Deep Learning for Lighting Simulations

Student
Peter Quinn
ECSE 499 – W2019

Supervisor
Derek Nowrouzezahrai

Abstract

High quality images and videos, such as those seen in media like film and advertisements, use a technique called ray tracing to produce highly detailed images. This technique takes significant time to generate images due to the millions of individual light rays that must be simulated and complex computations that must be performed.

The focus of this project is to improve the speed at which high quality images can be rendered by preferentially simulating light rays that have a significant contribution to the image. The goal is to do this using machine learning. A neural network will learn how the light is distributed in a scene and can then be used to select optimal paths to trace.

In this work, a neural network architecture known as a Real Non-Volume Preserving (RealNVP) network is used. This network can be trained to replicate an arbitrary complex multi-dimensional probability distribution, and subsequently generate new points from this distribution.

The network was implemented using the PyTorch python library. The training data was generated using the PBRT-V3 renderer. Once the network was trained, it was used to generate new samples that would hopefully contribute to the output image more reliably than randomly tracing rays. These new samples were loaded into PBRT-V3 and used to render images.

When tracing an equal number of rays, it was possible to observe an improvement in the quality of the images rendered using paths generated by the trained neural network compared to images rendered using randomly generated paths.

Contents

Abstract 1
List of Abbreviations and Notation 3
Introduction 4
Background 5
- Ray Tracing 5
- Computing Global Illumination in 3D Scene 6
  - Monte Carlo Integration and Path Tracing 8
  - Path Space Integral Formulation 10
- Deep Learning 12
- RealNVP Neural Network Architecture 13

Requirements and/or Problem 16
Design and Results 17
- Design 17
- Results 18

Impact on Society and the Environment 24
Conclusion 25
- Future Work 25

References 26
Appendix A: Additional Renders 27

List of Abbreviations and Notation

BRDF: Bidirectional Reflectance Function

CAD: Computer Aided Design

CDF: Cumulative Distribution Function

CGI: Computer Generated Images

LSTM: Long Short-Term Memory

MC: Monte Carlo

MIS: Multiple Importance Sampling

MSE: Mean Squared Error

NN: Neural Network

PDF: Probability Density Function

PSS: Primary Sample Space

RealNVP: Real Non-Volume Preserving

RNN: Recurrent Neural Network

RRF: Radiance Regression Function

SGD: Stochastic Gradient Descent

SPP: Samples per Pixel

Introduction

Computer graphics and computer-generated images (CGI) are ubiquitous in modern society. From special effects and animated films, to TV commercials and visualizations in CAD tools, computer graphics are used daily.

Physically based ray tracing is a powerful technique that can produce accurate photorealistic images. However, the main downside of this technique is the significant time it can take to generate a high-quality image. Generating a single image can take anywhere from minutes to hours depending on the complexity of the scene and the accuracy desired.

The calculations involved in simulating millions of light rays, bouncing an arbitrary number of times in a 3D scene, is very computationally intensive. Even on specialized hardware, it can take significant time to trace enough rays for the image to converge to an accurate representation of the scene. As such, it is very desirable to speed up the ray tracing process. Several techniques for this have been employed historically, such as multiple importance sampling (MIS) [1] and photon mapping.

In this project, we attempt to leverage recent advances in machine learning to speed up the convergence of images by using a neural network model to help select important paths in a scene. Here, important paths are the paths that transmit a significant amount of light along them, that is eventually directed towards viewer. Since the important paths contribute most significantly to the final image, by sampling these preferentially it should be possible to increase the rate of convergence and reduce the noise in the final image. This paper will focus on surface illumination.

Background

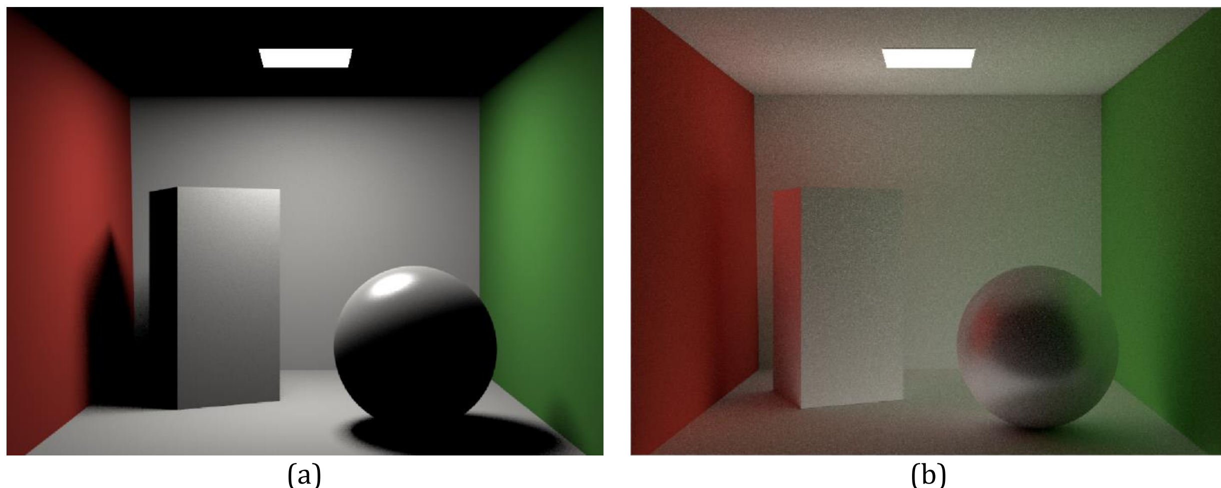
The background section is divided into three main sections. First an introduction to ray tracing is given. Second, the rendering equation is introduced and numerical techniques for solving it are presented. Finally, information on deep learning is presented.

Ray Tracing

Ray tracing the preferred technique for creating high-quality computer-generated images as this technique can render photorealistic images with a high degree of accuracy. However, the main downside of this technique is that it takes significant time to create images. This is due to the computational complexity of simulating millions of light rays travelling and bouncing through a 3D scene. As such, this technique has been restricted applications where images and videos can be rendered ahead of time, then simply played back. This precludes the technique from being used in interactive applications such as video game.

Most often in when rendering scenes, we are interested in computing the “global illumination” at the points we can see from our viewing position. Global illumination refers to the illumination at the point due to both the light coming directly from light sources (direct illumination) and the indirect light due to reflections off of other objects in the scene (indirect illumination).

An example of direct and indirect illumination in a scene with a single light source (emitter) is shown in Figure 1. The final scene that results from the sum of both direct and indirect illumination is show in Figure 2.



