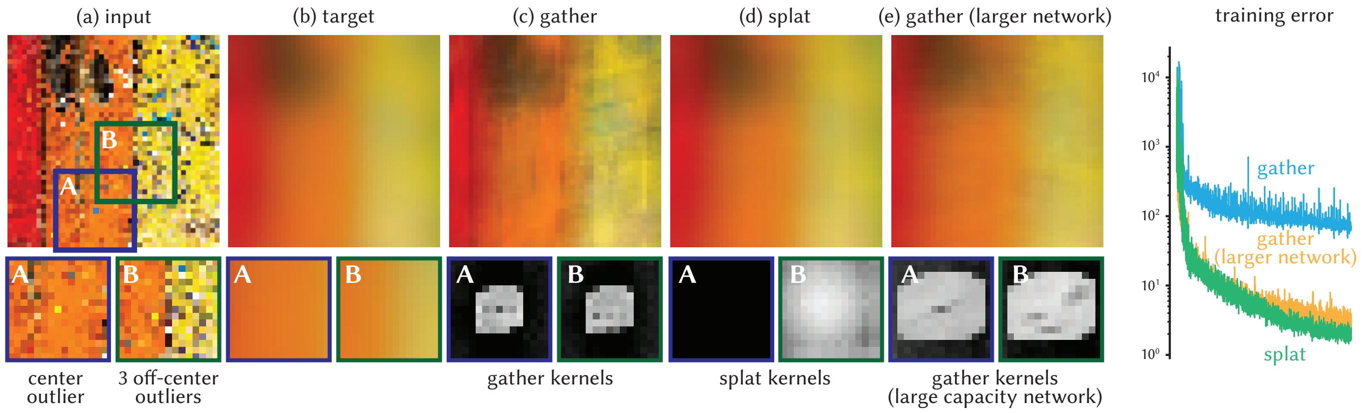

Fig. 4. We illustrate the benefits of our kernel-splatting approach with a controlled experiment in which the goal is: given unfiltered samples from a high-resolution texture, randomly corrupted by high-energy outliers (a), produce a properly anti-aliased and downsampled image (b). We train simple kernel-predicting networks to solve the problem on a small corpus of bitmaps with randomly added outliers. We purposefully limit the capacity of the kernel-predicting networks to have 3 convolution layers with 32 channels each. If there were no outliers, the network should simply predict a large Gaussian kernel for each pixel, which corresponds to the blur kernel we used to compute the target. However, outliers make the task less straighforward. In the *gather* case (c), each pixel needs to decide which of its neighbor is an outlier and consequently, the network needs to be able to predict all possible combinations of Dirac offsets to discard outliers, while still blurring the image. With a limited capacity, a gather-based network cannot learn the full Dirac basis. In order to minimize the training loss, it reverts to predicting kernels that are smaller than the allowed support (c) and therefore underblurs the image. A *splatting* network with the *same capacity* (d) does not suffer from this limitation. In the splatting formulation, each *sample* simply needs to decide whether it is an outlier by looking at its neighbors. If it is not, the network simply outputs the correct Gaussian kernel (d), bottom. Otherwise, it disables the sample's contribution by outputting an all-zeros kernel for this sample. The gather approach can be made to work by increasing the network capacity to 6 layers with 256 features each (e), over a 30-fold increase in the parameter count. With this capacity the gather network can learn the proper Diracs to exclude the central outlier in patch **A**, or the three neighboring outliers in patch **B** (dark areas inside the kernels of (e), bottom). We show the training loss for the three models on the right.

averaging sample embeddings has a fixed dimensionality regardless of the (potentially non-uniform) number of samples in the pixels. Thus, the context features can be processed using standard convolutional neural networks that take a fixed-dimension input. We feed the context features C^0 through a U-Net CNN [Ronneberger et al. 2015] that outputs a tensor of new features C^1, with the same dimensionality:

$$C^1 = \mathrm{UNet}(C^0;\ \boldsymbol{\theta}_C^0). \tag{4}$$

Upon completion, new sample embeddings E^1 are computed using the new context features C^1 in each pixel, again by separately applying a small fully-connected network to each sample embedding, taking the context feature from its pixel as an additional input:

$$E^1_{xys} = \mathrm{FC}(E^0_{xys}, C^1_{xy};\ \boldsymbol{\theta}_E^1) \quad \forall x, y, s. \tag{5}$$

The steps of updating sample embeddings, average pooling to context features, and updating context features are iterated for a few times (we use 3) before the final kernel prediction. This allows the samples to build a coherent picture of the image neighborhood and their own role in it.

3.3 Per-sample splatting kernels

Now that each sample is aware of the neighborhood patterns around it and the distribution of neighboring samples, it can decide which pixel to contribute to. That is, each sample uses the context as well as its own features to splat its radiance onto nearby pixels with weight:

$$K_{xyuvs} = \mathrm{FC}(E^n_{xys}, C^n_{xy};\ \boldsymbol{\theta}_K) \quad \forall x, y, s. \tag{6}$$

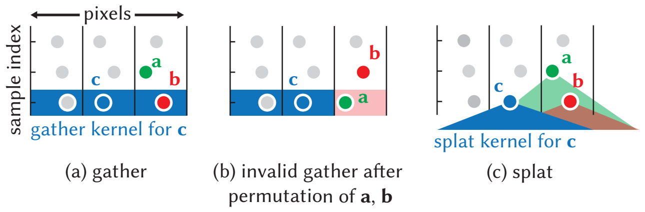

Fig. 5. Our network produces a fixed-size kernel per sample. In this formulation, gather kernels (a) would effectively couple pairs of samples yet only depend on the center sample. For instance, if samples **a** and **b** were swapped (b), the gather kernel at **c** would remain unchanged, thus assigning the wrong weight to **a**. With this permutation ambiguity, the best the gather network can do is predict uniform weights along the sample index dimension, which provides no benefit over a simpler pixel-based method. Splatting avoids this problem and can fully exploit the per-sample information because the kernels are directly associated with individual samples, while avoiding direct inter-sample interactions (c).

