KPCN Denoising for Monte Carlo Renderings

An Exploration of KPCNs as Applied to Individual Path-Traced Frames

Owen Jow
UC San Diego
owen@eng.ucsd.edu

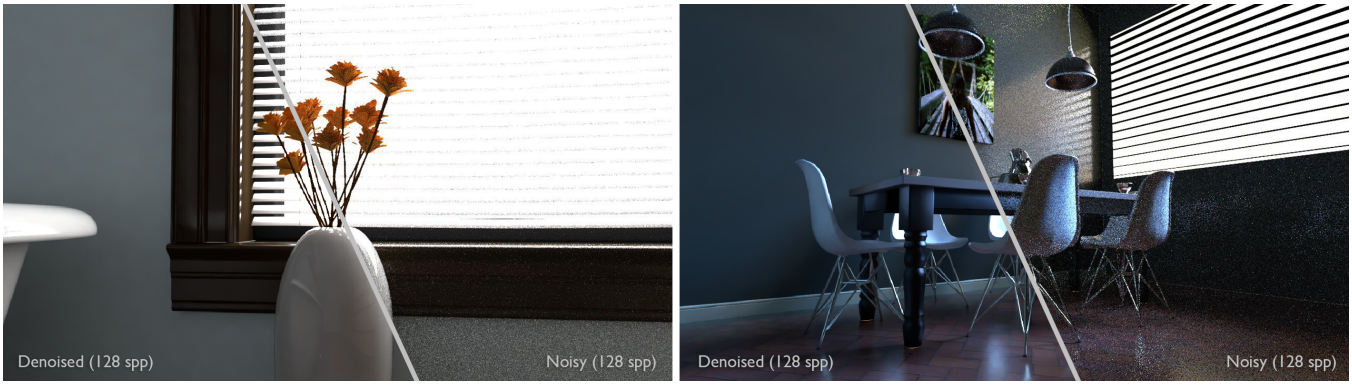

FIGURE 1: I reimplemented [1] and [4] to denoise individual path-traced frames using kernel prediction.

ABSTRACT

In this project, I applied KPCNs (kernel-predicting convolutional networks) as per [1] and [4] to denoise single path-traced frames. To my knowledge, I have replicated most presented findings with respect to direct prediction, kernel prediction, asymmetric loss, and multi-scale denoising. Additionally, over the course of many training attempts, I investigated the effects of several hyperparameter, architectural, or other changes that were not extensively discussed in either of the two reference papers. This report touches upon these different aspects of KPCN-based denoising methods, with the aim of imparting information helpful for navigating the often-murky field of hyperparameter and structural tuning in ML pipelines.

1 INTRODUCTION

In the years of late, computing subdomains have warped and deformed into increasingly nail-like entities under the hammer that is deep learning. Even the holy grail of physically-based rendering, Monte Carlo path tracing, has not been immune. The recent work of [1], [2], and [4] has turned the state of the art of reconstruction from few-sample path tracing into a collection of convolutional networks which predict either (a) denoised pixels directly or (b) local filtering kernels.

Here, I reimplement two of these papers (one an extension of the other), namely the ones by Disney/Pixar relating to kernel prediction. Although the more recent of these papers (KPAL) provides options for multiple path tracers and animated rendering sequences, I do not have immediate access to these things and so omit the associated modules (source-aware encoders and temporal denoising) from my implementation. Instead, I focus on the modules relevant to single-frame denoising, in particular the single-frame kernel-predicting denoiser (with or without asymmetric loss) and the multiscale wrapper around this denoiser.

Following this path, I obtain results comparable to each paper's while also exploring many other facets of KPCN-based denoising as described in Section 3.

2 METHOD

I denoise images using a specular KPCN and a diffuse KPCN, which each take a path-traced color image along with auxiliary buffers and predict a filtering kernel for every location (Figure 2). In order to better remove low-frequency artifacts, I include the option of denoising at multiple scales and blending the results. I also apply an asymmetric loss for increased preservation of detail.

While searching for the best results, I of course tried several data sampling/preprocessing strategies, network architectures, and training tricks. In the following subsections, I describe what I think of as my main denoising pipeline. For a condensed discussion of findings from my exploration, see Section 3.

2.1 Data

As a machine learning method, KPCN-based denoising requires large volumes of data for training. I used a dataset provided by Bako et al. which provides path-traced data (renderings and auxiliary feature buffers) for 1482 permutations of eight publicly available scenes rendered using Tungsten at five different sampling rates: 128, 256, 512, 1024, and 8192 spp. In my experiments, I train using 128 and 256 spp renderings as inputs and 8192 spp renderings as ground truth.

