Visualization Research Center Stuttgart

Masterarbeit

# Investigation of Volume Rendering Performance through Active Learning and Visual Analysis

Stephan Roth

| | |
|---|---|
| **Course of Study:** | Informatik |
| **Examiner:** | Prof. Dr. Thomas Ertl |
| **Supervisor:** | Valentin Bruder, M.Sc.,<br>Dr. rer. nat. Steffen Frey,<br>Gleb Tkachev, M.Sc. |
| **Commenced:** | 17. April 2017 |
| **Completed:** | 17. Oktober 2017 |
| **CR-Classification:** | I.6.4, D.4.8 |

## Abstract

Volume visualization has many real world applications such as medical imaging and scientific research. Rendering volumes can be done directly by shooting rays from the camera through the volume data, or indirectly by extracting features such as iso-surfaces. Knowing the runtime performance of visualization techniques enables for optimized infrastructure planning, trained models could also be reused for interactive quality adaption. Prediction models can make use of information about renderer and datasets to determine execution times before rendering. In this thesis, we present a model based on neural networks to predict rendering times, by using volume properties and rendering configuration. Moreover, our model actively intervenes the sampling process to improve learning while decreasing the amount of necessary measurements. For this, it estimates how likely a drawn sample will improve future predictions. Our model consists of multiple submodels, using their disagreement about certain samples as criteria for possible improvement. We evaluate our model, using different sampling strategies, loss functions and volume rendering techniques. This includes predictions based on measurement data of a volume raycaster, as well as a continuous setup with interleaved execution and prediction of an indirect volume renderer. Our indirect renderer utilizes marching cubes to extract iso-surfaces as triangle mesh from a density field and organizes them in an octree. This way, highly parallel sorting on the graphics card is enabled that is necessary for rendering transparent surfaces in correct order.

# Contents

# List of Figures

# List of Tables



# List of Listings

# 1 Introduction

Radiographs were a huge advantage for medicine. They enabled looking through skin and flesh to make bones visible. Techniques like magnetic resonance imaging (MRT) improved medical possibilities further by creating a three-dimensional density image of a patient.

Before 1975 there were first methods to visualize three-dimensional scans by a computer. First volumes were often composed of multiple two-dimensional images. With improving hardware and software, real three-dimensional scans were enabled. While taking density scans improved, visualizing became more complicated.

Todays graphical processing units (GPUs) are optimized to visualize huge amounts of triangles. Solid objects can be described by approximating their surface with triangle meshes. Typically, these triangles are drawn in a stream. Rendering with a rasterizer converts each triangle to a set of fragments that can be shown or overwritten by fragments in the foreground. Almost realistic graphics can be created this way with high frame rates and enabled todays state-of-the-art game engines.

However, volume visualizations is harder, because different optimizations are needed. Empty space between triangles does not affect render time because it is never handled. In three-dimensional scenes, most space does not contain any object, because only surfaces are shown. For volume visualization instead, there is a value at any position within the volume. A correct conversion into triangles is not an easy and often an impossible task. There is no need for clear surfaces in a volume.

Techniques for volume visualization are divided into direct and indirect. Direct volume visualization works based on rays that are shoot from the camera into the scene or simulate light along paths. Calculating rays enables to render volumes without preprocessing and in selectable quality. In contrast, indirect volume visualizations extract and visualize features of the volume. Typically these features are iso-surfaces at a specific density value.

Today, volume scans can have side widths of multiple thousands voxels. Beside visualization, memory management and further optimizations are necessary for high quality images. If thats not enough one need to compromise and decreasing quality or increasing execution time. The search for suitable parameters can take very long. There are also systems, based on genetic algorithms, that improve themselves by varying parameters according to become more efficient over time. However, if a suitable configuration should be found without rendering each candidate, performance prediction is needed. Evaluating frames costs time and can be very expensive, increasing with amount of features, rays or interaction that need to be calculated.

Todays GPUs can do more than rendering images. Creators of the open graphics language (OpenGL) added a compute shader to their standard in version 4.3. They reacted on existing implementations that misuse image rendering to do mathematical calculations. These compute shaders can be used for particle simulations, physics in games and more. The open computing language (OpenCL) was developed to enable such calculations on GPU even without any graphical context.

The power of a GPU to act as a large vector processor that handles single program fragments with a large bunch of different arguments helped another technique developed in the twentieth century. Neural networks are mathematical models based on linear algebra, that can be used for artificial intelligence and function approximation. They were originally developed for pattern recognition but with the advantage, that they are able to learn patterns by examples. Today, there is a huge amount of data available and storable. Companies with a focus on the Internet like Google, Facebook and Amazon are able to collect huge amounts of data about users. With increasing calculation power, neural networks can predict behaviors, play games and classify image contents or read and translate text. We intent to use this power to predict the performance of a GPU in the context of volume visualization.

In this thesis we present a model based on neural networks to predict performance for volume rendering applications based on its configuration and volume properties. Our model consists of multiple neural networks with equal structure that are trained with different training sets, sampled from a uniform distribution over the renderers configuration space. As a result, all neural network converge to the same optimum what makes their estimations comparable. Disagreement between them are used to predict effects of certain samples on the learning process before explicit evaluation. This way, we enable active sample selection to heavily reduce necessary amount of measurements and learning time for an accurate prediction. We evaluate our model using a given raycaster dataset created by the Visualization Research Center Stuttgart (VISUS) which consists of grid-based performance measurements. Afterwards, we will use our own indirect volume visualization to search, how our model behaves for renderer configurations from a continuous configuration space. Our own renderer utilizes marching cubes to extract iso-surfaces as triangle mesh from three-dimensional density images and organizes them in an octree. This algorithm creates a triangle mesh, based on cuboid cells of a three-dimensional grid over the volume. We sort triangle meshes highly parallel before drawing, to enable correct displaying of transparent surfaces.

## Outline

This thesis is structured in following chapters. In Chapter 2 we show foundations of our work. This contains detailed information about volume visualization and marching cubes, an algorithm we used to extract iso-surfaces from density scans. There is also a short information about performance prediction and a more detailed view on neural networks and their internal structure.

The main part of our work starts with of our indirect volume rendering approach in Chapter 3. We discuss different possible versions how extracted iso-surfaces are preprocessed for visualization. Assumptions and restrictions are named. We close this chapter with detailed information about our implementation.

Our prediction model is context of Chapter 4. There, we explain how our model is designed to select samples and learn to predict. We use a passive and two active learning approaches for our investigations. For training, two different loss functions are used.

Results of Chapter 3 and Chapter 4 are evaluated in Chapter 5. Analyses of datasets are contained, together with configuration and evaluation of our model. We describe how tabular data is sampled uniformly from non-uniform source distributions. Our model is evaluated in multiple configurations that are compared and discussed at the end of this chapter.

We summarize our these in Chapter 6, review limits and show, where potential for future work exists.

# 2 Foundations and Related Work

Three large fields are basis of this thesis. Our investigations focus volume visualization which is therefore described first. The first also contains the marching cubes algorithm we use to extract iso-surfaces from a density scan.

Our second section is about performance prediction in the context of volume visualization. Rendering large volumes is time consuming, but predictions can reduce the effort.

The third section in this chapter is about neural networks and deep learning. We explain important aspects like the internal structure of neural networks, but no deep mathematical backgrounds.

In the last section we will handle active learning and how it can intervene sample selection to improve learning.

## 2.1 Volume Visualization

In this thesis we focus on volume visualization as part of scientific visualization. It focuses three-dimensional data like scalar fields and visualizes them as two-dimensional image to enable a better understanding. These scalar fields can be dense, sparse, or grid based and do contain visual and physical properties. In the field of Magnetic Resonance Imaging (MRI) and Computed Tomography (CT), volume scans are used to generate three-dimensional voxel graphics containing densities. These are used to extract bones or veins. In simulations of astrophysical processes, light transport within the volume becomes a challenge.

Today we distinguish between direct and indirect volume visulalizations. Direct volume visualization applies no changes to the volume which is possible using approaches based on rays or paths. Indirect volume visualization instead and extracts features like iso-surfaces and focusses on informative and not realistic images. Its outcome is often a set of triangles that can be fast rendered by a rasterizer.

### 2.1.1 Indirect Volume Visualization

First volume visualization approaches where made before 1980. Because memory capabilities were small in comparison to existing volumes, these techniques focused on extracting features and are part of indirect volume visualization. Mazziotta and Huang [MH76] made a slice based approach to manually extract and display contours from three-dimensional

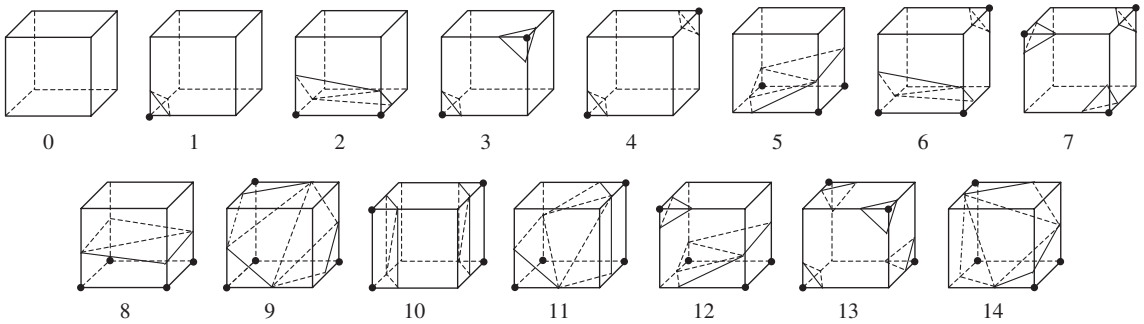



**Figure 2.1:** Triangulated cells of marching cubes. Corners with a black dot are on the inside of an iso-surface while others are outside. Through rotations and inverting of in- and outside, 256 combinations can be created without duplicates. [NY06].

scans. Several techniques to combine such contours were brought up Christiansen and Sederberg [CS78], Fuchs, Kedem, and Uselton [FKU77], and Keppel [Kep75]. All these slice based approaches have the disadvantage, that the space between slices is handled differently than within a slice.

Several years later, Lorensen and Cline [LC87] created Marching Cubes to extract iso-surfaces from volume data. Instead of slices, it handles voxels directly and independently from each other. Also, it makes use of lookup-tables. As a result, it can be implemented very fast and memory efficient.

To create a mesh with marching cubes, the volume needs to be divided by a three-dimensional grid with cuboid-shaped cells. The volume data must be evaluated only at cell corners. Additionally, for each corner must be evaluated, if its inside or outside. For example, iso-surfaces can be created, by applying a threshold to the corners density. For each grid cell, a lookup-value can be calculated from all corner values that refers to a cell similar to those shown in Figure 2.1. There are 256 different possible combinations, leading to iso-value aligned triangles within the cube. Only 15 combinations are unique, because of rotations and invertions (Switching inside/outside does not change the triangles).

The original marching cubes algorithm is further researched. A survey about this algorithm is written by Newman and Yi [NY06]. Very similar algorithms also exists, e.g. an also cube based approach [WMW86], implicit spatial surface approximations via functions [Blo88], or Marching Tetrahedra [TPG99] which utilizes tetrahedrons instead of cubes to prevent ambiguities.

Most indirect volume visualizations convert the volume into objects like triangles and therefore can profit from object-based rendering. Empty space is implicitly skipped, when objects are drawn one after another, enabling further optimizations like culling. However, there are also indirect volume visualization techniques, that do not use vertices or meshes. As an example, Drebin, Carpenter, and Hanrahan [DCH88] extract multiple regions out of the image, e.g. bones and fat from a CT scan, and use them as separate images. They are recombined in an image containing labelling colors and a separate density image. From this

density image, edges and gradients are calculated. Slices can be now drawn with shading and partial cutting of features, e.g. opening the skin of a scanned human to show bones.

### 2.1.2 Direct Volume Visualization

Indirect volume visualizations need always preparation and extra memory to store extracted features. Direct volume visualizations instead does no preprocessing of data. One technique is simulating a camera by shooting rays into the volume [App68; GN71]. Physical values can be mapped to colors or materials, enabling photo-realistic approaches [KPB12; Max95]. In literature this is known as volume ray casting [RPSC99], volumetic ray casting [CSSC00], volume ray tracing [Lev90] or volume ray marching [ZRL+08].

We use a given dataset, created by a ray casting algorithm. Like many direct volume visualization techniques, it is an image based render algorithm, that also needs to handle empty space in a volume. Therefore, most of them working image-based, meaning that rays are shoot from camera pixels into the volume. If inside the volume, values of are read along camera rays and combined.

Rays may interact differently with given volumes. There are simple techniques like summing of voxels along the ray or maximum intensity projection. Even iso-surfaces are possible by following the ray until a certain iso-value is reached. Therefore, only changes from lower-threshold to upper-threshold need to be handled like a surface. Normals can be simply calculated by local image gradients.

There are also techniques that are not easy to classify as direct or indirect, for example, the transformed volume could be rendered into another view-space aligned three-dimensional image and then directly visualized [VK96].

### 2.1.3 Challenges of Volume Visualization

Even on actual GPUs, volume visualization is not a simple task. With increasing accuracy of scanners and increasing memory, suitable storage structures need to be created to enable fast access. Object based renderings often utilize rasterizers, that convert each object into fragments. Each fragment is assigned to a pixel in the rendered frame. This enables several optimizations like culling or depth-tests to reduce the workload per pixel. Indirect volume visualizations can benefit from these techniques, but direct visualizations cannot. Instead of object based rendering, volume vizualization needs to handle the empty space between objects. Spatial structures like Octrees are often utilized to speed up ray tracing or to render with level-of-detail. At last, many visualizations work with transparency to show details below the surface. While direct volume visualizations can handle this easily, indirect once working with meshes need to invoke a sorting strategy for correct displaying.

## 2.2 Performance Predictions

Performance is a measurement that enables comparing of systems. It combines abilities and resource usage, whereby needed abilities increase and resource usage decreases performance. In context of computer science it describes usage of memory, execution time, hardware workload, page failures in databases and similar. If these values are known, they can be used for design decisions to build efficient systems.

It is not an easy task to measure performance for complex systems. For algorithms, analytical solutions are often used to predict runtime and memory usage. This is possible, because every single part is known and can be proofed mathematically. However, these solutions are only theoretically and may perform different depending on implementation and executing hardware. Analytical solutions can be used to approximate the real runtime for specific use cases. Several hardware manufacturers like NVIDIA and AMD offer benchmarks to proof abilities of their own hardware. Benchmarking is also used for other hardware like Random Access Memory (RAM), hard drives, libraries and more. But there is one problem: each benchmark is only a set of tests and may miss special cases. As a result, different benchmarks may return controversial results for same test subject. To handle this issue, own benchmarks or measurements within the own system are necessary to improve performance.

In the context of volume rendering, frame rate and volume resolution are important factors for performance. For interactive visualizations a response time of less than 100 ms is necessary to enable a user to navigate and modify the visualization. Performance prediction may help to improve this by approximating the runtime of a specific render configuration before rendering. However, it is not a simple task, because many factors need to be taken into account. On one hand, hardware abilities are important, like GPU type and manufacturer, available RAM and storage. Needed values are not simple to determine, because the internal structure of a GPU may matter, but is not always known. Operating system and network configurations may also play a role. On other hand, there are parameters of the renderer that have a large influence and are simple to determine, like volume resolution, desired image resolution, scene complexity, etc. These can also be modified and sometimes adjusted to improve performance on cost of quality. There are also self-tuning systems, reconfiguring themselves during visualization to increase frame rates and quality by fitting parameters and replacing functions.

This thesis is about performance prediction from measurements. We will focus on render times, however, it is a complex task, because of the high input space and an expected complex mapping to render time. Instead of interpolating between different measurements, we regress an approximation using neural networks.

## 2.3 Neural Networks and Deep Learning

Artificial Neural Networks (or short Neural Networks) are mathmatical models using linear algebra to fit a certain function. Deep learning is close assigned to neural networks, descibing their intern structure. They consist of multiple layers that transform data in a pipeline-like way.

Several books describe, how neural networks work and can be trained. See for example "Neural Networks and Deep Learning" by Nielsen [Nie15], "Backpropagation: theory, architectures, and applications" by Chauvin and Rumelhart [CR95], or "Deep Learning" by Goodfellow, Bengio, and Courville [GBC16].

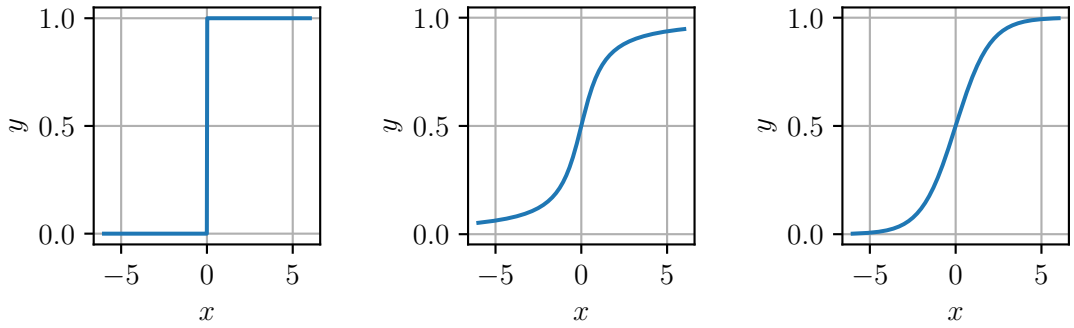
### 2.3.1 Internal Structure

Neural Networks are roughly based on the human brain. Rosenblatt [Ros58] developed the idea of perceptrons, which are artificial neurons. A perceptron is a single unit with multiple inputs, where information is read. This information is processed and generates a single output signal. Therefore, it can be modelled as a (high-dimensional) mapping function from input $\vec{x}$ to $y$. Rosenblatt defined inputs and output to be binary. In his perceptrons, all inputs where multiplied by internal weights $w_i$ and mapped to a binary value using thresholding. As a result, perceptrons work as classifiers and can be used to make simple decisions. Each can only decide between 0 and 1 or inside and outside of a class. However, their decisions are centered in the input space and do not perform well, if the correct solution is not centered as well. Adding an additional bias $b$ solves this issue and leads to following internal perceptron equation:

$$y = \sum_{x_i \in \vec{x}} w_i \cdot x_i + b = \vec{w} \cdot \vec{x} + b \tag{2.1}$$

Combining multiple perceptrons increases their decision power. If each neuron applies different weights to inputs, it is able to recognize a specific pattern within the input and as a result assigns meaning to its output. Maybe one perceptron has correct internal weights, but most certain none has, assuming a random initialization. However, there is another interpretation possible for perceptrons output. It can be used as a more abstract description of the input. When the input is plugged to outputs of other perceptrons, it becomes a transformed version of original input data and perceptons are enabled to make more complex decisions.

Often, neural networks are organized as a set of layers. Each layer is densly connected to its prior layer, but the first is connected to input data. The last layers output is also the networks output. It is allowed to have more than one output dimension, enabling classification for multiple classes. Layers that are not connected to input data nor network output are called *hidden layers*. To calculate one layers output, Equation (2.1) can be extended to a fixed number of $j$ perceptrons.




**(a)** Threshold **(b)** Normalized Arctan **(c)** Sigmoid

**Figure 2.2:** Comparison of different activation functions for neural networks. A threshold step function (a) saturates too fast and is not smooth. Normalized arctan (b) does not saturate fast enough. Sigmoid (c) is well suited for learning, as long as input values in $[-3, 3]$.

$$y_j = \vec{w}_j^T \cdot \vec{x} + b_j \Rightarrow \vec{y} = W\vec{x} + \vec{b} \tag{2.2}$$

The dot-product is rewritten as vector multiplication. As visible, stacking of all $j$ equations results in a simple matrix multiplication. Each layer transforms its input space into an output space with any dimension. Each dimension of this space represents a different feature or pattern and is therefore a more complex and abstract description of the input space. Adding more perceptrons per layer leads to more features and a better description. Adding more layers to the network enables more complex decisions.

Decisions of a neural network become more complex, the more neurons take part in the network. Each decision divides the incoming data into two classes 0 and 1 by thresholding. Learning means fitting a step function to the input space. Multiple step functions can weight different parts enabling later layers to make their decisions based on regions. These can be used to build all logical functions like `and` and `or`. However, it is impossible to create smooth approximations.

In prior equations, there is no threshold applied. It was omitted, because state-of-the-art neural networks do not use a threshold function for activation. Instead, more smooth functions like $arctan$ or $sigmoid$, shown in Figure 2.2, are applied to the outputs to map them into $[0, 1]^j$. These functions are continuous and monotonous increasing what supports optimizers that are based on activation gradients. Sigmoid has also the advantage, that it saturates faster, what decreases inner weights in each perceptron. Smooth activation functions enable finer decisions and very complex regression to arbitrary functions by reconstructing them piecewise. Each neuron now decides, if the output raises or falls for a certain input which enables smooth function approximation and regression. Even the gradient can be approximated by scaling the input. Binary decisions can still be made by applying a threshold on the output layer. Moreover, certainties are now derivable: For

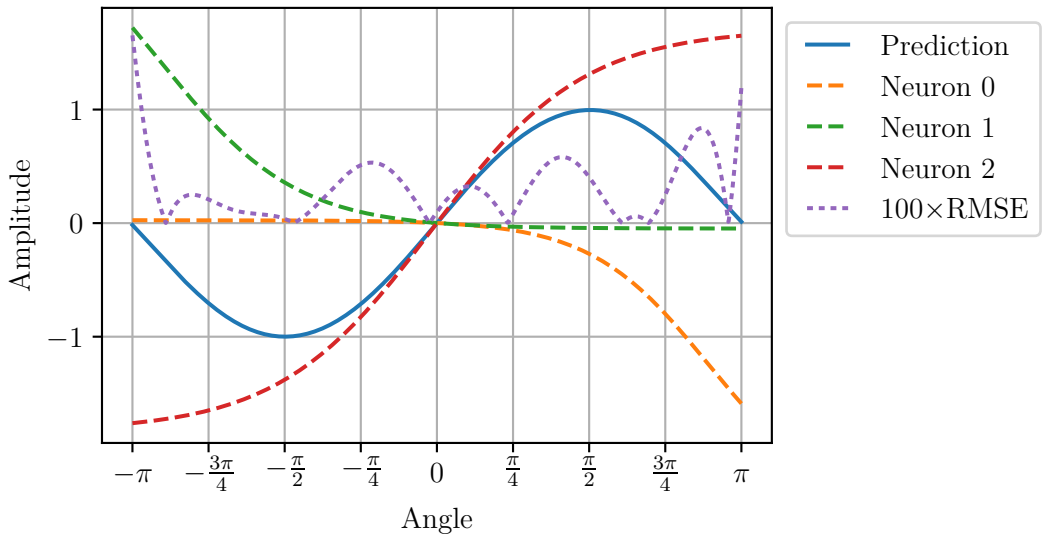



**Figure 2.3:** One period of the sinus function approximated by a neural network with one hidden layer of three perceptrons. Dashed lines show influence each. The root mean squared error (RMSE) increases at the borders, because there where no training samples outside.

multi-class decisions, the network can output a value for each class that states, how sure the decision is correct. Besides this enlarged possibilities for decision making, using these smooth functions improves also learning abilities like described in the next section.

The expressive power of perceptrons is large but cannot fit to any function. Their internal structure is only a linear transformation and cannot model higher functions like power or logarithm. Within a range close approximations are always possible, if the amount of used neurons is high enough. Input transformations can simplify the learning task. As an example, classifying random points into those with euclidean distance smaller 1 to origin and all others is very hard using only coordinates. Feeding this distance as an additional or the only input simplifies the task to a single decision that is easily learned. Whenever possible, such transformations should be applied to the networks input before the first layer. This leads to input selection as an important point of neural networks. On other hand, one should not extend the input if its not necessary. Deciding, which inputs are important and which can be dropped is also not easy. A single additional random input is able to slow down learning heavily.

### 2.3.2 Training a Neural Network

Neural networks are able to classify, regress and transform many types of data. The expressive power of neural networks is very large, caused by the also very large amount of free variables. Each neuron of a neural network has one weight $w_j$ per input and a bias $b$ that can be set to any value. Moreover, the error can only be measured at the output layer.

It must be transported backwards through the network and assigning the real error to each neuron. Rumelhart, Hinton, and Williams [RHW86] found that back-propagation of errors is a fast and accurate way to train neural networks. It is a well researched algorithm and further improved over many years [Bus98; CR95; Wer90; WL90].

In its core, back-propagation is an optimization algorithm. Therefore, a measurement is needed to calculate the correctness of network states. This is modeled as a cost function $C$ with pairs of predictions and ground truth values. The overall cost of the network is typically derived from the cost of multiple pairs by simple functions like mean or maximum. Latter can be very unstable, especially if costs are only approximated by random samples. A typical cost function is using the squared error or L2 norm. In some contexts, more complex cost functions are needed, e.g. in some classifications errors differ in cost, e.g. taking a toxic fish can cost a live while throwing away a good one causes hunger. Therefore, all free variables in the network define its state. Optimization happens by calculating a direction in the space of these variables, that will improve the overall cost.

If a prediction is made by a neural network, the input is propagated trough each layer. Each layer will generate a weighted sum of its input which we call activation $a$ plus a scalar bias $b$ to decentralize the estimation and improve its expression power. This value is referred as $z$. At last, a normalizing function $\sigma$ maps this value to the output $y$. Using $x$ as input (of the prior layer) we got $y = \sigma(wx + b) = \sigma(a + b) = \sigma(z)$.

At the output layer, the overall error of the network is calculated. Moreover, a gradient $\delta = \nabla C \odot \sigma'(z)$ can be approximated by evaluating multiple pairs of predictions and ground truth, with $\odot$ the element-wise multiplication or Hadamard product. Here, its also visible, that networks where $\sigma$ is a threshold learn slower than if $\sigma$ is derivable. Vector $\delta$ can now be used to alter weights of the prior layer according to their error. Now, this error vector is propagated backwards through the network until it reaches the input layer. One layers $\delta$ is calculated from a prior known $\delta$ as $\delta = (w^T\delta) \odot \sigma'(z)$ until the input layer is reached.