(a) (b)

Figure 1: (a) Direct illumination in simple scene. Note the lack of illumination on the ceiling and on the left side of the rectangular prism. These areas are not in direct view of the light. (b) Indirect illumination in the same scene with each light ray allowed to bounce up to 4 times. Images rendered in tinyrender using scenes provided as part of ECSE 546

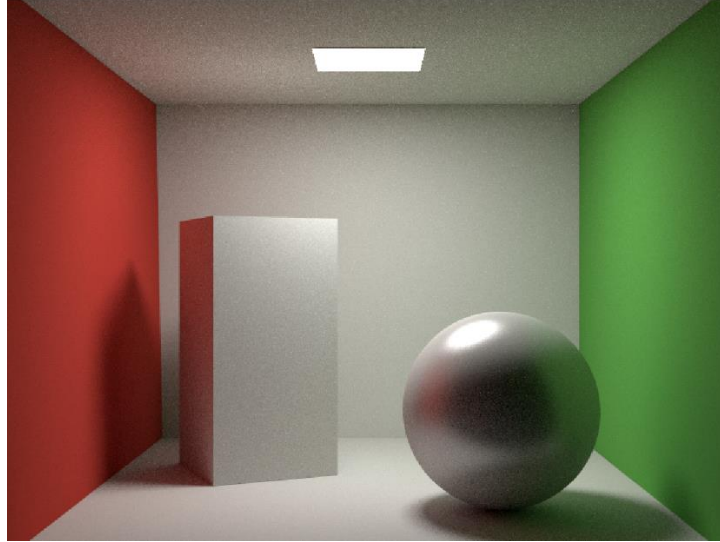


Figure 2: Global illumination in a simple scene. The global illumination is the sum of the direct and indirect illumination components. Indirect illumination was capped at 4 light bounces. Image rendered in tinyrender using scenes provided as part of ECSE 546

Computing Global Illumination in 3D Scene

The global illumination at a point $\mathbf{x} = (x, y, z)$ is given by the rendering equation [2]:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + L_r(\mathbf{x}, \vec{\omega}_o)$$

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) \cos \theta_i \, d\vec{\omega}_i$$

This equation shows that the outgoing radiance L_o (Units: $W \cdot sr^{-1} \cdot m^{-2}$) at the point is the sum of the radiance emitted by the point L_e and the reflected radiance L_r. It further shows that the reflected radiance can be written as the integral over the hemisphere, where the integrand is the product of the incidence radiance L_i, the bidirectional reflectance function (BRDF) f_r, and the foreshortening term $\cos \theta_i$ for every solid angle in the hemisphere.

The incident radiance $L_i(\mathbf{x}, \vec{\omega}_i)$ represents the amount of light energy incoming from a direction $\vec{\omega}_i$ towards the point **x.** The BRDF $f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$ describes the amount of light reflected in direction $\vec{\omega}_o$ at the point **x** that was incident from direction $\vec{\omega}_i$. The foreshortening term $\cos \theta_i$ is a factor that reduces the transmitted radiance when the incident radiance approaches from an extreme angle, where θ_i is the angle between the surface normal and the incident radiance. This factor ensures physical accuracy.

The parameters of the functions are shown below in Figure 3.

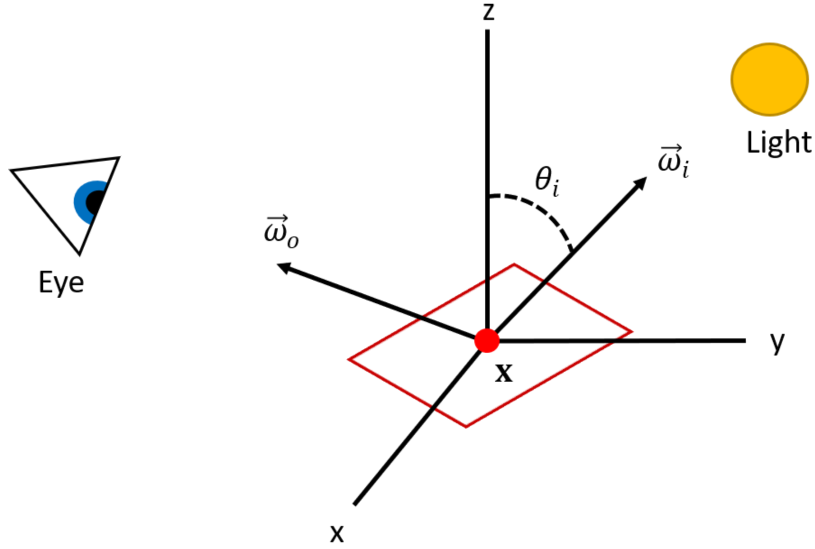


Figure 3: Diagram showing the relationship between $\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o$, the light source and the viewing point represented by the eye.

In practical implementations, global illumination must be calculated for each of the three colour channels (RGB). The red, green and blue components of the BRDF at any point will depend upon the colour of the surface the point lies on.

The incident radiance at point $\mathbf{x}$ from direction $\vec{\omega}_i$ is equivalent to the outgoing radiance from another point in the scene. This point will lie at the intersection point of the ray originating at $\mathbf{x}$ in direction $\vec{\omega}_i$ and the first surface it intersects with in the scene. Using this, we can reformulate the rendering equation into the following form:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_o(r(\mathbf{x}, \vec{\omega}_i), -\vec{\omega}_i) \cos \theta_i \, d\vec{\omega}_i$$

$r(\mathbf{x}, \vec{\omega}_i)$ is the ray tracing function which returns the point of intersection between the ray traced in direction $\vec{\omega}_i$ from point $\mathbf{x}$ and the first surface that this ray intersects. A diagram demonstrating this reparameterization can be seen in Figure 4.

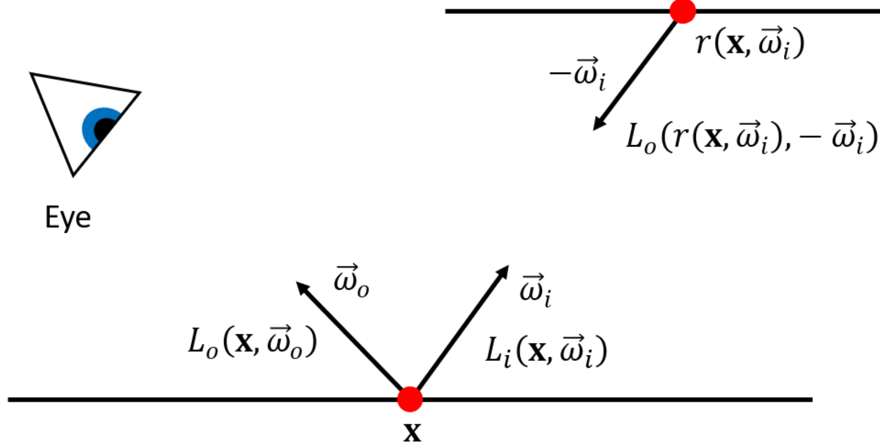


Figure 4: Reparameterization of incident light at point x in terms of outgoing radiance at another point found using the ray tracing function.

We can see that this equation now takes an interesting form. The dependence on the incident radiance $L_i(\mathbf{x}, \vec{\omega}_i)$ has disappeared and the outgoing radiance $L_o(\mathbf{x}, \vec{\omega}_o)$ now appears on both sides, where it appears inside the integral on the right-hand side. This makes the function recursive. This type of equation is known as a Fredholm equation of the second kind.

Solving the rendering equation is the primary challenge in rendering realistic scenes. Several algorithms exist that try to solve the rendering equation efficiently. This paper focuses on the path tracing algorithm. This algorithm relies on Monte Carlo (MC) methods to numerically approximate the integral term.

Monte Carlo Integration and Path Tracing

Monte Carlo (MC) integration is a technique for calculating a definite integral numerically using random numbers. This technique for integration is frequently in graphics to evaluate integrals.

If random variable X with probability distribution $p(x)$, and $\frac{f(x)}{p(x)} < \infty$ for all x then the following holds:

$$\int_{\Omega} f(x)du = \int_{\Omega} \frac{f(x)}{p(x)} p(x)du = E\left[\frac{f(X)}{p(X)}\right]$$

The expectation can be approximated using a finite number of samples N drawn from $p(x)$ using the following MC estimator:

$$\int_{\Omega} f(x)du \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

Essentially, the integral is approximated as the average of the integrand $f(x)$ weighted by $p(x)$ for N samples drawn according to the probability distribution $p(x)$. In the limit where $N \to \infty$, a strict equality holds.

