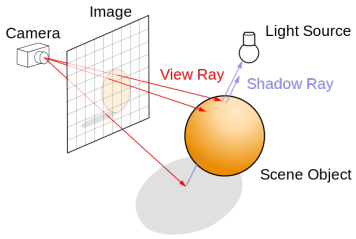
An Attempt at Adaptive Sampling for Photorealistic Image Generation: Learning Sampling Schemes for Monte Carlo Rendering

Winnie LIN, Stanford University, Email: winnielin@stanford.edu
Timothy WU, Stanford University, Email: thsuanwu@stanford.edu

***Abstract*—We take a machine learning based approach to adaptive sampling for Monte Carlo Rendering, by using geometric and lighting data obtained through prior renders of scenes. Using nonlinear kernels, we trained Support Vector Machines of high accuracy, but complications arose in the labelling of our data, resulting in slightly impractical results for the sampler itself.**

I. INTRODUCTION AND RELATED WORK

As a result of the constant drive towards improving aesthetic appeal, most companies involved in Computer Generated Imagery (CGI) in the industry either use or are shifting towards photorealistic ray tracing methods to generate their images. In essence, ray tracing is the implementation of a virtual camera, and the simulation of photons that the cameras light sensors receive from a mathematically described virtual scene.
We represent the image plane as a virtual plane that is normal to the direction of the camera. We trace the reverse path of photons from the camera into the virtual scene, record the intensity and color of the photon, then calculate the value of each pixel as the total sum of photons passing through it. A diagram is as follows:



The most general ray tracing scheme is *Monte Carlo ray tracing*, where pixels in the image plane are approximated as the average color of a collection of randomly sampled light beams (as known as *rays*) passing through the pixel. This scheme deftly handles a rich variety of object materials, complex lighting and particle effects, but the downside is it tends to be computationally expensive, due to the high number of rays needed per pixel to achieve a relatively noiseless render. This then contributes to and influences the interest in optimization.

In recent years, there has been an increased amount of machine learning related research in such optimization techniques, mostly in terms of smart filtering and postprocessing via lightfield analysis [1]. We were particularly inspired by the Kalantari paper [2], where Neural Networks where applied to postprocess and filter noisy renders, Zwicker et.al.'s comprehensive overview [1] on recent advances in optimization via light field analysis for Monte Carlo rendering, as well as Moon et.al.'s approach to using local regression on filter-based features for sampling [3].
While the aforementioned papers all described great results, we were wondering if machine learning approaches combined with only straightforward lighting and geometric data could produce good results, without having to resort to bandwidth dependent filtering and lightfield analysis. Considering that the pbrt [4] method for adaptive sampling is a simple model which sets two different sampling rates depending on whether the rays of a pixel hit only one shape in the scene, we thought it would be interesting to combine the two approaches into a simple but effective learned sampler.

II. OVERVIEW OF ADAPTIVE SAMPLING

Adaptive sampling is a technique where the number of ray samples per image pixel is heterogenuous. Ideally, the number of sampled rays of a pixel would depend on the pixel's rate of convergence to the ground truth pixel, the ground truth pixel defined as the result of sampling an infinite number of rays.In other words, we would only sample a high number of rays on pixels we predict are far from the ground truth, removing unneccessary or less impactful calculations from image generation.
The challenge is thus to predict when a pixel is close to the ground truth pixel, without having prior idea of what the ground truth pixel is.

III. METHODS

A. Objective

Given an input of a collection of ray samples for a pixel, by using a **Support Vector Machine** corresponding to this collection size,
our objective is to **predict whether we should increase the size of the collection or not.**
We use a model that has an individual support vector machine for each collection size (the justification of this will be given after the description of our features.) The proposed model will now be described in more detail.

B. Overview of Support Vector Machines

Support Vector Machines are used to separate labelled points in high dimensional spaces. In our case, the labels correspond to the distance of a pixel to it's corresponding ground truth pixel, and the point's position in space will be

determined by our features (as determined in the next section.) Given a kernel $K : \mathbb{R}^n \to \mathbb{R}$, a weighting $C \in \mathbb{R}$ that balances optimizing a clear separation (hyperplane with large margin) versus a separation for as many data points as possible, and a set of datapoints $x^{(i)} \in \mathbb{R}^n$ with labels $y^{(i)} \in \{-1, 1\}$, we find find the coefficients c_i corresponding to

$$\text{maximize}_{c_i} \sum_{i=1}^{m} c_i - \frac{1}{2} \sum_{i,j=1}^{m} c_i c_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)})$$
$$\text{subject to } 0 \leq c_i \leq C \ , \ \sum c_i y^{(i)} = 0,$$

As shown in Professor Ng's lecture notes, this is the dual problem of finding the optimal separating hyperplane of the labelled training data. Therefore, given any testing data x, we can predict its corresponding label y, or which side of the hyperplane x is on, by solving and checking the value of

$$f(x) = \sum_{c_i \neq 0} c_i K(x^{(i)}, x)$$

Against the intercept as determined by the support vectors $x^{(i)}$ where $c_i \neq 0$.