### 2.3.3 Challenges of Neural Networks

The idea of neural networks is very old. However, through the high amount of available data and computing power, they become more and more useful. Neural networks are well researched and widely used for all kinds of learning. They are used to predict image content, read text, controlling agents and more. Frameworks with efficient implementations exist, like Tensorflow which is developed at Google by Abadi et al. [ABC+16]. NVIDIA created cuDNN, a library for that are optimized to use and teach neural networks on compatible GPUs.

However, there are still challenges and aspects one needs to mind. First of all, there is no rule of thumb how to choose the amount of layers and perceptron count in each. Often trial and error leads to a network that has a suitable accuracy, but it is still an optimization problem. Several approaches exist like growing networks [Fri94; HVD01] and pruning of nodes [HSS05] to adjust the internal structure. There are even more complex ways to build

neural networks, e.g. by combining multiple to a tree. There are also recurrent neural networks, that got prior decisions as input.

Another large challenge is the selection and adjusting of input. Neural networks are based on linear algebra, therefore they are not able to apply other functions like power, logarithm, expressions or trigonometric functions. Dependent on the data, transformations simplify the task of the network heavily. As an example, classifying all real numbers into those within a circle of radius $r$ around the center and those outside is hard to solve having only Cartesian coordinates. But if the euclidean distance between a point and the center is given, the task comes down to a single decision. A designer has also to mind, that all inputs are in a range, where function $\sigma$ is not saturated. Otherwise, the vanishing gradient problem may arise. This function maps all real numbers into $[0, 1]$, meaning that even infinite values are mapped in this range. Therefore, large regions have almost no gradient. The standard steepest decent approach for back-propagation learns very slow in these regions because it uses the full gradient with length. A simple solution is scaling input values into a range, where $\sigma$ has a high derivative. For the sigmoid function this is in range $[-3, 3]$.

While neural networks are simple to use if the input is a fixed size vector of numbers it is far more complicated to use other data types. In context of classifications, names of classes can be numbers names or anything else. For the neural network, these numbers should not be used directly. Otherwise, they are mixed up in illegal ways, e.g. if class 1 and 3 are most likely, the network returns 2. Therefore, enumerations should be modeled as one-hot vectors, enabling more meaningful response. For real texts (not only names) handling is far more complex. In Tensorflow, there is an special input type for texts, where it is first mapped into a high dimensional space before fed to the network. This mapping is also learned and can be understood as clustering.

## 2.4 Active Learning

Neural networks are trained by feeding many pairs of input and ground truth also known as samples. Typically a network is trained until its error is small enough or convergence criteria are met. Active learning instead is able to intervene sample selection. By doing so, active learning can reduce cost and improve learning by deciding, which samples will offer a larger benefit.

Krogh and Vedelsby [KV95] showed that predictions are more accurate, if multiple estimations are made instead of a single one. This holds especially for instable learn algorithms like neural networks. Differences in initialization and training sets increase differences between all used estimators without loosing validity. This has two advantages: each estimator will have a different error. These errors cancel out, if the mean of all predictions is used. Krogh and Vedelsby called this standard deviation ambiguity. We also use this term.

Opitz and Maclin [OM99] compared Bagging and Boosting. These are two different approaches for active learning with multiple neural networks. Like Krogh and Vedelsby they stated, that simple averaging of multiple neural networks has an expected lower error

then each for its own. An infinite number of different neural networks therefore has an expected error of zero.

Bagging is especially suited for instable learning algorithms like neural networks and decision trees. Each learner gets a different sampled training set, therefore they learn a different error. The more networks disagree, the more stable is the result.

All training sets are sampled from same sample set. Therefore, all will converge to the same optimum. Using the ambiguity we have a measurement of agreement between all submodels. Samples can be voted by ambiguity. However, a small ambiguity does not mean a low error, but indicates that there is improvement possible.

Boosting on the other hand works by building a sequence of estimators. Samples move through this sequence until they are well estimated. However, because some samples are more often predicted than others, this algorithm is more sensitive to noise.

For good active learning it is important to select and evaluate only good samples, especially if evaluating is expensive. Schein and Ungar [SU07] reseached sample selection for classification. They stated a generalized active learning loop for sample selection, that is used in many variants:

1. Rank T randomly chosen examples and use top n for learning
2. Stop, if training set reaches desired size

For ranking, they used uncertainty sampling methods. For example Shannon Entropy, smallest margin which is the difference between most likely categories or query by committee where the most voted class is chosen. In each case, samples that improve the whole models certainty are used.

# 3 Indirect Volume Visualization

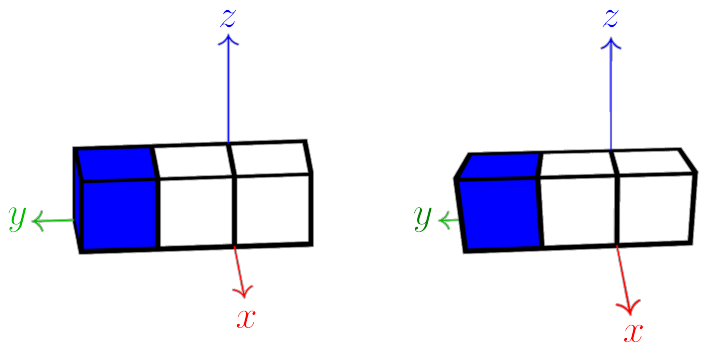
Indirect volume visualization is a technique to display three-dimensional image data like density scans. In this chapter we describe, how our approach is created and implemented. Our focus lays on static volume files that are displayed in random configurations. Therefore, preparation and setup-time will not be part of the render time. Further, no information of prior frames will be used to optimize results.

Our renderer extracts iso-layers from three-dimensional density images as triangle mesh and renders it with alpha-transparency and different colors. Beside drawing of the triangles, we added a GPU-based sorting to render time.

## 3.1 Extracting Iso-Surfaces

We use marching cubes to create iso-surfaces. It is developed by Lorensen and Cline [LC87] and described in Section 2.1.1. This algorithm works with cuboid-shaped cells, generating density-aligned triangles within each. It is an efficient algorithm, because it uses lookup-tables and linear interpolation. Cells lay on a predefined grid, that is independent from the volumes resolution. At least eight cells are necessary to visualize a surface and not more than the volumes resolution plus one are necessary. This is mainly enabled to add more configuration parameters for our learner. For simplification we define for the rest of this chapter, that grid size is one cell larger in each direction than the volumes resolution. Grid and volume are aligned such that centers of the grids corner cells are congruent to the volumes corners. If there is one more cell along an edge than voxels, both become equally sized. Moreover, grid points, where the densities are read during mesh generation, coincide with the voxel centers. The outer layer of grid points is always outside the volume and set to a density of zero to close any iso-surface at the volume borders.

Marching cubes generates a mesh that has some remarkable properties: First of all, there are no conflicts between triangles. They do never cross each other nor overlap in a loop, enabling object based sorting before drawing. Moreover, triangles do not share any points but edges and will never touch, when created different iso-surfaces. Secondly, vertices stay always on cell edges, triangle edges are always on cell surface and triangle surfaces are always inside one cell. Marching cubes, generates a fix maximal number of triangles per cell and iso-layer. These triangles will share edges with each other or with triangles from neighbor cells. There are never triangles in multiple cells or outside cells, which eases the challenge further. The problem can be split into sorting of cells and sorting of triangles within a cell.




**(a)** Orthografic camera **(b)** Perspective camera

**Figure 3.1:** The perspective sorting problem for cells. The blue cell on the left can be drawn in order for orthographic cameras (a), but not for perspective cameras (b), where the middle cell must be drawn last.

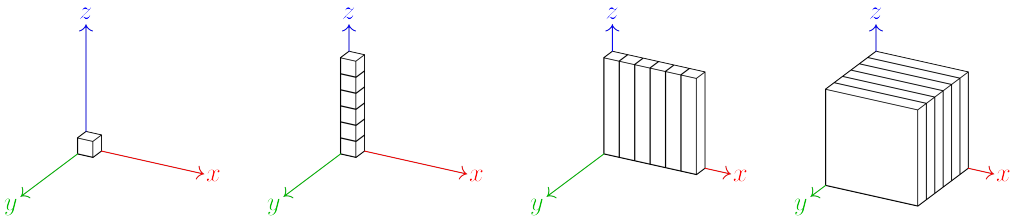
## 3.2 Drawing the Mesh

Our mesh consists of iso-surfaces. Per definition, iso-surfaces that describe different density values in a volume can never touch each other. However, higher densities are always contained in lower ones, making it necessary to draw with transparency to enable insights about what happens within the volume.

A common way to draw with transparency is using an $\alpha$-value, that describes how much influence one object or image has to the result. There are multiple ways to combine a foreground color with a background. Order-independent methods like a weighted sum or simple multiplication of colors exist, but their results do not look realistic. Naturally understanding, whats in front or back is not supported by these methods. Hence, we will use sorting to find the correct order of triangles before drawing. Triangles that are sorted can be combined by $c_{\text{result}} = \alpha \cdot c_{\text{front}} + (1 - \alpha) \cdot c_{\text{back}}$, where $c$ refers to a color. This method is quite more realistic, because the foreground can completely cancel out the background. However, drawing with transparency is not commutative. Meshes must be drawn from back to front.

We will use the standard OpenGL pipeline to draw our resulting mesh. This pipeline is object based and rasterizes points, lines and triangles. It is programmable and supports our prior described transparency handling. Drawing triangles from back to front is not supported by this pipeline, because it is a scene-dependent problem that is not solvable for the general case. Here, using marching cubes to generate the mesh has advantages. Some unsolvable sorting problems cannot appear in our mesh, like cirular overlapping triangles or crossing triangles.

Cells are non-overlapping, have exactly same shape and rotation, which is axes-aligned. Only the position varies, but is restricted to the given grid. To further ease the challenge, we assumed an orthographic camera and therefore only affine transformations when the




**(a)** Reference cell **(b)** Cell row **(c)** Cell plane **(d)** Whole volume

**Figure 3.2:** Drawing cells sorted by axes. In this example, $z$-axis has no influence on the distance and is therefore drawn first. $y$-axis has a larger influence than $x$-axis, visible by the shorter arrow. Hence, it is drawn in $x$-, then in $y$-direction.

volume is transformed to view space. This enables sorting cells in view space by distance to determine a valid draw order. For perspective cameras, these assumptions do not hold as shown in Figure 3.1.

Sorting all cells globally is problematic because of two aspects: We need at minimum a runtime of $\mathcal{O}(n \log n)$ and therefore should parallelize our implementation for faster execution. Global sorting is not always necessary, because there is more than one valid draw order. Transparency drawing is not commutative but associative and therefore each draw of a cell is valid if all cells behind it are already drawn. Such a reordering is undistinguishable from the drawn outcome of the global order. In other words, if we test an arbitrary camera ray all hit triangles need to be in order. This enables a simple segmentation of the volume into background and foreground. The only left challenge is drawing them in correct order.

For a perspective camera, another sorting approach is necessary. In perspective space, the distance of a cell is not sufficient to determine a valid sorting order, because transformed cells do neither share size nor shape. Cells, that ar further away may be in the foreground of nearer cells. Our main focus lays on predictions using neural networks and therefore we do not handle this issue.

In following sections we describe sketches of our renderer we tested during our development. Only the third was chosen to implement, because it is expected to be more suitable than both other approaches. We discuss advantages and disadvantages of each approach and how we expect each to perform on the GPU.

### 3.2.1 On-the-fly Mesh Creation Approach

If an orthographic camera is used, drawing cells in view space by distance will result in a correct image. Because alpha transparency is associative, we can simplify this sorting to axes sorting, as sketched in Figure 3.2. As sort key, we use their influence on the view distance. The longer this mapped vector becomes, the larger is the influence on the

resulting image. This influence can be positive or negative, whereby a negative influence is handled by drawing cells along this axis in reversed order.

This way, we can determine an order for all cells in the volume at once. The volume is divided into planes along the most influencing axis, into rows along the second most influencing axis and then into single cells along the last axis. Using this method we can simply iterate over all cells, creating triangle and afterwards sorting them within a cell before drawing. From marching cubes, we know the upper bound of triangles that are created. Hence, we can optimize all shaders to match this amount and all lower.

There are several advantages of this method. First of all there is no need for extra memory but the volume texture, that needs to be always accessible in read mode. Triangles in cells are recreated on every draw call, enabling modification of density values assigned to iso-surfaces without unnecessary overhead. Even extending this approach with a level-of-detail cell size adjustment and clipping is possible. Nevertheless, there are also disadvantages. Iterating over all cells cannot skip empty cells a priori, except if they are stored in an extra buffer. Most scanned volume files do contain huge areas of empty space around the scanned object. Even in generated volume file, a huge amount of cells is simply empty. While this is a weak disadvantage, that may be simple to optimize, there is a more complex. We utilize OpenGL to draw the volume. Iterating over all cells and then creating triangles implies usage of the geometry shader. This shader needs to generate and store all triangles of a cell and sorting them. The maximum amount of triangles, that can be created is restricted by graphics cards and therefore the amount of possible iso-layers is a fraction of this number. Creating a huge bunch of triangles breaks down performance very quickly, because they are executed one after another. A parallel creation and sorting would be better, but makes further buffers and data organization necessary.

### 3.2.2 Cached Mesh with OIT Approach

Beside the idea of sorting first and then drawing, there is the possibility to do it the other way round. If we create all triangles in the volume and storing them, the volume texture can be released. Instead of sorting these triangles, they are drawn using the OpenGL pipeline always in creation order. This enables storing them in special vertex arrays and using already implemented optimizations. Also, creation is made once and therefore a decreasing render time can be expected. Because, this drawing cannot result in a correct image for all possible view space transformations, each camera ray is sorted by its own. The last programmable part of the OpenGL pipeline is the fragments shader. The prior executed rasterizer divides triangles into fragments that are assigned to pixel positions in the rendered image. Additionally properties like camera distance, normal and more can be attached to fragments.

Instead of drawing fragments to the image plane, they are stored in a large buffer (often referred as A-buffer [Car84]), what enables sorting along camera rays in a second run. In literature this method to solve the transparency problem is called Order-Independent Transparency (OIT). Following challenges need to be solved: The amount of fragments that

are created during rendering depends on many factors like triangle count, their size and clipping. Therefore, the amount of fragments is not known a priori. Moreover, all fragments are assigned to a certain camera ray or pixel, and because sorting is only necessary along a ray, one list per pixel is the preferred way of storing them. Dynamic allocation is not possible using OpenGL 4.5. Therefore, a solution is restricting the amount of fragments per pixel to a fix number. They collect only $n$ closest fragments, which may result in visible image artifacts if more fragments are assigned. Using linked lists improves this issue, but increases sorting costs. However, if used buffers are too small, even with this approach fragments get lost. Multiple render passes are standard to first determine z-values to collect or to count the amount of fragments that are be drawn. Buffer size is critical, because it is restricted by graphics cards.

We tested linked list structures in a large buffer with fixed size. This size was selected based on triangle count to ensure that most triangles are drawn correctly. If our buffer was full, we calculated a weighted mean of fragment colors on the image plane using the alpha channel as weight. Cells are drawn sorted by axes. Those in the foreground where drawn first, such that buffer overrun only appears for cells in the background which improves the image. These cells are also most probable covered by other cells and therefore the error on the whole image is small. We further optimized fragment collection by early stopping if the amount of collected alpha values is larger than full covering. This reduces the amount of fragments and therefore more visible fragments can be collected.

Using our test implementation we found several disadvantages of this approach. First is the necessary large fragment buffer, especially if large images are rendered. Second disadvantage is sorting of fragments. Linked lists need to be resolved, making many random accesses to the fragment buffer necessary. Their length have a large variance, making data-dependent dynamic sorting necessary. GPUs are built to fast execute data-independent operations on different arguments. Therefore, this sorting becomes a bottleneck of the approach. Third, fragments that are not collected create artifacts. As only advantage, OIT can be used to solve any alpha problem, even if triangles overlap in a loop or cross each other such that their order switches. But this is not necessary in our case.

### 3.2.3 Cached Mesh with Octree Approach

Our final approach utilizes spacial structures and highly parallel sorting on cached mesh. Using marching cubes we generate the mesh as a large list of triangles. We built an octree over this triangle list, whereby triangles are sorted by their cell. Our octree is built top-down by iteratively splitting and reorganizing of the triangle list. We stop splitting, if the amount of contained triangles is equal or smaller than the amount of triangles that can be created per cell. Because we want to sort cells first and triangles second, splits of the octree are always made at cell edges, never through cells.

We sort the mesh using this octree to determine a valid draw order. Therefore, we iterate the octree until leaves are reached. In each iteration, existing children of the octree are sorted and handled from back to front. We want to reduce the amount of memory

writes and therefore defined, that the mesh is never changed. The result of our sorting is therefore a sorted index, that is used to access the mesh in a valid order. It is organized in ranges, whereby the root node has the complete sorted index. Each node splits its sorted index range such that each children gets a subrange in sorted order. The sorted index is recursively split until a leaf is reached. Because we ensure, that ranges are always in a valid order, the leaf can sort its triangles and write their indices into its assigned range. Even clipping can be done during this iteration with a tiny disadvantage. If nodes are clipped, their assigned ranges do not contain triangle indices and must be explicitly filtered.

## 3.3 Implementation

Our implementation is written in Python and OpenGL. Simplifying the interface between our learner and our renderer was main reason to choose Python instead of a faster and compiled programming language like C++. It enables evaluation through rendering as simple function call. However, our rendering pipeline consists of only a few lines of Python code that are only used for management.

### 3.3.1 Creating the Mesh

Building the mesh is very intensive on the Central Processing Unit (CPU). Because it is reused every frame, is has only a tiny influence on performance data we want to measure. We want to cache and reuse built meshes, because Python executes not very fast and therefore slows down learning unnecessarily. Caching enables fast switching between volumes and we expect many volume switches during learning.

Iso-layers depend on three parameters: volume file, grid size and density value. Latter is defined by a real value between 0 and 1. The grid size consists of three integer values larger 2, defining how many grid points exist for a certain axis. Even when there are only 256 different iso-values (volume files are byte-based), the amount of possible volumes is too large to be cached. We reduced the amount to a smaller subset of cubic grid sizes, defined by a single integer. For iso-layers we determine density values depending on the volumes density histogram.

Our goal is to create iso-layers that show a certain feature of the volume. Placing them at fixed positions is reasonable but is also able to miss the range of available densities. Therefore we choose them based on quantiles in the data. If only one iso-layer exists, it should be placed such that voxels are divided into two equally large groups. Two iso-surfaces should divide them into three equally sized groups and so on. Therefore, we create a cumulative histogram over a volume. It is normalized into the range [0, 1] and then simply split. However, if there exist too less iso-values, multiple surfaces get the same density value assigned. To prevent this, we add a tiny value to each bin of the histogram before normalization. The amount of voxels is typically much larger than this tiny value. It has not a large influence but ensures unique density values for each iso-layer.

When building iso-surfaces, our renderer first loads the image file on the graphics card. Python is not very fast in executing and therefore we create the mesh on the GPU. If necessary the volume is divided into multiple parts to ensure, all buffers fit onto the GPU. In an outer loop, we iterate over iso-values and in an inner loop we handle all volume parts. Created buffers are reused whenever possible. Afterwards, triangles are collected by Python and reorganized as Octree

### Creating Triangles

To generate the triangle mesh we use a highly parallel approach, that runs on the GPU in multiple steps. Our implementation is based on a on a C++ implementation of marching cubes by Bourke [Bou97], but adapted to run in OpenGL compute shaders. We build a pipeline-like structure, where each iso-layer is generated separately. To enable large volume files, we also split volumes into smaller cuboids, if generation buffers do not fit in GPU-memory. Our result is a set of triangles and their vertices that contain information about normals and an iso-layer index.

As first step of our mesh creation pipeline we estimate buffer sizes for vertices and triangles. Dynamic allocation is not possible in OpenGL 4.5, so we count and remember all cells that do contain triangles. Theoretically, each cell can contain up to five triangles. Using a buffer that fits the worst case is often not necessary because even in generated volumes almost all cells contain less than five triangles per cell. Here, lookup-tables of marching cubes can be used to get the amount of triangles per cell without creating them. The lookup-index for one cell only depends on densities in its corners. Each corner has an index $i \in \{0, 1, \ldots, 7\}$ and a density $d$. If density $d_i$ is larger than the surfaces iso-value, $2^i$ is added to the cells lookup-table. The lookup-table contains 256 entries, whereby indices of 0 or 255 are completely outside or completely inside an iso-layer and therefore no triangles need to be generated. For density scans like MRI, most cells do not contain any triangle.

If the amount of triangles is known, buffers with sufficient size can be generated. For vertices instead, buffer size must be approximated. We have two upper bounds for the vertex count, one dependent on full cell count and one dependent on counted triangles. From our cell-grid definition we know that each iso-surface is closed, because of our grid definition. Within the volume, each existing vertex stays on an edge between four cells and is therefore used by each. Reasoning is simple: If a vertex is created by one cell, there must be a density change from lower than threshold to higher than threshold. This change is recognized by all cells at these edge. Moreover, marching cubes is designed to connect all created vertices in a cell. Therefore, each cuboid-shaped cell, that has up to 12 vertices one per edge shares each vertex with three other cells. Our first upper bound for the vertex count is therefore $\frac{12 \cdot |\text{cells}|}{4}$. This bound can be far too large, even if only cells with triangles are taken into account, because almost never a cell has all 12 vertices. But we know, how many triangles will exist and that each triangle will have three different cell vertices. Our second upper bound is therefore $\frac{3 \cdot |\text{triangles}|}{4}$, whereby each vertex is again used by four cells.

When our buffers are created, we start our second step, which is the generation of the mesh. Here we use our prior gained knowledge about full cells and only run marching cubes for them. Vertices and triangles are written to different buffers, whereby triangles should contain vertex indices in the end. For this step, synchronization is important. Our prior made vertex count estimations do only hold, if double creation is explicitly disabled. Each thread needs to allocate space for its created triangles and vertices. This allocation is done by atomic counters, that increased by each thread before writing to buffers.

While atomic counters ensure, that triangles are written seamless to their buffer, they do not solve the double creation-problem of vertices. We need to assume, that up to four threads may try to create the same vertex at the same time. Our atomic counters ensure, that they will not override each others result but also cause that they will create duplicates. To handle this issue we created a vertex registration index buffer, where each potential vertex has a unique global index per definition. Theoretically, there can be one vertex on each edge of the cell grid. As follows, one drawback of this buffer is its size, that is calculated by

$$x \cdot (y+1) \cdot (z+1) + (x+1) \cdot y \cdot (z+1) + (x+1) \cdot (y+1) \cdot z$$

with $x, y, z$ dimensions of cell grid. This buffer is larger than three times the cell count and therefore much larger than the amount of real vertices. However, there is a huge advantage. This buffer enables vertex deduplication through synchronization. Each thread is now able to detect, if it needs to create a vertex or if this vertex is created by another thread. If one thread creates the vertex, it takes the next free position in the real vertex buffer by increasing an atomic counter. Afterwards it writes position, approximated normal vector and iso-layer-id. Latter is used while rendering for color selection.

Deduplicating vertices saves memory but complicates triangle creation. Triangles should contain the vertex index in the vertex buffer, but there are synchronization challenges, but atomic values. We use OpenGL 4.5 in our shaders, where no synchronization structures like monitors exist. Therefore, it is impossible to register a vertex by writing its real index to the registration buffer. Instead, we need to write a special “registered” value that is replaced after the real index is known. Between registration and writing of the index, other threads may ask for this vertex. They should never interpret the “registered” value as index but also should not delay until the real value is known. Therefore we decided to write the global unique index of a vertex to the triangle buffer instead of the real index. It will refer to our vertex registration buffer. Before we can use the triangle buffer, we need to correct this, by reading the real index of each triangle vertex from the vertex registration buffer. This is done in a third step of our creation pipeline which is invoked after the creation finished.

After one iso-layer is created, we transfere triangle buffer and vertex buffer to CPU memory to free GPU memory for the next iso-layer. If possible, buffers are reused but cleared. To enable simple concatenation of all created meshes, we use a vertex index offset when correcting indices.

### Building the Octree

The time, necessary to build the octree, is main reason to cache volumes instead of recreate them every time. It is written in Python and runs on the CPU, because it needs dynamic memory allocation which is not possible on GPU.

As first step, a root node is created. Each node of the tree has three properties: position, length, and children. The root node has position 0, meaning that its assigned triangles start at index 0. Its length is equal to the amount of triangles, because it contains all triangles. The last property children is not initialized. The creation runs iteratively and creates the octree as list of nodes. Therefore, an index that maps to the octree is increased every iteration. New nodes are added at the lists end, such that each new node is handled after all prior created once.

In the end, the octree should fulfill following properties: First, all children of one node are in one row, meaning that there is no node between them in the list, that has not the same parent. The position property of the parent maps to its first child and all others can be read by increasing this position by one. Second, the length parameter is always the number of triangles that are contained in all leaves in this subtree. This number is necessary for our parallel sorting, because it enables dividing the sorted index into ranges, that can be handled independently without destroying triangle order. Third, the children property needs to encode, how many and which of the up to eight children exist. Empty nodes should not be contained in the octree. Fourth, each node needs a position for culling nodes outside the view space before drawing. This is used to enable view frustrum culling during iteration and increase the problem size for performance measurements. Fifth, leaves are handled specially. They are marked by the children property (its empty, because they have no children). Also, position does not point to the next node, but to the first triangle in the triangle buffer, that needs to be handled by this leaf. Length has still the same meaning.

To achieve this, we reorganize the triangle buffer on every tree split. Per definition of the marching cubes implementation, there are only 5 triangles per iso-surface in one cell. If more are contained in a node, it is split. Splits are always made as close as possible to the nodes center but along cell borders, what is necessary to keep our assumptions about cell sorting. Therefore, the smallest unit, that can be created is a cell. All triangles assigned to the actual node are reorganized in the buffer, such that position and length of each children map correctly to assigned triangles. For later sorting it is important, that children are created always in the same order. The children property of the actual node is modeled as bit-vector where each existing children is marked. The position of a node is stored in compressed form. The bit-vector of the children property needs one byte. In standard OpenGL 4.5, all data has multiple of four bytes. We use left three bytes for node coordinates. These stay always within the volume and therefore between 0 and 1. To fit each dimension into a single byte, we store them as multiples of to 1/255. This is also the maximum error, that can be indicated which becomes important in the culling task. The position property of the actual node can now be set to the nodes list length before adding all children. For leaves, the children property is simply set to 0.