MC integration is an attractive technique because the method requires only three things:

1. A probability distribution function $p(x)$ that is non-zero everywhere $f(x)$ is non-zero
2. The ability to draw random samples according to $p(x)$

3. The ability to evaluate the integrand $f(x)$ at the point X

MC integration provides an unbiased estimator for an integral. It is easily extendable to higher dimensions, where $x \rightarrow \bar{x}$ and $f(\bar{x})$is scalar. MC integration is very useful in evaluating higher dimensional integrals because it does not become significantly more complex, we simply need to sample according to $p(\bar{x})$ and the random variable $X_i \rightarrow \bar{X}_i$ is now a vector.

The main downsides associated with MC integration are slow convergence of $O\left(\frac{1}{\sqrt{N}}\right)$ and noise associated with the variance due to the random sampling.

In the context of path tracing and the rendering equation, it is possible to satisfy all three conditions necessary for Monte Carlo for the integral term in the rendering equation. Our MC estimator for the integral will be:

$$\int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_o(r(\mathbf{x}, \vec{\omega}_i), -\vec{\omega}_i) \cos \theta_\mathrm{i} \, d\vec{\omega}_i$$

$$\approx \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(\mathbf{x}, \vec{\omega}_i(\bar{X}_i), \vec{\omega}_o) L_o\left(r\left(\mathbf{x}, \vec{\omega}_i(\bar{X}_i)\right), -\vec{\omega}_i(\bar{X}_i)\right) \cos \theta_\mathrm{i}}{p(\bar{X}_i)}$$

Unfortunately, attempting to solve even this estimator for a significant number of samples while allowing for several bounces of indirect light is not practical. The recursion in the rendering equation results in an exponential growth of the paths needed to trace to compute the radiance at a point, which would require an impractical amount of computer memory and compute time.

This limitation leads us directly to the path tracing algorithm. Instead of taking N samples at a point, we will instead take a single sample and repeat this for however many bounces. If a sufficient number paths for each pixel in the image are traced and have their radiance contributions averaged, the result will still converge to a realistic image.

If the path length is fixed to maximum depth, bias will be introduced into the estimation. This bias can be removed by employing a technique called Russian roulette which stochastically determines when to terminate the path with some probability p, and reweights radiance contributions of path segments according to p.

There are several different subtypes of path tracing including implicit path tracing, explicit path tracing and bidirectional path tracing [3].

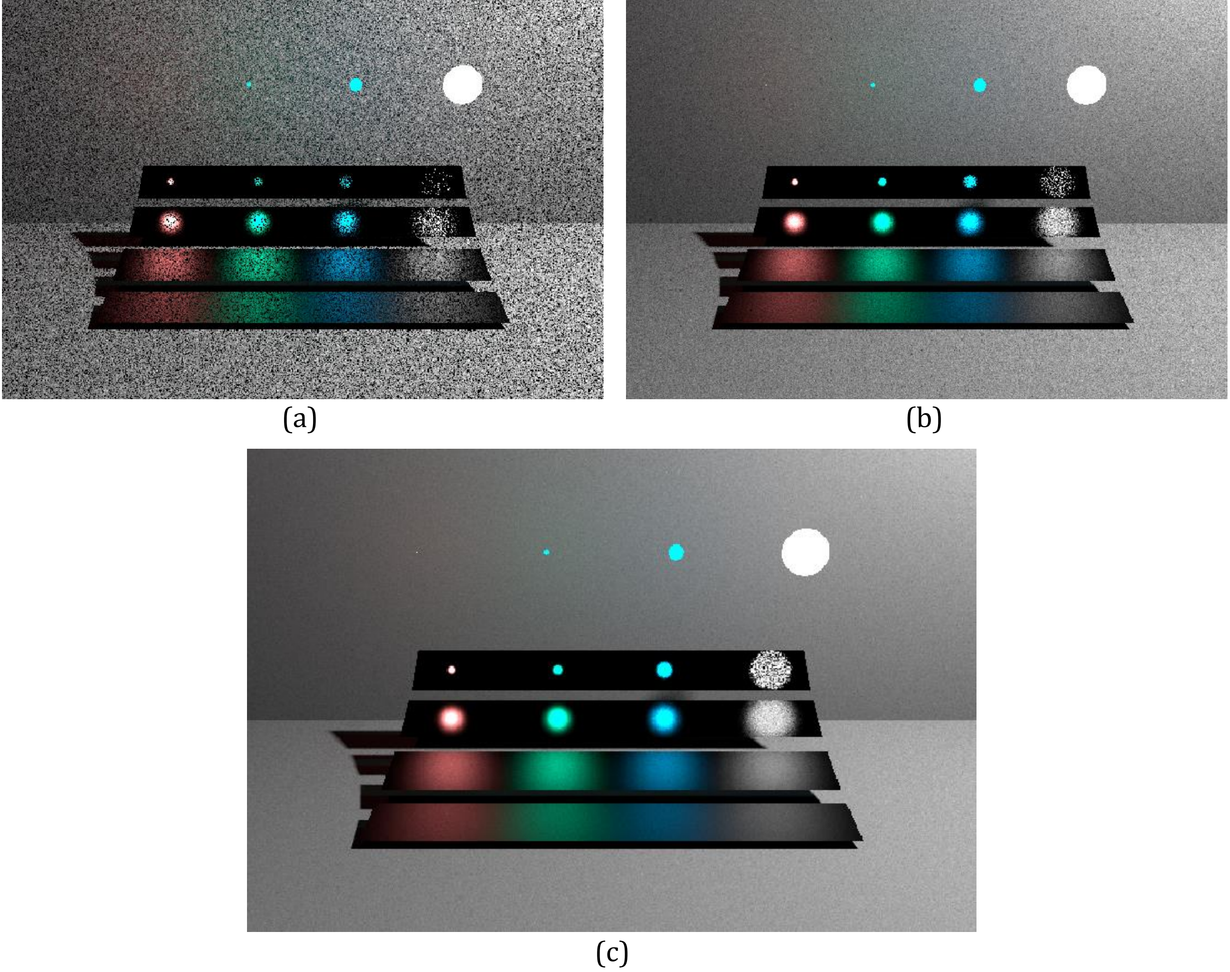


Figure 5: Images generated using MC integration (emitter sampling). The images use increasing numbers of samples when computing the integral. The images use 8, 64, 512 samples respectively

Path Space Integral Formulation

The solution for global illumination in a scene can also be considered using what are known as path integrals [4]. Path integrals consider drawing the image onto the screen by iterating over each pixel I_j and computing the radiance contribution to the pixel by adding up all the contributions from all the paths of arbitrary lengths that can be traced from the pixel. This summation can be expressed as an integral over "path space" P.

$$I_j = \int_P W_e(x_0, x_1) L_e(x_k, x_{k-1}) T(\bar{x}) d\bar{x}$$

Where $\bar{x}$ is a vector parametrizing the path, $W_e(x_0, x_1)$ is the response of the of the sensor (pixel) at location x_0 with radiance arriving from the direction of x_1, $L_e(x_k, x_{k-1})$ is the radiance emitted by the emitter at the termination of the path, and $T(\bar{x})$ is what is the throughput of the path. $T(\bar{x})$ is composed of the products of the BRDFs at all vertices and the geometry terms that result from the reparameterizations of directions to points in the BRDFs.

$$T(\bar{x}) = G(x_0, x_1) \prod_{i=1}^{k-1} f(x_i, x_{i+1}, x_{i-1}) G(x_j, x_{j+1})$$

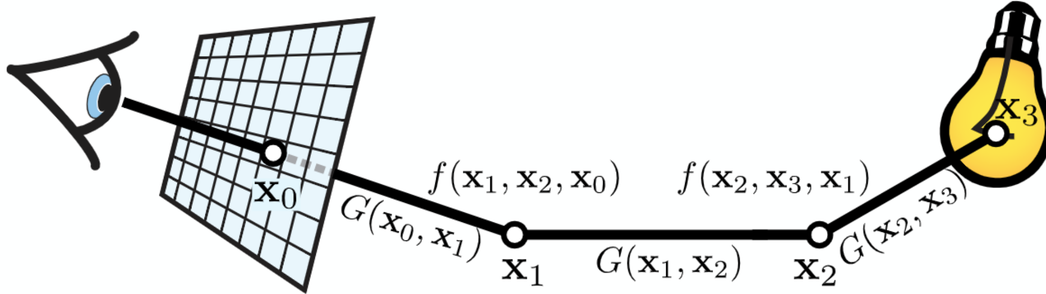


Figure 6: Visualization of a single path in path space and the terms involved in solving for the radiance at the pixel I_j (Source: ECSE 546 Path Space II lecture notes)

A MC estimator can be constructed for the path space integral.