Since it is intractable to train with the full 1280×720 buffers, I sample 500 65×65 patches from each image and its corresponding feature buffers and use those during training. The sampling is performed without replacement according to a PDF given by a weighted sum of the color and normal variances, in order to obtain a good distribution of informative patches (as opposed to, e.g., a

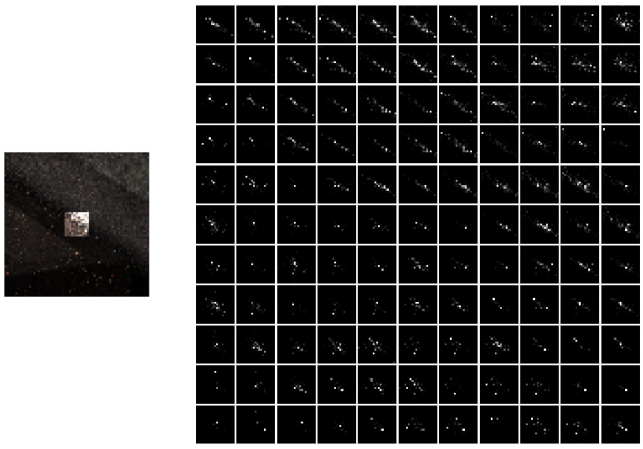

Figure 2: Predicted kernels for the highlighted region.

dataset full of patches which are of constant intensity when denoised). An example of the PDF and sampled patches can be seen in Figure 3.

In all, the network inputs (Figure 4) consist of

- log(1 + color)
- x-gradients of color, normals, albedo, and depth
- y-gradients of color, normals, albedo, and depth
- Relative variances of color, normals, albedo, and depth

In the above, *color* is either the specular color or the irradiance, depending on the network we're using (see Section 2.2). The irradiance is computed from the diffuse color by dividing out the albedo. We also re-scale the depth to be in the range $[0, 1]$.

This is a mixture of [1] and [4] network inputs; the idea to use gradients and depth comes from [1], while the idea to use a log transform of the irradiance and relative variance comes from [4]. Although [4] is the more recent paper, I found that my network performed better when I used gradients as opposed to raw (or log transforms of) auxiliary features as done in [4]. Presumably this is because the gradients make salient parts of the feature buffers more obvious; such seems to be the case for the gradients in Figure 4.

2.2 Architecture

For single-frame denoising, I use a 28-layer version of the architecture from [4] (see Figure 5 for a diagram). It is fully convolutional and composed mostly of residual blocks, each of which consists of two 3×3 convolutional layers wrapped in a residual connection. In each residual block, I include batch normalization and dropout (during training) after each convolutional layer. Neither paper mentions this, but I found that it helped stabilize the training process (the loss curves did not oscillate as drastically and seemed to converge toward better minimums). The output of the network is a set of 21×21 kernels, one for each pixel.

I also tried the vanilla 9-layer convolutional network from [1], which gave good results as well (Section 3.4.1).

As per [1], I have two networks: one for denoising diffuse data and one for denoising specular data. Each uses the same architecture. Thus, to denoise an image in full, I denoise diffuse and specular data separately, post-process their results (inverting the log transforms and the albedo divide), and finally add them together to form the final denoised color image.

2.2.1 Multiscale Module. The multiscale module serves as a wrapper around the single-frame denoiser. To perform multiscale denoising, I progressively downsample the input image $\mathbf{i}_1$ by a factor of 2 in each dimension using either average pooling or bicubic interpolation to form a three-level image pyramid $\{\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_4\}$. The auxiliary feature buffers are downsampled in the same way, and the variances are divided by 4 each time because downsampling effectively reduces the noise.