### 3.3.2 Drawing the Mesh

Our mesh drawing is implemented to sort triangles highly parallel. Thread synchronization is more difficult on GPUs than on CPUs. Reason is, that multi-threading behaves in different ways. On the CPU, threads are executed independently from each other and are time-scheduled by operating system. On the GPU instead, many threads run truly parallel, executing same commands but with different arguments. Data-dependent operations, that are executed only in some but not all threads force the rest to wait. Therefore, a good shader should not utilize branches. Early stopping instead does not increase runtime.

To sort all triangles, we need to iterate over our octree. It has only one entry point and hence, we decided to split its iteration into a single threaded start that enables sorting of the lower octree in multiple threads. The root iteration runs only a few levels deep and creates entry points for the multi-threaded continuation. The first iteration should not run too short. Otherwise, the amount of threads for the continuation is very restricted. However, running it too long is time-wasting. We found, that 3 to 5 levels are well suited for our volumes. For values outside this range large volumes cannot be rendered, because shaders execute too long and reach a timeout of the graphics card.

Our octree iteration works analogous to octree building. First of all a sort order is determined that is valid for all children on every layer. Corner points of the volume are transformed into view space and sorted by camera distance, which is possible, because of our orthographic camera. We get the sorting order of children without reading any position, because the relative order of children never changes and each of these corners can be assigned to a children on any layer. During the octree iteration startup nodes at a given level are stored to a buffer using atomic counters to determine the index. Leaves are also added to this buffer and not directly handled. For the octree continuation each thread reads its own subtree root from this buffer and iterates it until leaves are reached or cropped by view frustrum culling. Leaves are added to a leave access buffer, containing the leaves start in the triangle buffer, its length and starting position in the sorted index. Afterwards, each leave is sorted by its own thread using the leave access buffer.

To speed up execution, shaders are simulated and recompiled for every volume, grid size, iso-layer count and tree split. The upper described iteration through the octree is implemented using a stack and a `for`-loop. Stack size and maximum amount of iterations are inserted into the shader before compilation and can be optimized. Our prior mentioned choice to split the octree between the third and the fifth layer also depends on these shaders. If more than five levels of the octree are executed by one shader, the maximum amount of iterations is larger than 100000. This leads to crashes on our test computer with an Nvidia GeForce GTX 1080 as GPU. Even if the octree has more than 10 levels, far less iterations are executed in the second stage, because the deeper the layer, the more probable that it is sparse. Our leaf sorting is optimized to run in parallel on GPU. Therefore, no `if`-branches are used but minimum and maximum-functions. We use Batcher odd-even mergesort [Bat68] to create a sorting network. This is basically a list of switching instructions that are executed always in same order. Its runtime is therefore constant $\mathcal{O}(n(\log n)^2)$. Each switch gets two parameters $a$ and $b$ and ensures, that afterwards $a$ is

smaller or equal to $b$. This way, sorting can be made without any loop what also supports compiler optimizations and sorting in registers. Batcher sort works best for lists, that have a length of $2^n$ with $n > 1$. It reduces sorting the list to dividing the list into smaller parts of four elements, sorting them and then recursively merging sorted parts. In our implementation, we need to sort lists of length $5 \cdot l$ with $l$ the amount of iso-layers. We therefore construct a sorting network matching the next valid creation size and cropping all switches, where an out-of-index elements is accessed.

Drawing the volume is made using the OpenGL pipeline with a vertex shader and a fragment shader. We use a draw instants call with the amount of possible triangles. A pseudo triangle is used in every call, that consists of three enumerated vertices. The instance number is used to read the next triangle as set of three vertex indices from the sorted index. Using the given vertex number, all necessary attributes like position, normal, and iso-layer index are read from the vertex buffer. This way we do not need a geometry shader. Caused by octree clipping it is possible, that not all triangle indices are valid. If an invalid index is read, all vertices are mapped to a specific position outside the view space. This way, these triangles are implicitly discarded, because explicit discarding is only possible for fragments. The fragment shader gets normal, position and iso-layer index and maps latter to a material. Afterwards it illuminates the fragment using three different colored point lights. Example images of rendered images are shown in Figure 3.3

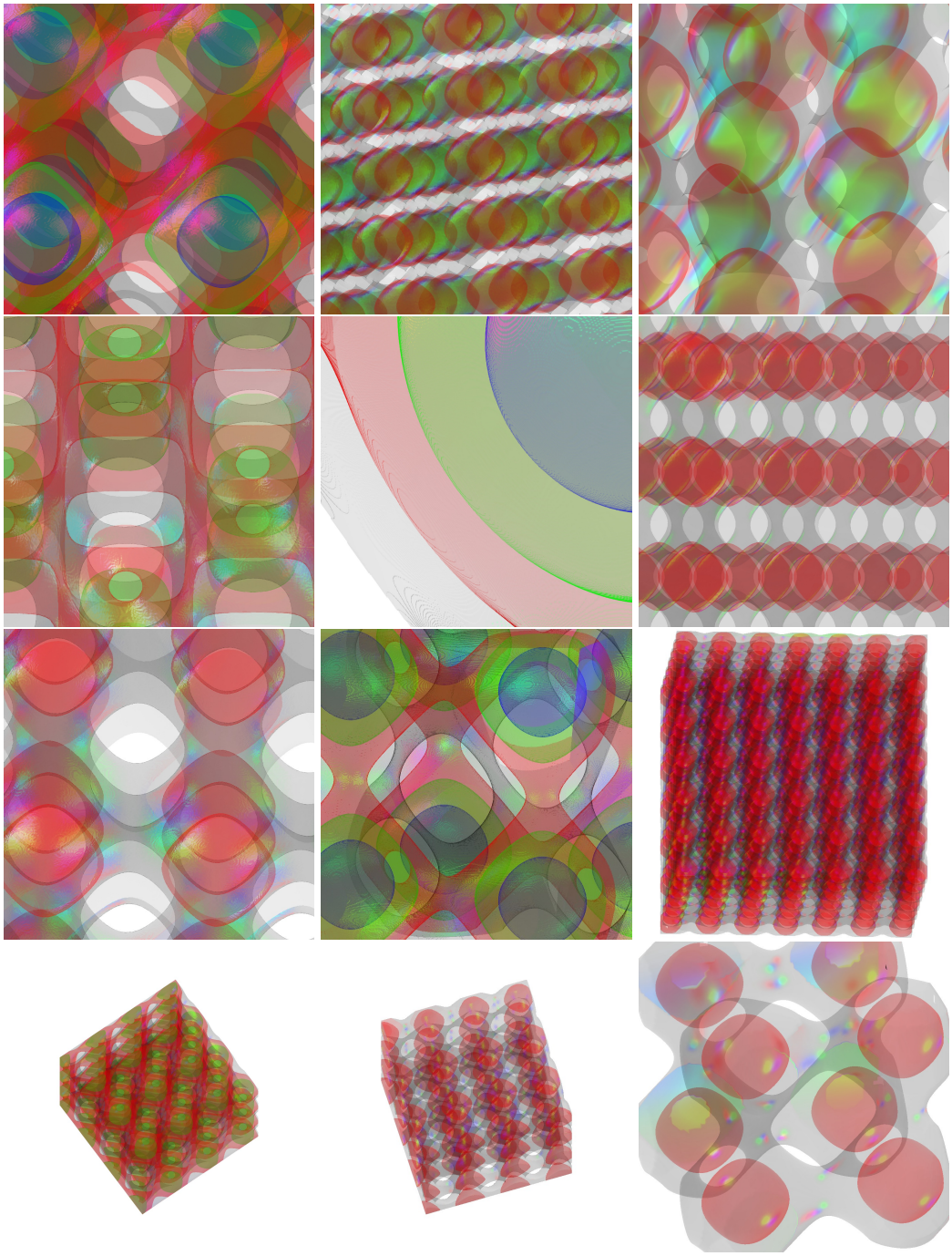


**Figure 3.3:** Example renderings of our indirect volume visualization.

# 4 Investigations through Active Learning

We want to predict performance for volume renderers under different configurations. These configurations include several parameters but also graphics cards. Our investigations are designed and tested using a given dataset offered by VISUS. It contains grid-based configurations, where execution time is measured by a volumetric raycaster on different graphics cards.

For our investigations we utilize neural networks, because they are able to approximate very complex functions by learning. If there is a mapping function between input parameters of a renderer and the frame calculation time, a suitable configured network can approximate it with a small error. We aim to reduce the necessary amount of measurements that are needed for an accurate performance prediction. Selecting good samples is challenging and should be done by the neural network itself using active learning and sample selection.

Even neural networks cannot approximate a function, if test data is not suitable. We show, how active learning can be utilized to improve random sampling. Also, different loss functions influence the result. We compare two different approaches and evaluate our network on a fix performance data set.

Our neural networks are first trained on a given dataset, where we have no control about where and how samples are selected and evaluated. We chose this dataset, because it enables a better comparing between different configurations, if all got the same input and expected output.

However, if samples can be chosen from a continuous distribution, using a fixed grid is not a good choice. Frequencies that are higher than the original grid get lost. Using hierarchical or dynamically growing grids may produce better datasets but need very long to be evaluated. Predicting performance using only a grid is possible by lookup but needs a good data organization and a runtime of at least $\mathcal{O}(d)$, where $d$ is the number of dimensions. Approximating the performance by a neural network simplifies this task, because all data can be reduced to a fixed number of parameters in the network. Of course, this may also loose some details, but by increasing the networks size, a suitable approximation can be found, if possible. However, with increasing size, training the network and predicting with the network becomes more and more expensive. Too large neural networks may also suffer from overfitting.

Approximating a value with a neural network has additional advantages. Neural networks do not depend on grid structures and enable a larger freedom of sample placement in the input space. Our main focus lays on designing the network and therefore we want to keep sampling simple. Our approach is to sample in multiple epochs. In each, a sparse subset

of the input space is uniformly sampled and estimated by the neural network. We do not expect, that naive random sampling is better then naive grid sampling. It is another task of our network to decide, which samples are really taken. Only those are evaluated to save time.

Following terms are often used in this chapter:

**Submodel** Our model consists of multiple submodels. Each is a neural network that predicts performance. Submodels are independently from each other.

**Sampling Epoch** In each sampling epoch, new samples are drawn and added to training data. Afterwards, submodels are trained with their full training data set for multiple learning epochs.

**Learning Epoch** Neuronal networks are trained in epochs. In each, internal weights and parameters are adjusted to predict training data as close as possible.

**Good Samples / Bad Samples** For learning, not all samples have same worth. Samples that stay in well matched regions do not help to improve a learning algorithm, therefore we call them bad samples. Good samples stay in regions, where prediction errors are high and therefore enable fast learning.

**Hyperparameters** In context of machine learning, parameters for architecture and about the learning process are called hyperparameters to distinguish them from learned weights and biases, that can also be refered as parameters.

## 4.1 Design of the Our Prediction Model

Our prediction model needs to solve two tasks. First task is approximate performance for specific input vectors. This is very simple, because this follows the main thought behind neural networks. The second task is more special and leads to active learning. The neural networks needs to influence its own training data and making decisions about how useful a sample is.

Our idea is based on a general active learning approach described by Opitz and Maclin [OM99]. Multiple submodels will predict a value for a new input vector and we will derive an ambiguity value for each. This ambiguity is afterwards used to take or reject a sample.

### 4.1.1 Model

Our model consists of $n$ submodels, where $n$ is a hyperparameter. All are neural networks with same architecture built of densely connected sigmoid layers. They differ in initialization but are also trained with different samples as described below. Additionally, we add an input transforming layer that is able to map enumerations to one-hot vectors, scales large

values to prevent saturation of the sigmoid layers and applies functions to transform input data. More details are described in section 'Implementation' on page 42.

According to Opitz and Maclin [OM99] we will take the mean of all submodel predictions as prediction of the whole model. Through differences in initialization and training sets for each submodel, exact matches of predictions can be assumed to simply not happen. Moreover, each model will have a different error which cancel out each other. From diverse submodel responses the ambiguity value is derived. It is used to select training data from a given set of samples, by testing how much submodels disagree. Areas, where disagreement is large, are not well matched and sampling there improves the overall model fast.

### 4.1.2 Estimating Ambiguity

There are several possibilities to measure disagreement between neural networks. Krogh and Vedelsby [KV95] defined ambiguity as sum over the distances between submodel prediction and models prediction. This is based on the assumption, that each submodel will have a different error. Therefore, they cancel out for an infinite number of submodels. This also means, that all submodel predictions are correct predictions plus additive noise.

If the mean of all submodel predictions is closer to the correct solution as most of the submodels, we can estimate that they are drawn from a normal distribution around the correct solution. Therefore, taking the mean of all submodels is consistent. Moreover, the standard deviation between submodel predictions can be used as measure of agreement. Therefore, we estimate ambiguity by calculating the standard deviation.

### 4.1.3 Iterative Training

Ambiguity is used to estimate, how good used submodels match the real performance. It is important, that ambiguity does not decrease too fast. However, this happens when all submodels are trained with same training data. They will learn same features but also learn same errors, which decreases ambiguity even for bad matched samples.

Our model is iteratively trained in a loop. We call each iteration a sampling epoch, because at the beginning, new samples are selected. For sampling we test three different approaches that are described in Section 4.2. Each returns a list of samples, that are added to individual training data pools. The difference ensures, that ambiguity values do not decrease too fast. If we would use the same training data for all, they will match even on bad approximated data. At the end of a sampling epoch, each submodel is trained on its training data. To evaluate the learning process, never selected samples are drawn after learning.

Ambiguity is affected by each new sample and will change over time. All submodels converge to the optimal performance approximation that depends on their configuration. Hence, a small overall ambiguity is a convergence criteria. Convergence to different local optima is possible but becomes unlikely with increasing amount of submodels. After our

model is trained until convergence, each of its submodel is as good as the whole model and can be used for performance prediction.

For training we tested two different loss functions. A widely used one is squaring the distance between prediction and ground truth. In literature it is called L2-Norm or Mean Squared Loss (MSE). Caused by the squaring it will optimize large errors stronger than small ones.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(\text{Prediction}_i - \text{Ground Truth}_i)^2 \tag{4.1}$$

However, if data contains more small values then large ones, even a small deviation may create a large relative error. Therefore we added a second loss function based on the Mean Squared Relative Error (MSRE).

$$MSRE = \frac{1}{n}\sum_{i=0}^{n}\left(\frac{\text{Prediction}_i}{\text{Ground Truth}_i} - 1\right)^2 \tag{4.2}$$

## 4.2 Sampling Strategies

Explicit search for good samples in continuous or high dimensional space is a complex task. Therefore we build our sampling on a drawing-and-reject strategy. All presented versions have in common, that samples are drawn uniformly from input space.

We use several parameters to control sample drawing. First parameter is draw size, which is used by all strategies and defines, how many new samples are drawn per submodel. Sharing of samples between different submodels is allowed but happens with low probability. Second parameter is pool size. It is used by our active approaches to define, how many samples are reviewed before evaluation. It can be understood as a reduction of the continuous input space to a sparse and finite subset. This parameter must fulfill two criteria to be useful. It needs to be larger than draw size (best larger than draw size times submodel count) and it should be large enough to enable exploring. The more dimensions the input space has, the more samples are needed. One need to keep in mind, that only a fraction is really evaluated.

### 4.2.1 Passive Sampling

We added a non-active sampling strategy to create a base estimation for model evaluation. Pool size is irrelevant for this, because all samples are immediately taken. Draw size instead matters much. As shown in Figure 4.1, for each submodel we add uniformly drawn samples and add them to their training sets. For real continuous input space, no sharing of samples can happen but we can expect it for tabular datasets.

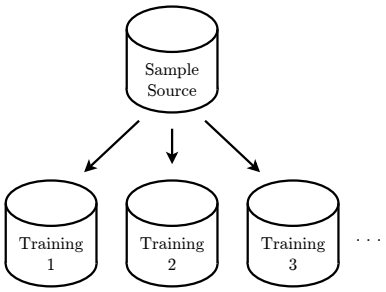



**Figure 4.1:** Passive sampling strategy. Each submodel samples its own training set independently.

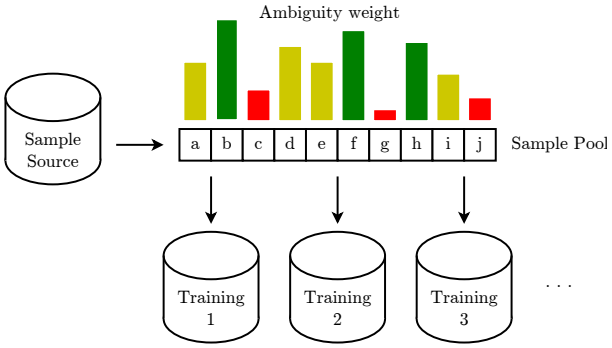



**Figure 4.2:** Active pool sampling strategy. First, a large pool of samples is drawn. Each sample is weighted by its ambiguity to create a new distribution. Afterwards, each submodel samples its own training set independently.

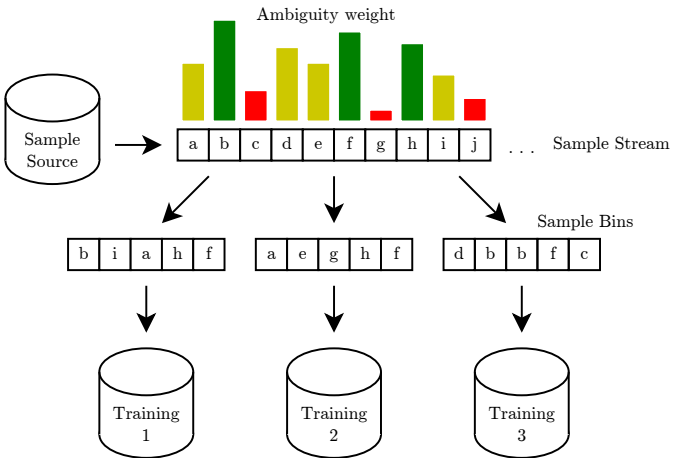



**Figure 4.3:** Active sample search strategy. Samples are drawn one after another in a sample stream. Each submodel has a set of sample bins. For each it decides based on sample ambiguities, if the new sample replaces the bins content. Sampling stops, when enough samples are drawn.

It is a very naive approach and therefore simple to evaluate. Nevertheless, it is able to get samples out of the whole input space. A neural network training with its data will therefore reach an optimal approximation. However, we expect this approach to need very much samples to converge.

### 4.2.2 Active Pool Sampling

For active sampling the pool size becomes relevant. As shown in Figure 4.2, a large pool of samples is drawn uniformly from input space. For each an ambiguity value is calculated from single submodel predictions. From ambiguity, the usefulness of a sample can be estimated. If a sample has a high ambiguity it is most probable useful but a small ambiguity value does not imply a low error at this sample. It can be a coincide, that all submodels agree. Therefore, we want to allow a small amount of bad samples to be used as training data. We do this by weighting all samples in the pool by ambiguity and then sampling submodel training data from this distribution. Implicitly this handles the expected shrinking of ambiguity values when submodels converge.

In comparison to passive sampling, our active approach is expected to need less samples for same learning result. However, there is a drawback caused by the shared sample pool. It is possible, that some samples have a far higher ambiguity than others. This may be caused by chance or outliers in the data. These samples have a higher probability to be drawn by multiple submodels, leading to a fast decreasing of ambiguity. If samples are decreased too fast, bad matched regions may be dominated by such samples. One way to handle this issue is selecting a very large pool size. This decreases the probability for these samples to be drawn. On the downside, this also decreases probabilities for all good samples as well. A more efficient way is data cleansing. Reduction of outliers reduces probability of bad matched regions. A simple technique to do this is using only the median of multiple evaluations. Depending on the problem, more or less samples are needed.

After a first evaluation we found that active sampling with a large pool is not much better than passive sampling. Reason lays in the amount of possible samples, causing that even good samples have a low probability of being taken compared to the probability that any bad sample is used. To increase the probability of good samples we added gamma correction. It comes from the context of image correction to transform brightness. We utilize it to correct weights of samples. Therefore, all weights are rescaled into the range $[0, 1]$. Afterwards, they are exponentiated with a fixed value $\gamma > 0$. If $\gamma$ is larger than 1, low probabilities are stronger decreased than high ones. If $\gamma$ is between 0 and 1 the opposite happens. A large $\gamma$ therefore enables faster exploring but sampling becomes more sensitive to outliers. However, it is an effective way to increase the contrast between high and low ambiguity values.

### 4.2.3 Active Sample Search

Drawing only a fixed sized pool of samples has following disadvantage: Ambiguity decreases over time. While in the first sampling epoch a high diversity between submodels can be expected, they converge to the same optimum, because they have the same internal structure. As a result, there are more and more well approximated regions.

Our idea is to increase sampling pool size by need. Therefore, we re-interpret the pool as a stream of samples, as shown in Figure 4.3. samples are drawn and based on their ambiguity selected or rejected. Each submodel gets bins to hold draw size samples. Sampling stops, if enough samples are chosen by all submodels or if a timeout is reached. The timeout is triggered if the amount of drawn samples reaches pool size. After each timeout, pool size is increased.

Our first version uses a simple threshold to reject and a fair sampling to select new samples. We refer to the maximum ambiguity of the prior sampling epoch as $A$. For the first epoch it is set to 1. A submodel has $b$ bins, each holding one sample. If a sample is drawn, first its ambiguity $a$ is calculated. Each sample is then taken with probability $(\frac{a}{A})^\gamma$ and else rejected. Afterwards, each submodel decides for each bin, if a taken sample replaces the bins content. Therefore, samples are numbered. Sample $s$ with $s \in \{1, 2, \ldots\}$ replaces the content of any bin with probability $\frac{1}{s}$. This way, each sample has the same probability for each bin.

Sampling stops, if enough different samples are drawn or if a timeout is reached. We expect, that multiple submodels are used and defined enough samples to be equal to draw size times submodel count - 1. The timout is necessary to prevent infinity loops for small $A$. By accident, $A$ may be such large, that less than a percent of all drawn samples have a chance to be selected by a submodel. However, through a high amount of drawn samples, the same problem like without the $\gamma$-correction appears. Drawn samples cannot be expected to achieve any advantage in learning. A break by timeout results in a learning epoch with less additional samples but in the next epoch, $A$ will be far smaller and more convincing.

We intended, that searching has several advantages, like a higher ambiguity of selected samples and therefore a faster learning with less evaluations. Choosing adjusts to prior results, meaning that the better the approximation, the longer it searches for bad approximated regions. On one hand this enables a better sample selection but also prolongs sampling. There are further disadvantages. It is very hard to analytically proof, that $A$ has no unexpected side effects. For it is the maximum ambiguity of the prior sampling epoch it is very sensitive to outliers and noise. A high data quality is needed to prevent unexpected behavior.

As a result, we decided to drop factor $A$. Instead, we compare each drawn sample with the actual bin content. Samples with high ambiguity should be accumulated while low ambiguities should be dropped. However, the ambiguity is not an estimate of the real error and therefore we want to allow a small fraction of samples with a low ambiguity. A low amount of bad samples further stabilize almost good matched regions and do increase ambiguity in bad matched regions, where ambiguity is low by accident.

A new sample with ambiguity $a_S$ is drawn from the stream, with $S$ the samples number. Each bin has an ambiguity $a_b$ assigned, which is inherited from the contained sample. If no sample is contained, $a_b$ is defined to be 0. Samples with a higher ambiguity should be taken more often than those with lower. Therefore, we calculate an improvement probability by $p(\text{improve}) = \frac{a_S}{a_S+a_b}$. If both ambiguities match, this value becomes $\frac{1}{2}$ and if $a_b$ is 0, our improvement is 100% which is the case for the first sample. Combining it with fair sampling, a sample replaces an already chosen one with $p(\text{fair}) \times p(\text{improve})$. Drawing is therefore not fair anymore but becomes more fair with increasing ambiguity.

## 4.3 Implementation

Our implementation is written in Python 3.5 using Tensorflow 1.2 by Google [ABC+16]. Python is a scripting language that is not quite fast as compiled program languages like C++ but offers many modules for scientific work like math, visualization and more. Tensorflow is a machine learning module offering a rich Application Programming INterface (API) for Python where machine learning models like neural networks can be created in a very abstract way. Also, the learning process is optimized to utilize available hardware capabilities like math libraries, multiple CPU kernels or graphics cards.

Our implementation consists of three parts. We have a sampler which draws from input space and evaluates them. Next, there is a learner, that also creates neural networks and collects training data. At last we have an evaluator, that is only known by a sampler and used to evaluate samples on learners request. Each part is implemented independently and therefore can be exchanged.

### 4.3.1 Sampler

To access any data from our neural network we use a sampler as interface. Beside sampling it manages transformation and enhancing of data in a predefined way. Evaluation of samples is also handled via the sampler because it already needs access to the dataset and its definition.

There are two sampler versions. First one samples from a static dataset, containing grid-based data. The second sampler gets and evaluator and information about ranges and types of input. Our indirect volume visualization technique was used as evaluator.

There are actually two different samplers. First is the table sampler that prepares tabular data for drawing and expects grid-based input data. We used a given dataset from the VISUS Institute in Stuttgart which contains measurements of a volumetric raycaster on different graphics cards. Second one is an evaluator-based sampler which draws samples without evaluation but also works as interface for evaluation. It uses our indirect volume visualization to evaluate selected samples. There is a slightly interface difference, because table sampler implicitly evaluates all drawn samples and the evaluator needs an explicit

call. Also, parameter ranges for the evaluator must be explicitly set while the table sampler simply reads them.

### Modeling Input Data

Both versions have in common, that each input dimension is modeled as a column of a big table. Each has an assigned name that is used in configuration files. There are following column types:

**Enumerations** Defined by valid values. The table sampler parses all given table files to determine them. For the evaluator, they must be defined in a dataset configuration. Scalar columns can be modeled as enumerations, which is done for the table sampler, where it is important that drawn numbers exist. Even floats can be sampled this way, if they are equal to the last bit. For the given dataset, angle $\phi$ of the camera position was modeled as enumeration.