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{W_e(x_{i,0}, x_{i,1}) L_e(x_{i,k}, x_{i,k-1}) T(\bar{x})}{p(\bar{x}_i)}$$

This estimator relies on being able to sample paths according to some joint pdf $p(\bar{x})$. This can be done through normal path tracing, using some local sampling strategy at each point on the path and then computing the total probability of the path as the product of the probabilities at each vertex.

Another way of constructing a path is to sample from a unit hypercube of dimension $2(k+1)$, where k is the length of the path. This space is referred to as Primary Sample Space (PSS) [5]. This method for constructing paths assumes that all points lie on surfaces and are parametrized by two parameters, which can be thought of as the ϕ, θ directions in which to trace the next segment from the current point.

Deep Learning

Deep learning is a branch of machine learning that has gained signification traction over recent years. Deep learning is done though neural networks, which are computing systems designed to vaguely mimic the way organic brains work.

The principles behind deep learning are relatively simple to understand. The inputs are related to the outputs using a large network of interconnected nodes, called units. A neural network can be viewed as very flexible function. In the limit case, with an infinite number of units, it can be considered an infinitely flexible function [6].

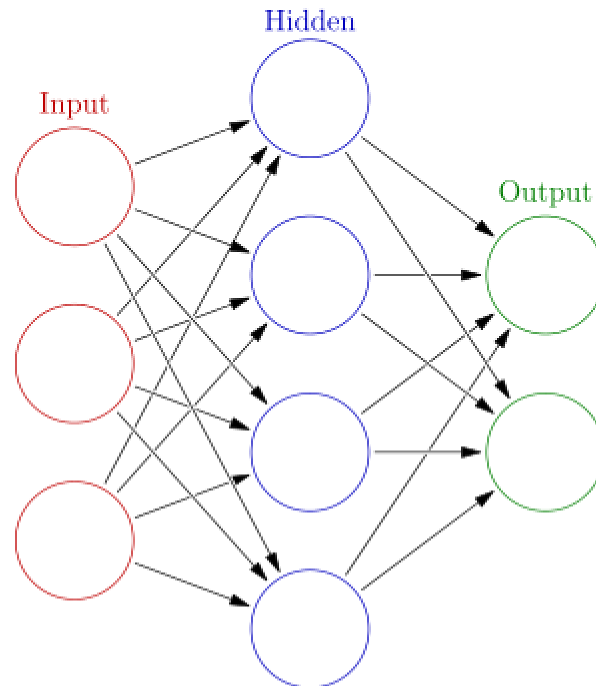


Figure 7: The interconnections between units in a neural network (Source: https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg)

A node processes its inputs according to a function, that is a composition of two simpler functions. The first function is simply a linear function with some slope (called weights) and a bias term. After the inputs are passed through the linear function, they are passed through the second function, called the activation function. The activation function is non-linear. This non-linearity is important because it allows the network to establish non-linear relations between the inputs and outputs, which is necessary for any reasonably complex problem.

The weights and biases of the individual nodes are collectively called the parameters of the network. The parameters are not set by the programmer. Rather, they are learned by the network itself using an objective function (also called an error function) and labelled training data.

The objective function is a function in terms of the parameters of the network (or simply in terms of the outputs) and it describes how well the network is performing. Usually, the lower the value of the objective function (the error) the better the network is performing. Using the training data, the error in the output calculated by the network for a given input can be computed, and by taking the gradient of the objective function at that point, we can use some sort of optimization algorithm for

minimization to reduce the error at that point. One of the most popular methods is known as Stochastic Gradient Descent (SGD), which simply adjusts the parameters according to some fraction of the gradient at the single point being tested.

As long as our objective function is good (i.e. it is in terms of the parameters of the network and its gradient can be computed) and our network is sufficiently flexible (often referred to as being sufficiently "deep" in the literature), our network should be able to learn the correct parameters that match the inputs to the outputs.

Once the network has been trained to a sufficient degree of accuracy, it can be used to make predictions based on new data that it has never has seen before.

One of the drawbacks of the standard deep learning architecture is that it has no memory in terms of the sequence of inputs. This means that if there is some pattern or order in the sequence of the inputs we feed to the network, our network will never be able to recognize it and use it to improve. This limitation has been recognized and lead to the development of more powerful neural network architectures.

RealNVP Neural Network Architecture

A specific type of neural network architecture called Real Non Volume Preserving (RealNVP) will be very useful for this work [7]. The advantage of this type of network is it can learn an arbitrary probability density function from a set of samples (a distribution), and the learned PDF is then able to be sampled from. It is also possible to compute the value of the PDF for a given sample. This model falls under the class of machine learning models known as generative models.

The model works by using part of the input vector to parametrize a non-linear scaling and translation factor on the rest of the input vector. Passing the input through several layers of these scaling and translation operations and using different sections of the input vector each time, allows a complex, nonlinear mapping to be created. Neural networks are used to calculate the scaling and translation factors. As such, a complex relationship between the input vector and the output factors can be learned.

This operation is also easily invertible because the part of the input vector that is used to compute the scaling and translation factors is preserved (i.e. the scaling/translation operation is not applied to them), the scaling and translation factors used in the forward mapping can be recomputed using the output vector and used to invert the scaling/translation done on the other components. In the paper that introduces this architecture, the mapping from the primary space to the latent space is denoted by $z = f(x)$ where x is in the primary space, and the mapping from the latent space to the primary space is denoted by $x = g(z)$ where z is in the latent space. Using the scaling and translation factors calculated, the Jacobian of the mapping can also be computed, which allows for the probability of generating sample x to be calculated.

When training a RealNVP network, the loss function used is based on the Jacobian of the mapping between the primary space and the latent space, and the distribution chosen for the latent space. The Jacobian is given by:

$$\left|\frac{\partial f(x_i; \theta)}{\partial x_i}\right|$$

The network is trained using a maximum likelihood approach, to find the optimal parameters θ^*. The samples given as training data, the set S, are in the primary space and are mapped to the latent space. The training process seeks to find the parameters of the network that maximize the sum of the latent space pdf evaluated for each training data point after it has been warped. This process learns the function $f(x)$.

$$\theta^* = \arg\max_{\theta} \sum_{i\in S} \log\left(\left|\frac{\partial f(x_i;\theta)}{\partial x_i}\right| p_Z\big(f(x_i;\theta)\big)\right)$$

$$= \arg\max_{\theta} \sum_{i\in S} \log\left(\left|\frac{\partial f(x_i;\theta)}{\partial x_i}\right|\right) + \log(p_Z\big(f(x_i;\theta)\big))$$

Note that the log function used here is the natural logarithm.