This is also a fully-connected network, applied per-sample. The coordinates u, v index the neighboring pixels. The reconstruction of the final image can now proceed as per Equation (1).

3.4 Why scatter and not gather?

We argue that scattering, or splatting, is a more natural operation for the sample-based setting. As a thought experiment, a defocused sample only needs to figure out its circle of confusion to splat to other pixels, while in the gathering formulation a pixel needs to find all samples whose circle of confusion overlaps with it. Similarly, from a sample's point of view, "am I an outlier given my neighborhood?" is a simpler question, whereas from the pixel's perspective, "is the sample two pixels to the right an outlier?" is a more difficult question. Figure 4 presents a 2D experiment simulating anti-aliasing with outliers, demonstrating the difference between splatting and gathering. The splatting approach results in much higher performance than gather for a given network size.

It is also difficult to achieve permutation invariance with a sample-based gathering approach. Splatting avoids the problem. The optima of the two formulations are fundamentally different and ours is superior. The gather approach cannot do better than a pixel-space model because it predicts conservative kernels that are optimal for the *expected* sample values. Our splats can adapt to *individual* samples. Figure 5 illustrates this.

3.5 Model and implementation details

All the fully-connected networks we use for per-sample operations consist of a stack of 3 fully connected layers with 128 features each (implemented as 1×1 convolutions on the sample tensor). The intermediate embedding operators have $D_E = 128$ output features. The kernel-prediction module outputs $441 = 21 \times 21$ weights in log space, which we exponentiate to enforce non-negativity of the K_{xyuvs}.

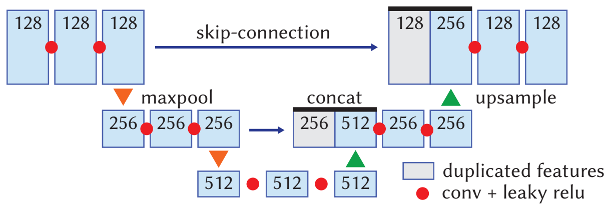

Fig. 6. Details of the UNet architecture we use to propagate the context featues spatially.

The U-Nets use zero-padded 3×3 convolutions, 2×2 maxpooling operators for downsampling and bilinear interpolation for upsampling. The skip connections are concatenation-based. The architecture is illustrated in Figure 6. We use leaky ReLU as activation functions, with a negative slope 10^{-2} for both the U-Nets and the fully-connected networks. We do not use any weight-sharing scheme, so each sub-network has its own set of parameters.

Implementing the kernel splatting efficiently in current deep learning frameworks such as PyTorch [Paszke et al. 2017] is not trivial. We do not want to instantiate the full 5D kernel tensor K_{xyuvs} in Equation (1) and (6), nor do we want to run into race conditions when two samples contribute to the same pixel at the same time. We implement our kernel-splatting operation in Halide [Ragan-Kelley et al. 2012] and leverage the compiler to inline the 5D kernel computation and generate efficient gradient code for training [Li et al. 2018]. Effectively, this amounts to transposing the kernels and reordering the loops to make our splats as efficient as gather kernels, without resorting to atomic adds.

4 DATASET AND TRAINING PROCEDURE

Deep learning models require a large amounts of training data. We procedurally generated a large dataset of around 300,000 renderings with resolution 128×128. We also rendered a validation set of around 1,000 images following the same procedure, in order to monitor the training progress. We describe the scene generation process in Section 4.1.

For each training example, we render a reference image at a high sample count (4096 samples per pixel) for which we only keep the per-pixel radiance (averaged over samples); this is our denoising target. We also render a low sample count input buffer (e.g. 4–32 samples per pixel), but this time we maintain extra information about the individual sample, stored as auxiliary feature buffers. This is the input of our model. We give a detailed description of the features in Appendix A. A few of our training scenes are shown in Figure 7.

4.1 Scene generation procedure

For our model to generalize, it is important that our training data be diverse and covers a variety of complex light transport situations. We developed a random scene generator that produces a combination of outdoor and indoor scenes. Our indoor scenes were adapted from the SunCG dataset [Song et al. 2017].

Geometry. To add geometric diversity, we randomly insert extra objects from the ShapeNet database [Chang et al. 2015] and apply random geometric transformation (scale, rotation, translation).

For the outdoor scenes, we populate the scene by sampling candidate positions on the ground plane using Poisson disc sampling with randomized radius, within and slightly beyond the camera frustum. Out of these candidates, we keep up to 50 positions to insert an object, randomizing its altitude, and object-space transform. For indoor scenes, we select at random a room from the SunCG dataset, keeping its furniture. We also add random objects within the room's bounding box. We always reject object locations that are too close to the camera to minimize the number of occluded (all black) images.

Camera and distributed effects. We randomize several camera parameters including the camera's field of view, its depth-of-field settings (focusing distance, aperture), and its shutter speed. For indoor scenes, we select the camera positions uniformly at random among the pre-computed positions given in the SunCG data. We then randomize the camera's up vector.

We activate motion blur and depth-of-field effects, each with 50% probability. When motion blur is activated, we add a random linear translation motion to 50% of the scene objects.

