A Learning Approach for Adaptive Sampling in Monte Carlo Rendering

Fu-Jun Luan

Abstract
Monte Carlo rendering is widely used in photorealistic images generation, which can simulate a series of real-world visual effects such as soft shadow, glossy reflection, caustics, color bleeding, motion blur, depth of field, and so on. However, because the integration domain contains difficult paths with multiple scattering or inter-reflections, the final image often contains high variance, which can be seen as, the noise. One direction to solve this problem is to do adaptive sampling and reconstruction, which generates final images by adaptive sampling and filtering with single or multiple iterations. In this paper, combined with machine learning, we focus on the adaptive sampling phase which tries to adaptively throw more samples to high error region. As a result, our method leads to better adaptive sampling strategy than state-of-the-arts, and better final images in both visual qualities and MSEs.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction
One key area in Monte Carlo rendering community is sparse sampling and reconstruction, which contains two parts, one is adaptive sampling, and the other is reconstruction. In this paper we focus on adaptive sampling phase. The basic idea of adaptive sampling is to estimate high error regions with some pixel error criterion, and then adaptively throw more samples to those regions.

There are mainly two directions in adaptive sampling, multidimensional adaptive sampling [Hachisuka et al.2008] and image space adaptive sampling [Rousselle et al.2011, 2012; Li et al.2012]. Our method belongs to the image space one, which formulas a sampling map in image space, and adaptively throw samples according to the sampling map.

Sampling Metric. The sampling map is generated by calculating the sampling metric pixel by pixel. The sampling metric should guide more samples to larger MSE regions. In addition, since human eyes are sensitive to dark regions [Overbeck et al.2009], the sampling metric should also guide more samples to darker regions. Following the sample way as [Rousselle et al.2011, 2012; Li et al.2012] did, the sampling function for a pixel $i$ is determined by

$$S(i) = \frac{\text{MSE}(F(c_i))}{l(F(c_i))^2} + \epsilon$$

where $\text{MSE}(F(c_i))$ is the MSE of the filtered pixel color according to ground truth image, $l(F(c_i))^2$ is the squared luminance of the filtered pixel color, and $\epsilon$ is to prevent null denominator, which sets to 0.01 in implementation. After calculating the sampling metric for every pixel, we then normalize the sampling map to sum up to $N$, which $N$ is the sample budget per iteration.

2. Main Algorithm
Our key insight is that the complex relationship between the optimal sampling map and a series of features (which are available in the rendering pipeline) can be well approximated using machine learning methods. In this paper, we use a neural network as the regression model since it’s simple yet efficient. We use a number of scenes to generate training data, which contains features as inputs to the neural network, and optimal sampling maps as the target of the neural network, according to the ground truth image of each training scene.

2.1. Neural Network Structure
We use a neural network with two hidden layers, which contains 8 nodes for the input layer, 20 nodes for the first hidden layer, 10 nodes for the second hidden layer, and 1 node for the output layer (shown in Figure 1).
Figure 1: Illustration of our neural network structure: We use 8 input nodes, 20 nodes and 10 nodes for two hidden layers, and 1 node for output layer. The activation functions for hidden layers and output layer are Sigmoid function and Softplus function, respectively.

The activation function for hidden layers is Sigmoid function 

\[ f(x) = \frac{1}{1 + e^{-x}} \]

and for output layer is Softplus function

\[ f(x) = \log(1 + e^x) \].

With the nonlinear property of Sigmoid function, the neural network can learn complex relationships efficiently. The output range of Softplus is always positive which makes it optimal choice to predict sampling metric in the output layer.

2.2. Features for Neural Network

In this subsection we list all the features we have used as inputs to the neural network. Fujun: This is the key part that we should work on to improve the performance of our NN, e.g. we should find a suitable way to use scene features like normal and texture as inputs.

SURE Metric. Stein [1981] proposed the Stein’s Unbiased Risk Estimator (SURE) which is an MSE estimator. According to [Li et al.2012], it can be well used for Monte Carlo adaptive sampling and reconstruction problem, and we use it as a feature to our neural network. (Note that we use the sampling metric from [Li et al.2012] which divides the luminance of pixel color, instead of directly using SURE estimated MSE.)

MAD Metric. According to Kalantari and Sen [2013], median absolute deviation (MAD) is an efficient metric to estimate noise using wavelet transform. We make use of it as a feature to our neural network.