After the model has been successfully trained, samples can be drawn from the prior distribution in the latent space and warped to the primary space using $g(z)$ to obtain a sample x in the primary space that follows the distribution of the data seen during training. The probability $p_X(x)$ of obtaining sample x can be calculated:

$$p_X(x) = \left|\frac{\partial f(x;\theta)}{\partial x}\right| p_Z(z)$$

$p_Z(z)$ is the probability of drawing the sample latent space sample z and $\left|\frac{\partial f(x;\theta)}{\partial x}\right|$ is the Jacobian of the warping function.

An example of a RealNVP architecture learning to map between a crescent moon distribution primary space and a 2D multivariate gaussian is presented below.

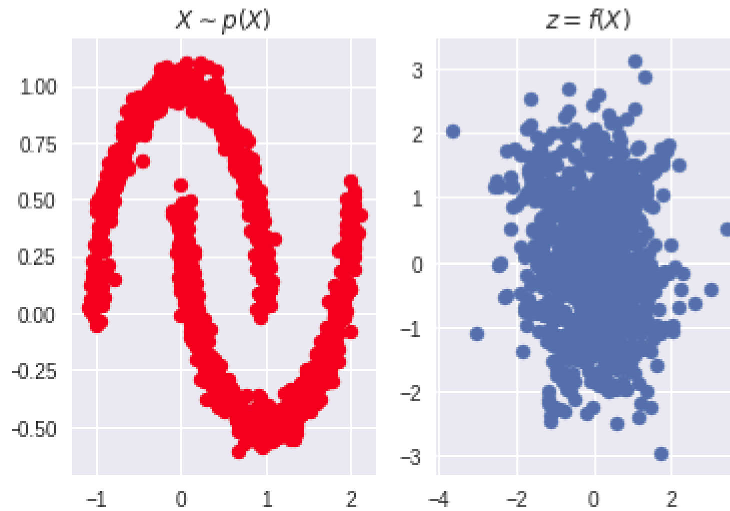


Figure 8: Samples are generated according a distribution (left, red) in the primary space and mapped to the latent space (right, blue) by the RealNVP network. The network is trained to map the values from the primary space in such a way that the probability of the sum over all samples is maximized in the latent space given a prior distribution (Gaussian with zero mean and identity covariance matrix in this example)

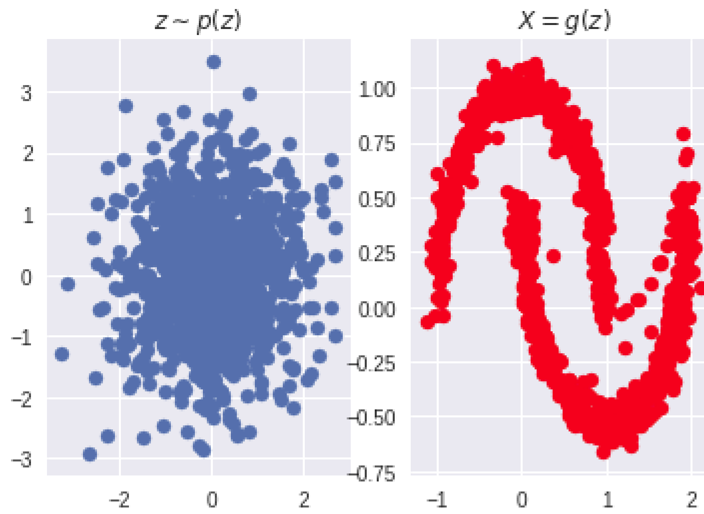


Figure 9: Samples are generated in the latent space (left, blue) and mapped to the primary space (right, red) using the inverse of the function learned during the training. The samples in the primary space resemble the distribution the network was trained on.

Requirements and/or Problem

One of the main issues currently facing ray tracing is the amount of time it takes to render a scene. Tracing the ray through the 3D scene involves checking if the ray intersects with any object in the scene. This is a very costly operation and is the main reason ray tracing takes significant time. Several acceleration techniques for minimizing this problem exist, such as back face culling and bounding volume hierarchies (BVH).

Related to this first problem, another issue that causes ray traced images take significant time to converge is that it takes many samples for the Monte Carlo integrations to converge to the correct value. Often, paths that do not contribute much to the integral are traced, involving the costly retracing operation. Tracing paths with small contributions to the integral results in slower convergence.

It is desired that this system focuses on indirect illumination. Direct illumination in scenes is handled well by explicit path tracing/next event estimation, while indirect illumination must be calculated using an integral or an approximation. It is also planned for this system to focus on surface illumination. Volumetric effects will not be considered.

The goal of the system in this work is to speed up the convergence of the rendered image, by speeding up the convergence of the Monte Carlo estimator. The goal is to increase the proportion of rays that are traced along paths that contribute significantly to the indirect illumination at the point of interest. By tracing rays that contribute significantly to the MC integral, the integral will converge faster, resulting in fewer iterations needed to compute the lighting.

There are some constraints needed for this strategy to offer advantages over simply tracing more rays. In order to see a reduction in the time taken to render an image, the process for choosing a path with sufficient contribution should take less time than simply tracing more rays. The system will also have to have a good degree of accuracy, in order to ensure that we are not taking the time to compute or check for paths with high contribution, and then tracing the ray only to have a small contribution.

Design and Results

Design

This work closely follows the approach taken by Zheng and Zwicker [8], but with some important differences. In the Zheng work, the operation performed by the neural network architecture was viewed as mapping a sample from a $2k + 1$ unit hypercube onto itself non-linearly, where k represents the number of bounces the light path takes in the scene. Essentially, the prior distribution in the latent space is a $2k + 1$ unit hypercube, which maps to a primary space that is also a $2k + 1$ unit hypercube.

In this paper, the mapping is from a $2k + 1$ multivariate gaussian that has an mean of zero and all the dimensions are independent. This was done because it was found during some initial experiments with the RealNVP architecture that training the network while using a $2k + 1$ unit hypercube as the prior distribution was difficult because this distribution does not have full support in the $2k + 1$ space. The issue arose when the network tried to map values to outside the hypercube during training, which resulted in issues when calculating the gradient for updating the parameters of the network. It was not determined how Zheng and Zwicker overcame this difficulty. Additionally, this change also brought the RealNVP model used here more in line with the implementation of the authors who originally described the RealNVP, who also used a gaussian to represent their latent space.

The work this semester started out by modifying the PBRT-V3 renderer [9] to render a scene given samples in primary path space. This involves generating an array of samples in a $2k + 1$ unit hypercube, and then using these values to parametrize the directions of the rays traced in the scene, as well as the final direct connection to a light source. The parameter k represents the maximum number of bounces a light path. In this work, we use $k = 2$ because this describes the light paths that contribute the most indirect lighting to a scene, and are therefore important to the quality of the final image. These paths can be challenging to find when tracing rays randomly. The objective is to have the RealNVP network generate this $2k + 1$ sample, which should have a higher chance of contributing light to the output image than if the numbers were simply chosen randomly.

The work then shifted to implementing the RealNVP neural network. The code for this was first experimented with by implementing it in Python and testing it on synthetic data. The construction of the code was based on a minimal working example [10], but was then modified to more closely resemble the network use by Zheng and Zwicker.

It was originally planned to implement the network directly in C++ by making use of C++ PyTorch distribution, Libtorch. It was thought that this would make the integration with the C++ code used in the PBRT-V3 renderer simple. However, several runtime errors were encountered while using Libtorch that were unable to be resolved. It was also discovered that Libtorch lacks several of the "quality of life" libraries that are included in the Python version of PyTorch, most notably the distributions library which allows for the creation of a multivariate gaussian distribution that can be sampled from.