I then use the single-frame denoiser to denoise each image in the pyramid, producing a set $\{\mathbf{i}'_1, \mathbf{i}'_2, \mathbf{i}'_4\}$ of denoised images at different scales. Next, I blend the results at adjacent scales according to a trained scale compositor (Figure 6). The scale compositor predicts a per-pixel $\alpha \in [0, 1]$ map via a six-layer convolutional network and then applies it according to the following blending equation at each pixel:

$$\mathbf{o}' = \mathbf{i}'_f + \alpha \left(\mathbf{U}\mathbf{i}'_c - \mathbf{U}\mathbf{D}\mathbf{i}'_f \right)$$

where $\mathbf{i}_f$ represents the denoised fine-scale image, $\mathbf{i}_c$ represents the denoised coarse-scale image, $\mathbf{U}$ represents a 2×2 nearest-neighbor upsampling operator, and $\mathbf{D}$ represents a 2×2 average pooling downsampling operator. $\mathbf{o}'$ is the final denoised output at the finer scale. This equation describes a replacement based on α of denoised finer-scale low frequencies $\mathbf{U}\mathbf{D}\mathbf{i}'_f$ with denoised coarser-scale low frequencies $\mathbf{U}\mathbf{i}'_c$.

The main benefit of the multiscale module is that we can use it in lieu of a very deep network to avoid low-frequency artifacts in denoised images. Normally, we would require a large effective receptive field or filter footprint to properly denoise low frequencies; however, increasing depth and/or filter sizes in the network can be expensive. Multiscale denoising serves as a cheap way to obtain a large spatial support and eliminate persistent low-frequency noise.

When training, the weights of the single-frame denoiser are frozen and only the scale composition module is updated.

The effect of the multiscale module can be seen in Section 4.1.

2.3 Training

I optimize the networks according to an asymmetric loss based on the symmetric mean absolute percentage error (SMAPE) mentioned in [4], which is supposed to be stabler for HDR images and empirically gave better-looking results than the L1 loss of [1].

SMAPE is defined as

$$\ell(\texttt{out}, \texttt{gt}) = \texttt{mean}\left(\frac{|\texttt{out} - \texttt{gt}|}{|\texttt{out}| + |\texttt{gt}| + 0.01} \right)$$

where the mean is taken over all pixels and channels.

2.3.1 Asymmetric Loss. The asymmetric loss at some pixel and channel is defined as

$$\ell'(\texttt{in}, \texttt{out}, \texttt{gt}, \lambda) =$$
$$\ell(\texttt{out}, \texttt{gt}) \cdot (1 + (\lambda - 1) H((\texttt{out} - \texttt{gt})(\texttt{gt} - \texttt{in})))$$

where H is the Heaviside step function (1 if its argument is positive, 0 otherwise). The slope parameter λ is a positive real number which is greater than or equal to 1.

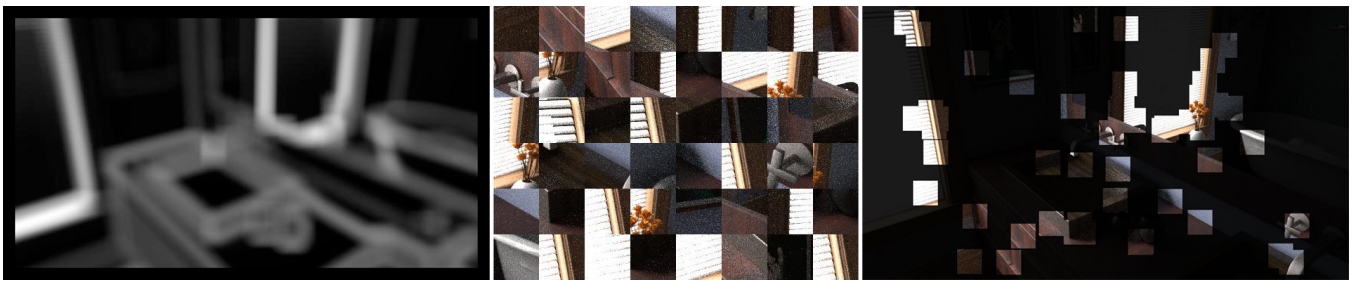

Figure 3: An illustration of sampling coverage. From left to right: *sampling PDF, sampled patches, overlaid sampled patches.*

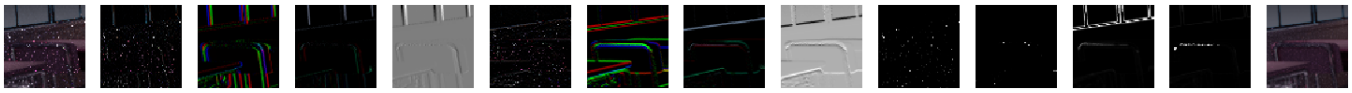

Figure 4: An example of diffuse network inputs used during training. From left to right: *irradiance* (1), *x-gradients for irradiance, normals, albedo, and depth* (4), *y-gradients for irradiance, normals, albedo, and depth* (4), *relative variances for irradiance, normals, albedo, and depth* (4). The rightmost patch is the ground truth irradiance, which is naturally only provided when training.