C. Pipeline

The method with which we incorporate our trained support vector machines into the rendering pipeline is as follows: For each pixel,

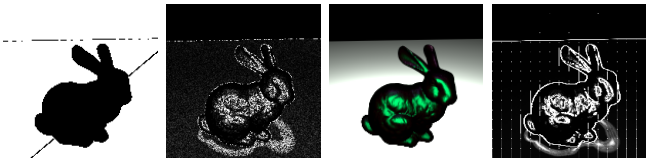
1) We start by sampling a fixed minimum samples, and record relevant data.
2) Using $f(x)$ of the first layer of Support Vector Machines, predict whether we need to continue sampling or not.
3) If we need to continue sampling, we add an extra collection of ray samples (doubling the size of the sample collection), run it through the next layer of Support Vector Machines, and continue until termination as determined by the hypothesis or the cap we set on the maximum number of samples.

D. Features

Intuitively, we predict that we need more samples if the pixel corresponds to a part of the scene with complex geometry and/or complex lighting. With that in mind, we set our features to be:

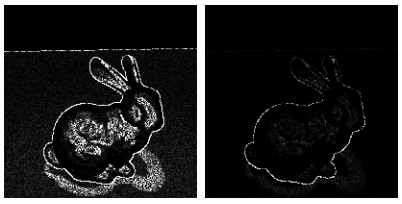
- **The number of intersection shapes:** This term captures the complexity of the geometry. If the collection of rays of a pixel hits multiple shapes, we know that it might correspond to the edge of an object's silhouette or an edge on the object's surface.
- **The difference in X,Y,Z color channels between the old ray collection and the newly sampled rays:** If the two collections of rays corresponding to the same pixel are distinctly different in color, we think this is a good indication of complex lighting or complex color textures. This data is easy for us to obtain and compare, as the way our model is designed gives us a set of old samples and a set of new ones of equal size.
- **The mean of X,Y,Z color channels of the overall ray collection:** This feature was added when we found that we were not able to train models on data sets without it, due to the high dependency of the previous features on the actual color value of the ray collection.
- **The variance in light source illuminance:** If our ray collection has a high variance in power/illuminance, we think that it corresponds to complex lighting. This term is fast to compute using a recursive relationship between the two sets of samples within our old ray collection. Given the variance σ_a, σ_b and mean μ_a, μ_b of two separate sets of samples, with the fact that the two sets are of the same size, we derived
  $\mu_{a \cup b} = \frac{1}{2}(\mu_a + \mu_b)$ and
  $\sigma^2_{a \cup b} = \frac{1}{4}(2\sigma^2_a + 2\sigma^2_b + (\mu_a - \mu_b)^2)$.

Other features we considered include the average number of bounces of a ray before it reaches the pixel, and the surface area of the convex hull of the normal direction of the ray-object intersection. However, given the restrictions within the existing pbrt pipeline and the complexity of computation, we limited ourselves to the ones described above. Here are some visualizations of the features corresponding to shape, difference in Y channel, combined mean of XYZ channels, and variance of illuminance, left to right:



E. Motivation of multiple layers

Coming back to our model design, it can be noted that our normalized feature values themselves are highly dependent on the number of samples we have; for example, a higher number of samples would naturally give us a higher variance in illuminance and a lower color difference between the two subsets of the ray collection. To illustrate, these are visualizations of the color difference for the layer of 8 samples and 512 samples, respectively:



Using the layer count itself as a feature would further increase the complexity of our SVMs and increase the number of support vectors (which we will see later on is rather a major problem already.) With this in mind, we arrived at the decision to use multiple layers of SVMs.

F. Data collection and labelling

In our data collection stage, we repeatedly render scenes sampled with a uniform number of rays, varying the overall sample number from 8 rays to 512, and collect feature data corresponding to each sample rate during this process. Afterwards, we record the color distance between each pixel of

sample rate less than 512 and its corresponding pixel of 512 rays, then label the feature data according to some threshold of the color distance.

