Path Tracer Image Reconstruction Using Denoising Autoencoder

Tristan Axelsson Smith

***Abstract*— In this report the implementation of a path tracer and denoising autoencoder is presented. The path tracer outputs noisy images that are reconstructed using the autoencoder. The reconstruction worked well on images that where trained on but did not generalize well.**

I. Introduction

In the world of computer graphics the two most common ways of drawing 3D objects on the screen is rasterization or ray/path tracing. Rasterization being the dominant technique in real time applications because of its performance, and ray/path tracing being used more in offline settings for its photorealistic nature. Ray tracing for real time use does exist but does not perform as well as rasterization and using path tracing for real time applications does require a large decrease in the sample count. This gives rise to very noisy images.

In the field of machine learning autoencoders are artificial neural networks that learn to represent their input in a compressed way. An autoencoder has an encoding and decoding stage. In the encoding stage the network compresses the input to some compressed representation and in the decoding stage the encoded data is then decoded to form the input. The autoencoder is therefore trying to learn the identity function. A denoising autoencoder has noisy input and tries to learn to denoise the input.

For this project a monte carlo path tracer for global illumination with a denoising autoencoder was implemented. The path tracer generates a noisy image for the denoising autoencoder to denoise.

II. Related Work

Ray tracing is a well known technique in computer graphics [1]. What it and path tracing try to solve is the famous rendering equation [2]. Convolutional neural networks have been used for image denoising before [3] [4] [5]. Previous work on denoising monte carlo rendering has also been made [6] [7] [8], also using machine learning for real time [9] and offline use [10].

III. Method

First the path tracer is presented and after that the denoising autoencoder.

A. Path Tracer

A path tracer, not to be confused with a ray tracer, uses monte carlo integration to solve the rendering equation. Rays are traced from the camera for each pixel. Then the ray bounces around the scene until it hits a light or some bounce limit is reached [11]. At the first intersection many rays can be traced from that point. The final color for that point is the average from all the rays. This is how monte carlo is used for path tracing. Since only Lambertian reflection is used the color of a point is view independent.

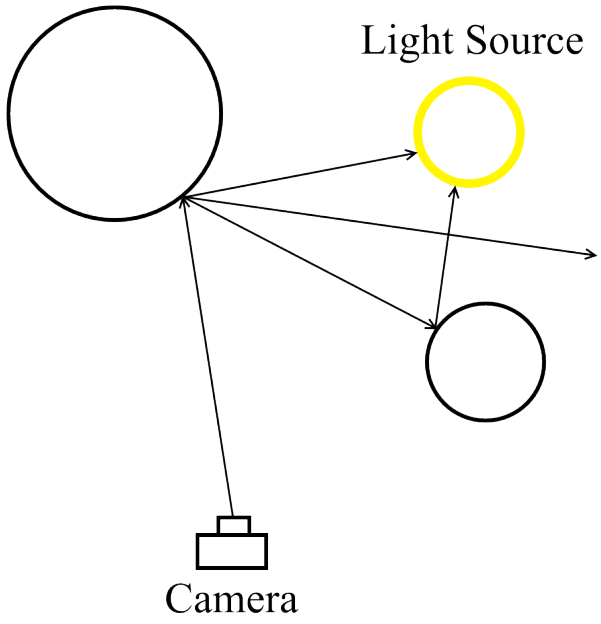


Fig. 1: Diagram showing rays for a single pixel with 3 SPP. The first ray from the camera intersects the scene and creates 3 rays from that point. From the first intersection one ray hits a light source, one misses and one hits another circle creating a new ray from that point.

The path tracer was implemented in C++ using OpenGL. A quad drawn to fill the screen giving a fragment for each pixel. For each pixel a ray is shoot into the scene. At the first intersection the distance from the camera and the normal is saved. At that point if the surface is a light source then the color of that pixel will be the color of the light source. If the surface is not a light source then from that point many rays are created pointing in the hemisphere aligned to the normal of that point. The number of rays is defined as the number of *samples per pixel* (SPP). In the implementation later intersections of these created rays do not recursively create more rays like the first intersection. Instead they create a single ray from each intersection. This is shown in figure 1.