Materials and Textures. We randomize materials per-object, choosing between the various material models offered by PBRT v2 (metal, glass, mirror, plastic, etc). We slightly bias the distribution towards diffuse materials (20% of the random choices). We randomly assign textures and bump maps, with various UV scaling parameters. We used the *Describable Textures Dataset* [Cimpoi et al. 2014] as the source of our textures.

Lighting. We downloaded 111 HDR images from the HDRI Haven website [Zaal 2016] and used them as environment maps to light our

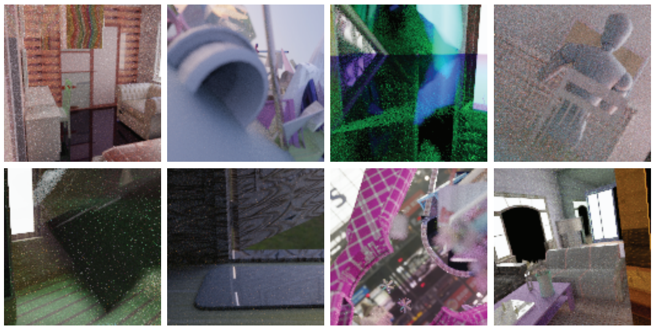
Fig. 7. A selection of the random indoor and outdoor scenes we generate for training.

outdoor scenes, and some of our indoor scenes. For indoor scenes, we also randomly add 1 to 5 point or area light sources.

Rejection sampling. Despite our careful parameterization of the scene generator, sometime the scene is too simple, or renders to pure black (e.g. the lights or the camera are occluded). We remove those pathological scenes from our dataset.

4.2 Rendering the input and target data

All our training, validation and testing data is generated with the PBRT v2 renderer [Pharr and Humphreys 2010]. We use PBRT's default *perspective* camera model and the default *path* surface integrator. We instrument the pathtracer to save, for each sample, its radiance and a 74-d vector of features (Appendix A). We fix the path length to 5 bounces and disable the Russian roulette termination criterion. We also disable the volume integrator for our experiments. To avoid discrepancies between the noisy input and the ground truth rendering, we fix the scaling of ray differentials used by PBRT for mip-mapping to be independent of the sample count. While more modern, faster renderers are available, we chose PBRT to facilitate the comparison to previous works that are implemented atop the public PBRT code ([Kalantari et al. 2015; Rousselle et al. 2012, 2013; Sen and Darabi 2012]). We use PBRT's default low-discrepancy sampler [Kollig and Keller 2002] to obtain the samples. The ground-truth images are produced using the same parameters, although at a higher sample count. For these, the pixel values are obtained from the samples with a box filter. The set of samples used for the ground-truth is generated from different random seeds and are independent from those of the noisy input. This is important to avoid correlation between the input and target so that, e.g. the network does not learn to copy outliers. To maximize diversity, we randomize the output resolution of the output images and render a *single* random 128×128 crop for each synthetic scene we generate. Varying the resolution allows us to cover a wide range of spatial frequency content in the output, without changing the scene geometry or textures.

4.3 Training details

Our network is trained to minimize the expected L_2 loss between our tonemapped output and the tonemapped ground-truth:

$$\mathbb{E}_{I \sim \mathcal{D}}\left(||\tau(I) - \tau\left(I_{\mathrm{gt}}\right)||^2 \right), \tag{7}$$

where $\tau(x) = \frac{x}{1+x}$ is the tonemapping operator, $\mathcal{D}$ the training dataset and I_{gt} the ground truth corresponding to our reconstruction I. At train time, we randomly select a number of samples per-pixel. Our models are trained with sample counts between 2 and 8. Note that although our loss favors a specific tone-mapper, our model outputs linear, high dynamic range radiance. We found that the key property for stable training is that the loss limits the influence of overly bright samples. We also experimented with the relative L_2 and SMAPE [Vogels et al. 2018] loss functions; both gave qualitatively similar results.

The weights for the convolutional layers are initialized according to He et al.'s recommendation [2015] and the biases to 0. We optimize the network parameters using the ADAM solver [Kingma and Ba 2015]. We found that a batch size of 1 fully utilizes the GPU while larger batches make the training I/O-bound (each batch contains all the samples of a 128×128 image). We set the learning rate to 10^{-4} and leave the remaining parameters of the ADAM optimizer to the values recommended by the authors. Our model is implemented in PyTorch [Paszke et al. 2017]. All the models we present were trained on a NVIDIA Titan X (Pascal) GPU until the loss on a held-out validation set stopped improving (typically 3–4 days). Because we work with samples instead of pixel inputs, storage and I/O bandwidth can become the bottleneck in training our model.

5 RESULTS

We assembled a dataset of 55 test scenes for evaluation and converted them to the format expected by our renderer. The scenes were collected from publicly available sources [Bitterli 2016; Pharr and Humphreys 2010; Pharr et al. 2016]. Like our training set, these scenes were rendered using PBRT v2. In addition to previously published techniques, we compare our model to several baselines in an ablation study that highlights the benefits of both our per-sample prediction, and our splatting kernels (Section 5.2).

Throughout this section we evaluate a denoiser's performance quantitatively using two metrics: relative Mean Squared Error (rMSE) and structural dissimilarity [Wang et al. 2004], DSSIM = 1 − SSIM. We refer the reader to the supplemental material for full-resolution images and more details on the quantitative evaluation.