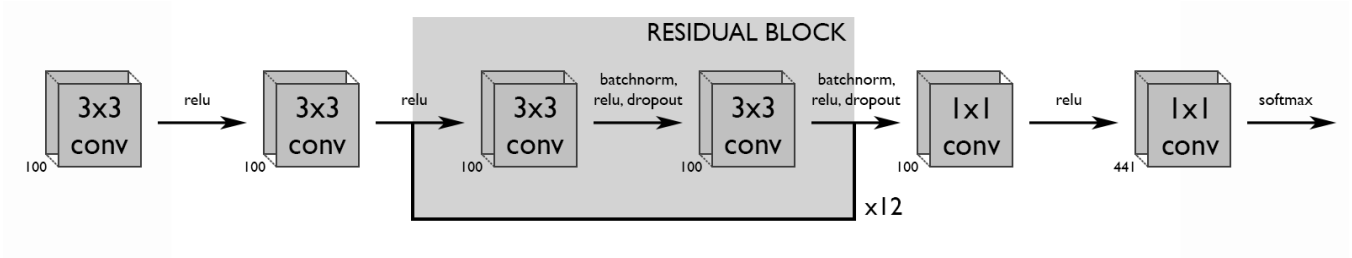

Figure 5: Single-frame KPCN. *Not pictured*: (a) the input, which consists of the color image and a collection of feature buffers, and (b) per-pixel filter application (weighted reconstruction), which happens after the kernel is predicted via the softmax at the end.

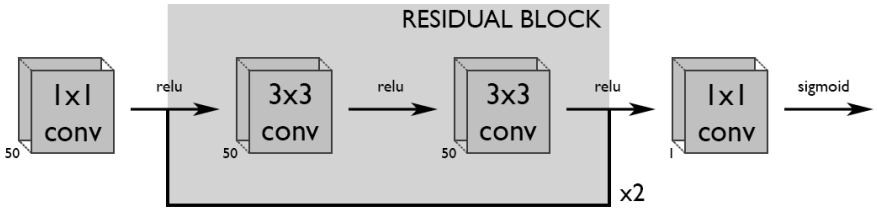

Figure 6: Scale weight (α) prediction module. *Not pictured*: (a) the input, which consists of a concatenated fine-scale denoised image and an upsampled coarse-scale denoised image, and (b) blending, which occurs via the equation presented in Section 2.2.1 after α has been predicted.

The asymmetric loss tells the network to prefer noisier (input-side) output to a degree characterized by λ. This is desirable in some cases because noise corresponds to detail, and some parts of the image might look better if they are slightly noisy as opposed to slightly over-blurred.

According to [4], the authors train a special loss specialization module which sits right before the end of the network and allows per-pixel λ maps to be specified when only a small amount of computation remains. The point of this is that λ can be adjusted on a spatially varying basis and applied at an interactive rate. However, I wasn't expecting to have any artists use my system and did not implement per-pixel λ specification primarily because I didn't have time to construct the necessary inference interface. Instead, I simply trained the entire image on a uniform λ parameter and offered no option to change it during inference (meaning a loss specialization module was not required). The effect of the asymmetric loss can be inspected in Section 4.2.

2.3.2 Other Notes. For the most part, I trained with a batch size of 12, an initial (exponentially decaying) learning rate of 1e-4, and

a dropout keep probability of 0.7. I also utilized gradient clipping, which turned out to be important for stability's sake.

Like [1], the diffuse and specular networks were trained independently and then fine-tuned as part of a cohesive denoising system (i.e. supervised on the final post-processed color output) with an initial learning rate of 1e-6. In the initial phase, I trained each network for about one epoch (60000 iterations) due to time constraints, although the loss seemed to be close to converging. In the fine-tuning stage, I trained for about a sixth of an epoch (10000 iterations).

I used TensorFlow as my deep learning framework, and Adam as my optimization algorithm.

3 INVESTIGATIONS

In this section, I'll describe my observations with respect to different components of the KPCN implementation.