The first alternative tried was to embed Python code into C++, which could be used to construct the functions related to the neural network and executed when needed. This was done successfully, however the resulting hybrid code was extremely slow to the point where it was not practical to use. The operations involving the neural network were the primary source of the slow down. It could possibly to improve the speed of these operations to a point where they could be useful by

loading the neural network onto a CUDA compatible GPU. However, this hardware was not available at the time.

The solution was to divide the code into four distinct sections: training data generation, neural network training, new sample generation, and final rendering using neural network generated samples. The training data for a given scene was generated using PBRT-V3 and exported to a CSV file. The CSV containing the training data was uploaded to a compute server (Google's Colaboratory) which was used to train the neural network using GPUs. Once the network was trained, an appropriate number of new samples were generated and exported to CSV files. The CSV files containing the new sample data were then loaded into PBRT-V3 and used to produce the final render of the scene.

It was found that some data processing had to be done on the samples generated by PBRT-V3. The parameters used to construct a light path were saved if they contributed a non-zero amount of radiance to the final image. In addition, the value of maximum component (RBG) of the radiance was also saved. It appears there were some occasional numerical calculation issues during the rendering that results in extremely high values of radiance being saved. These outliers had to be removed prior to training to ensure the distribution did not become skewed towards these samples.

The distribution used to train the RealNVP model was created using weighted resampling of the generated data, where the weights were proportional to the radiance of the sample. Once the training data was constructed appropriately, the training procedure was done as described in the RealNVP section above.

Results

The RealNVP network had 8 coupling layers. Each fully connected layer contained 40 neurons. The masks used alternated between masking even and odd dimensions. The total number trainable parameters in the model using this construction was 92 400. This construction was used because it is similar to the one found to be effective by Zheng and Zwicker [8]. The network was trained using the Adam optimization algorithm.

The network and algorithm were tested on two different scenes, both available as part of the PBRT-V3 software. One was a simple scene containing a teapot on a table lit by a single area light source. The other was a much more complex scene based on a hotel in Mexico. The scene is lit from the sky and has many areas of with indirect lighting.

The size of the training data set varied depending on the scene and the path sampling scheme used. The training data set was created sampling a fixed number of camera rays, and saving any path of the proper length that contributed to the image. Therefore, scenes and sampling schemes that produced fewer valid paths ended up with smaller training datasets.

Scene	Samples Generated
Teapot (BSDF)	35 632
Teapot (uniform)	15 300
Teapot (cosine)	14 650
San Miguel (BSDF)	138 699

Table 1: Training samples per scene generated using 16spp

In each case, the network was trained for 7 epochs. The training was done using Google Colaboratory GPU runtimes. The training phase takes approximately 5 minutes for the teapot scenes and about 10 minutes for the San Miguel scene.

Loading in the new data from CSV files does appear to translate to significant overhead when rendering a scene. There are several seconds of difference between the rendering times using randomly generated samples and loaded samples for the teapot scene, and several minutes for the San Miguel scene. Table 2 shows a comparison between random and learned sampling for the two test scenes. When doing the teapot scene, different techniques for sampling/parameterizing the first bounce were tested: uniform, BSDF, and cosine importance sampling. It was found that using BSDF sampling resulted in the highest number of valid paths and was therefor selected to use with the San Miguel scene. The images produced BSDF sampling are presented in the main body of this text (Figure 10, Figure 11, Figure 12, Figure 13) while the images produced with uniform and cosine sampling can be seen in Appendix A: Additional Renders.

When comparing the rendered images where learned sampling was used to images that used random sampling, we can see that the learned sampling method was able to find more paths of the desired length, and a higher percentage of these paths produced a non-zero contribution to the final image. The effect of finding more contributing paths can be seen in highlighted areas of the rendered images (Figure 10, Figure 11, Figure 12, Figure 13) where the 2-bounce indirect lighting contribution is significant. This demonstrates that the RealNVP network can successfully learn an approximating representation of the distribution of incoming radiance in path space for a given scene, and then generate samples that can be used to construct new paths that have a higher probability of contributing incoming radiance to the final image when compared to random sampling.

Scene	Sampling Scheme	Camera Rays Traced	Render time	2 bounce paths found	Non-zero 2 bounce paths	Percentage (Non-zero/total paths)
Teapot	BSDF, Random	1 081 600	11.7s	126 022	35 632	28.27%
Teapot	BSDF, Learned	1 081 600	31.7s	224 922	117 329	52.16%
Teapot	Cosine, Random	1 081 600	9.1s	77 640	14 650	18.87%
Teapot	Cosine, Learned	1 081 600	31.1s	195 009	90 016	46.16%
Teapot	Uniform, Random	1 081 600	9.1s	86 295	15 300	17.73%
Teapot	Uniform, Learned	1 081 600	27.8s	182 152	74 588	40.94%
San Miguel	BSDF, Random	6 904 064	331.9s	6 027 277	138 739	2.30%
San Miguel	BSDF, Learned	6 904 064	529.1s	5 415 854	319 964	5.91%

Table 2: Comparison of random sampling vs. learned sampling

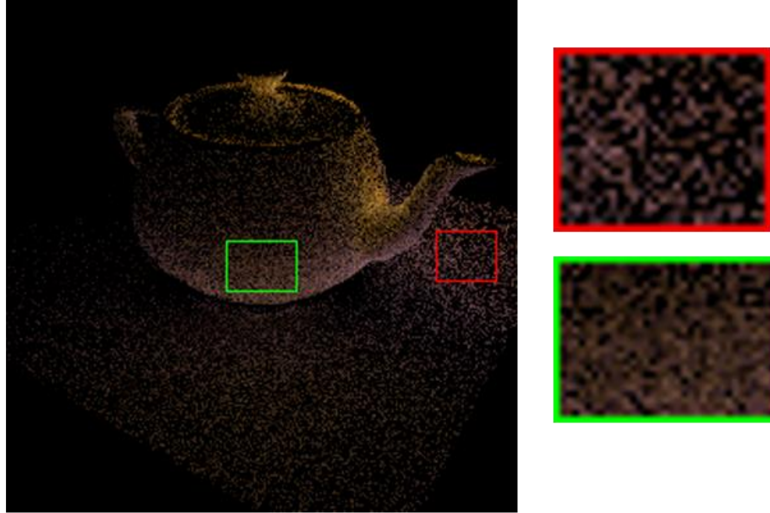


Figure 10: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and random BSDF sampling

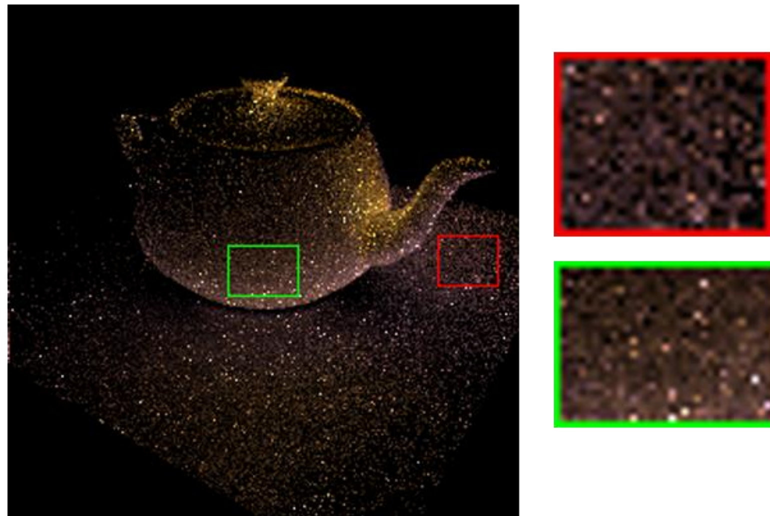


Figure 11: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and learned BSDF sampling. We can see that the highlighted areas are more filled in when compared to the same scene using random sampling.