5.1 Comparison to previous work

We compare our technique to a state-of-the-art pixel-based deep denoiser [Bako et al. 2017] as well as previous methods that do not use deep learning [Bitterli et al. 2016; Kalantari et al. 2015; Rousselle et al. 2012, 2013; Sen and Darabi 2012]. We use the original, publicly available implementations for all the methods (with minor code changes to make them compile with a modern GCC version).

All the methods use a combination of color, albedo, normal and depth buffers as auxiliary features. Except for Bitterli et al. [2016] and Bako et al.[2017], the denoisers in our comparison compute features directly within PBRT, so we made no attempt to alter them. For these two methods, we transform our features (averaged per pixel) so they match the method's expected format and normalization, including the specular/diffuse separation, albedo un-premultiplication and log transform for Bako et al [2017].

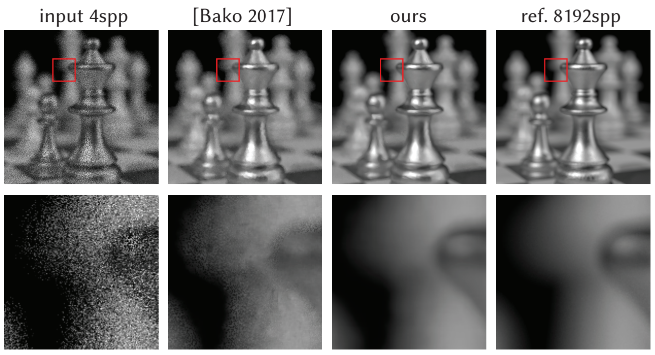

Fig. 8. Previous work, like Bako et al. [2017], assume an easy-to-compute, noise-free albedo buffer is available. In many cases involving distributed effects (e.g. motion blur, depth-of-field), this assumption does not hold and computing a clean albedo requires many rays to be cast. In contrast, our method does not decouple irradiance and albedo and, because it has access to the individual samples, can denoise the moving or defocused object properly.

The noisy inputs of our dataset tend to have a very high dynamic range, which causes numerical instability in Bitterli [2016]. We remedy this by separating the radiance into a diffuse and a specular component. We compress the dynamic range of the specular term with a log-transform and denoise the two components independently before re-combining them. This modification has been previously reported to improve the quality of this algorithm's output [Bako et al. 2017].

We fine-tuned Bako et al's deep learning model on our data, since the authors report their model is expected to generalize poorly when used with a different renderer than that it was trained with. We report the test error for both the fine-tuned and unaltered version. Visual comparisons can be seen in Figure 10. We provide more comparisons with varying number of samples in supplementary material. We summarize our quantitative evaluation in Table 1, testing the methods with various level of noise by changing the number of samples.

Bako et al. [2017] make the strong assumption that one can quickly compute a perfectly clean albedo buffer to feed as input to the network. This assumption is violated when, for example, dealing with distributed effects like depth-of-field or motion blur. Because [Bako et al. 2017] divide by the albedo to denoise the irradiance on diffuse surfaces, their method breaks in this case (Figure 8). This is particularly striking at low sample counts. It also explains why our *PixelGather* ablation (see below) performs better numerically.

5.2 Model ablation

We study the benefits of treating samples individually and kernel splatting by selectively de-activating them on our architecture, leading to four choices: *PixelGather*, *PixelScatter*, *Gather*, and *Ours*.

- *PixelGather*. We reduce the sample features to mean and variance and treat this as a one sample per pixel input, and feed into our architecture. This model is closest to Bako et al.'s method [2017], but normalizes for architectural difference and does not use the albedo un-premultiplication.