Dual-Buffer Variance. Rousselle et al. [2012] proposed a non-local means method aims at Monte Carlo denoising technique, in which he obtained output variance estimation by operating on two separate buffers. We follow this method and split the samples to two buffers, then filter them using the same reconstruction filter respectively. After that, we estimate the filtered pixel color’s variance by calculating the squared difference between two buffers pixel by pixel, and smooth it using a cross-bilateral filter. We use it as a feature to our neural network.

Image Gradients. Image gradients are useful in image processing community, such as panorama stitching and Poisson blending. We use the X-axis and Y-axis gradients of filtered MC image as the features to our neural network.

Contrast Metric. Hachisuka et al.[2008] proposed a contrast metric as an error metric. However, we find that this metric can be improved for our project if we take filtered pixel color into consideration, by calculating the difference between the filtered pixel color and the average color of samples in that pixel. Our metric is simply 

\[ \varepsilon(i) = |F(c_i) - \hat{c}|/\hat{c} \],

where \( F(c_i) \) is the filtered color, and \( \hat{c} \) is the average color of samples in pixel \( i \). We use it as a feature to our neural network.

Derivative of Filter. We calculate the derivative of filter \( dF(c)/dc \) and use it as a feature.

Variance. According to central limit theorem, if \( Y \) is the estimated pixel color by Monte Carlo technique, and \( y \) is the ground truth color of that pixel, then

\[ Y \rightarrow N(y, \sigma^2/n), \]

where \( \sigma^2 \) is the variance of samples in that pixel, \( n \) is the number of samples. We use \( \sigma^2/n \) as a feature to our neural network.

Please note that some of the maps (e.g. SURE Sampling Map, Ground Truth Sampling Map) can be very noisy. According to Rousselle et al. [2012], we should pre-filter them using a low-pass filter to ensure high error region to be well sampled. To understand that, consider a scene with DOF effect (shown in Figure 2), the noisy sampling map will contain many holes and outliers in out-of-focus region, this smoothing step allows pixels that miss the intersection of primary rays to still get samples, if there local nearby pixels have high error.
2.3. Training

We use the resilient backpropagation (RPROP) [Riedmiller and Braun 1993] as the training algorithm, which takes advantage of the signs of gradients to speed up the training process on flat error surfaces where the magnitudes of gradients are small.

Intuitively thinking, we may train the neural network pixel by pixel, feeding one pixel’s features to the neural network, minimizing the error between the output of the neural network and the ground truth sampling metric in that pixel. This way we can accumulate the error gradients once we have processed a pixel, then we repeat the same step for the rest of pixels in each training scene.

However, this is not the optimal choice. It’s because the normalization step the renderer will do before adaptive sampling, which normalizes the whole sampling map to sum up to \( N \), where \( N \) is the number of sample budget for that iteration. As a result, our sampling map which is obtained by looping over the whole image recording the output of the neural network will be normalized by the renderer. Because of the normalization step, the map can be influenced by any pixel in the image, which leads to unstable sampling maps and poor results.

To avoid this problem, we propose a normalization block (Figure 3) and integrates it into the training process, that is, for each training scene we loop over the whole image recording the output of our neural network, then normalize the outputs of the neural network to sum up to the number of sample budget just as the renderer will do before adaptive sampling, after that, we compute the error between the normalized outputs of neural network and ground truth sampling map, then accumulate the error gradients back through the normalization block. After looping over the whole training scenes, we update the weights of our neural network. This is one iteration of our training process.

**Backpropagation with Normalization Block.** In order to achieve such a training system, we must obtain the error gradients formulation with the normalization block.

To measure the error between our normalized outputs of the neural network and the ground truth sampling map, we use the least mean log squares (LMLS) method [Liano 1996] instead of standard MSE since LMLS can handle outliers more robustly:

\[
E_i = \log(1 + \frac{1}{2} (x'_i - y_i)^2),
\]

where \( x'_i \) is the normalized output of the neural network for pixel \( i \), \( y_i \) is the ground truth sampling metric for pixel \( i \). Then the error gradient for \( x'_i \) is

\[
\frac{\partial E_i}{\partial x'_i} = \frac{(x'_i - y_i)}{1 + \frac{1}{2}(x'_i - y_i)^2},
\]

with the relationship \( x'_i = \frac{\sum_j x'_j}{N} \), where \( N \) is the total number of pixels in that training scene image, using the chain rule, we formula the error gradient for \( x_i \):

\[
\frac{\partial E_i}{\partial x_i} = \sum_{j=1}^{N} \frac{\partial E_i}{\partial x'_i} \frac{\partial x'_i}{\partial x_i}
\]

this way we obtain the error gradients for neural network output node, and we can run training process using normal gradient descent algorithms.