3.1 Multiscale Denoising

In principle multiscale denoising sounds fantastic, but I had a difficult time attaining particularly improved results using the method described in the paper. The first problem I hit was that the α map seemed to tend toward 0 during training, signifying "instead of blending, just use the denoised fine image" and making the results essentially the same as those without multiscale denoising. To get around this, I added an extra weighted term to the loss which provided a linear error signal when the mean α value was less than 0.5.

$$\ell_{\text{new}} = \ell + \gamma \cdot \max\{0.5 - \texttt{mean}(\alpha), 0\}$$

In practice, I usually set γ to 0.01. This extra error term was incorporated twice, once for each application of the scale compositor to a pair of adjacent scales. The reason for setting a threshold at 0.5 is a hypothesis on my part that on average the blending should be able to use equal amounts of the coarse and fine scales, and a quick visual inspection of results for alternative threshold settings suggested that other thresholds induced worse outcomes. Moreover, with this setting I observed predicted α maps that resembled the one shown in [4].

3.1.1 Kernel-Predicting Scale Compositor. In my quest to construct a decent multiscale denoising module, I did a quick literature search and, finding [3], decided to try adopting some of its ideas for my own purposes. It also performs denoising using an image pyramid. For each pair of adjacent scalings $\{\mathbf{i}_f, \mathbf{i}_c\}$ in the pyramid, Choi et al. use a learned filter to upsample $\mathbf{i}_c$ (the denoised output of the coarser layer, *or* a raw image at the coarsest scale) and a learned filter to denoise $\mathbf{i}_f$. Then they sum the upsampled and denoised components to produce the final denoised output of the pyramid level. Accordingly, the filters also implicitly perform blending.

As compared to the approach of Vogels et al., [3] proposes more levels of the image pyramid (although this is flexible), selects ("learns") per-pixel filters from a linear FIR filter bank according to local structure analysis, uses *learned* filters for upsampling, and operates on raw noisy inputs as opposed to having access to KPCN-denoised outputs at different scales. They also denoise and blend independently at each scale, downsample with bicubic interpolation for increased noise reduction, and supervise outputs at all scales with appropriately downsampled versions of the reference image.

I easily adopted the latter two ideas. Meanwhile, I assumed that KPCN denoising would already provide fine-scale denoising functionality and that I would just have to perform upsampling and blending of each coarser scale. For this, I tried using kernel prediction to predict per-pixel 21×21 upsampling/blending kernels and then adding the result to the denoised finer-scale image at each stage of the multiscale processing, i.e.

$$\mathbf{Ui}'_c[k] = \left(\sum_{j \in F} \mathbf{f}[j] \mathbf{i}_c[k/2 + j] \right)$$

where $\mathbf{Ui}'_c[k]$ is pixel k in the denoised output at the finer level, F is the set of valid filter offsets, and $\mathbf{f}$ is the upsampling/blending filter for the coarser image.

To blend, we apply the same α-blending process as before. α is predicted by a separate head of the network, at the same level as the one that predicts the kernels.

There is one confounding factor in this experiment in that my extended network is more complex than the one in [4] and thus has some advantage in representational capacity. In an attempt to facilitate comparison, I use the same scale composition network as before with the only exception being the extra kernel-predicting head. Thus, the only difference is in the upsampling.

Results-wise, I haven't seen any noticeable benefit (or deterioration) from this method. However, it seems to train to near-convergence almost instantaneously (in a couple hundred iterations) and exhibits a smoother loss curve when I let it run for longer. In theory, this extension removes some of the limitations of the single-frame denoiser, since there is the option to perform additional filtering with the "upsampling" kernels. As evidenced by my observations, it may also enjoy something of the increased convergence rates associated with kernel prediction that were discussed in [4].

3.2 Asymmetric Loss

Again, I train with a constant asymmetric loss slope over all spatial locations. As expected, this makes the output generally more (or less) noisy everywhere as I increase (or decrease) the value of the slope parameter.

For the most part, I set $\lambda = 2$ because it appeared to provide a more visually pleasing balance of blurriness and detail as compared to $\lambda = 1$ and $\lambda = 8$. However, I only ran this check on partially-trained networks without multi-scale denoising, and was comparing the output to the 8192 spp renderings, which definitely retain some noise and might have made e.g. $\lambda = 2$ results look undeservedly better than those of $\lambda = 1$ at times..

3.3 Data

3.3.1 Preprocessing. Unlike [1], I take the log transform of both the irradiance and specular color data. I observed better results from this than using raw irradiance as [1] suggests, perhaps due to the data's high dynamic range. It also has the benefit of making single-network training (Section 3.4.5) more sensible, since the same transform is performed on both color inputs.