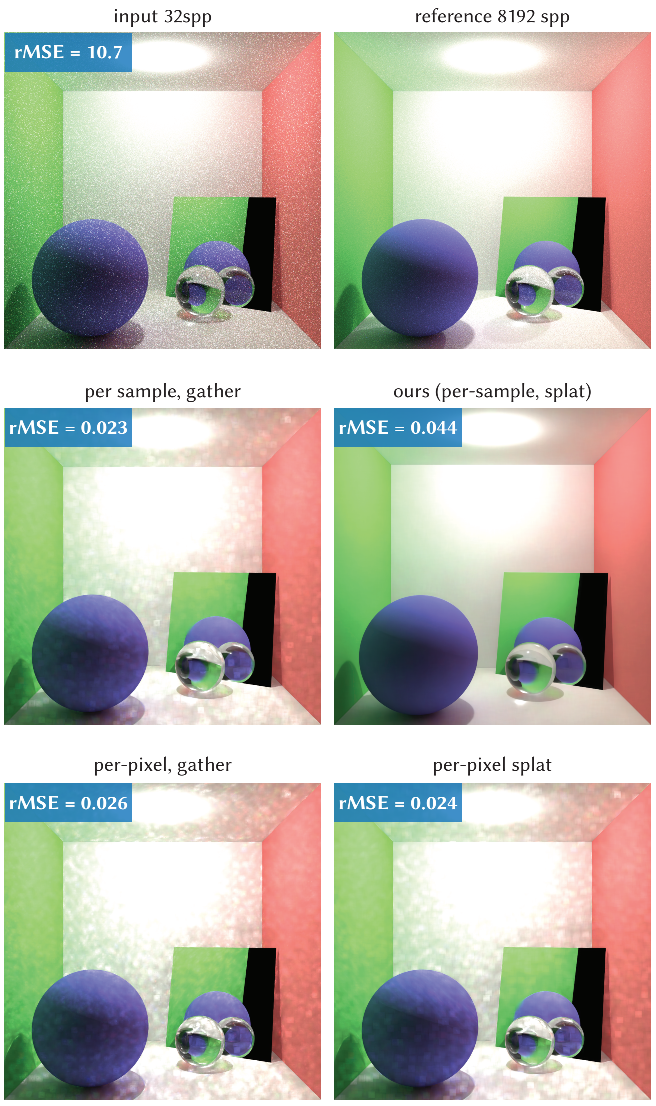

Fig. 9. We show that both treating sample individually and kernel splatting are necessary for good denoising. We compare to three other baselines by switching to gather and collapsing samples to pixel statistics. For example, our sample-based model can more easily separate the radiance contribution from high-energy, sharp specular light paths from softer global illumination components (bottom right). All the other baselines produce artifacts that are dominated by the outlier samples, producing similar artifacts with low-frequency noise. In this hard scenario, all of the methods fail to reproduce the extremely undersampled caustics caused by the mirror onto the ground (see the reference). Our model falls back gracefully and produces a much cleaner image, even though its rMSE is slightly higher.

- *PixelSplat*. We reduce to one sample per-pixel like *PixelGather*, but we replace the gather kernels by our new splatting operation.
- *Gather*. We treat each individual sample independently, but filter them with gather kernels [Bako et al. 2017].

Table 1. Error comparison on the test set with several standard metrics for 4 to 128 sample-per-pixel inputs respectively. Our model was trained on 2–8spp inputs. We report both the vanilla and finetuned (f.t.) version of Bako et al.'s method [2017]. All the metrics are computed so that a lower value is better. We provide the full analysis and additional metrics in the supplemental material.

		input	ours	Bako [2017]	Bako (f.t.)	Bitterli [2016]	Kalantari [2015]	Rousselle [2012]	Sen [2012]
4spp	rMSE	17.3054	**0.0482**	1.0867	0.4932	1.0847	1.5814	2.0416	1.0352
	DSSIM	0.4648	**0.0685**	0.1294	0.1173	0.1014	0.0869	0.1575	0.1009
8spp	rMSE	7.4732	**0.0382**	8.1238	0.8471	0.9149	1.6684	2.0721	0.5670
	DSSIM	0.3995	**0.0599**	0.1190	0.0940	0.0818	0.0708	0.1175	0.0987
16spp	rMSE	11.0416	**0.0315**	21.3297	0.2934	0.9488	1.8151	2.0481	0.3348
	DSSIM	0.3373	**0.0542**	0.1128	0.0762	0.0700	0.0600	0.0898	0.0986
32spp	rMSE	16.5478	**0.0274**	31.4400	0.1427	1.1344	1.7398	1.6447	0.2731
	DSSIM	0.2775	**0.0510**	0.1106	0.0619	0.0630	0.0516	0.0685	0.1005
64spp	rMSE	12.0608	**0.0261**	0.1359	0.1553	0.8747	1.6228	1.6974	—
	DSSIM	0.2223	0.0494	0.0407	0.0536	0.0566	**0.0393**	0.0547	—
128spp	rMSE	1.7717	**0.0254**	0.0757	0.1229	0.9039	1.7394	1.8176	—
	DSSIM	0.1750	0.0488	**0.0353**	0.0432	0.0565	0.0414	0.0466	—

Table 2. We studied the importance of the per-sample computation and the splatting kernel (as opposed to gather) by selectively turning off these components. Both improve the reconstruction error. *PixelGather* outperforms Bako [2017]. This is because at low sample counts, the albedo channel is noisy and the albedo un-premultiplication reduces image quality.

		ours	Gather	PixelGather	PixelSplat
4spp	rMSE	**0.0482**	0.0573	0.0823	0.0589
	DSSIM	**0.0685**	0.0744	0.0741	0.0752
8spp	rMSE	**0.0382**	0.0512	0.0882	0.0620
	DSSIM	**0.0599**	0.0635	0.0628	0.0639
16spp	rMSE	**0.0315**	0.0579	0.0539	0.0505
	DSSIM	**0.0542**	0.0556	0.0542	0.0553
32spp	rMSE	**0.0274**	0.0634	0.0431	0.0466
	DSSIM	0.0510	0.0506	**0.0479**	0.0490
64spp	rMSE	**0.0261**	0.0742	0.0417	0.0568
	DSSIM	0.0494	0.0475	**0.0431**	0.0446
128spp	rMSE	**0.0254**	0.0722	0.0398	0.0532
	DSSIM	0.0488	0.0455	**0.0394**	0.0413

Our sample-based splatting network outperforms these alternatives on our test set (see Table 2). Figure 9 compares these alternatives on a modified Cornell box scene with difficult specular light transport, where the input image is severely contaminated by noise and outliers. This example shows that both the individual treatment of the samples and the splatting kernels are required to resolve the image details and filter out the outliers.

5.3 Performance

Our technique is most beneficial at low-sample counts since its complexity grows linearly with the number of samples (the fully connected networks operate per-sample). In practice, for the high sample count examples, we stream the samples between the GPU and the main RAM or disk to bound the memory usage. This streaming operation applies to the per-sample processing steps (Equations (2), (5) and (6)). We report the runtime of several denoisers including ours, for a few different sample counts in Table 3 below. The denoisers were evaluated on the same machine (running an 8-core Intel i7-6900K CPU @ 3.20GHz, and an NVIDIA Titan Xp GPU). We used the reference implementation provided by the authors for this benchmark.