**Latitude** For angle $\theta$ of the camera position we added an extra column type. It is basically a float value but with special handling. The table sampler draws $\theta$-values weighted, because they where sampled equally distant in angular space and therefore had a higher density at poles.

**Float / Integer Scalars** For our evaluator approach scalars are sampled in a given range. We also enabled transformations after uniform sampling. For example, latitude columns are internally modeled as float columns with transformation.

**Copy / Cache** The dataset contained data that strongly depend on other input columns. Sampling this data may result in non-existing configurations. However, these values may be more precise or informative than the sample-able column. In this case one column is sampled an dependent columns are read. For the tabular sampler this was made for resolutions of images and volumes, that were always equal in each dimension. For the evaluator approach, columns like triangle count where cached and inserted into drawn samples depending on volume configurations.

**Static** Static columns are not sampled. They always have one specified value and therefore do not provide any advantage for learning. However, they can be used as parameter for generators.

**Generator** To enhance data, generator columns use predefined functions to combine or transform other columns. As input columns, only existing columns are valid, no other generated columns. Output column names must also be unique. For missing input values, static columns can be used.

**Label** The label column marks columns with ground truth. The table sampler returns its value with drawn samples but in a separate table. The evaluator approach fills this value only on explicit evaluation and also does not know its value at sampling.

### Prepare Drawing

Both sampler approaches need a different preparation of given data to enable sampling. For tabular sampling this is very data depending and for our evaluator approach it is more related to a given definition.

**Prepare the Tabular Sampler** We aim that samples are drawn uniformly from input space. To achieve this goal with table sampler, all given table files are first parsed. Contents are merged and preprocessed for faster reading and searching. Therefore we used a dataset configuration containing further information like input and output columns, their data type, and sampling strategies. The provided dataset contained columns with constant content for future usage. We dropped these columns by leaving them from the definition.

For each network configuration an own subversion of the dataset is derived. These subversions are mainly used for data reduction by cleansing and filtering. For example, each configuration was evaluated five times in the given dataset. There are outliers in the data but they can be reduced by applying the median before sampling. Also, our learner learns each given GPU separately, which enables subset extraction. We enabled restriction of columns to a smaller value set or range. Columns that are contained in the dataset but not used by the network are also dropped.

Data reduction speeds up sampling and as a side effect also learning. After reduction, an inverted index is built over each sampled columns of the networks input. We modeled them as mappings from column and value to a list of ranges, where this column has this value. As side effect we got a list of all distinct values. Because we have grid-based data, these lists are very short, even for float columns. This implies, that each column is an enumeration with a sparse set of values. As disadvantage, it works only if all grid points are evaluated. Different grids can not be mixed together. Weights for sampling are also determined in this step.

To speed up learning, generated columns are precalculated. Afterwards, it is possible to drop more columns, if they are not used as input but for generated columns. They can still be correctly sampled using the inverted index.

If a value is sampled, each dimension is handled independently. From the inverted index, valid values are picked uniformly or weighted (e.g. for latitude columns). Afterwards, their appearance in the tabular data is read from the inverted index. Each is defined as a sorted list of ranges, and simple to search for overlappings. Such a search may return more than one entry for a specific input configuration, because other columns, that are not relevant, differ in their value. In this case we do a random picking and return one of the found samples.

**Prepare the Evaluator Approach** The evaluator approach is far simpler to prepare than tabular sampling. There is almost no preparation necessary, because there is no source to read valid values and therefore they need to be parameters in the dataset configuration.

Generated columns cannot be precalculated, because samples are not known a priori. However, cached columns must be precalculated or evaluated at sample time. Using our evaluation as example, values like volume resolution, generated triangles and octree size need much time to be calculated. We know that our evaluator will cache processed volume configurations and hence our sampler will render each necessary configuration to fill cached columns, before first samples are drawn.

A cached column is modeled as a black box function mapping from specific columns to a value. We assume, that these values never change, otherwise they cannot be cached. As input we allow only columns with sparse valid data like enumerations, constants or integers. Floats are not allowed. All possible configurations of these columns are enumerated and evaluated. The evaluator will fill in missing parameters with standard values and render them. This implicitly triggers preparation and caching of volume files. Afterwards, the result is split into small key-value-stores, that can be used while drawing to fill in cached columns. It is very important, that mappings from columns to cached columns is sound and complete.

### 4.3.2 Training the model

Our learner creates our models based on user configurations. These configurations are written into separate files and contain general information about dataset and specific neural network hyperparameters.

Each model consists of several submodels to enable sample selection without evaluation. In the network configuration the amount of submodels is set. They are build equally structured but with different initializations. The outcome of our model is defined as mean over all submodel predictions.

Each submodel is based on an Dynamic Neural Network Regressor (DNNRegressor) that is already implemented in Tensorflow. It is a neural network that can be used as black-box with tiny configuration. Some where made fix for all used networks and never changed during learning. We replaced the steepest decent optimizer with the adam optimizer [KB14], because it learns much faster. As activation function, sigmoid is used. During our investigations we modified the internal structure by setting amount and size of hidden layers and different input layers. Networks will learn using the absolute error in Equation (4.1) or the relative error in Equation (4.2). These two loss functions set as command line parameter or explicit in the network configuration.

We are using Tensorflows real valued columns as input for our neural network model. Each column that is named in the network configuration is tested, if it is a number type and added. Text and enumerations are not allowed as direct input but as transformed vector. Scaling factors from dataset configuration are applied by the sampler before a submodel gets any sampled data. Setting weights in the predefined DNNRegressor is supported officially. Otherwise, scaling the internal weights of the first layer is better.

After initialization learning is done in an iterative loop. Samples are selected using one of the strategies described in Section 4.2. We skip ambiguity estimation in the first epoch, because Tensorflow does not allow predictions of models, that never seen samples. Instead, ambiguities are set to 1 for all samples which leads to fair sampling for active pool sampling and almost fair sampling for active sample search. The result is a list of unique samples and information about selections of each submodel. Samples, that are selected by at least one submodel are evaluated by the sampler if necessary. For the table sampler, evaluated data is already contained in the samples and therefore not evaluated again. Afterwards, each submodel is trained.

Training of a submodel also works iteratively. The training set of each submodel contains all prior drawn samples with their ground truth. New samples are added and afterwards models are trained for a fix number of epochs. This number should be selected high enough such that submodels fit to new samples. Over time, each submodel will converge to a slightly different solution because of their individual training sets. The more samples are drawn the closer they match and therefore ambiguity decreases.

To measure the error of our neural network we evaluate a fix number of samples that were never seen by the neural network. These samples are drawn uniformly and predicted by the model. We keep track of drawn samples, ambiguity values and several statistical information over errors and ambiguity like minimum, maximum, mean squared error, mean, median, standard deviation and variance. Measures are made in absolute and relative way. After this data is written to disk, the learning loop starts again with sampling new data.

### Evaluating Samples

For table sampling, evaluation of input configuration is done implicitly. Drawn samples already contain their ground truth and it is up to the learner not to use them for learning. While this speeds up learning by reducing communication, this is not possible using an interactive evaluator. In this case, samples were evaluated only on explicit call enabling also sample modification if needed (we do not do that).

There are some input values, that depend on others in very complex ways and need evaluation to be detected. For example, we sampled different volume files for rendering. These are modeled as an enumeration, but are not fed to the neural network. Modeling them as one-hot vector has the disadvantage that flexibility is lost and on each change, e.g. new volumes, the whole network must be retrained from scratch because input dimensions changed. Instead, more describing properties like the volumes resolution, amount of created triangles, etc. promise a better learning but need evaluation. Our indirect volume visualization is designed such that these values are constants and therefore can be cached after a single evaluation. Afterwards, these values can be simply filled in after the real sampling step.

### Configuration Files

There are two types of configuration files used. One defines the data, its type, transformations and sampling details. The second one defines the architecture of our model, consisting of input, output, hidden layers and submodel count. The training process is configured via command line by pool size, draw size, gamma and how many training epochs are made after each sampling.

Our dataset configuration file is used to define, where to read samples and how. The given dataset used for the table sampler consists of multiple text files containing csv-like tables. Instead of commas, spaces where used to separate data. Strings had no quotes but also no spaces and therefore they can be read without problems. Each file had a file header to identify columns. We read all text files in the given directory and cached them using pandas, a Python module for tables and data analysis.

Following description will show column types and ranges in the raycaster dataset. We also used further information that is not contained in the data to enable a correct interpretation. The dataset was grid based using uniform spacing for all numeric columns. However, many parameters where never changed because the training set. We extracted following data:

**device** Names of ten different graphics cards.

**volume** Names of 22 different volume files.

**volRes** Resolution of the volume file. Was split into volRes.x, volRes.y, volRes.z. All used volumes are cubic and therefore all three columns are equal.

**imgRes** Four different resolutions of the rendered image. Like volRes they are splitted into columns imgRes.x and imgRes.y containing equal content.

**tffId** A binary value (zero or one) that was set only for volRes = 1024.

**viewRot** The camera was rotated around the volume by quaternions. Through analysation of the raytracer source code we determined that these can be mapped to sphere coordinates with fix radius 3. Also, the camera will always look to the origin, which is also the center of the volume. viewRot.x is equal to $\phi$ and viewRot.y corresponds to $\theta$. Each is in range $[0, 2\pi)$ and has 16 values. Because in sphere coordinates $\theta$ has only an extend of $\pi$, each camera position is contained twice but 32 times at the poles. Only difference between these rotations is the implicit rotation around the cameras view direction. The third parameter viewRot.z would have done the same rotation but was never varied.

**execTime** The last column with varying value is the output column. For each distinct configuration there are five entries. They are measured as batch and sometimes contain startup-displacement and outliers. There are almost no values twice in the dataset.

**Listing 4.1** JSON-formatted configuration file of the given raytracer dataset. Columns defined by input and output can be used by our model. These columns are extended by generators (defined as function name, argument columns, new column names). Factors defined under scale are applied to columns before they are fed to the neural network. All columns that can be sampled are listed unter sampling together with their strategy. Not listed columns have strategy copy by definition.

```
{
  "file": {"type": "csv", "separator": " ", "header": true,
  "source_path": "volume_raytracer/", "file_filter": ".*[.]txt" },
  "input": {"device": "enum", "volume": "enum",
        "volRes.x": "float", "volRes.y": "float", "volRes.z": "float",
        "imgRes.x": "float", "imgRes.y": "float",
       "viewRot.x": "float", "viewRot.y": "float",
          "tffId": "float", "stepSize": "float"        },
         "output": {"execTime": "float"},
          "scale": {"volRes.x": "1/1024", "volRes.y": "1/1024", "volRes.z": "1/1024",
        "imgRes.x": "1/1024", "imgRes.y": "1/1024" },
  "generate": [
    ["multiply", ["volRes.x", "volRes.y", "volRes.z" ], ["volRes"]],
    ["multiply", ["imgRes.x", "imgRes.y"], ["imgRes"]],
    ["enum2vec", ["device"], ["<device>"]],
    ["sphere2cart", ["viewRot.x", "viewRot.y"],
    ["camPos.x", "camPos.y", "camPos.z"]]
  ],
  "sampling": { "device": {"strategy": "enumerate", "values": "existing"},
             "volume": {"strategy": "enumerate", "values": "existing"},
           "volRes.x": {"strategy": "enumerate", "values": "existing"},
           "volRes.y": {"strategy": "copy"}, volRes.z": {"strategy": "copy"},
           "imgRes.x": {"strategy": "enumerate", "values": "existing"},
           "imgRes.y": {"strategy": "copy"},
          "viewRot.x": {"strategy": "enumerate", "values": "existing"},
          "viewRot.y": {"strategy": "enumerate_latitude", "values": "existing"},
             "tffId": {"strategy": "enumerate", "values": "existing"},
           "stepSize": {"strategy": "enumerate", "values": "existing"}}
}
```

**Listing 4.2** JSON-formatted configuration file of the given raytracer dataset. This example shows a network that using multiple input-columns and generated columns defined in Listing 4.1. Parameter model_dir refers to a folder, where log-files are stored, “?” is replaced by command line argument.

```
{
  "constraints": [["apply", "median", "to output"]],
  "input": ["<device>", "volRes.x", "volRes.y", "volRes.z", "volRes",
  "imgRes.x", "imgRes.y", "imgRes",
  "camPos.x", "camPos.y", "camPos.z", "tffId", "stepSize"],
  "hidden_units": [64, 32],
  "output": ["execTime"],
  "sub_models": 5,
  "model_dir": "?/all_in_one_network",
  "name": "all_in_one_network"
}
```

To extend these input data we added generators that transform input columns. We assumed, that the amount of pixels is more relevant than the image measurement. Also, neural networks are not able to square input values and therefore it needs to be done before they get it. They are described in Section 5.1.1 on page 54.

In Listing 4.1 our configuration for the raytracer dataset is shown as used by our sampler. As visible, we added the amount of pixels and voxels as generators. Also, viewRot is converted to Cartesian coordinates. The radius is set by default but can also be set using a static column as third. Names of the input and output columns are used to map columns read from the dataset correctly. These names must therefore exist. The scale-entry contains factors that are applied to each input column before generators. They should scale input in the range of roughly $[-3, 3]$, where the used sigmoid layers of our learning model are not saturated. Otherwise, learning is slowed down heavily.

The whole dataset is freed from unused columns (they do not appear as input nor output) and afterwards cached for faster access. From this raw dataset, a subset is used by neural networks enabling faster data access through reduction.

When the dataset is configured, networks can be attached. This is done by defining inputs and outputs that are used, whereby inputs can also be generated columns. Next our sampler will calculate a hash sum based on dataset and network configuration and use it as cache name. If a cached table with this name exists, it is used. Otherwise, the raw dataset is loaded and reduced. Data reduction happens on explicit and implicit constraints. Explicit constraints are defined in the network configuration and allow reduction by functions over the output column. In our example in Listing 4.2 the median of the output is used, but we also enabled mean, max and min. The median is more sound and will most probably not contain outliers. A good alternative will be min, because in our case outliers that are far smaller than the real result are not very common. Also, filtering is allowed, what is necessary to train a specific GPU. Typical comparing operations are implemented. Implicit contraints are made by input and output of the network. Unused columns are dropped and generated columns are precalculated. For sampling, the dataset and an inverted index are stored.

In Listing 4.1 there are no values explicitly given for sampling but the special value existing which leads to reading values from file. Further, there are some special challenges for the GPU. For the evaluator approach, these values must be explicitly set. We use OpenGL 4.5 which has no opportunity to select a specific device if multiple are mounted to the computer. The device parameter is therefore set by hardware and read while initializing. For network configuration we added a special %device% variable that can be used into model_dir and name and is replaced by the devices name.

# 5 Evaluation

In this chapter we will evaluate our learning model and the indirect volume visualization. Our learning model will be evaluated with both, the given raycaster dataset and our renderer.

In the first half of this chapter we will handle data used in learning. We will analyze them by content and parameter, whereby the focus of the raycaster datasets lays on context and correction of distributions to enable uniform sampling. For our own renderer we focus performance and effects of different parameters on render time. In this case, uniform sampling is simple, because we are not restricted to a small set of sample positions.

## 5.1 Analyses of datasets

In this section we analyze both of our utilized datasets. They have several similarities like volume resolution, camera position and image resolution. We name parameters of each and how they can be used for learning. Our analysis of the raycaster dataset focuses, how non-uniform distributions can be weighted to enable uniform sampling. Parts of this sampling are reused in our own indirect volume visualization. However, sampling is far simpler, because arbitrary configurations can be rendered and evaluated. Therefore, its analysis focuses performance of our own rendering and influence of each parameter on execution time.

### 5.1.1 Raycaster Dataset

First, we analyzed the given dataset to determine column types and ranges. They are shown in Table 5.1 and make visible, that there are large similarities between some columns. We also used further information that is not contained in the dataset to enable a correct interpretation. We extracted following data:

**device** Names of ten different graphics cards.

**volume** Names of 22 different volume files.

**volRes** Resolution of the volume file, split into **volRes.x**, **volRes.y**, **volRes.z**. All used volumes are cubic and therefore all three columns are equal.

**imgRes** Four different resolutions of the rendered image. Like volRes they are splitted into columns **imgRes.x** and **imgRes.y** containing equal content.

| Name | Values | Minimum | Maximum | Mean | Median |
|---|---|---|---|---|---|
| device | 10 | Ellesmere | TITAN X (Pascal) | | |
| volume | 22 | chameleon.raw | zeiss.raw | | |
| volRes.x | 10 | 128 | 1024 | 818.55 | 1024.00 |
| volRes.y | 10 | 128 | 1024 | 818.55 | 1024.00 |
| volRes.z | 10 | 128 | 1024 | 818.55 | 1024.00 |
| imgRes.x | 4 | 512 | 2048 | 1280.00 | 1280.00 |
| imgRes.y | 4 | 512 | 2048 | 1280.00 | 1280.00 |
| tffId | 2 | 0 | 1 | 0.50 | 0.50 |
| viewRot.x | 16 | 0.00 | 5.89 | 2.95 | 2.95 |
| viewRot.y | 16 | 0.00 | 5.89 | 2.95 | 2.95 |
| stepSize | 5 | 0.25 | 4.00 | 1.55 | 1.00 |
| execTime | 1134577 | 0.00 | 0.78 | 0.02 | 0.01 |

**Table 5.1:** Raycaster dataset ranges. Column Values contains the amount of different values. There are strong similarities between volRes.*, imgRes.* and viewRot.*.

**tffId** A binary value (zero or one) that was set only for **volRes** = 1024.

**viewRot** The camera was rotated around the volume by quaternions. From the raycaster source code we determined that these can be mapped to sphere coordinates with fix radius 3. Also, the camera always looks toward the origin, which is also the center of the volume. In typical sphere coordinate notation **viewRot.x** is equal to $\phi$ and **viewRot.y** corresponds to $\theta$. Each is in range $[0, 2\pi)$ and has 16 values. Because in sphere coordinates $\theta$ has only an extend of $\pi$, each camera position is contained twice but 32 times at the poles. Only difference between these rotations is the implicit rotation around the cameras view direction.

**execTime** The last column with varying value is the output column. For each distinct configuration there are five entries. They were measured as batches of fife samples and partly contain startup-displacement and outliers.

**Corrections for Uniform Sampling** Figure 5.1 gives further insights about columns with non-uniform value distributions. While enumerated values like device and volume are uniformly distributed, volume size, which depends on the volume, is not. There are far more volumes with a side length of 1024. Our sampler chooses volumes by resolution and afterwards by name. As a result, all volume sizes are equally probable to be used, but there is much more and possible disagreeing data for high volume resolutions. Results for these large volumes is on one hand more reliable, because it has more training data. On other hand this locally larger training set may indicate a higher error in evaluation. Along with this, there is another irregularity, that is not visible in Table 5.1 or Figure 5.1. Values for tffId seem to be uniformly distributed, but they are not. It is the only parameter that breaks the uniform sampling grid, that was used in the training set, because tffId is always 0 but

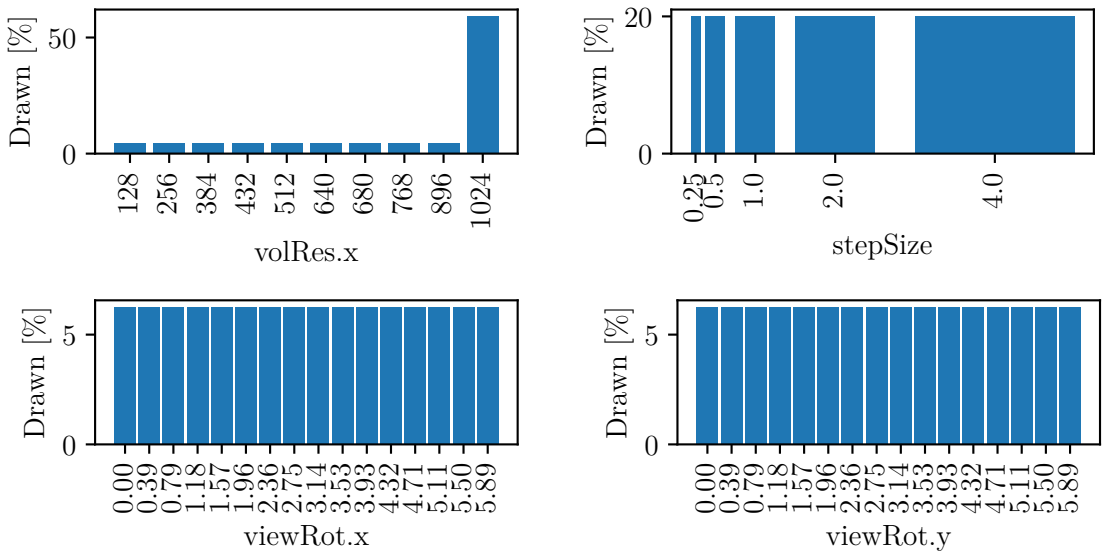


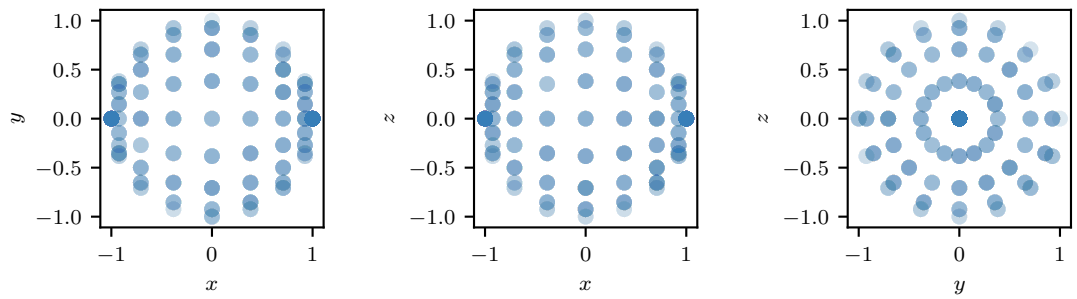

**Figure 5.1:** Non-uniform value distribution in the raycaster dataset. Bar widths correspond with our sample weight. All volRes.* are equal but sampled far more often for resolution 1024. For stepSize, all values are equally often sampled, but uneven distributed by value. Non-uniform distribution for viewRot.* is implicit, because sphere coordinates are non-uniform.

for volumes with a resolution of 1024. Nevertheless, we included this value in learning, even if it has no meaning for smaller volumes.

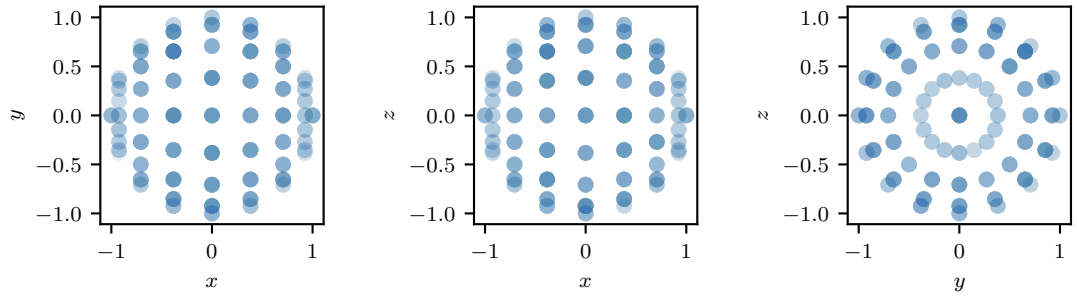
For the camera position we implemented a special latitude column, because sampling angles uniformly leads to a higher point density at poles than at the equator, as shown in Figure 5.2. We normalized point probabilities by weighting each by its nearest neighbor region. These regions can be defined as rectangles in angle space. This space can be divided in columns defined by angle $\phi$ and rows defined by angle $\theta$. All columns have same size what infers $\phi$ to be not relevant for normalization. As a result, it can be sampled uniformly. On other side, $\theta$ is relevant. Viewing rows in three-dimensional space shows that they are spherical segments. Their area is calculated by $A = 2\pi Rh$ with $R$ the spheres radius and $h$ the height of the spherical segment measured parallel to the axis between both poles. $R$ scales all spherical segments equally and therefore can be dropped. Height $h$ is calculated by $h = |cos(\theta_1) - cos(\theta_2)|$ with $\theta_i$ laying in the middle of two existing theta values that are neighbors. Our result is an array of weights that is used for sampling. In Figure 5.2, there is a visual proof, that weighting the latitude by their nearest neighbor region results in uniform sampling in three-dimensional space.

Parameter stepSize is not uniform in a different way. Here, values are uneven distributed and therefore we need to compensate it. Analogous to latitude weighting for uniform camera position sampling, we weight each value by its nearest neighbor region. Value $v_i$ gets

$$\text{weight}(v_i) = \frac{v_i - v_{i-1}}{2} + \frac{v_{i+1} - v_i}{2}.$$




**(a)** Sampling by index samples more often at poles




**(b)** Sampling with weighted latitude samples the equator more often

**Figure 5.2:** Camera position sampling. If samples are drawn uniformly in angular space (a) poles are far more often sampled than with corrected distribution (b).

Because, stepSize consists of powers of 2, we can simplify this equation to

$$\text{weight}(v_i) = \frac{v_i - v_i/2 + v_i \cdot 2 - v_i}{2} = \frac{2 \cdot v_i - v_i/2}{2} = \frac{3}{4} v_i \sim v_i.$$

As a result, sampling stepSize proportional to its own value is sufficient.

**Extension of the Dataset** To extend these input data we added generators that transform input columns. We assumed, that the amount of pixels is more relevant than the image measurement. Also, neural networks are not able to square input values and therefore it needs to be done before usage. To handle this we added following generators:

**multiply** Gets two or more columns and simply multiplies them. Simple scaling with scalars should not be done with this generator, because we have a better alternative for this. Used to combine volRes.* to voxel count and imgRes.* to pixel count.

**enum2vec** Converts an enumeration into a one-hot vector. For each unique entry, an own column is created, that always contains 0 but 1 for entries where the enumeration value equals the column name. As drawback, a neural network must be recreated if the enumeration changes, loosing all learned weights. To use this generator in a network, a special column name is defined as visible in Listing 4.1, line 15 and

Listing 4.2 line 3. This name is replaced by all enumeration values. Used to convert device names and volume names.