3.3.2 Iterative Error-Based Sampling. Although I didn't really get the chance to evaluate it, I also added the ability to iteratively

sample more patches from the dataset according to the denoiser's current error on each image. In theory, this could help build up a better and more educational distribution of data for the network so that it can learn to handle the sampling patterns it has trouble with. This is similar to the idea of RL methods such as DAgger (dataset aggregation).

3.4 Architecture

3.4.1 KPCN vs. KPAL Architecture. By "KPCN architecture," I mean the vanilla 9-layer CNN; by "KPAL architecture," I mean the deep residual CNN. Trained to near-convergence, my observation is that these networks produce visually similar outputs, but the KPAL architecture exhibits error values which are just a little bit lower (on the order of 1e-4 or 1e-5 for MrSE and DSSIM).

3.4.2 KPCN vs. DPCN. In agreement with [1], I found that DPCN converged more slowly than KPCN during training but ultimately produced results of similar, perhaps slightly lower quality.

3.4.3 Network Depth. Generally, deeper residual networks seem to perform marginally better but require more memory and more time to execute. One benefit of having a deeper network is that the effective receptive field is larger, meaning low frequencies can be denoised more easily.

3.4.4 Batch Normalization. I found batch normalization useful for training the deep residual network architecture from [4]. In particular, it aided me in avoiding NaNs, which makes sense because by normalizing activations throughout the network batch normalization helps stop things from exploding or vanishing.

3.4.5 Single Network. In the style of [4], I tried using a single network to denoise both diffuse and specular data. It worked, seemingly as well as individual diffuse and specular networks if the validation loss is anything to go by. The main advantages of having a single denoising network are probably computational savings and better data efficiency during training, along with memory savings during inference.

3.5 Training

3.5.1 Batch Size. From limited experiments, it seemed that higher batch sizes were better for training. However, due to memory limitations ~24 was the highest batch size my computer could reasonably handle.

3.5.2 Learning Rate. Although Bako et al. report a learning rate of 1e-5, I found that the slightly more aggressive learning rate of 1e-4 gave rise to better and swifter training. I attribute this mostly to differences between the Tungsten-rendered scenes I was using and the paper's production data.

4 RESULTS

See Figures 7-14 for a collection of results from the single-frame component of my denoising pipeline. For each denoised image, I have also computed the MSE (mean squared error), MrSE (mean relative squared error), and DSSIM (structural dissimilarity) to facilitate comparison with other works' results.

Since the 8192 spp (reference) renderings still contain noise, in many cases the denoised output actually looks *better* in a no-noise sense because it gets rid of noise that the reference images still possess. This is evident from many regions in the included figures. On the other hand, there is some loss of detail at times.

4.1 Multiscale Denoising

See Figures 15 and 16 for an illustration of the effect of multiscale denoising on a shallower 10-layer network. Note that I made the network shallower so that the multiscale noise removal would be more pronounced. Again, multiscale denoising helps remove low-frequency noise in a less expensive fashion than deepening the network at a single scale.

While I saw a reduction in low-frequency noise from multiscale denoising, the results were not exactly comparable to those given by a deeper residual network. However, this might in some ways be a limitation of the underlying denoiser – I only trained each 10-layer network for 16000 iterations (about a quarter of an epoch).

4.2 Asymmetric Loss

See Figure 17 for a comparison of the effects of different settings of λ. As previously stated, higher values of λ increase the network's propensity toward detail preservation.

5 CONCLUSION

In this project, I implemented a KPCN denoising pipeline according to [1] and [4] with the exception of a few modules that I could not use. On top of the base KPCN method from [1], I included multiscale denoising and optimization based on asymmetric loss functions (both from [4]), which allowed for better low-frequency denoising and/or detail preservation. Along the way, I also explored a number of architectural and training settings which I described to some degree in this report. Ultimately, I found the quality of results to be as promised and conclude that kernel prediction is a veritably effective means of denoising Monte Carlo renderings.