Table 3. Runtime cost of several denoisers (in seconds) to process a 1024×1024 image. Our model's performance scales linearly with the number of samples. The runtime of pixel-based methods is constant; we report it in the last column only. The algorithm of Sen et al. [2012] did not terminate for the two highest sample counts. All the other denoisers remain significantly faster than what it would take to pathtrace additional samples to convergence. The PBRT timings are averaged over several renderings of the *sanmiguel* scene (see supplemental).

spp	4	8	16	32	64	128
PBRT rendering	15.4	30.6	61.3	121.4	245.1	491.8
ours	6.0	10.1	18.9	35.9	67.0	156.5
Bako [2017]						14.6
Bitterli [2016]						21.9
Kalantari [2015]						10.4
Rousselle [2012]						13.3
Sen [2012]	281.2	638.1	1603.1	4847.8	—	—

5.4 Discussions and Limitations

Performance. Instead of focusing on pure performance, our work is motivated by the fundamental question of how much information can be extracted from very few samples, where denoising is more challenging.

For simpler PBRT scenes used for research, our denoiser can use close to 25% of the total rendering time (Table 3). With more complex materials, geometry and lighting, production scenes typically take much longer to render — Bako et al. [2017] report over 100 core hours to render a single 1920×1080 images with 128 samples per pixel. Since scene complexity does not affect the denoiser's runtime, our technique would be most useful in such settings, e.g. to quickly produce clean previews at low sample rates.

Still, the linear performance scaling of our algorithm as we add more samples can become an issue. In particular, at much higher sample counts algorithmic changes would likely lead to a more efficient solution. For example, in spirit of the histograms of Delbracio et al. [2014], models then could learn to pre-aggregate similar samples to be denoised together would be more scalable. Storing the radiance and features for all the samples can also become cumbersome at high sample counts. A recurrent formulation may be an option to avoid storing all the samples when computing the context features. Such a model would make progressive rendering easier but would suffer from the lack of permutation invariance with respect to the samples.

Video denoising. We argue that temporal consistency in video denoising is an issue orthogonal to maximally exploiting the information from the samples. Techniques like [Bonneel et al. 2015; Chaitanya et al. 2017; Schied et al. 2017; Vogels et al. 2018] could be applied to our model to enforce temporal consistency, and conceptually our kernel-splatting model can be easily extended to splat in the temporal dimension. We nonetheless show an example video output in the supplemental material, in which each frame (with a different random seed) is processed independently by our model.

Sample decomposition. While we treat each sample individually, our *samples* are still aggregates of information over path length, BRDF layers, and next event estimations. Previous works have shown that further decomposing these information can help the denoising process (e.g. [Mehta et al. 2014; Ward et al. 1988; Zimmer et al. 2015]). Our aggregation of radiance over the path vertices means that enabling Russian roulette and generalizing to arbitrary path lengths is feasible with our current model, although some of the auxiliary features would need to be adapted or disabled since we currently store some of the features at each vertex of a depth-5 path (Appendix A).

6 CONCLUSION

We propose a new convolutional neural network for denoising Monte Carlo renderings. The key innovations that explain the success of our method are the use of samples rather than pixel statistics and splatting rather than gathering. For this, we introduce a new kernel-splatting architecture that is invariant to input permutations and accepts arbitrary numbers of samples. We show that our approach is robust to severe, heavy-tailed noise in low sample count settings and excels at rendering scenes with distributed effects such as depth-of-field, achieving significantly-reduced error on our extensive tests.

A EXTRACTING SAMPLE FEATURES

To properly disambiguate samples from their neighbors, we need to associate them with features that describe the properties of light paths. Our network consumes two kinds of features: *global*, shared by all samples, and *local*, specific to each.

Our global features are the lens aperture radius, the camera's field of view, and the lens focusing distance. These quantities are useful to, e.g. determine the circle-of-confusion of each sample for defocusing effects.

Sample coordinates. For proper anti-aliasing, depth-of-field and motion blur effects, we record each sample's (x, y, u, v, t) coordinates, all in the range $[0, 1]$. (x, y) are the sub-pixel sample position. (u, v) are coordinates of the ray's intersection with the camera lens. They lie in the unit disk. $t \in [0, 1]$ is a time coordinate indicating the moment the sample was taken relative to the camera's shutter duration.

Radiance. The radiance samples computed by a renderer typically have a high dynamic range (covering several orders of magnitude), in part because of low-probability, high-energy paths like specular bounces. This high dynamic range is challenging for the stochastic optimization of the network's weights. We therefore compress the sample's radiance using a log-transform before feeding it to the kernel-predicting network. Note that while the kernel predictor consumes log-radiance, the splatting weights still act on the *linear* radiance.

We follow previous work [Bako et al. 2017; Chaitanya et al. 2017], and futher decompose the log-radiance features into a diffuse and a specular component. Here again, this decomposition only affects the kernel prediction. In particular, unlike [Bako et al. 2017], we do *not* use separate networks to reconstruct the diffuse and specular components.

Geometry and materials. To capture the scene geometry and guide the denoiser, we attach to each sample the depth and normals at the first diffuse bounce. We also keep track of the texture hit at the first diffuse bounce with an RGB albedo channel. We also record 5 boolean features to characterize the material interactions at each vertex of the light path (reflection, transmission, diffuse, glossy, specular). Since we fix the path length to 5, this gives 25 features.

Lighting information. We also encode light sampling information. First, with a binary visibility term for the first bounce and second, with a pair of angles encoding the sampled light's direction in the camera's spherical coordinates, for each path vertex ($1 + 2 \times 5 = 11$ numbers). Finally, we store the conditional log-probabilities of sampling this light direction according to the BRDF, and the direct light sampling algorithm, which accounts for 4 numbers per path vertex. Overall, the samples are represented as 74 floating point numbers. Together with the 3 global features, these constitute the input of the kernel-predicting network.