Choosing the direction to bounce a ray can be done uniformly across the hemisphere. But it can be beneficial

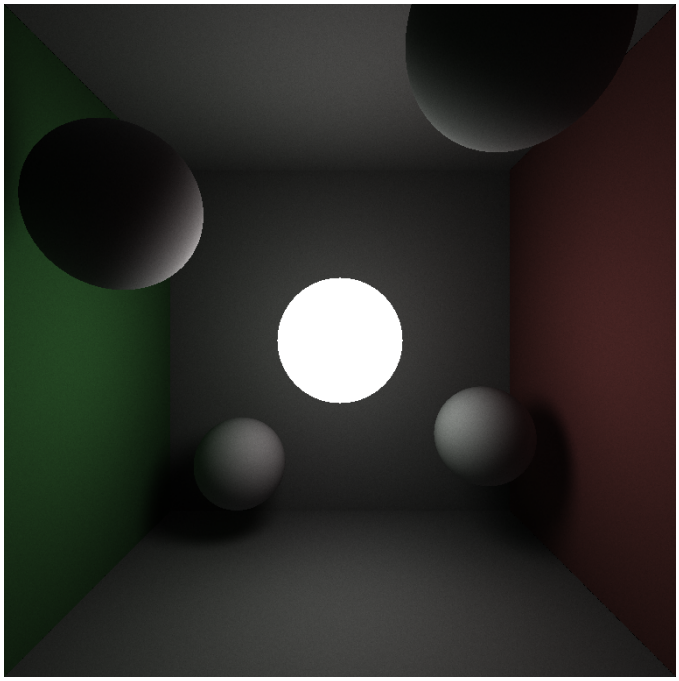


Fig. 2: The scene used in the implementation. The number of samples per pixel is 16384

to let the direction be cosine weighted. Since the intensity of the light is calculated from the dot product it is better to choose direction that can make a larger impact on the end color [11] [12].

The scene used in the implementation is an imitation of the Cornell box. Inside the box there are five spheres that move around scene. One of the sphere is a light source emitting a white light. The scene is shown in figure 2.

When the color of a pixel is found it is stored in a texture. The depth of the first ray and the x and y components of the screen space normal are also stored into a texture. These will be the input for reconstruction later.

B. Denoising Autoencoder

The output of the path tracer will be very noisy. Instead of using a higher number of samples per pixel a denoising autoencoder is used to reconstruct the image. The structure of the network used in the implementation is based on [9] with the exception of not using recurrent convolution. It was implemented in python using the machine learning library Keras [13]. The network consists of layers of convolutions and pooling/up-sampling. The input to the network is the output of the path tracer which is The path tracer outputs six channel 768x768 images divided into 36 128x128 chunks. The first three channels are the colors, the next two is the x and y components of the screen space normal, and the last is the depth. The network structure is shown in figure 3. Skip connections [4] are used in the network, connecting convolutional layers of the encoding and decoding stage symmetrically. The skip connection serve to avoid the information loss that can happen with autoencoders. A 3x3 filter was used for all the convolutions.

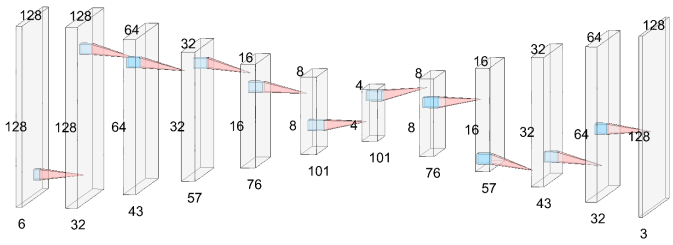


Fig. 3: The structure of the denoising autoencoder. After each convolution leaky ReLU is used except for the last layer that is linear.

C. Training

To train the network the path tracer outputs high sample count images and their corresponding noisy versions. The noisy image will be used as the input for the network. The output of the network will then be compared to the high sample version. The loss function that was used is the mean absolute error [9]. Different optimizers where tested and Adam was choosen because of its faster convergence. The parameters used where: learning rate = 0.001, β_1 = 0.9, β_2 = 0.99. For training 7200 images where used. The images where split into ten equal parts. Nine of the parts where used to train the network and the rest where used to evaluate it. The 7200 images come from 200 768x768 images. Each image has different camera and sphere positions. The camera is panned in a random direction. To make the network better understand the noisy images each pair of images are have the same position for the camera and spheres but the seed used for choosing random ray directions is different. This forces the network to learn to output the same image that has different noisy input. The network was trained for iterations of 20 epochs. After 20 epochs had passes the network was saved to the disk and the next iteration of epochs started. In the final implementation 16 samples per pixel where used in the path tracer for the noisy images.