REFERENCES

[1] 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 Transactions on Graphics (Proceedings of SIGGRAPH 2017)* 36, 4, Article 97 (2017), 97:1–97:14 pages. https://doi.org/10.1145/3072959.3073708

[2] 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 Trans. Graph.* 36, 4, Article 98 (July 2017), 12 pages. https://doi.org/10.1145/3072959.3073601

[3] Sungjoon Choi, John Isidoro, Pascal Getreuer, and Peyman Milanfar. 2018. Fast, Trainable, Multiscale Denoising. *CoRR* abs/1802.06130 (2018). arXiv:1802.06130 http://arxiv.org/abs/1802.06130

[4] 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 Transactions on Graphics (Proceedings of SIGGRAPH 2018)* 37, 4, Article 124 (2018), 124:1–124:15 pages. https://doi.org/10.1145/3197517.3201388

FIGURE 7: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.00561 / MrSE: 0.00313 / DSSIM: 0.000327

FIGURE 8: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.00792 / MrSE: 0.00541 / DSSIM: 0.00476

FIGURE 9: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.000220 / MrSE: 0.00209 / DSSIM: 0.000169

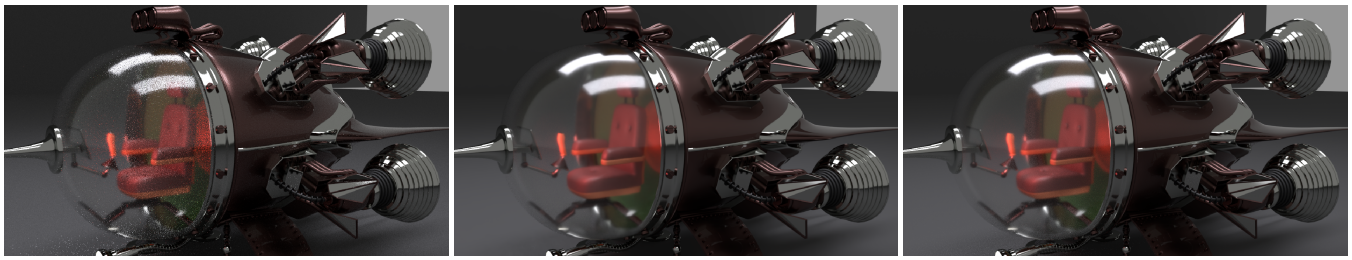

FIGURE 10: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.000340 / MrSE: 0.00162 / DSSIM: 0.00311

FIGURE 11: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.000824 / MrSE: 0.00420 / DSSIM: 0.00557

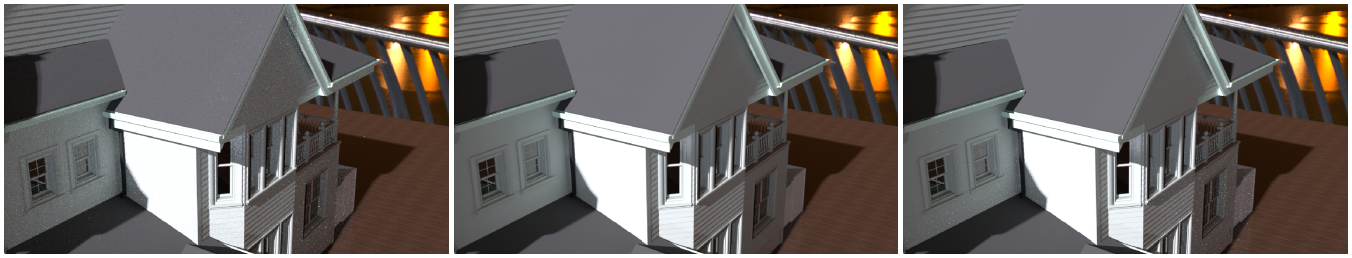

FIGURE 12: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.000922 / MrSE: 0.00310 / DSSIM: 0.000536

FIGURE 13: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.0137 / MrSE: 0.000990 / DSSIM: 0.0000289

FIGURE 14: From left to right: *in* (128 spp, validation set), *out*, *reference* (8192 spp). MSE: 0.0000630 / MrSE: 0.00237 / DSSIM: 0.000597

FIGURE 15: From left to right: *without multiscale denoising*, *with multiscale denoising*, *relative absolute diff between the two*. With multiscale compositing, most of the low-frequency noise on the walls (and everywhere else) goes away. The base single-frame denoiser is the same in both cases; the only difference is that in the multiscale setting we operate at multiple scales and blend the results.