ACKNOWLEDGMENTS

We thank the anonymous SIGGRAPH reviewers for their valuable feedback. This work was partially funded by the Toyota Research Institute.

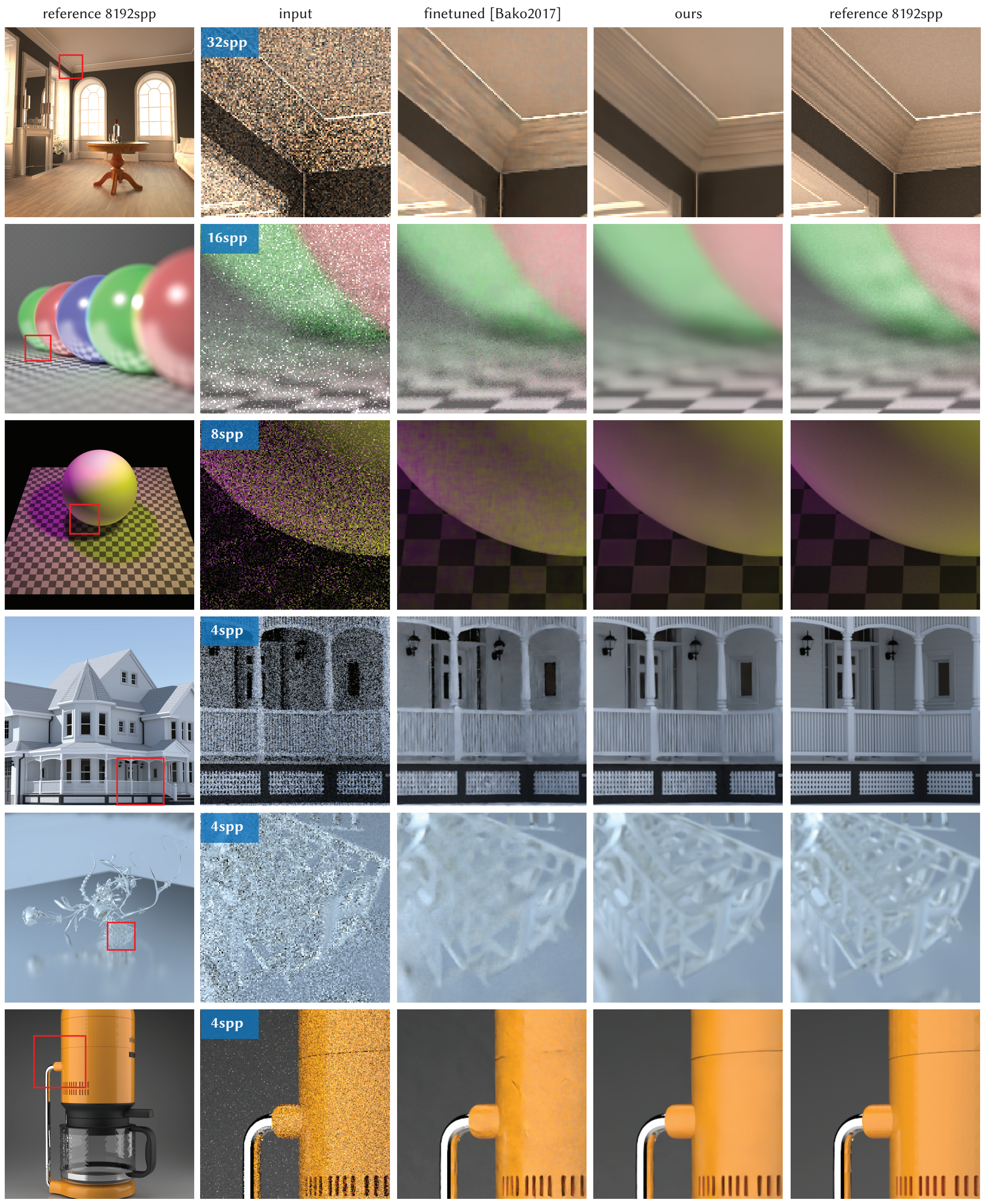

Fig. 10. We compare to previous deep learning denoising method [Bako et al. 2017] on a benchmark containing a variety of lighting scenarios, including distribution effects and diffuse and specular global illumination. Bako et al.'s method is usually used with a higher sample count, and when this number goes down, the method fails to distinguish between noise and features, and either oversmoothes or undersmoothes. In contrast, our method is able to reconstruct crisp images, due to the increased differentiation afforded by samples and our splatting approach. We choose the number of samples to match the complexity of the scenes. The supplementary material contains more results with different numbers of input samples. In general we found that 8 samples per pixel gives consistently good results.

REFERENCES

Miika Aittala and Frédo Durand. 2018. Burst image deblurring using permutation invariant convolutional neural networks. *ECCV* (2018).

Steve Bako, Thijs Vogels, Brian Mcwilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. 2017. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM SIGGRAPH* (2017).

Pablo Bauszat, Martin Eisemann, S John, and M Magnor. 2015. Sample-based manifold filtering for interactive global illumination and depth of field. *Computer Graphics Forum* (2015).

Pablo Bauszat, Martin Eisemann, and Marcus Magnor. 2011. Guided Image Filtering for Interactive High-quality Global Illumination. *Computer Graphics Forum (Proc. EGSR)* (2011).

Laurent Belcour, Cyril Soler, Kartic Subr, Nicolas Holzschuch, and Fredo Durand. 2013. 5D covariance tracing for efficient defocus and motion blur. *ACM TOG* (2013).