Originally, we set our threshold to be the just noticeable difference (JND) for the human eye as determined in Lab color space [5]. However, we were actually unable to train all our SVMs with this labelling, nor with any other threshold that was uniform throughout all SVM layers (libSVM threw out NaN values.) We hypothesize that the labelling was simply too unbalanced; if the SVMs of lower samples were trainable, then ones with the higher samples, which would have color distance closer to the 512 samples, would yield labellings that were too skewed towards "having distances smaller than the global threshold," and vice versa. In the end, we settled on varying the threshold for each layer to be the mean color difference for all of the pixels; this gave us perfectly balanced labelling (it partitions the data exactly in half) and great training results, yet it is not what we'd hope for for the sampler itself.

G. Software Implementation

We wrote our sampler within the C++based pbrt [4] framework. For the prediction stage, we linked libSVM [6] into the pbrt sampler. For the training stage, we wrote our data collector also as an extension of pbrt, parsed and labelled data with Python scripts, and trained our models with Python scripts and libSVM. We also wrote a simple visualizer with libpng [7] to visualize the features and results.

IV. Results: The Good, the Bad and the Ugly

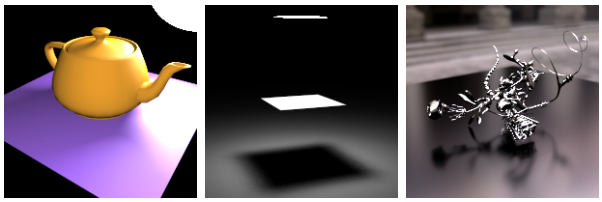
Specifically in that order.

A. Model Accuracy

While with a lot of labelling schemes and kernels our data was not trainable at all, (in particular, linear kernels and a globally uniform labelling scheme never worked,) our models are extremely accurate with the right parameters. In particular, the radial basis kernel

$$K(x^{(i)}, x) = e^{-\frac{||x^{(i)}-x||_2^2}{2\sigma^2}}$$

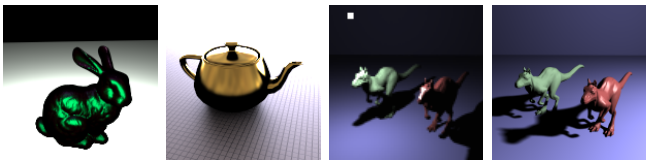
with balanced data labelling gave rise to predictions consistently above ninety percent. We collected data on 120 thousand pixels obtained from these three images, which were chosen respectively for their color variance, shadow variance, and object complexity:



We then trained our SVMs once on the entire set of training data, once on one tenth of the pixels randomly sampled from the data collection, and tested them both on the training data and data obtained from rendering various other scenes. Here is an accuracy chart of our model tested on training data, and another on a subset of our testing scenes:

	Original Data (121324 samples)	Sampled Data (12132 samples)	10-fold Cross Validation
Layer 1	92.37%	90.6858%	90.488%
Layer 2	92.6717%	90.9083%	90.76%
Layer 3	92.919%	91.1474%	90.9413%
Layer 4	93.0789%	91.3452%	91.1474%
Layer 5	93.1737%	91.3699%	91.1639%

The following chart corresponds to testing data from the bunny scene as shown in the features section, and another teapot scene and a kangaroo scene all rendered at 150x150 pixels. This were trained on the SVMs of the filtered dataset:



	Bunny	Teapot	Kangaroo	Bright Kgr
Layer 1	94.202%	95.7809%	98.8553%	98.9255%
Layer 2	94.3467%	95.9081%	98.9562%	99.0176%
Layer 3	94.3774%	96.0177%	98.9343%	99.1009%
Layer 4	94.3906%	96.0046%	98.7501%	99.0702%
Layer 5	94.1011%	95.9914%	98.1974%	98.5966%

While we predicted that the teapot would have a higher accuracy rate due to its similarity with the teapot scene in the training data, it was rather perplexing how high the accuracy of the killeroo scene was. Originally we thought that it was due to the fact that most of the scene was unlit and uniformly black, but even increasing the lighting did not reduce the accuracy.

B. Support Vectors

The number of support vectors was surprisingly high, with the models trained on the original data and the sampled data as shown below:

	Original Data (121324)	Sampled Data (12132)
Layer 1	29807	4526
Layer 2	27820	4246
Layer 3	20144	3982
Layer 4	25373	3800
Layer 5	24803	3757

However, as shown previously, performing cross validation on our results continued to yield high accuracy in the 90%s, which to some extent decreases the probability that we might be overfitting. A TA mentioned that a high number of support vectors was often the case when training data includes RGB/XYZ color values directly as features, since the variance within the color space is so high.