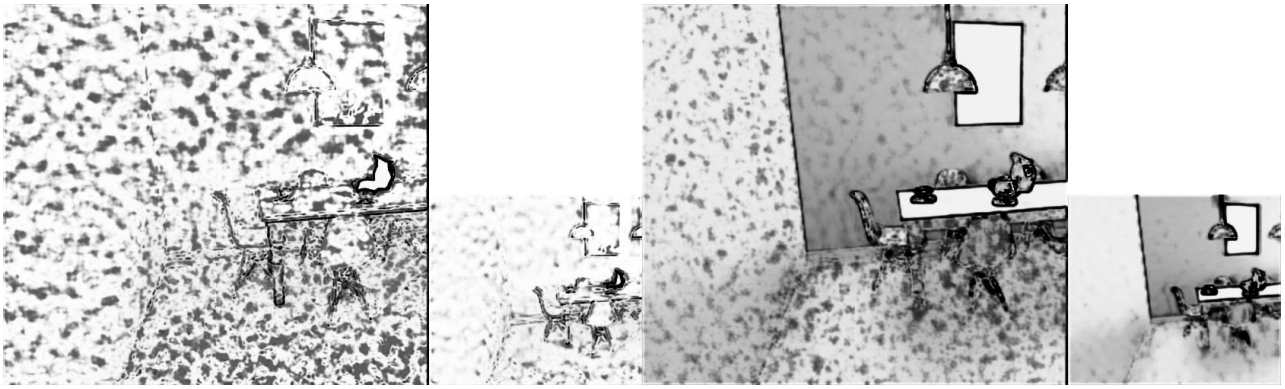

FIGURE 16: Predicted α maps for the multiscale processing in Figure 15. From left to right: *diffuse first-level* α, *diffuse second-level* α, *specular first-level* α, *specular second-level* α. The brightness of α describes how much weight we're giving to the coarse-scale values in the blending process. Not coincidentally, the α maps somewhat resemble the lower-frequency noise in the scene.

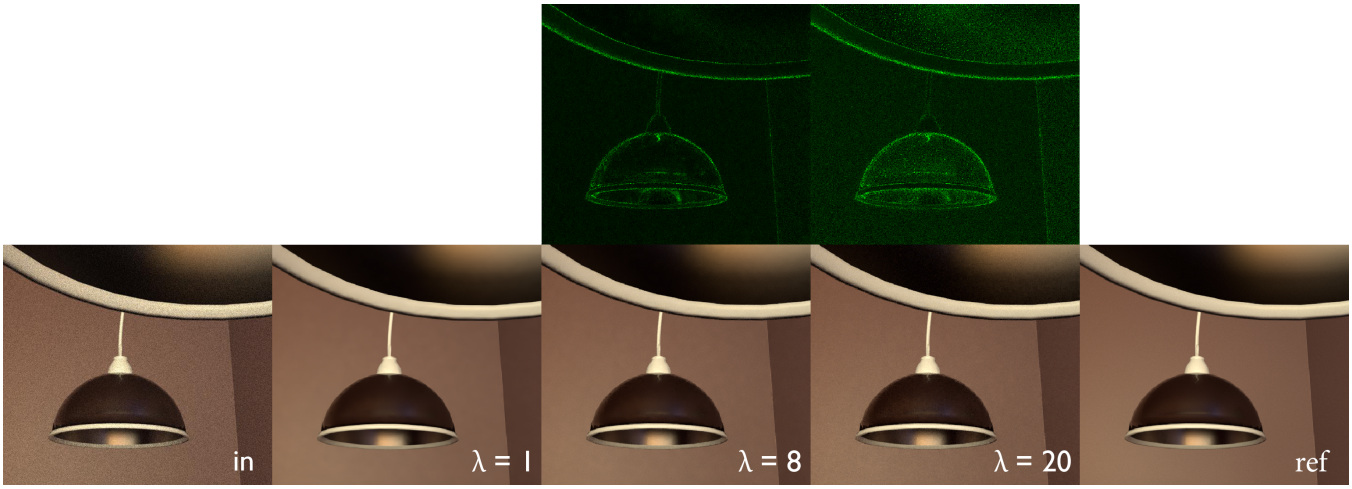

FIGURE 17: An image from the validation set, denoised by three networks which have been trained using $\lambda = 1$, $\lambda = 8$, and $\lambda = 20$. As λ increases, preference for noise over blurriness increases as well. It might be difficult to see at this scale, but there is a distinct increase in residual noise between adjacent outputs. It is particularly noticeable (at least at full size) around the yellowish highlight on the top right, on the surface of the light fixture in the middle, and on the walls in the bottom right corner of the image. The greenish images above the $\lambda = 8$ and $\lambda = 20$ outputs are the corresponding outputs' relative absolute diffs with the $\lambda = 1$ image, provided for convenience of comparison.