Benedikt Bitterli. 2016. Rendering resources. https://benedikt-bitterli.me/resources.

Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José A. Iglesias-Guitiàn, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák. 2016. Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings. *Computer Graphics Forum (Proc. EGSR)* (2016).

Nicolas Bonneel, James Tompkin, Kalyan Sunkavalli, Deqing Sun, Sylvain Paris, and Hanspeter Pfister. 2015. Blind Video Temporal Consistency. *ACM SIGGRAPH Asia* (2015).

Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM SIGGRAPH* (2017).

Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. 2015. ShapeNet: An Information-Rich 3D Model Repository. *CoRR* (2015).

M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. 2014. Describing Textures in the Wild. *CVPR* (2014).

Mauricio Delbracio, Pablo Musé, Antoni Buades, Julien Chauvier, Nicholas Phelps, and Jean-Michel Morel. 2014. Boosting Monte Carlo rendering by ray histogram fusion. *ACM TOG* (2014).

Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X Sillion. 2005. A frequency analysis of light transport. *ACM SIGGRAPH* (2005).

Kevin Egan, Florian Hecht, Frédo Durand, and Ravi Ramamoorthi. 2011. Frequency analysis and sheared filtering for shadow light fields of complex occluders. *ACM TOG* (2011).

Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM SIGGRAPH* (2009).

Eduardo SL Gastal and Manuel M Oliveira. 2012. Adaptive manifolds for real-time high-dimensional filtering. *ACM SIGGRAPH* (2012).

Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM SIGGRAPH* (2008).

Kaiming He, Jian Sun, and Xiaoou Tang. 2010. Guided image filtering. *ECCV* (2010).

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *ICCV* (2015).

James T. Kajiya. 1986. The Rendering Equation. *ACM SIGGRAPH* (1986).

Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. 2015. A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM SIGGRAPH* (2015).

Nima Khademi Kalantari and Pradeep Sen. 2013. Removing the noise in Monte Carlo rendering with general image denoising algorithms. *Computer Graphics Forum (Proc. EG)* (2013).

Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *International Conference on Learning Representations* (2015).

Thomas Kollig and Alexander Keller. 2002. Efficient multidimensional sampling. *Computer Graphics Forum (Proc. EG)* (2002).

Jaakko Lehtinen, Timo Aila, Jiawen Chen, Samuli Laine, and Frédo Durand. 2011. Temporal light field reconstruction for rendering distribution effects. *ACM SIGGRAPH* (2011).

Jaakko Lehtinen, Timo Aila, Samuli Laine, and Frédo Durand. 2012. Reconstructing the indirect light field for global illumination. *ACM SIGGRAPH* (2012).

Thomas Leimkühler, Hans-Peter Seidel, and Tobias Ritschel. 2018. Laplacian kernel splatting for efficient depth-of-field and motion blur synthesis or reconstruction. *ACM SIGGRAPH* (2018).

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM SIGGRAPH* (2018).

Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. 2012. SURE-based optimization for adaptive sampling and reconstruction. *ACM SIGGRAPH Asia* (2012).

Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. 2017. An Efficient Denoising Algorithm for Global Illumination. *High Performance Graphics* (2017).

Michael D McCool. 1999. Anisotropic diffusion for Monte Carlo noise reduction. *ACM TOG* (1999).

Soham Uday Mehta, JiaXian Yao, Ravi Ramamoorthi, and Fredo Durand. 2014. Factored Axis-aligned Filtering for Rendering Multiple Distribution Effects. *ACM SIGGRAPH* (2014).

Bochang Moon, Nathan Carr, and Sung-Eui Yoon. 2014. Adaptive rendering based on weighted local regression. *ACM TOG* (2014).

Ryan S Overbeck, Craig Donner, and Ravi Ramamoorthi. 2009. Adaptive wavelet rendering. *ACM SIGGRAPH Asia* (2009).

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Matt Pharr and Greg Humphreys. 2010. *Physically based rendering: from theory to implementation.*

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation.*

Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. 2017. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. *CVPR* (2017).

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM SIGGRAPH* (2012).

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. *International Conference on Medical Image Computing and Computer-Assisted Intervention* (2015).

Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. 2012. Adaptive rendering with non-local means filtering. *ACM SIGGRAPH Asia* (2012).

Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. 2013. Robust denoising using feature and color information. *Computer Graphics Forum (Proc. PG)* (2013).

Holly E. Rushmeier and Gregory J. Ward. 1994. Energy Preserving Non-linear Filters. *ACM SIGGRAPH* (1994).

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination. *High Performance Graphics* (2017).

Pradeep Sen and Soheil Darabi. 2012. On filtering the noise from the random parameters in Monte Carlo rendering. *ACM TOG* (2012).

Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. 2017. Semantic Scene Completion from a Single Depth Image. *CVPR* (2017).

Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. 2018. Denoising with kernel prediction and asymmetric loss functions. *ACM SIGGRAPH* (2018).

Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE TIP* (2004).

Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. 1988. A Ray Tracing Solution for Diffuse Interreflection. *ACM SIGGRAPH* (1988).

Greg Zaal. 2016. HDRI Haven. https://hdrihaven.com.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. *NIPS* (2017).

Henning Zimmer, Fabrice Rousselle, Wenzel Jakob, Oliver Wang, David Adler, Wojciech Jarosz, Olga Sorkine-Hornung, and Alexander Sorkine-Hornung. 2015. Path-space Motion Estimation and Decomposition for Robust Animation Filtering. *Computer Graphics Forum (Proc. EGSR)* (2015).

M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon. 2015. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Computer Graphics Forum (Proc. EG)* (2015).