C. Renderer Accuracy and Efficiency

Despite the promising accuracy of the model itself with collected data, as of now it does not work as well in practice. In particular, rendering the bunny scene under the adaptive sampler, which should sample between 8 and 512 pixels, took between 80.7 to 85.2 seconds under multiple renders. Using the standard pbrt sampler, rendering 512 samples took between 90.4 to 92.6 seconds, but rendering with 8 samples takes 1.2 to 1.3 seconds with no noticeable difference.

Below is a visualization of the number of samples per pixel (brightest color corresponds to 512 samples, darkest to 8,) and it shows us that for some reason, in this scene, the relatively simply flat ground plane requires up to 512 samples.

V. Future Work

There are still many improvements that could potentially make this sampling scheme work. We have listed a few of these below:

1) A better labelling scheme or pipeline structure: We think the normalized labelling scheme really affected the final results, yet we haven't found a better way to label our data.

2) More complicated scenes: Due to restrictions on the computing power of my laptop (we did not figure out the Farmshare task submission framework in time, and my computer repeatedly froze on some of the more complex scenes we attempted,) all the scenes we've used have been relatively simple, and are most likely not representative of scenes that would actually benefit from such a sampling scheme.

3) Experimentation with Neural Networks: While all attempts at using linear classification failed rather spectacularly, it would be interesting to try to apply neural networks to our problem, given the success of nonlinear functions in our current SVMs and the complexity of our data.

4) Experimentation with more features: As we mentioned before, we only included 8 features (bundled into 4 main categories) due to time constraints and the complexity of data retrieval. However, rewriting more parts of pbrt would give us access to a wider array of data that we could probably use to our advantage.

For example, we originally only considered features that were constrained within the pixel due to the limitations of the Sampler superclass of pbrt, but as previous research in related areas have shown, features dependent on image filtering across pixels [3] have yielded good results.

5) Optimization: The fact that the radial basis kernel (which involves many exponential calculations) was the only one that produced satisfactory results made the prediction stage rather expensive, and thus resulted in a adaptive sampler that was not significantly faster than sampling at the highest rate. However, upon the suggestion of a friend and further examination of the libSVM source code, we realized that if we implemented a lookup table for the exponential values instead, a huge speedup may potentially occur. During the presentation session, someone also suggested that working consistently in Lab color space instead of rgb/xyz might give us better quality SVMs with lower number of support vectors (which we still need to look into.)

VI. Conclusion

In summary, our intended model was complicated by the inability to use the same labelling scheme throughout all layers, and the label normalizing approach we took in the end to bypass this was neither as accurate nor as fast as we had hoped.

However, simply from a SVM quality standpoint, the models we obtained had high accuracy all above 90 percent, even with scenes that were highly dissimilar from the training scenes. Some main points we discovered through adjusting the training parameters were:

- The radial basis kernel yielded optimal results, while linear kernels did not work at all.
- A filtering of our training data reduced the number of support vectors while still retaining a high accuracy for the test data.

The accuracy in our trained models and the large potentials in optimization lead us to continue to believe that adaptive sampling is an area that may benefit from maching learning techniques!

VII. Code

Code for this project is currently available at https://github.com/winnie1994/229project if interested.

Acknowledgments

We'd like to thank Albert Haque for his advice and feedback on the project, Professor Ng for the great course, and all the TAs for making a course of this size possible! We'd also like to acknowledge the rendering course taught by Professor Hanrahan in the Spring of 2015, which was extremely helpful in setting up the framework of this project, and Radiant Entertainment, whose flexibility in intern work schedule allowed Winnie to sightsee at SIGGRAPH and gain her first exposure to Machine Learning in Computer Graphics.

Despite all the unexpected twists and turns and complications and less than stellar results, working on such an unpredictable project all in all was rather exciting. Additionally, we were exposed to many potential problems that might arise in future machine learning adventures, and are grateful for this particular learning experience.

References

[1] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon, "Recent advances in adaptive sampling and reconstruction for monte carlo rendering," *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 34, no. 2, pp. 667–681, May 2015.

[2] N. K. Kalantari, S. Bako, and P. Sen, "A Machine Learning Approach for Filtering Monte Carlo Noise," *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2015)*, vol. 34, no. 4, 2015.

[3] B. Moon, N. Carr, and S.-E. Yoon, "Adaptive rendering based on weighted local regression," *ACM Trans. Graph.*, vol. 33, no. 5, pp. 170:1–170:14, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2641762

[4] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Elsevier, 2010.

[5] E. N. D. G. Sharma, W. Wu, "The ciede2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations," *Color Research and Application*, vol. 30, no. 1, 2005.

[6] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] G. V. Simon-Pierre Cadieux, Eric S. Raymond, "libpng version 1.2.4," 2002.