**sphere2cart** Converts sphere coordinates into Cartesian coordinates. Sphere coordinates contain ambiguities, like all periodic functions. Converting them into Cartesian coordinates removes the need to learn the jump between 0 and $2\pi$ in the dataset. However, this comes on cost of loosing the cameras view direction. For the given dataset, where the camera is always a fix distance from the volume, we expect no side effects. However, if the radius is variated or camera position is decoupled from its rotation, these data should be also used as input.

**Scaling of Values** We used the sigmoid function visible in Figure 2.2c for activation between neural network layers. It has only a small range, where it grows almost linearly. In this range, we can expect a good learning rate, but for values larger than 6, the sigmoid function is almost one and its gradient almost zero. This region is often called saturated, because even large changes do not change the outcome much. The vanishing gradient causes slow learning and to avoid this, we divide volRes.* and imgRes.* by 1024. The columns viewRot.* and stepSize were not scaled, because viewRot.* is converted to Cartesian coordinates and implicitly normalized into range [-1, 1]. Parameter stepSize is an exponential growing value, where most values are lower 3.

### 5.1.2 Indirect Volume Visualization

In this section we discuss our volume renderer and show performance measurements. A detailed description about our implementation can be read in Chapter 3. We used marching cubes and an octree structure to create and organize the mesh. Each leaf of the octree contains multiple triangles. We first iterate the octree to determine a sort order of leaves and afterwards triangles are sorted to create a sorted index buffer for drawing. Using the OpenGL pipeline we draw triangles into a frame buffer object (FBO) using simple shaders. Our pipeline for sorting and drawing has following steps, whereby each is executed on GPU and managed by CPU.

**Initialization** Clears `Sorted Index Buffer` with invalid index. Resets counters.

**Octree Startup** *Single-threaded.* Iterates over the upper part of the octree. Writes octree nodes into `Subtree Buffer` using an atomic index counter. Reserves ranges in the `Sorted Index Buffer` for each subtree.

**Octree Continuation** *Multi-threaded.* One thread per entry in `Subtree Buffer`. Iterates until all leaves are handled or clipped. Writes into a `Leave Access Buffer` using an atomic index counter.

**Leaf Sort** Starts one thread per entry in `Leave Access Buffer`. Sorts triangles and writes their indices to the `Sorted Index Buffer`.

| Name | Values | Minimum | Maximum | Mean | Median |
|---|---|---|---|---|---|
| Initialization | 13893 | 0.00 | 3.16 | 0.22 | 0.08 |
| Octree Startup | 258573 | 0.40 | 249.36 | 19.72 | 7.98 |
| Octree Continuation | 178123 | 0.31 | 193.57 | 5.05 | 2.09 |
| Leaf Sort | 211756 | 0.00 | 1115.62 | 19.75 | 3.08 |
| FBO Drawing | 225154 | 0.03 | 654.56 | 16.62 | 4.87 |
| Screen Drawing | 69693 | 0.11 | 123.18 | 0.89 | 0.26 |
| execTime | 335860 | 1.33 | 1620.74 | 62.25 | 27.76 |

**Table 5.2:** Render Times of our Renderer. All times in milliseconds.

**FBO drawing** An instantiated draw call is used to draw all triangles. Triangle attributes like position are read from read-only buffers using `Sorted Index Buffer`.

**Screen drawing** The content of the FBO is drawn to screen.

The analysis of the render pipeline is based on 360 000 samples that are also used for evaluation of our learner. We used only samples that were drawn by the passive sampling strategy, because these samples are expected to be drawn uniformly and independently from each other. Rendering needed 32.5 hours of rendering without time for setup.

We measured each step of our pipeline independently by GPU time using OpenGL query commands. Results are presented in Table 5.2. Column Values shows the amount of unique values.

In the following we will analyze the main parts of our pipeline which are octree iteration and leaf sorting. Initialization of the pipeline only contained clearing of counters and the sorted index buffer. Screen drawing was a simple copy-shader to show the FBOs content on screen. Both are not important, because of their far smaller complexity. There is no ability to improve these parts but dropping the screen drawing. FBO drawing cannot be improved much without changes in octree iteration and leaf sorting. Its task is visualizing triangles under the assumption, that they are already sorted. Its performance therefore depends on the amount of triangles in the sorted index buffer and its fragment shader complexity. These parts should be free to be modified by later users and therefore take no part in our analysis.

**Analysis of Octree Iteration** Both octree iteration do basically the same. Their only difference is, that octree startup is single threaded and will write a buffer of subtree entries. It starts always at the root of the octree. Octree continuation instead starts a thread at each of these subtree entries and iterates until all leaves are handled or cropped by clipping. No parameters differ between both but tree split has a large influence. This influence can be seen in Table 5.3. Increasing tree split increases octree startup time and decreases octree continuation. Octree size increases exponentially with depth and so duration of the octree

| Tree Split | Min. | Max. | Mean | Median | Min. | Max. | Mean | Median |
|---|---|---|---|---|---|---|---|---|
| 3 | 0.40 | 30.30 | 2.63 | 2.51 | 0.43 | 193.57 | 8.18 | 4.36 |
| 4 | 0.49 | 34.01 | 11.03 | 9.16 | 0.33 | 176.75 | 4.17 | 2.05 |
| 5 | 0.54 | 249.36 | 45.42 | 29.75 | 0.31 | 109.81 | 2.81 | 1.34 |
| | **(a)** Octree Startup | | | | **(b)** Octree Continuation | | | |

**Table 5.3:** Octree Split Analysis. All times in milliseconds.

iteation. However, octree startup times increase much faster than octree continuation times decrease. This is caused by thread count.

A closer analysis, visualized in Figure 5.3 shows a more detailed image. The sum of each coordinates (or Manhatten distance to the origin) is the complete octree iteration time. Therefore, an octree split of 5 heavily increases this time, as visible in the upper left subgraph. In the further we refer to each different colored sample set in the upper left graph as cluster. With increasing tree split, distributions within a cluster become less high but wider. As a consequence, increasing the split value has minimally advantage in the octree continuation but a huge disadvantage for octree startup. Both lower subgraphs show the intern structure of each cluster, that is also scaled with tree split. All clusters are structured similarly, but partly hidden behind others. The upper right subgraph shows, that subtrees have a large influence on the render time, especially when tree split is large. This is because the possible amount of subtrees grow exponentially with tree split. Octree startup can also reach leaves but has no access to the leaf buffer. Instead, it stores leaves as subtrees which are used as entry points for continuation. We decided to handle leaves this way to simplify the renderers structure. The octree startup therefore iterates through the whole octree, if the tree split value is too high. It completely removes the advantage of our intended multithreaded octree sorting.

Another interesting aspect, visible in Figure 5.3, are lines in each subgraph but they are differently colored for triangle count visible in the lower right plot. The amount of triangles is connected the slope of each line together with tree split, but a high amount of triangles does not necessarily cause a high iteration time. Instead, a larger grid resolution leads to a higher amount of triangles. We assume, that visible effects are caused by the number of iso-layers. If the number increases, more triangles exist, but also more triangles are allowed in a single leaf. Therefore, this parameter can decrease the amount of leaves and shrink the octree. The upper right subgraph also shows that the amount of subtrees influences the distance of each point in a line too the origin. There are also lines with a high amount of triangles at the end, that are shorter than others with less triangles. We assume, that this also depends on the octrees density, because it was not balanced.

**Analysis of Leaf Sort** As visible in Table 5.2, leaf sort has the largest maximal time. During the octree iteration, leaves can be dropped by view frustrum culling. Each visible leaf contains a limited number of triangles and is sorted using batchers odd-even merge

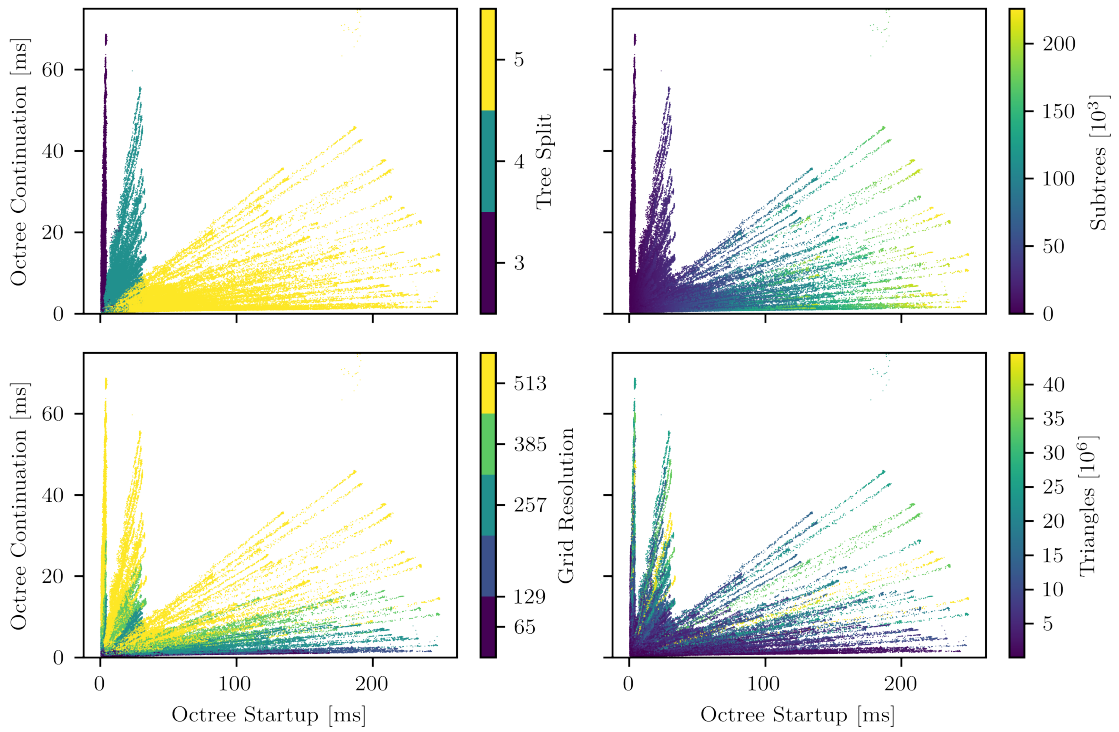



**Figure 5.3:** Visual Octree Split Analysis. All plot share data and scale. Parameter tree split divides samples into three clear clusters. They share their internal structure as visible in both lower graphs. There are lines, dependent on triangles. Subtree count depends on octree startup time.

sort. We chose this sorting algorithm, because it is well suited to be executed by a graphics card. Two parameters mainly influence execution time of this shader. First is the amount of triangles, that need to be sorted which is five times the iso layer count. Batchers odd-even merge sort has an execution time of $\mathcal{O}(n \log^2(n)))$. In our case $n$ is the number of triangles per leaf and as a result, adding a new iso-layer increases sorting time. Another factor is the amount of leaves that need to be sorted. For each leaf, a thread is started and our intention is, that if the graphics card can execute its maximal number of threads at the same time and all will finish at the same time.

In Figure 5.4 it is visible, how leaf sort depends on iso-layer count. This parameter also influences triangle count visualized as color. Leaf size first increases, when adding new layers. This may be volume dependent, as well as the further decrease. If more iso-layers are used, each node is allowed to contain more triangles and therefore can be larger in regions with sparse triangle placement. From the graph is is visible, that increasing amount of iso-layers slows down sorting. Also, execution time seems to grow with the squared amount of leaves. The amount of triangles seems also to be important. There is no early stopping implemented but when the shader is called without valid parameters.

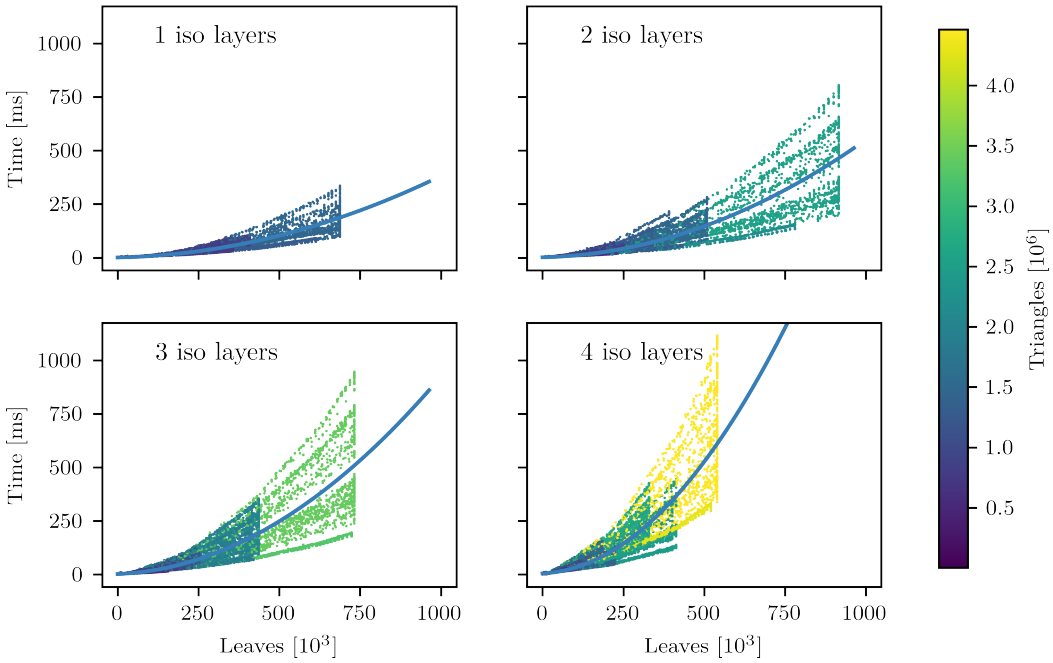



**Figure 5.4:** Visual Leaf Sort Analysis with fitted polynom of degree 2. Visible lines are caused by different volumes. All subgraphs share scale.

We therefore assume, that this raise in execution time is caused by internal graphics card optimizations and calls. Longer execution times seem to be very difficult in handling.

Because the leaf sorting algorithm is very simple, further optimization is difficult. To speed up execution, the overall amount of leaves need to be minimized by further optimizations during octree iteration. As an example, on can add further clipping, e.g. by estimated occlusion, or level of detail approaches.

**Analysis of Sample Selection** In this section we look at our sampling method and if its uniform as intended. In Table 5.4 we show parameters of the renderer that where sampled. Device is the an exception, because our tests were executed independently on different machines. Therefore, this column can differ in sample count and is not expected to be uniformly. The distribution of all parameters is visualized in Figure 5.5 and show several exceptions. There are a few, that are visible but not important. Both exceptions in gridRes and iso-layer count are caused by extension of these parameters during learning. We did not restart from scratch, because of the long training times.

During evaluation we found, that our cam_dist parameter was not correctly sampled, as visible in Figure 5.5. This resulted in slightly oversampling in the volumes center at distance 0. Sphere coordinates are more dense at poles and at the origin, which in our case

| Name | Values | Minimum | Maximum | Mean | Median |
|---|---|---|---|---|---|
| device | 4 | GeForce GTX 1070 | Quadro M6000 24GB | | |
| volume | 7 | freq_1.dat | freq_8.dat | | |
| cam_dist | 360238 | -1.00 | 1.00 | -0.00 | -0.02 |
| cam_phi | 360238 | 0.00 | 3.14 | 1.57 | 1.57 |
| cam_theta | 360238 | 0.00 | 6.28 | 3.14 | 3.14 |
| gridRes | 5 | 65 | 513 | 259.54 | 257.00 |
| imgRes | 2048 | 1 | 2048 | 1023.64 | 1023.00 |
| iso_layer_count | 4 | 1 | 4 | 2.44 | 2.00 |
| tree_split | 3 | 3 | 5 | 4.00 | 4.00 |

**Table 5.4:** Value Ranges for Sampling with Renderer

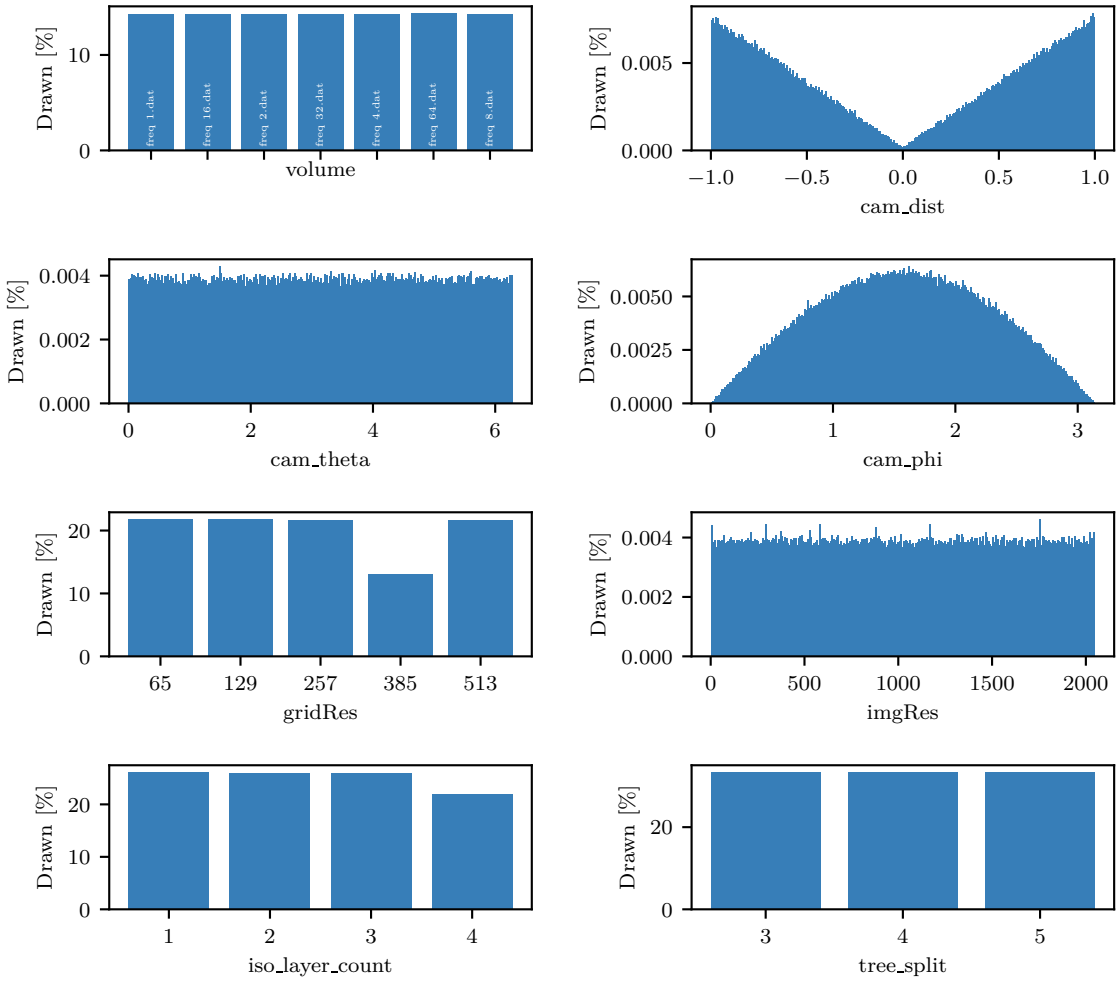



**Figure 5.5:** Value distribution in raycaster dataset. Ranges of gridRes and iso-layer count were extended during learning. Camera parameters cam_dist and cam_phi are corrected to enable uniform camera position sampling.

corresponds with the volumes center. The density decreases to the power of three with the distance. We compensated it with the power of two, which improved sampling but not corrected it.

**Extending of Samples** We expected, that parameters we chose for sampling were not sufficient for learning. Similar generators like for the raytracer dataset are applied. Parameters like volume name cannot be expected to contain any relevant information. However, after processing the volume there are far more interesting and relevant attributes of generated meshes accessible. These attributes stay the same even if the mesh is recreated. Therefore, we extended samples to improve learning by values listed in Table 5.5.

These extensions were applied at sampling, enabling the learner to use them for sample selection. Values for cached columns were read during initialization. As side effect, all cached volumes were loaded into RAM.

**Influence of Single Parameters on Execution Time** We use all prior described parameters to determine how parameters affect overall execution time. Therefore, we first have a closer look, how the execution time is split by each pipeline part and afterwards, how single parameters behave in comparison to execution time.

In Table 5.2 we showed a small set of statistics about our render time. For a more detailed interpretation we plotted all of our samples into scatter plots in Figure 5.6 against execution time. The initialization does not have a large influence. A rescaled version looks equal to Figure 5.7b. This is mainly because the sorted index buffer that is cleared during initialization is sized to hold an index for each triangle. Screen Drawing has many outliers between 50 and 100 milliseconds, that are most probable caused by the operating system callbacks. However, most often its duration is short. FBO Drawing has also a broad linearly increasing influence, which most probably corresponds with the amount of fragments to draw. There is also a transformed version of plots from Figure 5.7 visible, because triangles and vertices are accessed in this step.

The more interesting parts are again our octree iteration and leaf sort. Starting with the octree startup there is a visible similarity with our visual octree split analysis in Figure 5.3. We can also assume, that the structure is similar influenced by the tree split parameter. Also, same clusters are visible.

The octree continuation has a different structure. It is impossible to decide, which parts belong to a certain tree split value. Our octrees are not balanced and as a result, for each split parameter there are large and small subtrees together. This stage of the pipeline has a broad and almost linear influence on the overall execution time. However, its influence is rater small.

The plot for leaf sorting shows, that leaf sorting has an almost linear influence on the execution time. There is a second slightly higher branch, raising faster for shorter times and then aligns with the linear part. This may be influenced by our sorting algorithm. Batchers

| Name | Values | Minimum | Maximum | Mean | Median |
|---|---|---|---|---|---|
| *Depending on cam_dist, cam_phi, cam_theta* | | | | | |
| camPos.x | 360238 | -2.99 | 3.00 | 0.00 | 0.01 |
| camPos.y | 360238 | -3.00 | 3.00 | 0.00 | 0.00 |
| camPos.z | 360238 | -3.00 | 3.00 | 0.00 | 0.00 |
| *Depending on gridRes* | | | | | |
| gridPts | 5 | 274625 | 135005697 | 40945660.52 | 16974593.00 |
| *Depending on imgRes* | | | | | |
| pixels | 2048 | 1 | 4194304 | 1397440.16 | 1046529.00 |

**(a)** Generated Columns

| Name | Values | Minimum | Maximum | Mean | Median |
|---|---|---|---|---|---|
| *Depending on volume, iso_layer_count, tree_split* | | | | | |
| subtrees | 62819 | 0 | 225871 | 17409.49 | 6234.00 |
| *Depending on volume, iso_layer_count* | | | | | |
| leaves | 276 | 0.00 | 9161250.00 | 676961.64 | 236858.50 |
| octree_size | 276 | 15252 | 11079682 | 1145345.27 | 408953.00 |
| triangles | 276 | 26612 | 44648276 | 3893767.02 | 1404252.00 |
| vertices | 276 | 13308 | 22323102 | 1946725.05 | 702132.00 |
| *Depending on volume* | | | | | |
| volRes | 1 | 512 | 512 | 512.00 | 512.00 |
| voxels | 1 | 134217728 | 134217728 | 134217728.00 | 134217728.00 |

**(b)** Cached Columns

**Table 5.5:** Value Ranges of Sample Extension

odd-even merge sort has an execution time of $\mathcal{O}(n \log^2(n))$ which could correspond with the upper bound. In our algorithm this render time is multiplied with the number of octree leaves that are sorted, which can be a cause of the linearity.

In the following we discuss the influence of single parameters and if they should be used in future learning. Learning is complicated and wrong or unnecessary data slows it down.

We found, that there are too many parameters describing memory usage on the GPU. We listed them all in Figure 5.7 to show that clear coherences exist. Other values are very equal and we assume that triangle count itself would be sufficient, because it is used in initialization and FBO drawing, influences octree structure and vertex count depends on it.

In Figure 5.8 there are our sampled sphere coordinates and their conversion into Cartesian coordinates. We chose to sample in sphere coordinates for two reasons. First, this was also

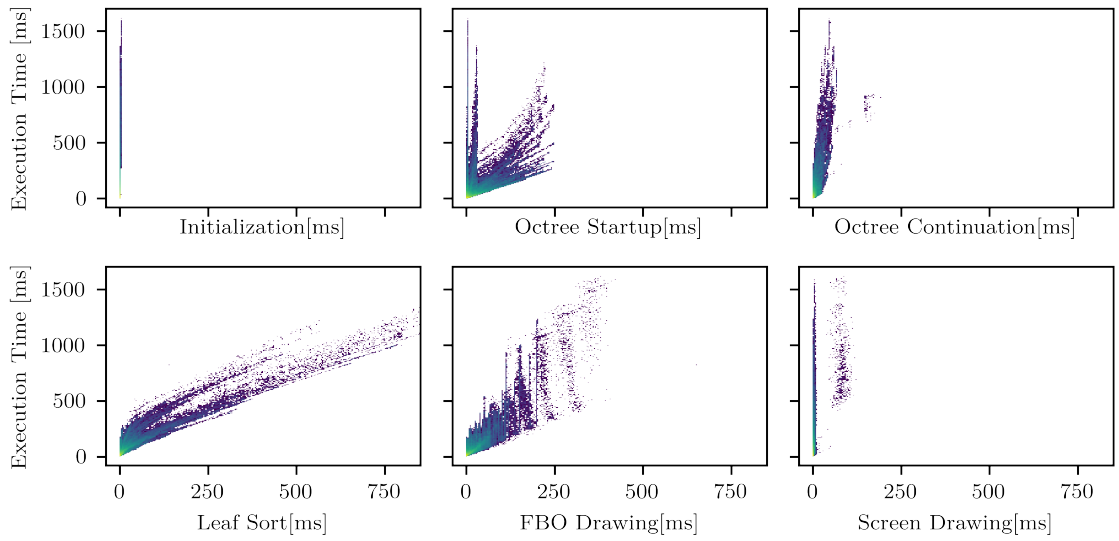
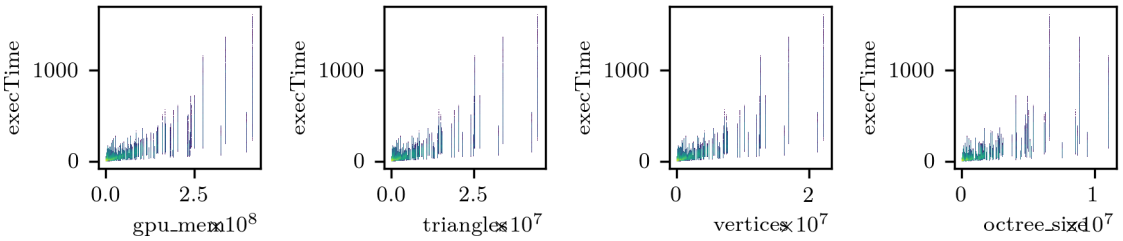


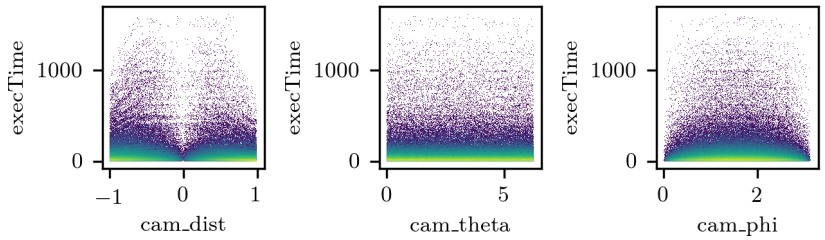

**Figure 5.6:** Influence of each part on the render pipeline. All plots share axes and drawn samples. Brighter colors refer to higher densities. Octree startup shows strong correspondence to Figure 5.3, while octree continuation, leaf sort and FBO drawing grow almost linear.