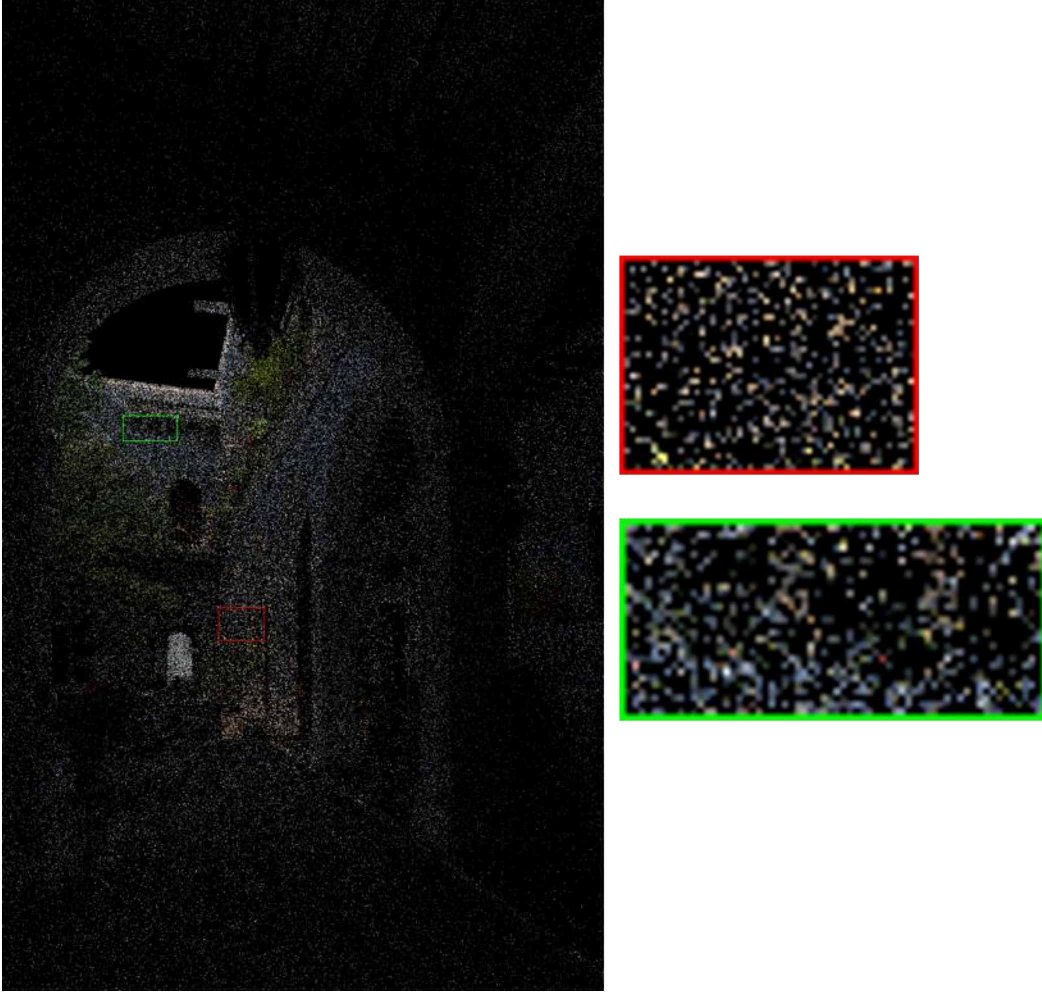


Figure 12: 2-bounce indirect lighting "San Miguel" scene rendered using 16 SPP and random BSDF sampling

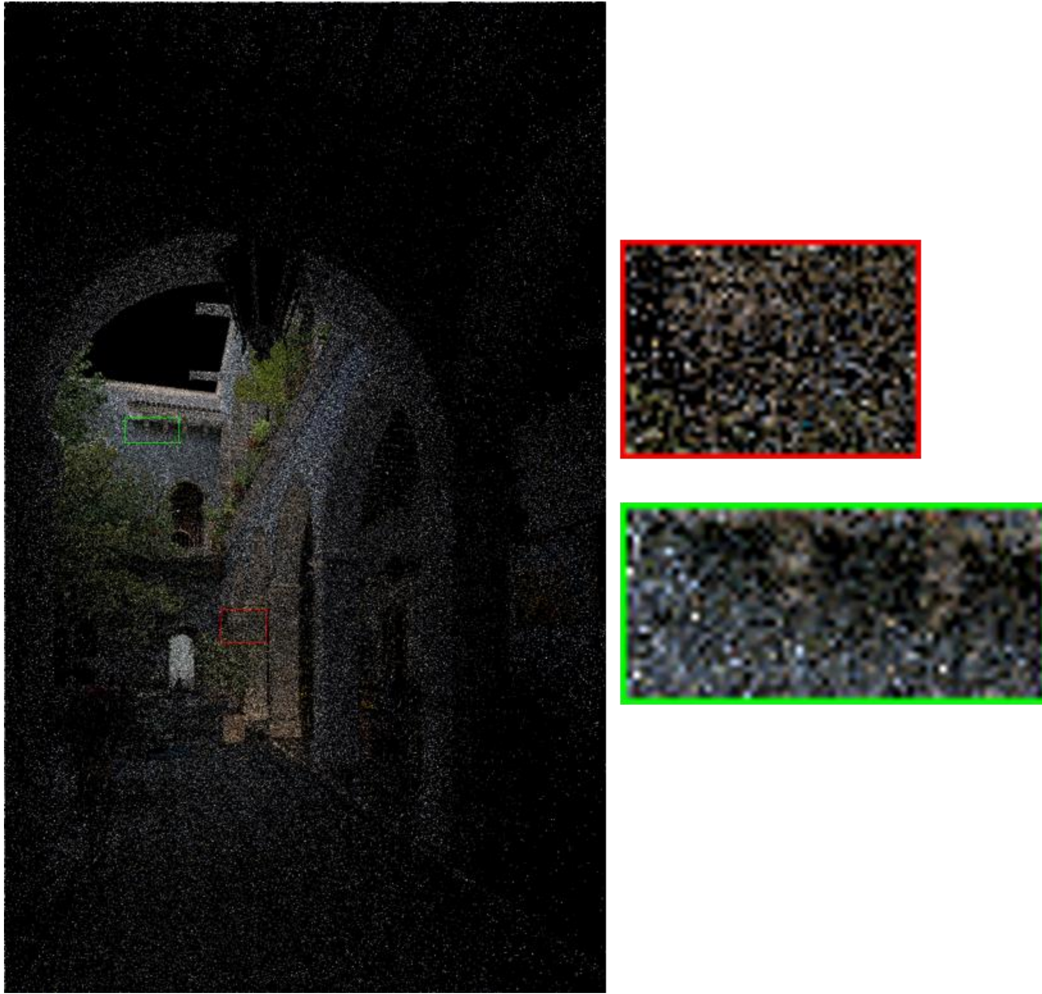


Figure 13: 2-bounce indirect lighting "San Miguel" scene rendered using 16 SPP and learned BSDF sampling. We can see that the highlighted areas are more filled in when compared to the same scene using random sampling.

A downside of using learned samples is the increased presence of "fireflies" in the rendered images. These are areas of high brightness (usually single pixels appearing white) that are produced when a path with a low probability of being constructed is traced and it contributes to the image. Because the path sample was unlikely, it becomes heavily weighted in the Monte Carlo estimator and produces a bright spot in the image. Finding a way to remove or smooth out these fireflies will be a subject of future work.