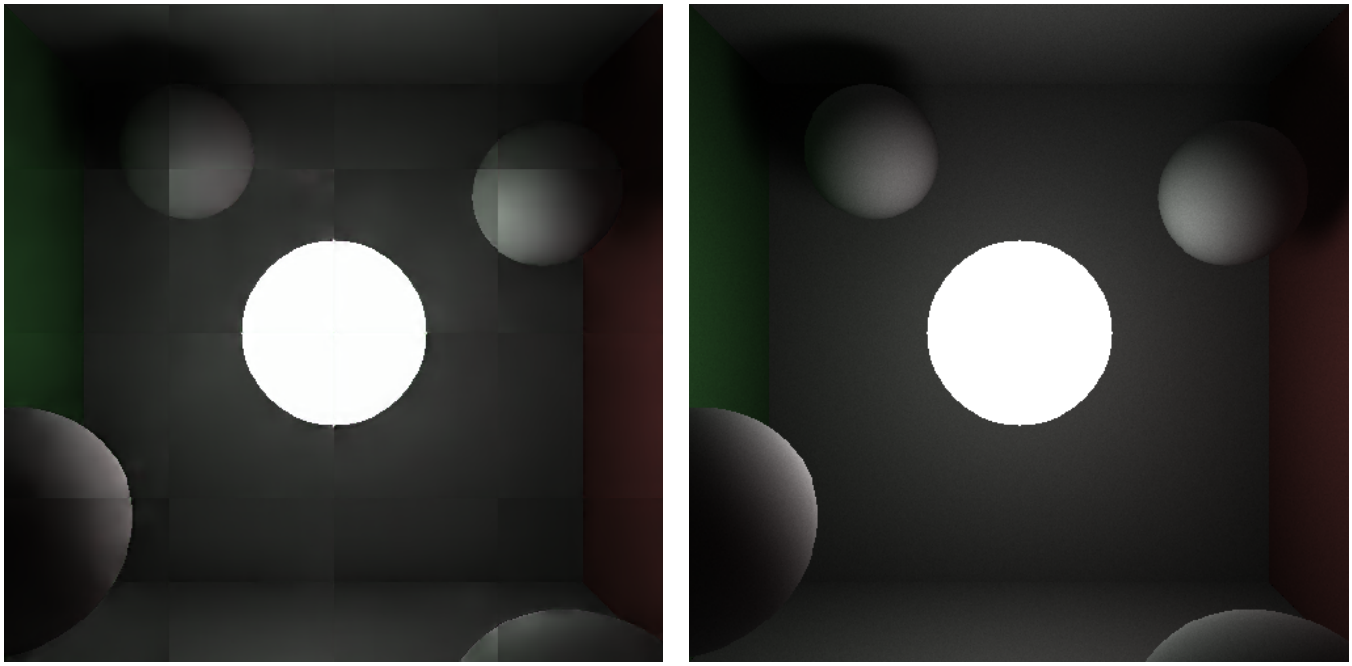
IV. Results & Analysis

The network used was trained for 9 hours on a GTX 980. The loss for the evaluation data converged to 0.006 and the loss for the training data converged to 0.0048. Figure 4 shows the the input, output and reference images. On the reconstructed image blocky artifacts can be seen. These are from applying the network on 128x128 chunks of the image. To combat this one could apply the filter in a overlapping manner and interpolate the borders to the output of the closest image.

The network did not generalize well. This can be seen in figure 5. The blocky artifacts are even more prevalent. The proposed solution of applying the filter overlapped may work if the overlap is large.

(a) Color channels of noisy input. 16 SPP (b) The reconstructed image (c) Reference. 16384 SPP

Fig. 4: Images that the network trained on.



(a) The reconstructed image (b) Reference image

Fig. 5: Image that the network did not train on

V. Conclusions

To conclude in this report the implementation of a path tracer outputting noisy images that where reconstructed using a denoising autoencoder was presented. The network worked well on images that where trained on, but did not generalize well. The method of reconstructing monte carlo rendering using machine learning is an interesting solution for real time use. With NVIDIA's release of the RTX lineup of graphics cards that include both ray tracing and tensor cores makes this method even more viable to investigate.

References

[1] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[2] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[3] Viren Jain and Sebastian Seung. Natural image denoising with convolutional networks. In *Advances in Neural Information Processing Systems*, pages 769–776, 2009.

[4] Xiao-Jiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using convolutional auto-encoders with symmetric skip connections. *arXiv preprint arXiv:1606.08921*, 2016.

[5] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 341–349. Curran Associates, Inc., 2012.

[6] Soham Uday Mehta, Brandon Wang, Ravi Ramamoorthi, and Fredo Durand. Axis-aligned filtering for interactive physically-based diffuse indirect lighting. *ACM Trans. Graph.*, 32(4):96:1–96:12, July 2013.

[7] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik Lensch. Edge-avoiding à-trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, pages 67–75. Eurographics Association, 2010.

[8] Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. Sure-based optimization for adaptive sampling and reconstruction. *ACM Transactions on*

Graphics (TOG), 31(6):194, 2012.

[9] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4):98:1–98:12, July 2017.

[10] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Trans. Graph*, 36(4):97, 2017.

[11] Jean-Colas Prunier. Global illumination and path tracing. `https://www.scratchapixel.com`. accessed 2019-01-13.

[12] Edd Biddulph. Lambertian reflection without tangents. `http://www.amietia.com/lambertnotangent.html`, 2017.

[13] François Chollet et al. Keras. `https://keras.io`, 2015.