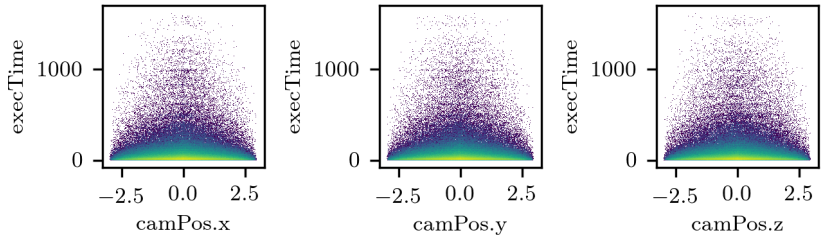

**(a)** Used GPU Memory **(b)** Triangle Count **(c)** Vertex Count **(d)** Octree Size

**Figure 5.7:** Influence of Memory and Large Buffers shows strong coherences.

used in the given raycaster dataset and therefore enables a better comparing. Next, it is simpler to sample on a sphere surface when using sphere coordinates. From angular values in Figure 5.8a we cannot determine anything but, that our sampling influences them. The longitude is sampled uniformly and the latitude is compensated at the poles. Here, point densities only show, that there is a low influence on execution time. The distance instead has interesting properties. Low values around the center, where the camera distance is 0 are caused by our sampling, that was less often sampled in this region to compensate the sphere density by radius. The sign of the distance encodes, if the camera is looking toward or away from the volumes center. As visible, a negative distance increases render time. In this case the volumes center is visible and therefore more triangles need to be drawn. The conversion into Cartesian coordinates shows a different image. Each of the coordinates has same influence as visible in Figure 5.8b. The highest density is in the center as expected




**(a)** Sphere Coordinates (Distance, theta, phi)




**(b)** Cartesian Coordinates

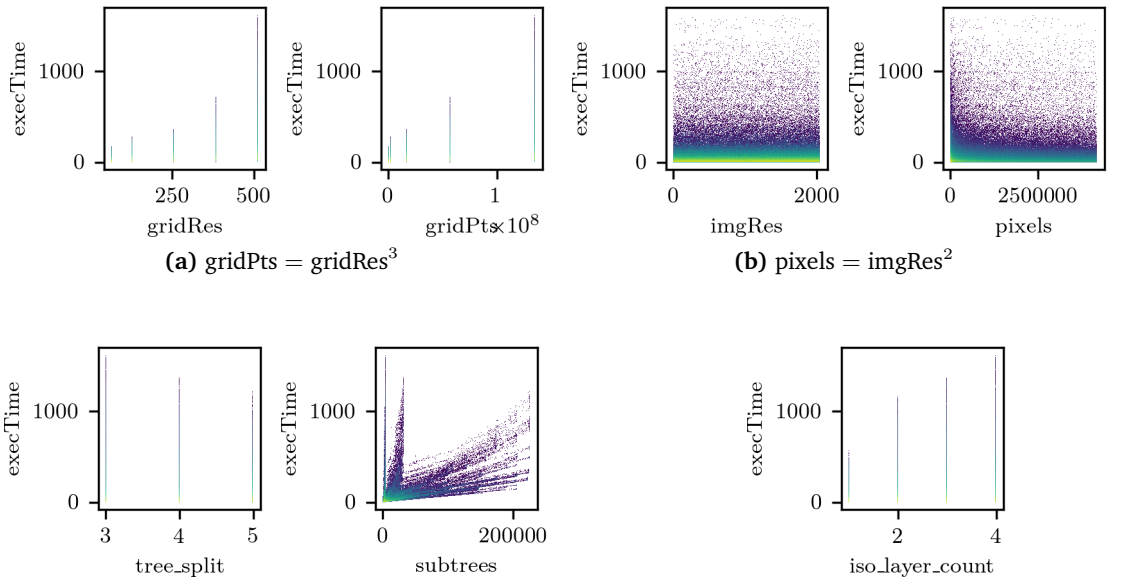
**Figure 5.8:** Influence of Camera Position

and here are also the highest render times. If the camera is near the center, the volume will fill the whole screen and therefore more pixels need to be drawn. However, there is no significant raise or fall of execution time visible and therefore we can state, that camera position has almost no effect.

There are several different parameters describing resolutions. We left the volumes resolution aside because there is only one value as shown in Table 5.5. A static value does not carry information and therefore should not be used for learning. However, our grid resolution carries real information about triangle mesh resolution and could be important. Our plots in Figure 5.9a show, that there is a clearly visible influence on execution time. Sampled values form clear lines, because there where only a few grid resolutions. All of these lines are more dense at the bottom. For grid points itself, there is an almost linear influence visible, using the upper ends of each line. It infers also, that the influence of grid resolution is to the power of three. This proofs that our extension of parameters was useful in this case.

For image resolution, visible in Figure 5.9b, there is another picture. Resolution is far more dense all have same influence on execution time. Even a high resolution does not result in a visible change. The pixel count shows higher values for small numbers of pixels, but this is more probable caused by sample density and not by influence. Pixel count is the square of image resolution and therefore there are more small values than large ones.

We already discussed the effects of tree split in Section 5.1.2. We found that a tree split value of four was optimal when comparing both octree iteration steps. Plotting this parameter against execution time in Figure 5.9c shows that increasing tree split decreases the overall render time. Therefore, setting tree split to five is better in this terms. Plotting subtrees




**(a)** gridPts = gridRes$^3$ **(b)** pixels = imgRes$^2$

**(c)** Tree split influences subtree count **(d)** Iso-layer count influences mesh attributes

**Figure 5.9:** Influence of other parameters on execution time

against execution time also supports this, assuming a similar cluster location like shown in Figure 5.3.

At last, iso-layer count also increases render time as shown in Figure 5.9d. It has influence on triangle count and this way also affects vertex count, octree size and initialization. Sorting is also affected, because this parameter increases the allowed amount of triangles per cell and therefore makes leaf sorting more complicated.

## 5.2 Performance Prediction

In this section we train and evaluate our model on prior analyzed datasets. We start using tabular data from the raycaster dataset and afterwards compare these results to performance with our own indirect volume visualization. We had several different configurations, containing two loss functions, three sampling strategies and several network configurations.

### 5.2.1 Raycaster Dataset

We run two different tests on the dataset. In our first test run we focused general questions like learning possibility and data selection. Therefore we build three different network configurations, whereby one was trained for a specific graphics card and two for all available

graphics cards at once, whereby only one of these knows the graphics card. We run all tests quasi-parallel using a train scheduler that runs our learner with different configurations.

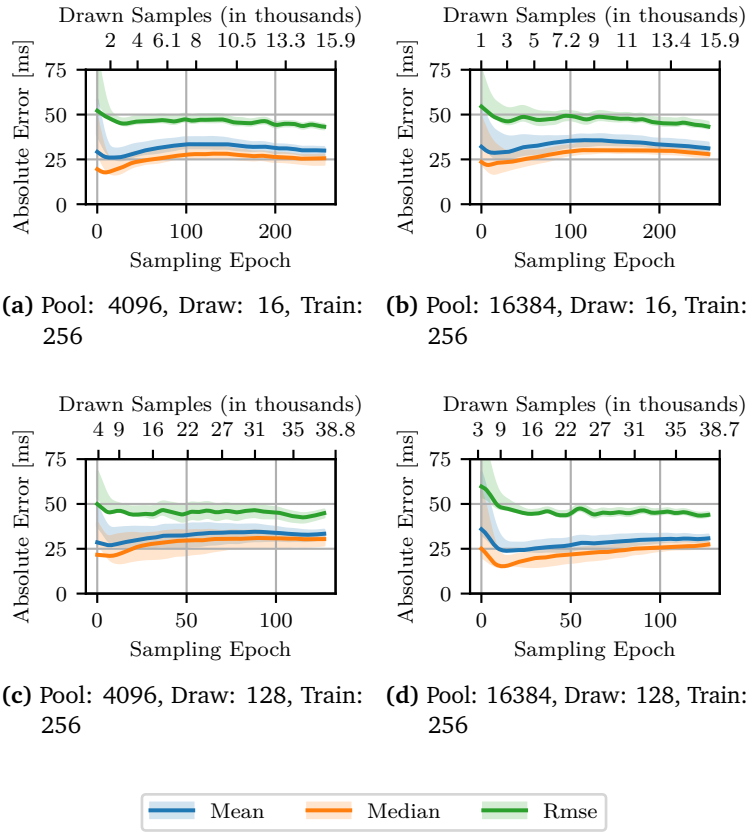
Evaluation is made at the end of a sampling epoch after each submodel trained on its training set. Therefore, 2048 samples are drawn at the eand of a sampling epoch after each submodel was trained. We collect statistics about error and ambiguity. All samples, that where never drawn or rejected by the sampler form our evaluation set.

**Choosing Parameters for Neural Networks** Choosing hyperparameters for neural networks is not an easy task. As described in Section 2.3, neural networks regress our value mapping by fitting multiple step functions to the input and prior network layers. Therefore, its complexity needs to be high enough. However, a too complex network may overfit its training data and costs more training time. Therefore, explore the possible configuration space to determine, how hyperparameters affect learning.

Our starting configuration consists of a network with two hidden layers with each a size of 32 perceptrons. As input we had three possible configurations. The “all-in-one network”, shown in Listing 4.2, shows the largest configuration and the only one, that has 64 perceptrons in the first hidden layer. Reason is the far larger input, because the GPU name is fed as one-hot vector with ten entries. Our other configuration are equal but without the graphics card name. They consist of a “general network” that should predict performance for all graphics cards at once without knowing them and individual networks, that are trained for exactly one graphics card.

Results showed, that training for multiple graphics cards at once has several disadvantages. Our “general network” performed worse than all others caused by contradictory values in the dataset. Our “all-in-one network” performed better, but not as good as any individual network. Additionally, the “all-in-one network” needs to be retrained on every change in the set of used graphics cards. Individual predictors are more practically to train, because even if a computer has multiple GPUs mounted, OpenGL does not offer an opportunity to select a specific one.

Afterwards, we started to switch hyper-parameters of the learner itself. Two parameters were modified: pool size and draw size. The pool defines, how many samples are drawn and weighted by ambiguity. For active sample search it defines an upper bound for sampling to enable a timeout. Draw size defines, how many new samples are taken by each submodel. Their influence can best be seen for active pool sampling and therefore, we show results of several short tests in Figure 5.10 only for this strategy. Remark, that the upper row ((a) and (b)) contains more epochs, because tests where made with an external timeout. Our results show that increasing draw size leads to an enormous increase of the training set but no improvement in learning. Pool size does not seems to have much effect, but slightly increases the error in Figure 5.10b and decreases it also slightly in Figure 5.10d. There is a possible explanation for this behavior: All networks in Figure 5.10 where trained without our gamma correction of ambiguity values. We suspect, that a larger pool size increased the probability that any bad sample is drawn. In this case bad samples are well known and therefore do not have an advantage for learning. If draw size is also increased,




**(a)** Pool: 4096, Draw: 16, Train: 256

**(b)** Pool: 16384, Draw: 16, Train: 256

**(c)** Pool: 4096, Draw: 128, Train: 256

**(d)** Pool: 16384, Draw: 128, Train: 256

**Figure 5.10:** Short parameter tests for the GeForce GTX 1080 with two hidden layers of 32 neurons and sample strategy active.

more unknown samples are drawn again and learning is speed up. Decreasing the pool size has two disadvantages: more sampling epochs are needed, because our samples are more sparse in the input space. Also, good samples are more likely to be drawn by multiple submodels, what speeds up decreasing of ambiguity for single values. If these values are outliers but drawn from too many submodels, their can distort the approximation.

Another interesting aspect of all graphs in Figure 5.10 is the rise of all errors after a good start. This effect might come from a local minimum of the performance prediction, that is found very fast but new samples lead to another one. Both upper graphs show, that error will decrease after a while.

From these insights about hyperparameters we determine, that the ambiguity value needs to be modified for large pool sizes. A size of 16384 enables a finer search in the input space in reliable time. However, bad samples are more likely to replace good ones. To handle this issue, we introduced our $\gamma$-correction, explained in Section 4.2.2. It increases the contrast between high and low ambiguities and therefore increases the probability of high samples to be drawn. This is also another reason for a large pool size, because for too small pools $\gamma$-correction leads to smaller differences between submodel training data. We choose a $\gamma$

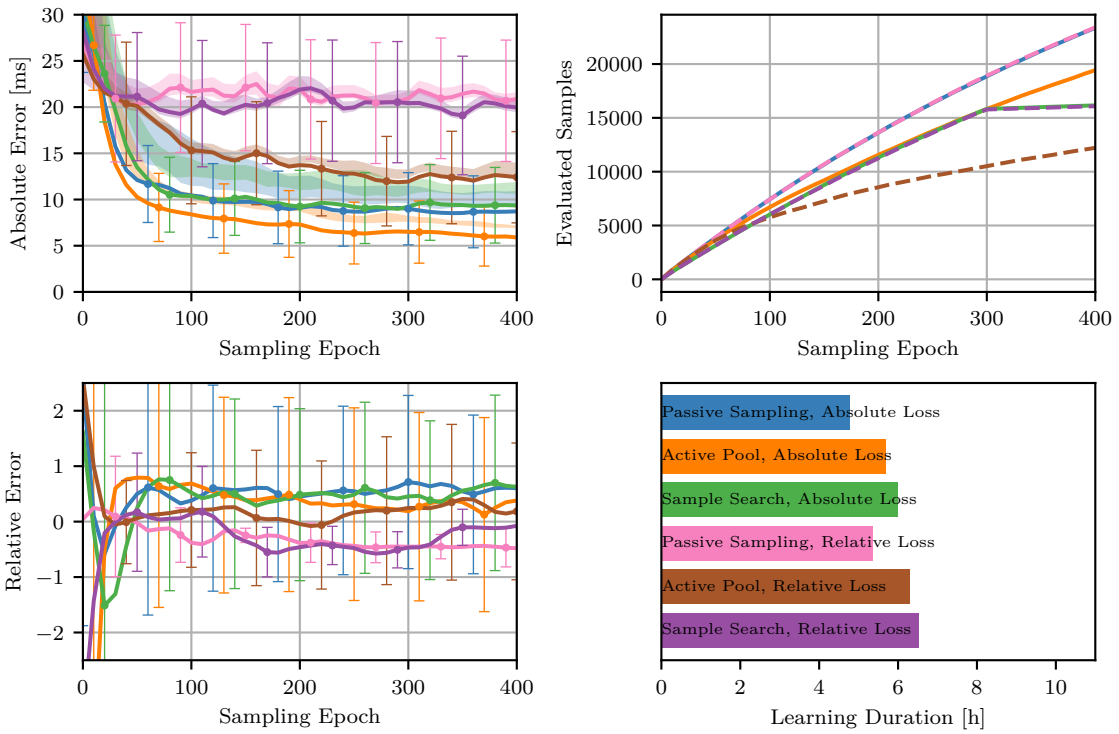



**Figure 5.11:** Mean error of Dual Layer Network. Error bars show squared standard deviation. Learning with absolute loss leads to smaller absolute errors, while active pool sampling leads to small relative error.

of 4 for our next training stage. Also, we increased the amount of learning epochs. Each submodel therefore will better fit to its own training set, what increases ambiguity in bad approximated regions. Figure 5.11 contains results of our changes. As visible, the error is far smaller than for any configuration in Figure 5.10.

**Refining the Architecture of Our Model** After choosing base parameters for our neural network, we tested five different network configurations with all sampling strategies and both loss functions. For these networks, we used prior chosen pool size, draw size, gamma and amount training epochs. Our five networks differ in their internal structure. First, we evaluated our Dual Layer Network with two hidden layers, where each has a size of 32 perceptrons. Our Single Layer Network is configured with a single hidden layer of size 64. In contrast, our Triple Layer Network gets three hidden layer of size 32. To measure the effect of submodel counts we test an Increased Submodel Configuration, that is equal to Dual Layer Network but with nine submodels. At last we use both loss functions side by side in a Dual Loss Network. It has six submodels, whereby one half trains with absolute and the other half with relative error.

Figure 5.11 shows, how our first network configuration behaves using different sampling strategies and different loss functions in training. There are several interesting aspects

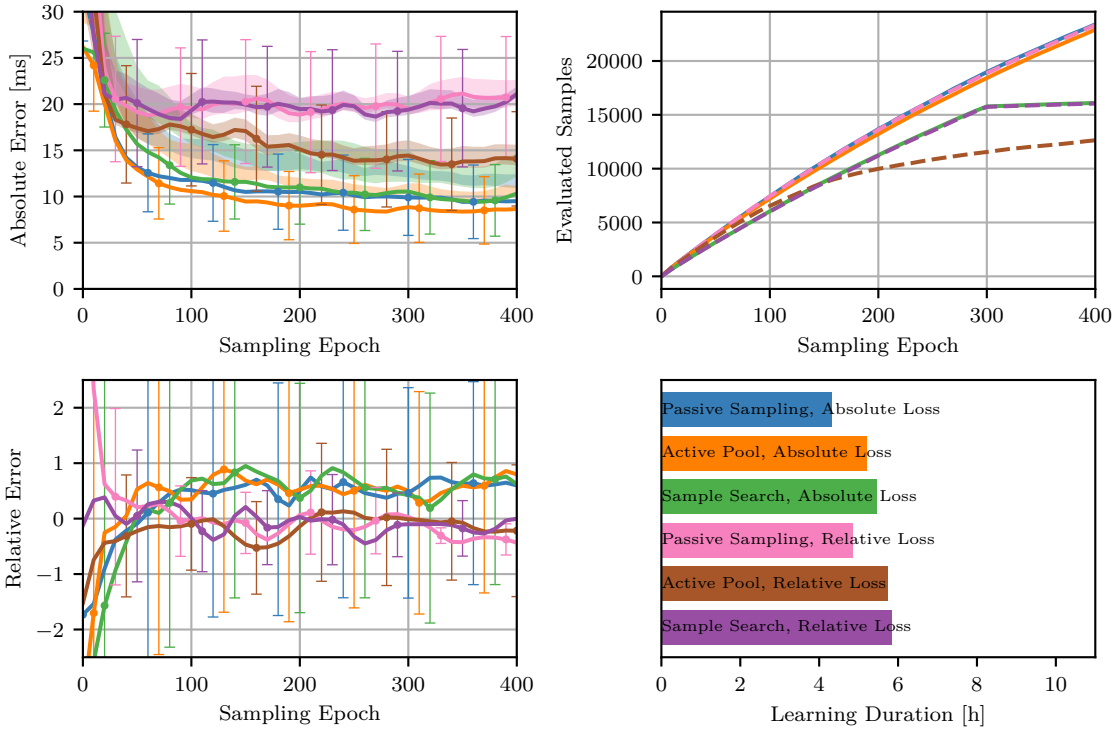



**Figure 5.12:** Mean error of Single Layer Network. Error bars show squared standard deviation. Smaller networks need less training time, but have a worse optimum for absolute error. Relative error shows clear grouping between different loss functions.

visible. First of all, passive sampling needs more samples than any other strategy. Reason lays in its simple structure, where each submodel samples independently from the pool of all available samples. As a result, training sets share almost no samples. Instead both active sample strategies do increase the probability of samples to be drawn by multiple submodels. Active pool sampling has the lowest absolute error independently for each loss function. Even for relative error it is very low and most often closer to zero than other sampling strategies with same loss. Sample search instead behaves often like passive sampling but with less samples and more training time. All graphs seem to be almost converged but for relative error, that is very fluctuating and less stable.

Graphs for all network configurations share their structure to support comparing. We use our Dual Layer Network as reference for comparing and afterwards discuss which one is the best approach. Comparing Dual Layer Network with Single Layer Network, which is visible in Figure 5.12, one can see that the amount of drawn samples is almost the same, but increased for active pool sampling with absolute loss. Absolute error is increased for pool sampling, but relative error improved. Moreover, there is clustering by loss function visible for relative error. Training and sampling needs slightly less time.

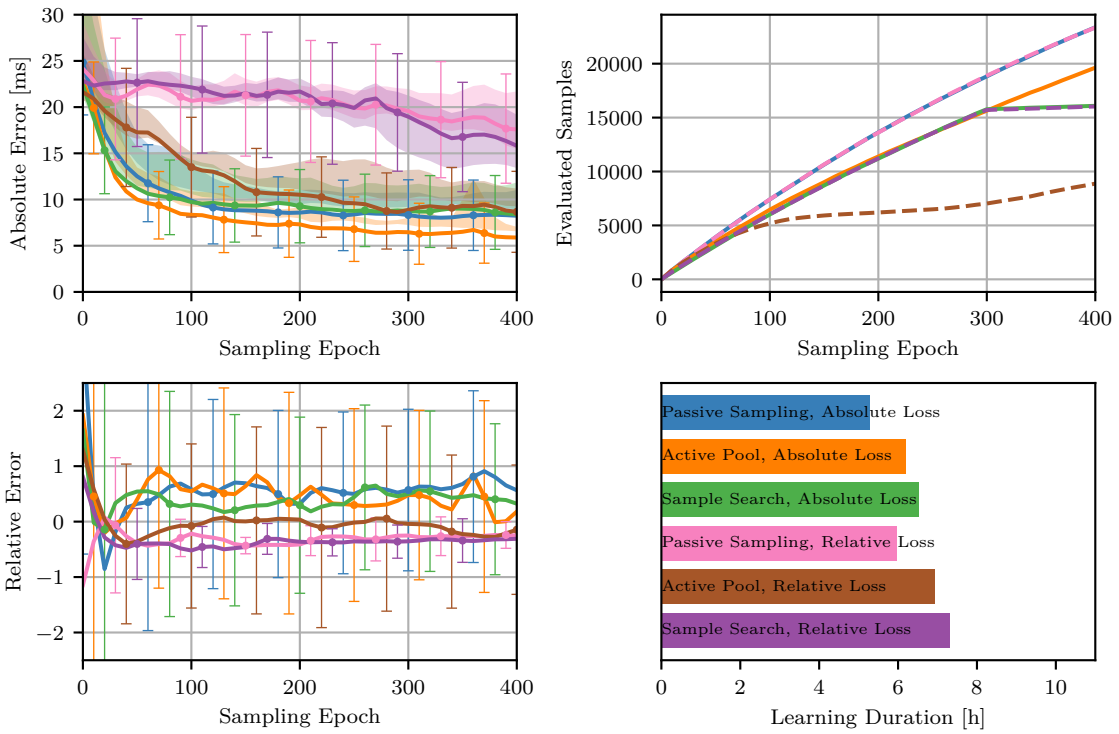



**Figure 5.13:** Mean error of Triple Layer Network. Error bars show squared standard deviation. Absolute errors are improved for all runs. Learning with absolute error is less stable for relative error.

Performance measurements of Triple Layer Network are shown in Figure 5.13. If the amount of hidden layers is increased, performance becomes better for relative loss. In combination with active pool sampling it needs far less samples than other results with comparable error. Even passive sampling and sample search leave their local minimum and improve further. Training with absolute error had not improved much.

Training with more submodels, as made for Increased Submodel Configuration in Figure 5.14, increases training time and amount of drawn samples. There is almost no improvement visible compared to our first network configuration. However, relative errors are more stable. Here, another effect of the tabular sampler is visible, especially for active pool sampling with relative loss. Its training set size increases in a bend curve and seems to converge. This is caused by samples, that are drawn multiple times. With increasing training sets, drawing an already known sample becomes more likely.