Upon review of the network employed and the number of samples used in training, it was observed that compared to the number of parameters in the network (92 400), relatively few training samples were used (~15 000 samples for the teapot uniform and teapot cosine, ~35 000 for teapot BSDF, and ~139 000 for San Miguel BSDF). It would be interesting to see if using more samples and more epochs would lead to further improvements in the ability of the network to learn valid paths or reduce the presence of fireflies in the image.

When training the network, the training data was not split into train and test sets. This was done because it was desired to use as many samples as possible to train the network because a relatively few numbers of samples were available. It was also thought that overfitting would not be a significant problem with this application because only a single specific scene is being focused on at a time, and there is not really a need for the model to learn generalizations. If the code were to be improved in order to make it easier to generate and process more samples when training, it would be good to investigate the impact of splitting the data and performing a detailed analysis of how the loss on the test set changes with different numbers of epochs, samples, and parameters in the network.

In order to produce images with global illumination while taking advantage of these learned paths would require combining these paths with other paths of different length. Further research in how to achieve this is needed.

Impact on Society and the Environment

This project will be purely software based, and thus have very little environmental impact. The main environmental impact that will result from this project is energy consumption. Machine learning on large servers often consumes a lot of electricity. Depending on how this electricity is produced (i.e. coal vs. solar or wind) it can have varying levels of environmental impact. There should be no significant difference in terms of the environmental impact between the product in the development stage and as a marketed product.

A possible negative side effect of this technology is that advanced photorealistic computer graphics could produce images that are indistinguishable from real images. This could lead to the creation of fake images that could be used to deceive people for malicious purposes, or to produce images of public figures in compromising scenarios in order to defame them. This is something that society has already begun to experience, with widespread "fake news".

There are many benefits to several industries that would result from improved computer graphics. The film industry increasingly relies on computer generate images to replace practical effects and add that "movie magic" to the film. Increasing the speed at which digital artists could iterate on their work would result in superior results in the same time frame. Additionally, more complex effects that would have historically been deemed to take too long to render would now be able to be done. Animation films would in particular directly benefit from decreased render times. We could see animated environments with more complex lighting and more detailed and expressive characters. Some of these improvements could also translate to video games, if ray tracing ever became sufficiently fast to be real-time.

Another industry that would benefit immensely from improvements in computer graphics would be the computer aided design (CAD) industry. CAD is often used to visualize parts, products, and environments. Improving the quality and the speed at which the visualizations can be produce would have a substantial impact. Designers could iterate on their work faster and produce better results. This would result in economic benefits for companies.

Conclusion

This is an interesting problem in an active area of research. This work demonstrates the ability of carefully designed neural network architectures to learn and then preferentially sample paths in a way that is compatible with the typical Monte Carlo integration process used in rendering. Using this method to generate indirect lighting ray paths, we can see some reduction in variance and a resulting improvement in the quality of the images produced when compared to random path sampling.

Future Work

Some immediate follow ups to this work would be to implement a rendering algorithm that included the learned paths when computing global illumination in a scene. This would help to determine exactly how much these paths contribute to the final look and convergence rate of a fully rendered image. Determining how to reduce the number of fireflies in the images would also be a priority.

Areas to build upon this work include exploring the possibility of learning longer paths and rendering a scene using multiple sampler generators each specialized for different path lengths is a particularly interesting area. Further areas of research could include seeing how a technique similar to this could be extended to volumetric interactions. More experimentation and optimization with the precise construction of the neural network would also be interesting.

Some more implementation specific and practical improvements to be made would be unifying the different operations in this process (i.e. generating training data, neural network training, generating new samples, and rendering final image) into a single program and making use of a GPU to speed up the neural network computations. This would allow for a more detailed comparison between the overall time used for this method vs. more traditional rendering methods. It would also avoid the need to store files containing the training data and the new samples.

References

[1] E. Veach and L. J. Guibas, "Optimally combining sampling techniques for Monte Carlo rendering," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995, pp. 419-428: ACM.

[2] J. T. Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, 1986, vol. 20, no. 4, pp. 143-150: ACM.

[3] E. Veach and L. Guibas, "Bidirectional estimators for light transport," in *Photorealistic Rendering Techniques*: Springer, 1995, pp. 145-167.

[4] E. Veach, "Robust monte carlo methods for light transport simulation," Stanford University PhD thesis, 1610, 1997.

[5] C. Kelemen, L. Szirmay-Kalos, G. Antal, and F. Csonka, "A simple and robust mutation strategy for the metropolis light transport algorithm," in *Computer Graphics Forum*, 2002, vol. 21, no. 3, pp. 531-540: Wiley Online Library.

[6] B. C. J. F. o. S. Csáji, Etvs Lornd University, Hungary, "Approximation with artificial neural networks," vol. 24, p. 48, 2001.

[7] L. Dinh, J. Sohl-Dickstein, and S. J. a. p. a. Bengio, "Density estimation using Real NVP," 2016.

[8] Q. Zheng and M. J. a. p. a. Zwicker, "Learning to importance sample in primary sample space," 2018.

[9] M. Pharr, G. P. D. Humphreys, and W. Jakob, Physically based rendering : from theory to implementation, Third edition. ed. Cambridge, MA: Morgan Kaufmann, 2017. [Online]. Available: http://www.myilibrary.com?id=958343.

[10] A. Arsenii. (2018). *Real NVP PyTorch a Minimal Working Example*. Available: https://github.com/ars-ashuha/real-nvp-pytorch

Appendix A: Additional Renders

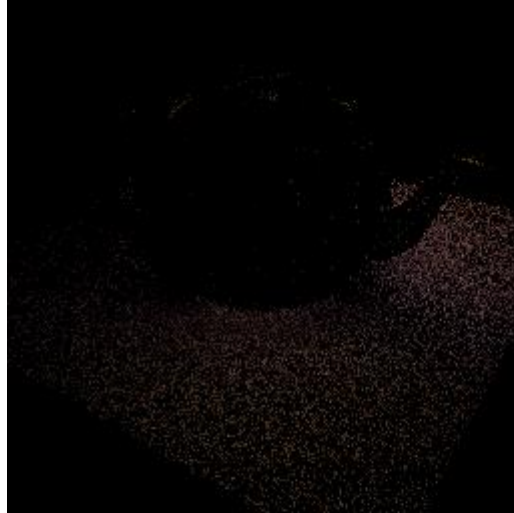


Figure 14: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and random cosine sampling.

Figure 15: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and learned cosine sampling.

Figure 16: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and random uniform sampling.

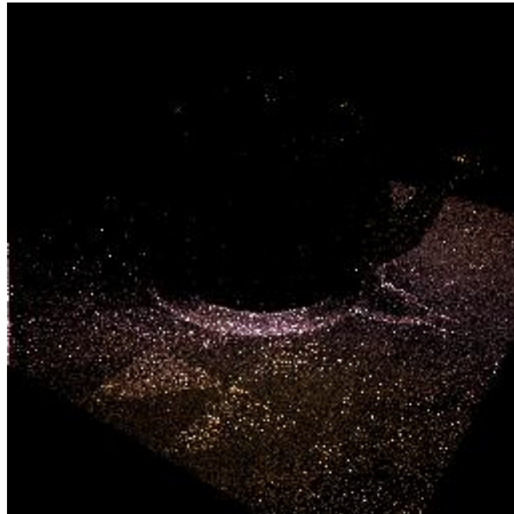


Figure 17: 2 bounce indirect lighting "Teapot" scene rendered using 16 SPP and learned uniform sampling.