Our Dual Loss Network was created after all others and in parallel to tests of our network on real graphics cards. Reason was the long time each network needs to converge. As a result, our fifth network configuration has also three hidden layers like Triple Layer Network. It has six submodels, whereby one half of trains with absolute and the other half with relative loss. This network seems to learn slower and is not converged after 400 sampling epochs. Also, it is not as accurate for passive sampling. Training with pool sampling is

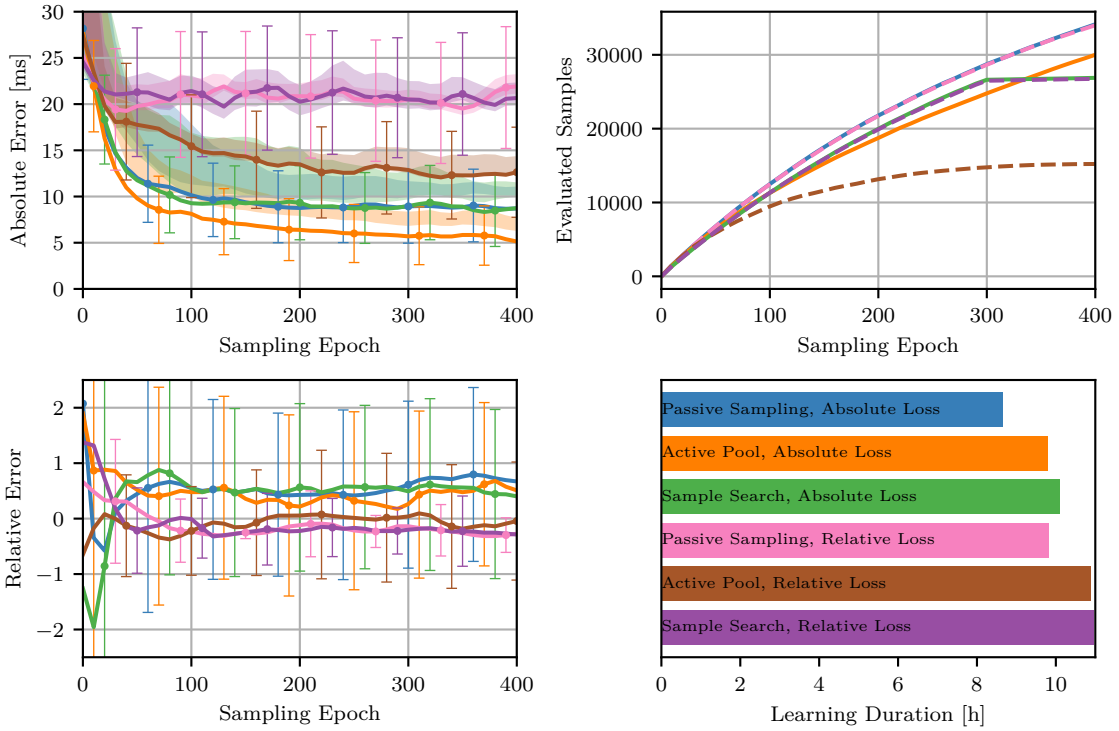



**Figure 5.14:** Mean error of Increased Submodel Configuration. Error bars show squared standard deviation. Increasing submodel count leads to smoother learning curves and slightly improves the error but costs far more evaluations and more learning time.

slightly worse than our first configuration but disagreement of submodels is far higher. The relative error instead is almost zero. Interestingly, this configuration takes far less samples when searching for good samples, while at the same time keeping disagreement between different submodels high. This especially means, that training sets differ much because otherwise all submodels would agree.

There are two aspects of sampling strategies visible in our network configuration. Active pool sampling has always the smallest error and therefore can be used, if an accurate solution is necessary. Sample search instead should be used with a combination of both loss functions, when measurements are very expensive.

To enable better insights about the error we collected pairs of predictions and ground truth using the same samples that where used to plot our error graphs. We draw one for each sampling strategy whereby each contains three different loss function configurations. Passive sampling is visualized in Figure 5.16, active pool sampling in Figure 5.17, and active sample search in Figure 5.18. A good performance prediction should converge to a diagonal, which is therefore drawn into all graphs. Also, we applied linear regression to the data to show tendencies. The area around our linear approximation shows how stable our regression is. Because there are many small values we color them by density, whereby

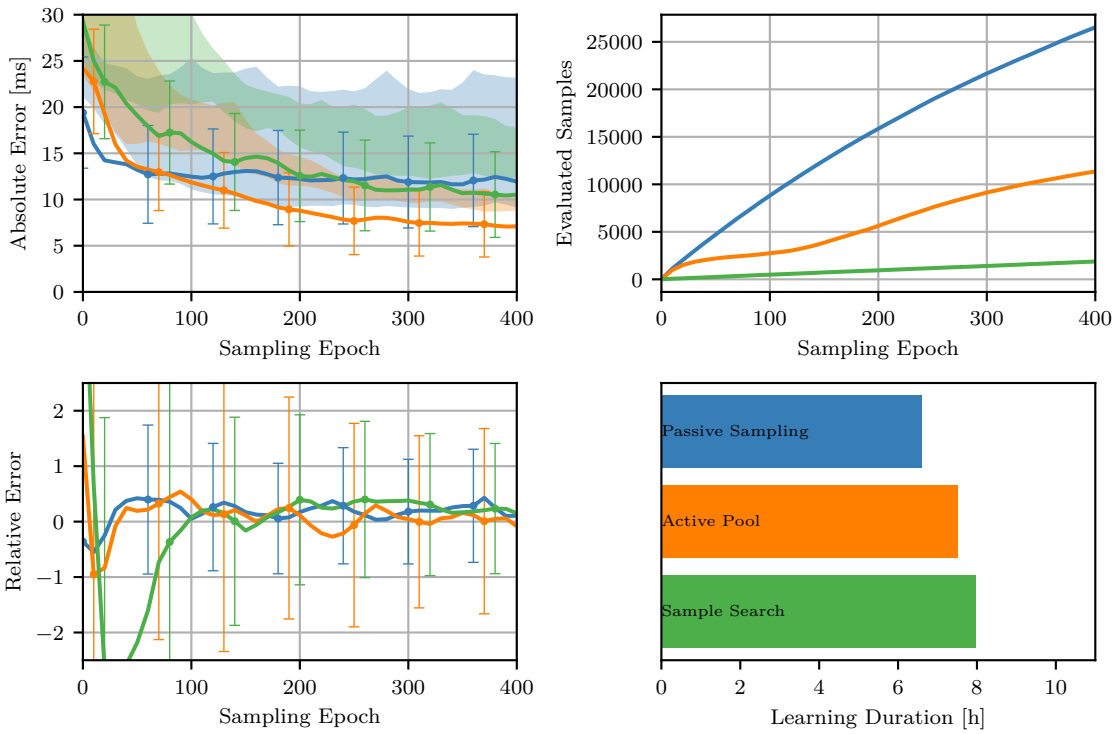
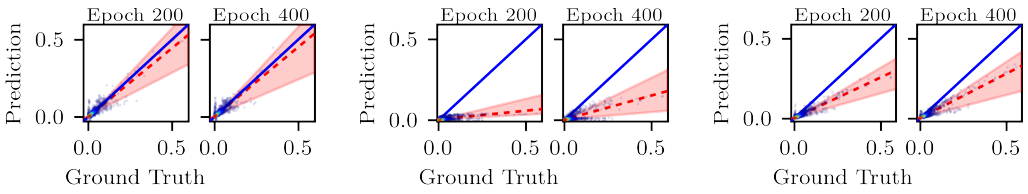



**Figure 5.15:** Mean error of Dual Loss Network. Error bars show squared standard deviation. Using both loss functions side by side decreases samples and learning time for active sampling. Areas behind absolute error lines show that there is a far higher disagreement between submodels. Relative errors are very small while absolute errors are slightly worse than our optimum in Figure 5.13

yellow indicates maximal density and dark blue indicates low density and outliers. The evaluation happens after learning. Using our linear regression as indicator, it is clearly visible, that training with absolute error indicates the smallest error independently from sampling strategy. We think, that the reason is the weighting of error, because absolute loss uses the squared error and therefore mainly fits large samples. Interestingly, the prediction tends to underestimate the ground truth in every graph. Even the upper bound of our linear regression is never far above.

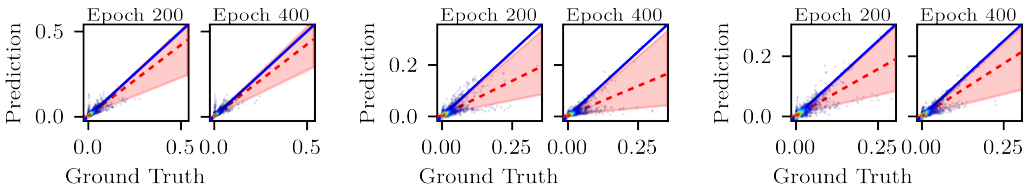
For the absolute loss, there is a bunch of samples with a small ground truth, that are overestimated. They form a vertical line on the left side, that grows for active pool sampling over time. This may be caused by sample selection when submodels disagree for these samples. In comparison to the large bunch of samples that are almost correctly predicted, they all seem to be outliers and therefore it is possible, that they are caused by noise in the dataset. Training with relative loss shows also a vertical line of similar samples, but far smaller and decreasing.

Training with relative loss leads to underestimation of samples, especially for passive sample selection. In this case, active pool sampling improves the prediction more than




**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

**Figure 5.16:** Comparison of Prediction and Ground Truth for passive sampling. Usage of relative error leads to underestimating.



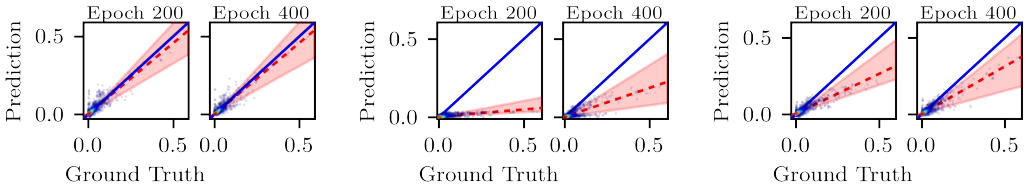

**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

**Figure 5.17:** Comparison of prediction and ground truth for active pool sampling. Predictions improve faster than for passive sampling. Two main diagonals are visible, one consists of underestimated samples.

sample search. However, it also enlarges the error range. The upper and the lower bound of our linear regression mark two different clusters. One is very good predicted and always around the main diagonal. The other is predicted too low.

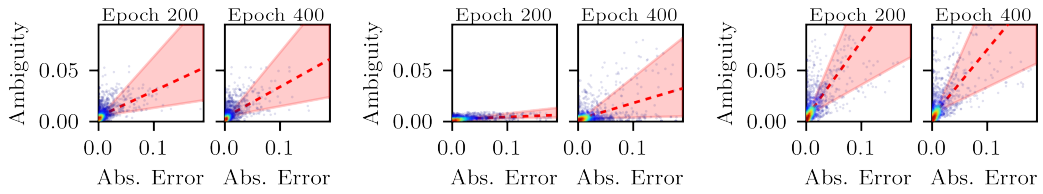
Our Dual Loss Network combined both loss functions. Results show, that this mixing is possible and results not in improving but blending between both results. Both disadvantages, the left cluster for absolute loss and the lower cluster for relative loss, are weakened. However, it also tends to underestimate.

We can follow, that training with relative loss leads to underestimation. A possible cause lays in our sample selection and therefore in our ambiguity value. This value indicates, how much submodels disagree, but not necessarily how large the error is. In the following




**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

**Figure 5.18:** Comparison of Prediction and Ground Truth for active sample search. Learning with relative loss is especially bad.




**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

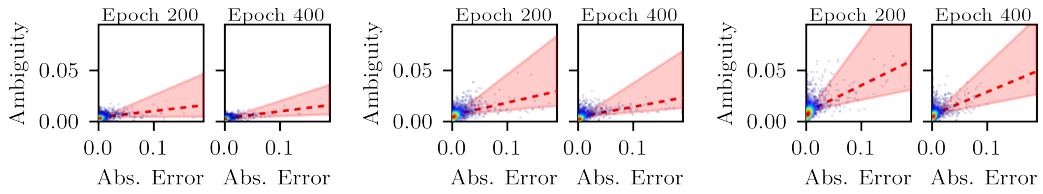
**Figure 5.19:** Comparison of ambiguity and absolute error for passive sampling. There are almost no changes except for relative sampling, that started with an especially low ambiguity.

we analyze, if the ambiguity can also approximate the error. Figures 5.19, 5.20, and 5.21 are structured analogous to prior used plots for prediction and ground truth.

In prior Figures 5.16 to 5.18 it is visible, that there are only few high values in the data. This also explains, why in the first epoch, there are only a few outliers with high errors. The big but not very dense blob on the left side shows, that even for very small errors there can be a high ambiguity. However, this blob becomes compressed near the origin over time. For passive sampling, there are several interesting aspects visible in Figure 5.19. Training with absolute loss decreases ambiguity slowly. Even after 400 epochs, ambiguity can be large for small errors and small for large errors. If trained with relative loss, ambiguities are wider spread. Until epoch 200 learning seemed to focus on large samples and learning small ones between epoch 200 and epoch 400. For the combination of both loss functions in Figure 5.19c, the linear regression of ambiguity and error is closer the main diagonal and therefore best to predict the model error. Even in Figure 5.15 it is visible, that ambiguity is very high for this network configuration, but Figure 5.16b indicates, that this supports good sample selection.

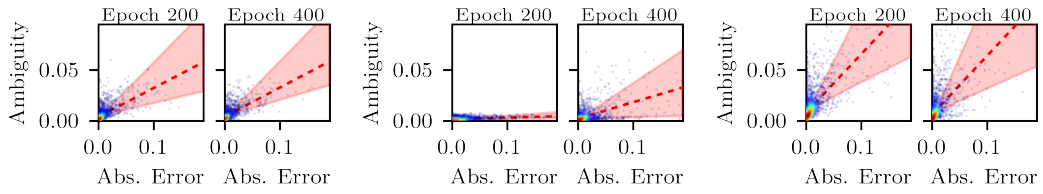
For active pool sampling, ambiguity changes differently over time as visible in Figure 5.20. Ambiguity is in the whole figure far smaller than the real error. However, ambiguity is higher than for passive sampling, which seems to be a side-effect of active sampling. This is especially interesting because, active pool sampling results in the lowest overall error.

Interestingly, Figure 5.19 about passive sampling and Figure 5.21 about active sampling search are again very similar. Their value ranges overlap that much, that slightly differences may be caused by random, e.g. the sample selection for the evaluation. Even the cluster at the bottom in Figure 5.21b in epoch 400 is similar. Comparing this to error plots in Figure 5.13 and Figure 5.15 we can conclude, that both methods are almost equal but active sample search needs less samples.




**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

**Figure 5.20:** Comparison of ambiguity and absolute error for active pool sampling. Samples are drawn to the origin, meaning that error and ambiguity are decreased in parallel.




**(a)** Absolute Loss **(b)** Relative Loss **(c)** Both Loss Functions

**Figure 5.21:** Comparison of ambiguity and absolute error for active sample search. Strong similarities to passive sampling in Figure 5.19 are visible.

### 5.2.2 Indirect Volume Visualization

We executed our learner on four different graphics cards with all of our different sampling strategies and both loss functions. Because tests were started during prior tests run, we did not used Dual Loss Network but Triple Layer Network.

Our measured results are shown in Figure 5.22 to 5.24. All plots are equally formatted like in prior ones. The render times of our renderer are larger than for the given raycaster dataset (compare execTime in Table 5.1 and Table 5.2). Therefore, an higher error can also be expected.

In Figure 5.22, there are visible spikes in the learning curve. These are caused by an external change in the dataset. Our first input configuration consisted of gridRes, imgRes, iso_layer_count, tree_split and all generated and cached values listed in Table 5.5. We recognized during learning that the camera position was contained in the input space but not the cameras direction. It makes a huge difference if the camera is looking towards the volume center (cam_dist $> 0$) or if its looking away (cam_dist $< 0$). This also explains why learning is slow at the beginning compared to results in Figure 5.23 and 5.24. We changed the input of our model and recreated it. Instead of throwing all collected data away, we reused it as if the model never changed. The visible spike is caused by the uninitialized model.

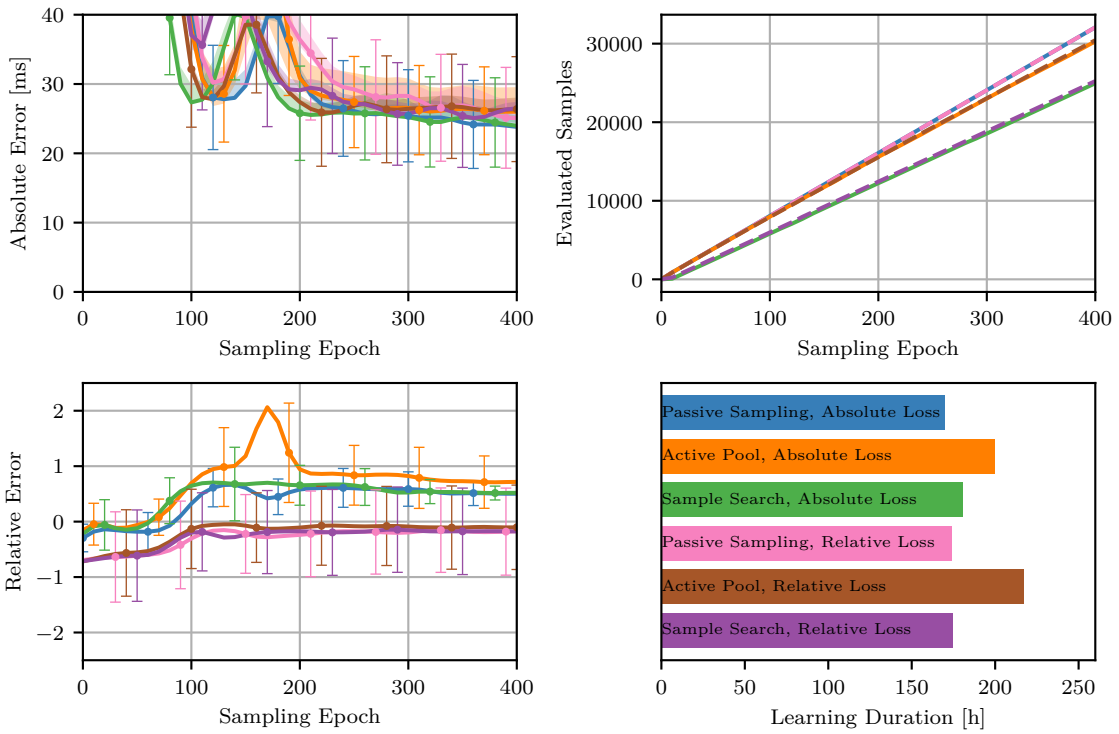



**Figure 5.22:** Error of GeForce GTX Titan. A bad sample selection leads to an high absolute error. Number of evaluated samples raises linear.

When comparing those graphs with spike to those without there are important aspects for future training. First of all, when training with data, that is not suitable, learning is very slow. When the input is corrected, learning is never the less slower, than when training from scratch. The reason lays probably within the neural networks. A new neural network has a random initialization that is most probable wrong. This results in faster learning because there are clear errors. At the spikes we fed prior sampled data that was also active sampled and therefore as bad as our prior input selection. Nevertheless, networks used them to become better but afterwards, learning was harder. They were trapped in a local minimum and needed to get more good samples to find a better one. Even after 400 sampling epochs, where each contained 512 learning epochs, they were not converged. Both networks that were trained with correct data from start on converged before 200 sampling epochs were complete (see lower half of Figure 5.24).

In comparison with Figure 5.13 the relative error is more stable for our renderer. This can be caused by our longer execution times per frame that seem to be less prone to noise. Training time is not comparable, because the necessary time to render each frame multiple times and evaluate never seen frames is only included in our renderer, not in the given dataset. Interestingly, trainings on the given dataset seemed to keep learning until the end, while learning our renderer stopped earlier.

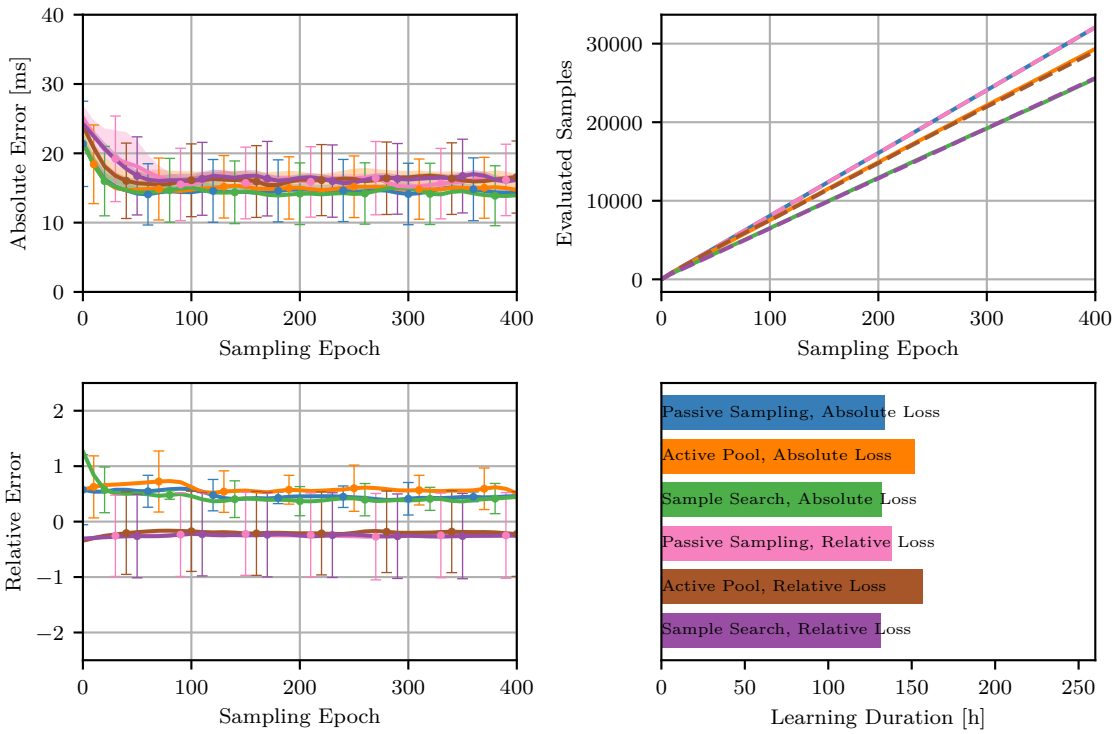



**Figure 5.23:** Error of GeForce GTX 1070. With good input, learners converge fast for our renderer.

Like before for the given dataset, we compared the prediction of our model to the collected ground truth. We used the GeForce GTC 1070 in our plots in Figure 5.25, because it performed best in the test. The results are well comparable to our prior results in Figure 5.16 to 5.18. However, there are some new patterns while others are gone. For the given dataset, there was a cluster of small valued samples that were overestimated when training with absolute error. This is not repeated and neither the underestimated cluster when training with relative loss. This clearly points to data as the source of these clusters. In Figure 5.25 instead, there are clearly visible horizontal lines in the plot. They are most probably caused by volume files like lines in Figure 5.3. On one hand this shows, that more different grid resolutions and volume files are necessary to become better. We follow from the empty space between these lines, that there is not enough information to improve these artefacts.

Relative error again will underestimate samples more often than overestimating. That this effect coincides with prior results of the raycaster dataset points to a general disadvantage of this loss function. However, there is also a positive aspect. Prior mentioned horizontal line artefacts do also exist but are more dense, even if the overall error is larger.

Our scatter plots in Figure 5.26 show that again, ambiguity is not an error measure. Ambiguity values are drawn to the origin but after a several epochs there are still high ones, that were not drawn. Again, horizontal lines are visible. There is a cluster of samples,

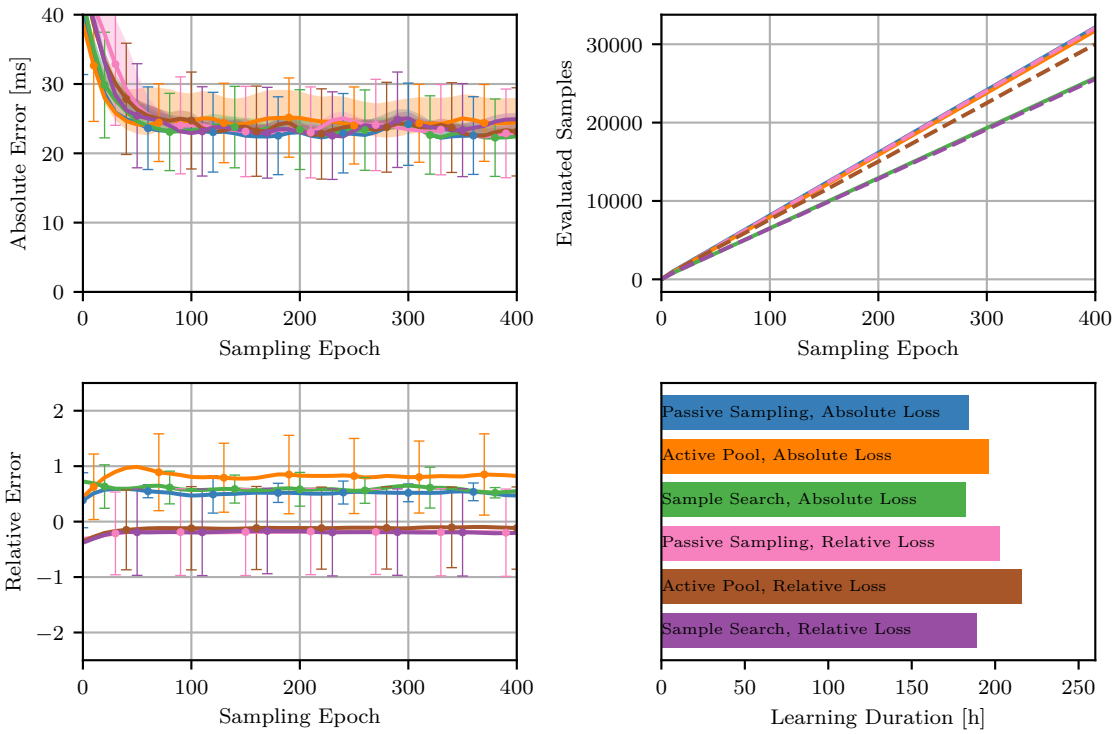
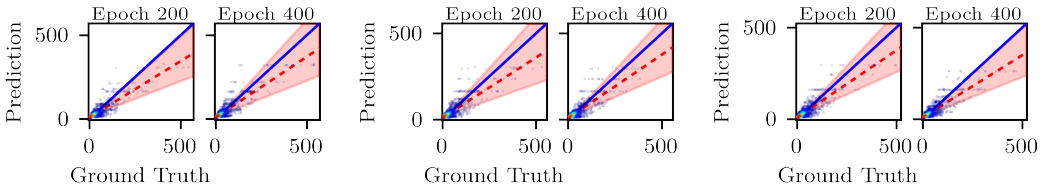



**Figure 5.24:** Error of Quadro M6000. Learning converged to an higher optimum as for Figure 5.23 and needed more time. Search is again better than pool sampling, especially for relative error.



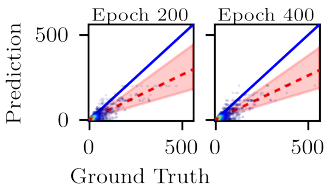

**(a)** Passive Sampling, Absolute Loss



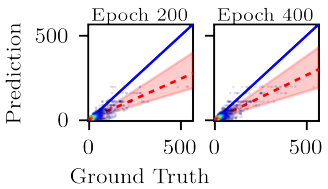

**(b)** Active Pool Sampling, Absolute Loss



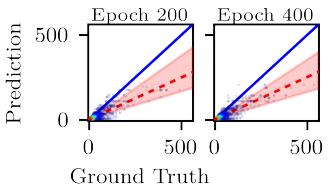

**(c)** Sample Search, Absolute Loss




**(d)** Passive Sampling, Relative Loss

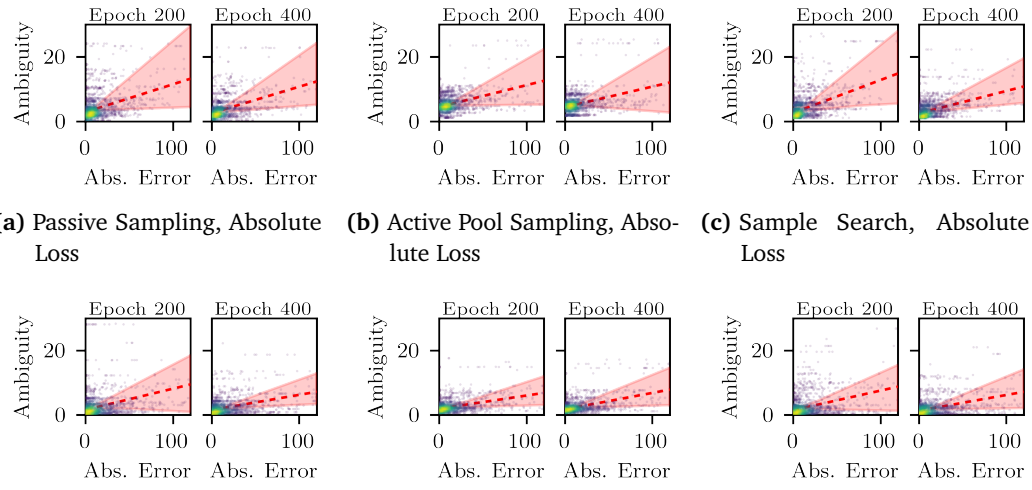



**(e)** Active Pool Sampling, Relative Loss




**(f)** Sample Search, Relative Loss

**Figure 5.25:** Comparison of Prediction and Ground Truth for GeForce GTX 1070. Estimations are similar for all sampling strategies and loss functions. There are visible line-like artifacts in each.




**(a)** Passive Sampling, Absolute Loss

**(b)** Active Pool Sampling, Absolute Loss

**(c)** Sample Search, Absolute Loss

**(d)** Passive Sampling, Relative Loss

**(e)** Active Pool Sampling, Relative Loss

**(f)** Sample Search, Relative Loss

**Figure 5.26:** Comparison of Prediction and Ground Truth for search sampling. Errors and ambiguities are far higher than for raycaster dataset. Ambiguity is larger, when training with absolute loss.

were the ambiguity is almost zero but the error is larger. Our model would have problems finding these specific samples, if none is chosen randomly.

## 5.3 Discussion of Results

We described both of our datasets by their content, meaning and parameter distributions. First is a tabular dataset of measurements by a volumetric raycaster. Second is not a real dataset but an evaluator approach, where our model can select any valid configuration to measure our renderers performance. They share several parameters like device, volume, camera position and image resolution. Other parameters depend on the visualization type.

Afterwards, we discuss the learning process and insights we can derive by comparing differences between learning on each dataset. We name limits of each part and how they may be handled in future. We discuss expressiveness of our results, together with found advantages and disadvantages.

### 5.3.1 Datasets

The given raycaster dataset contained grid-based samples. Our sampler normalizes distributions to enable uniform sampling. This needed several adjustments and can still not be fully achieved. Dependent variables must be sampled together, like volume file and

its resolution, disabling a uniform distribution for both. Volumes of resolution 1024 are overrepresented and more often rendered. Our sampling reduces their probability of being drawn, however, there are far more details for this volume size. Additionally, one parameter – tffId – was not varied for smaller volumes and as a result its influence predictions are only valid for large volumes. For our own renderer, same problems appear, when attributes of volumes are added to the training set. However, because our volumes has all the same size and grid resolution can be chosen, only mesh attributes like triangle count are non-uniform in drawing. This issue cannot be handled easily. However, we suspect, that active learning can solve this issue, if the pool size is large enough. Pool sampling transforms all of our uniform distributions by submodel predictions. However, another way would be switching of distributions. Samples can be drawn from different distributions, where uniform sampled parameters are switched for dependent parameters.

In comparison to the dataset created by our indirect volume visualization, the given raycaster dataset contained less volume properties. Only volume resolution was offered, because it is the most important value for direct volume visualization. We instead added a huge bunch of attributes like triangle count and instead of using the volumes resolution we chose to replace it by a grid resolution. This higher amount of attributes is a possible cause for faster convergence.

Both datasets contained redundant data. For the raycaster dataset, resolutions were duplicated, because all volumes were cubes and rendered images were squares. For our own renderer, we collected these values only once. Instead we had four attributes of our created meshes, that all showed memory usage with a slightly different scale. We do not think that there is a further disadvantage from this but a possible learning slow down.

Our expansion and deeper analysis of the camera position showed us that converting into Cartesian coordinates creates a more smooth representation. Moreover, it showed, that the effect of each axis in the space is equal. However, if the camera is free to move in three-dimensional space, like for our renderer, it is important to add the cameras orientation. In our case, the origin of the visualized volume was always in the center of the image plane but sometimes behind the camera. Therefore, the distance to the origin encoded viewing direction. Rendering is more expensive if the center is in front of the camera, because the visible volume part contains more triangles.

The main part of our indirect volume visualization is sorting of triangles. We divide sorting of triangles into two tasks. Properties described in Chapter 3 allow us to encapsulate triangles in cuboid cells. Sorting of cells and sorting of triangles within each cell are different tasks. Sorting cells happens in an octree iteration, which is divided into a single-threaded startup and a multi-threaded continuation. We needed to limit parameter tree split, that defines how deep startup iterates the octree. A value smaller 3 leads to crashes of the continuation and a value larger 5 leads to crashes in startup. This limitation also leads to an upper bound for the octree size, because none of these shader is able to handle more than 5 levels of the octree. A better solution would be to iterate in more than two steps. If each would iterate about three steps of the octree, each will be very fast and many threads can be used in parallel. This solves also another issue: Our octree build by

splitting always in the center of a node. Therefore, it becomes a long linked list in worst case. A better solution would be a balanced kd-tree, but then clipping of nodes needs more information.

Our leaf sorting has the longest maximal execution time. However, it is a very simple shader that is optimized for parallel execution and therefore we cannot shorten render time by improving this shader. However, its runtime raises to the square of leaves that are reached during octree iteration. Therefore, reducing the amount of leaves is the most certain point of optimization. Possible optimizations are combining of cells in a level-of-detail approach or clipping by occlusion. Latter might need multiple render passes to detect visible cells. However, optimizations by occlusion are not possible, if each layers transparency is too high.

Our own renderer is not able to sample graphics cards itself, because OpenGL 4.5 does not allow GPU selection, even if multiple are mounted. Because we determined, that training a model per graphics card and renderer is best according to input selection, we do not see a disadvantage in it.

In our opinion, grid resolution is most important for render time. It controls the amount of triangles and therefore octree size and octree iteration time. Leaf sorting also depends on the amount of leaves, that directly depends on triangles. Next, tree split is also important. It enables decreasing of render times through controlling parallelism.

The largest drawback of our renderer is its slow mesh creation. Otherwise, a larger set of volumes, iso layer counts and grid resolutions could have been used. We built the renderer to create a flexible environment, where our performance prediction can learn. For this task, our renderer can be used, however it might perform better when improving this issue.

### 5.3.2 Prediction Learning

In Section 5.2 we determined a good configuration for our neural networks. Our results showed, that increasing the draw size also increases the amount of made measurements but does not improve learning. We think, that a larger draw size also increases the probability for bad samples to be drawn. They further decrease ambiguity in well matched regions and therefore do not offer much advantage for learning. Moreover, if less samples are drawn, learning can better focus those with an high error and decreasing the error for each submodel with respect of its own training set. This on other hand keeps ambiguity high by shifting each submodel differently. We therefore conclude, that a small draw size and a large amount of training epochs should be used.

For a similar reason, the pool size should be chosen very high to enlarge the amount of good samples within. Because there where more important parameters, we did not change our gamma value for ambiguity correction. However, we think that it depends on the pool size. We can only conclude, that it has a very positive effect on training. However, if chosen too high, it leads to an overlap of training sets between submodels and we expect, that this will lead to bad learning, because ambiguity is decreased too fast.

Evaluation of our different network configuration showed, that a network with three hidden layers performs best. We estimate, that an even larger network will be even better, but also will need more training time and samples to converge. Also, increasing the amount of perceptrons should improve learning by refining the networks possibilities.

Our ambiguity value was thought as a measurement of submodel agreeing. When training with absolute loss, ambiguity is decreased in every sampling epoch, but is often far lower than the absolute error. It becomes worse when training with relative loss, which is probably caused by a measurement mismatch. Ambiguity is based on the absolute predictions of each submodel and therefore behaves badly, when the relative error is used for learning. In this case, a relative ambiguity needs to be redesigned. Parallel usage of submodels that train with absolute loss and same amount of submodels that train with relative loss increases ambiguity. For sample search it becomes the best error approximation we found according to drawn samples.

When reviewing our sampling strategies, active pool sampling reached the smallest error for the raycaster dataset. Even if results of our second evaluation with our indirect volume visualization showed, that this is not always the case, its results are near the optimum. This behavior is visible for all both loss functions and their combination, wherefore we state, that active pool sampling is the most accurate strategy. When training with our renderer, the amount of drawn samples is almost the same like for passive sampling. This is caused by continuous columns, because it is impossible to draw a sample twice. For tabular data instead it is possible and becomes more probable over time what explains the small amount of samples drawn by active pool sampling with relative loss. A larger $\gamma$ for ambiguity correction may decrease drawn samples by increasing the amount of samples that are drawn by multiple submodels. It is not a disadvantage, when samples are shared. For sample search such sharing is very probable and it behaves not worse than passive sampling.

Our model is able to predict performance for both datasets. However, while it converged very fast for continuous input space of our indirect volume visualization, it needed far longer for grid-based data of the raycaster dataset. Learning with the given raycaster dataset converged to a smaller error. We assume, that this behavior is caused by the value range within each dataset. The used raycaster rendered images faster than our render approach and had a mean execution time of 25 ms. Our renderer instead had needed 62 ms as mean. This corresponds with the relative difference between both approaches.

For learning there are several differences. First of all the learner seems to learn longer for the raycaster dataset. There is still a small improvement visible until the 400th sampling epoch. Instead, our volume visualization seems to be converged before 200 sampling epochs. On one hand this can be caused by the dataset itself. It is possible, that for our renderer, the optimum is simpler to find and therefore convergence happens much faster. On other hand, this can be caused by increased freedom for the sample selector. Even if not all parameters could be chosen with fine resolution, it was possible for camera position and rendered image resolution. We think, that their influence can be estimated much more

accurate by the network and along with this, estimating the effect of other parameters became simpler. None of these possibilities can be proofed without further tests.

### 5.3.3 Limitations

There are several issues that forced us to limit some of the parameter ranges. Tree Split was restricted to values 3, 4, and 5 because higher and lower values had the same effect. In each, one of the octree iteration shaders (startup or continuation) needed more than 100000 internal iterations. This resulted in crashes, even bluescreens under Windows. Our volume selection and grid resolution were limited to a tiny set, because mesh creation was implemented in Python and therefore not very fast. Creating new meshes needed several minutes and would have slowed down learning heavily. Our implementation caches created meshes for reuse. Our set of seven volumes and five different resolutions needed twenty gigabytes, that we held in RAM.

There are some limitations in hardware for our renderer. The utilized graphics cards needs enough storage to store the whole volume in memory during mesh creation. For rendering, octree, sorted index, triangles and vertices where in GPU memory to speed up execution. In our tests these buffers where smaller than 500 MB, as shown at the bottom of Table 5.5. It is possible to process larger data but not interactively.

Our renderer was tested on seven graphics cards, but only four of them were able to run our shaders. We found, that our code is incompatible with AMD graphics cards, resulting in immediate crashes before any configuration is processed. We also tested our implementation on a GeForce GTX 1080 to enable a better comparing with our dataset experiments. Unfortunately, this computer or its operating system seemed to be broken. Execution crashed several times randomly even on small volumes and triggered bluescreens with a WATCHDOG_VIOLATION code.

# 6 Conclusion and Future Work

In this thesis we presented a model and sampling strategies to predict performance using active learning in context of volumetric visualization. We used a given dataset that was created by a raycaster and implemented an own renderer for indirect volume visualization. Both offered us a large set of data we used for our visual analysis.

Our renderer consists of two parts. First we preprocess each volume before rendering and extract iso-surfaces as triangle mesh using marching cubes. Triangles are organized in a spacial octree and cached on disk for future usage. Creating of the octree was implemented in Python and as a result too slow to take part in execution time. When rendering, we first created a sorted index for all triangles. It does not create a global solution but a local one.

We divided triangle sorting into two parts. To apply marching cubes, the volume is divided into cuboid cells. Triangles are created within and touch triangles in neighbor cells. As an important property, triangles can never cross nor be in multiple cells. Therefore we first sort cells by iterating the octree and afterwards sorting triangles within each visible octree leaf.

In our visual analysis we found that the given raycaster dataset contained grid-based data, that was not uniformly for every parameter. Before learning we wrote a sampler that handles this issue by weighting. Also, we found that spherical coordinates were used to describe camera positions. This has the disadvantage, that ambiguities like $0 \equiv 2\pi$ are not visible. A neural network cannot learn a smooth solution for this, therefore we decided to convert the camera position into Cartesian coordinates. The cameras view direction always pointed to the volumes center. For our renderer we enabled further movement of the camera towards the volumes center and through. Therefore, we needed to add the cameras distance along the view vector, because its a difference, if the camera in in front or behind the center.

In our investigations we used a given dataset created using a volumetric raycaster and a our own renderer to create another dataset during learning. Both contained the input parameters of the used renderer and attributes of visualized volumes. We build a model using neural networks to learn a mapping function from this input space to execution time. Because the input space can be high in dimension we decided to build our model based on a uniform sampler over parameters. Our model then actively selected or rejected drawn samples to improve learning.

Our model was build as a set of equally structured but differently initialized neural networks. From each prediction we derived two values. On one hand, the mean of all neural networks

was used as the models prediction. After a few sampling epochs this outcome was better than each submodel prediction. This is possible, because each submodel has learned a different error and after a while they cancel out each other. On other hand, the difference between all submodels can be used as measurement, how good the mapping function learned a certain sample. A large disagreement does not correspond with a large error, but shows that at least one submodel has not reached the optimum at this point. Therefore, taking samples with a high ambiguity supports learning.

Ambiguity will decrease very fast, when all submodels train on same data. However, error will decrease very slow in this case because submodels will all fit a sparse subset of the real mapping function. Instead, our submodels sampled all different training sets with small overlaps. Because all samples are drawn from the same distribution, all submodels will converge to the same optimum. If they share as few samples as possible, they become more reliable.

We tested three different strategies to sample. Our base value was estimated using passive sampling where each submodel samples independently and without usage of ambiguity. Results showed, how our model behaves without active learning. We improved this sampling in two ways. First, we uniformly sampled a large amount of samples and calculated their ambiguity. This ambiguity was used to weight each drawn sample before each submodel draws its own subset. For large pools, many bad samples with small ambiguity are drawn, because their number increases during learning and so increases the probability that a bad sample is drawn. To handle this issue, we used gamma correction, that increased contrasts between high and low ambiguities. Our second active learning strategy explicitly searches for good samples and draws as many as necessary. One after another, ambiguity is evaluated and samples are distributed over submodels until enough distinct samples are chosen by all submodels. On one hand this increases the amount of samples that are drawn, but it also decreases number of evaluated samples.

Our results showed, that for the given raycaster dataset, active learning achieves smaller errors than both other sampling strategies. This advantage does not generalize for our learner, where it converged faster than both other functions but reached a worse local optima. Sample search instead behaved like our passive sampling but needed far less samples.

We also tested two different loss function for learning. On one hand we tested training with mean squared loss that depends on the absolute error. On other hand we utilized the mean square of the relative error to train the model. Each behaved like expected, but in comparison training with absolute loss resulted in smaller absolute and comparable relative errors. Additionally, the training with relative loss tends to underestimate in its predictions. We also tried a combination of both loss functions. Therefore, we build a model our of six submodels, where each half was trained with a different submodel. Results showed, that this improves the relative error and delivers small absolute errors. Further, the amount of samples is heavily decreased for both active strategies.

## 6.1 Limitations

Both training sets that where used in our evaluation had less than 10 different parameters in their configuration. Each was restricted to a small set of valid values, but render size and camera position for our own renderer. Therefore, we can suspect, that our model is able to efficiently learn in a high-dimensional and continuous configuration space, but this is not shown by our results. We restricted used volumes and resolutions of triangle meshes, because volumes needed long to be preprocessed and would have slowed down learning heavily. Other parameters were restricted to prevent the renderer from crashing. Improving volume preparation efficiency and detecting crashes are no simple tasks but enable usage of our model in larger domains.

We determined configuration and architecture for our model such that learning on the given raycaster dataset resulted in an prediction with small error. However, this configuration behaved differently when training with our own renderer. As a result, we got a faster convergence but a higher error, that might be improved with a different network configuration.

Our indirect volume visualization is based on several assumptions. The whole triangle mesh and the complete octree need to stay on GPU. Therefore, some configurations in resolution and iso layer count will result in buffers too large to be drawn. Moreover, we found that our octree iteration is restricted in iterations per execution. To handle this we needed to restrict parameters and graphics cards.

## 6.2 Future Work

There is further potential in our investigations for future work. We showed, that active sampling strategies and usage of different loss functions can be used for accurate performance predictions. However, our model simply took the mean of all available submodels. A more complex model could weight each submodel by accuracy to achive even better. Even differently structured submodels can used this way within one model. We suggest, that even exchanging of submodels during learning to fit configurations to datasets is a possible way. As result, a well configured set of submodel configurations can be achieved and applied on further graphics cards.

However, such a model will need far more time to converge to an optimum but can also learn a structure to match future graphics cards faster. For our renderer the presented model needed about a week per sampling method and loss function to converge to a useful result. Training with fine sampled sets of measurements speeds up training heavily as visible in our evaluation of the raycaster dataset. We suggest to create such a dataset on a specific graphics card for future test. As a secondary advantage, this enables training of multiple different machines while ensuring comparability of results.

# Bibliography


[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283 (cit. on pp. 20, 42).

[App68] A. Appel. "Some Techniques for Shading Machine Renderings of Solids." In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 37–45. DOI: 10.1145/1468075.1468082. URL: http://doi.acm.org/10.1145/1468075.1468082 (cit. on p. 15).

[Bat68] K. E. Batcher. "Sorting Networks and Their Applications." In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 307–314. DOI: 10.1145/1468075.1468121. URL: http://doi.acm.org/10.1145/1468075.1468121 (cit. on p. 32).

[Blo88] J. Bloomenthal. "Polygonization of implicit surfaces." In: *Computer Aided Geometric Design* 5.4 (1988), pp. 341–355 (cit. on p. 14).

[Bou97] P. Bourke. *Polygonising a scalar field*. 1997. URL: http://paulbourke.net/geometry/polygonise (cit. on p. 29).

[Bus98] M. Buscema. "Back propagation neural networks." In: *Substance use & misuse* 33.2 (1998), pp. 233–270 (cit. on p. 20).

[Car84] L. Carpenter. "The A-buffer, an antialiased hidden surface method." In: *ACM Siggraph Computer Graphics* 18.3 (1984), pp. 103–108 (cit. on p. 26).

[CR95] Y. Chauvin, D. E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995 (cit. on pp. 17, 20).

[CS78] H. N. Christiansen, T. W. Sederberg. "Conversion of complex contour line definitions into polygonal element mosaics." In: *ACM Siggraph Computer Graphics*. Vol. 12. 3. ACM. 1978, pp. 187–192 (cit. on p. 14).

[CSSC00] J. J. Choi, B.-S. Shin, Y. G. Shin, K. Cleary. "Efficient volumetric ray casting for isosurface rendering." In: *Computers & Graphics* 24.5 (2000), pp. 661–670 (cit. on p. 15).

[DCH88] R. A. Drebin, L. Carpenter, P. Hanrahan. "Volume Rendering." In: *SIGGRAPH Comput. Graph.* 22.4 (June 1988), pp. 65–74. ISSN: 0097-8930. DOI: 10.1145/378456.378484. URL: http://doi.acm.org/10.1145/378456.378484 (cit. on p. 14).

[FKU77] H. Fuchs, Z. M. Kedem, S. P. Uselton. “Optimal surface reconstruction from planar contours.” In: *Communications of the ACM* 20.10 (1977), pp. 693–702 (cit. on p. 14).

[Fri94] B. Fritzke. “Growing cell structures—a self-organizing network for unsupervised and supervised learning.” In: *Neural networks* 7.9 (1994), pp. 1441–1460 (cit. on p. 20).

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 17).

[GN71] R. A. Goldstein, R. Nagel. “3-D visual simulation.” In: *Transactions of the Society for Computer Simulation* 16.1 (1971), pp. 25–31 (cit. on p. 15).

[HSS05] G.-B. Huang, P. Saratchandran, N. Sundararajan. “A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation.” In: *IEEE Transactions on Neural Networks* 16.1 (2005), pp. 57–67 (cit. on p. 20).

[HVD01] J. Herrero, A. Valencia, J. Dopazo. “A hierarchical unsupervised growing neural network for clustering gene expression patterns.” In: *Bioinformatics* 17.2 (2001), pp. 126–136 (cit. on p. 20).

[KB14] D. Kingma, J. Ba. “Adam: A method for stochastic optimization.” In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 45).

[Kep75] E. Keppel. “Approximating complex surfaces by triangulation of contour lines.” In: *IBM Journal of Research and Development* 19.1 (1975), pp. 2–11 (cit. on p. 14).

[KPB12] T. Kroes, F. H. Post, C. P. Botha. “Exposure render: An interactive photo-realistic volume rendering framework.” In: *PloS one* 7.7 (2012), e38586 (cit. on p. 15).

[KV95] A. Krogh, J. Vedelsby. “Neural network ensembles, cross validation, and active learning.” In: *Advances in neural information processing systems*. 1995, pp. 231–238 (cit. on pp. 21, 37).

[LC87] W. E. Lorensen, H. E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm.” In: *ACM siggraph computer graphics*. Vol. 21. 4. ACM. 1987, pp. 163–169 (cit. on pp. 14, 23).

[Lev90] M. Levoy. “Efficient Ray Tracing of Volume Data.” In: *ACM Trans. Graph.* 9.3 (July 1990), pp. 245–261. ISSN: 0730-0301. DOI: 10.1145/78964.78965. URL: http://doi.acm.org/10.1145/78964.78965 (cit. on p. 15).

[Max95] N. Max. “Optical models for direct volume rendering.” In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108 (cit. on p. 15).

[MH76] J. C. Mazziotta, H. K. Huang. “THREAD (Three-dimensional Reconstruction and Display) with Biomedical Applications in Neuron Ultrastructure and Computerized Tomography.” In: *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*. AFIPS ’76. New York, New York: ACM, 1976, pp. 241–250. DOI: 10.1145/1499799.1499839. URL: http://doi.acm.org/10.1145/1499799.1499839 (cit. on p. 13).

[Nie15] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com (cit. on p. 17).

[NY06] T. S. Newman, H. Yi. "A survey of the marching cubes algorithm." In: *Computers & Graphics* 30.5 (2006), pp. 854–879 (cit. on p. 14).

[OM99] D. W. Opitz, R. Maclin. "Popular ensemble methods: An empirical study." In: *J. Artif. Intell. Res.(JAIR)* 11 (1999), pp. 169–198 (cit. on pp. 21, 36, 37).

[RHW86] D. E. Rumelhart, G. E. Hinton, R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–538 (cit. on p. 20).

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 17).

[RPSC99] H. Ray, H. Pfister, D. Silver, T. A. Cook. "Ray casting architectures for volume visualization." In: *IEEE Transactions on Visualization and Computer Graphics* 5.3 (1999), pp. 210–223 (cit. on p. 15).

[SU07] A. I. Schein, L. H. Ungar. "Active learning for logistic regression: an evaluation." In: *Machine Learning* 68.3 (2007), pp. 235–265 (cit. on p. 22).

[TPG99] G. M. Treece, R. W. Prager, A. H. Gee. "Regularised marching tetrahedra: improved iso-surface extraction." In: *Computers & Graphics* 23.4 (1999), pp. 583–598 (cit. on p. 14).

[VK96] A. Van Gelder, K. Kim. "Direct volume rendering with shading via three-dimensional textures." In: *Proceedings of the 1996 symposium on Volume visualization*. IEEE Press. 1996, 23–ff (cit. on p. 15).

[Wer90] P. J. Werbos. "Backpropagation through time: what it does and how to do it." In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560 (cit. on p. 20).

[WL90] B. Widrow, M. A. Lehr. "30 years of adaptive neural networks: perceptron, madaline, and backpropagation." In: *Proceedings of the IEEE* 78.9 (1990), pp. 1415–1442 (cit. on p. 20).

[WMW86] G. Wyvill, C. McPheeters, B. Wyvill. "Data structure forsoft objects." In: *The visual computer* 2.4 (1986), pp. 227–234 (cit. on p. 14).

[ZRL+08] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, H.-Y. Shum. "Real-time Smoke Rendering Using Compensated Ray Marching." In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 36:1–36:12. ISSN: 0730-0301. DOI: 10.1145/1360612.1360635. URL: http://doi.acm.org/10.1145/1360612.1360635 (cit. on p. 15).

All links were last followed on October 10, 2017.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature