Graz University of Technology
Institute of Computer Graphics and Vision

University of Ljubljana
Faculty of Computer and Information Science

Tim Oblak

Recovery of superquadric parameters from depth images using deep learning

MASTER'S THESIS

MASTER'S STUDY PROGRAMME
COMPUTER SCIENCE

Graz, 2020

Graz University of Technology
Institute of Computer Graphics and Vision

University of Ljubljana
Faculty of Computer and Information Science

Tim Oblak

Recovery of superquadric parameters from depth images using deep learning

MASTER'S THESIS

MASTER'S STUDY PROGRAMME
COMPUTER SCIENCE

Supervisor: dr. Peter M. Roth
Co-supervisor: prof. dr. Franc Solina

Graz, 2020

Acknowledgments

I would like to thank the whole team on the Superquadrics project for giving me the opportunity to work on this interesting topic and providing me with support during its course. Prof. dr. Peter Peer, Assoc. Prof. dr. Vitomir Štruc and Assist. Prof. dr. Aleš Jaklič helped me greatly at the operative level, motivated and steered me into the right direction. I'm grateful to Prof. dr. Franc Solina for accepting the role of my mentor and providing me with various thoughts and ideas from his pursuit of superquadrics. I would also like to thank dr. Peter M. Roth, who offered me mentorship during my stay in Graz, allowed me the freedom to research freely for his seminar course and kindly helped me with my technical writing. Despite the initial difficulties, I am glad my hypothesis was solid and I was able to complete this work highly motivated.

I am thankful, that even during this strange year, that was 2020, I was fortunate enough to be part of many positive stories and interesting opportunities. From the bottom of my heart, I thank my partner, Sara, who inspired me and selflessly offered her love and support. I am beyond grateful to my parents, who gave me encouragement and most importantly, a sense of security, which enabled me to take bigger risks in life.

Tim Oblak, 2020

Contents

Abstract

Abstract

Povzetek

1 Introduction **1**

1.1 Challenges 4

1.2 Contribution 6

1.3 Thesis structure 6

2 Related work **9**

2.1 Early volumetric reconstruction 9

2.2 Superquadric parameter recovery 11

2.3 Deep learning in 3D 13

3 Superquadrics **19**

3.1 Superellipses 19

3.2 Parametric equation and superellipsoids 21

3.3 Superquadric inside-outside function 22

3.4 Superquadrics in general position 23

3.5 Superquadric vocabulary 25

3.6 Visualization 26

4 Neural networks **31**
4.1 Artificial neural networks . . . 31
4.2 Learning . . . 36
4.3 Convolutional neural networks . . . 39

5 Experimental setup **45**
5.1 Dataset . . . 45
5.2 Model architecture . . . 47
5.3 Training and hyper-parameters . . . 49
5.4 Metrics . . . 50

6 Supervised learning of superquadric parameter recovery **53**
6.1 Motivation . . . 53
6.2 Problem formulation . . . 55
6.3 Occupancy loss . . . 56
6.4 Implementation details . . . 58
6.5 Experiments and results . . . 60
6.6 Discussion . . . 72

7 Unsupervised learning of superquadric parameter recovery **73**
7.1 Motivation . . . 73
7.2 Problem formulation . . . 75
7.3 Least squares loss . . . 75
7.4 Differentiable render loss . . . 78
7.5 Implementation details . . . 81
7.6 Experiments and results . . . 82
7.7 Discussion . . . 95

8 Conclusion **97**
8.1 Summary . . . 97
8.2 Future work . . . 98

List of used acronmys

acronym	**meaning**
ANN	artificial neural network
FFNN	feed-forward neural network
CNN	convolutional neural network
RNN	recurrent neural network
IoU	intersection over union
SQ	superquadric
MSE	mean squared error
MAE	mean absolute error

Abstract

Title: Recovery of superquadric parameters from depth images using deep learning

Reconstruction of 3D space from 2D image data has always been a significant challenge in the field of computer vision. Simple geometric entities are used to describe larger, more complex objects or entire scenes. This representation of the environment allows an autonomous agent to manipulate and interact with it's surroundings. Superquadrics are parametric models, able to describe a wide array of 3D objects using only a few parameters, which makes them a suitable representation in such tasks. In this work, we explore the possibility of using deep learning techniques to successfully recover parameters of a single superquadric from depth images. We present a new framework, which enables us to train deep learning models able to interpret the ambiguous nature of superquadrics in general position. We propose multiple loss functions for usage in supervised and unsupervised learning scenarios. On a synthetic depth image dataset, our best CNN regression model achieves an IoU accuracy of 95% and a speedup of a factor of 240 compared to the classic iterative recovery method.

Keywords

superquadrics, parametric models, reconstruction, 3D, deep learning, convolutional neural networks, CNN, parameter recovery

Abstract

Titel: Vorhersage von Parametern für superquadrische Modelle aus Tiefenbildern mit Deep-Learning

Die Rekonstruktion von 3D-Objekten aus 2D-Bilddaten war im Bereich der Bildverarbeitung schon immer eine große Herausforderung. Dazu werden einfache geometrische Objekte verwendet, um größere, komplexere Objekte oder sogar ganze Szenen zu beschreiben. Diese Darstellung der Umgebung ermöglicht es einem autonomen Agenten, seine Umgebung zu manipulieren und so mit ihr zu interagieren. Superquadrics sind parametrische Modelle, die eine Vielzahl von 3D-Objekten mit nur wenigen Parametern beschreiben können, was sie zu einer geeigneten Darstellung für solche Aufgaben macht. In dieser Arbeit wird untersucht, ob es mithilfe von Deep-Learning-Techniken möglich ist, die Parameter einzelner Superquadrics erfolgreich aus Tiefenbildern wiederherzustellen. Dazu wurde ein Deep-Learning-Framework entwickelt, das es ermöglicht, Modelle zu trainieren, die die Mehrdeutigkeit von Superquadrics auch in einer allgemeinen Position interpretieren können. In diesem Rahmen wurden mehrere Loss-Funktionen sowohl für überwachtes als auch nicht überwachtes Lernen verwendet. Das beste CNN-Regressionsmodell erreicht eine IoU-Genauigkeit von 95%, wobei die Berechnung im Vergleich zur klassischen iterativen Reproduktionsmethode um einen Faktor von 240 beschleunigt werden konnte.

Schlüsselwörter

Superquadrics, parametrische Modelle, Rekonstruktion, 3D, Deep Learning, Convolutional Neural Networks, CNN

Povzetek

Naslov: Pridobivanje parametrov superkvadrikov iz globinskih slik s pomočjo globokega učenja

Rekonstrukcija trodimenzionalnega prostora z dvodimenzionalnih slik je že od nekdaj pomemben izziv na področju računalniškega vida. Za opis kompleksnih objektov ali celotnih scen se uporabljajo preprosti geometrijski elementi. Predstavitev okolja na takšen način avtonomnemu agentu omogoča upravljanje z vsebovanimi elementi ali pa možnost reagiranja na določene dogodke v okolici. Superkvadriki so parametrični modeli, s katerimi lahko opišemo širok nabor trodimenzionalnih objektov z uporabo majhnega števila parametrov, in so zato primerni elementi za predstavitev okolja. V tem delu raziščemo možnosti uporabe metod globokega učenja v namen uspešne pridobitve parametrov superkvadrika iz globinskih slik. Predstavimo novo ogrodje za učenje modelov globokih nevronskih mrež, ki so sposobni razbrati dvoumnost superkvadrikov v splošni poziciji. V sklopu tega dela predlagamo več funkcij napake, s katerimi lahko modele učimo na nadzorovan ali nenadzorovan način. Na sintetični podatkovni zbirki naš najbolj uspešen CNN regresijski model doseže 95% IoU natančnost in pa 240-kratno pohitritev izvajanja v primerjavi s klasično iterativno metodo.

Ključne besede

superkvadriki, parametrični modeli, rekonstrukcija, 3D, globoko učenje, konvolucijske nevronske mreže, CNN, pridobivanje parametrov

Chapter 1

Introduction

One of the pioneers of visual neuroscience, David Marr, starts his book Vision [1] with the following though: "What does it mean, to see? The plain man's answer (and Aristotle's, too) would be, to know what is where by looking. In other words, vision is the process of discovering from images what is present in the world, and where it is." The quote gives a nice insight into how the research community partitioned computer vision into two major branches: recognition and reconstruction.

3D reconstruction has therefore been one of the main problems of computer vision since it's beginnings. In the context of reconstruction, we are not only concerned with the shape and appearance of objects in our environment, but also their relative pose in relation to some origin. To describe and interpret, geometry is used as the main language. In the spirit of a bottom-up approach, the result of a reconstruction is a set of elemental geometric entities. This can include primitive volumetric shapes, voxels, or, even simpler, points, which may be further connected into a surface or mesh. All of these have their advantages and disadvantages when it comes to scene or object description. The choice of these building blocks allow us to control the complexity of a description; a trade-off between accuracy and it's ease of use. The more closely a reconstruction resembles the target object or scene, the harder it is to manage it computationally.

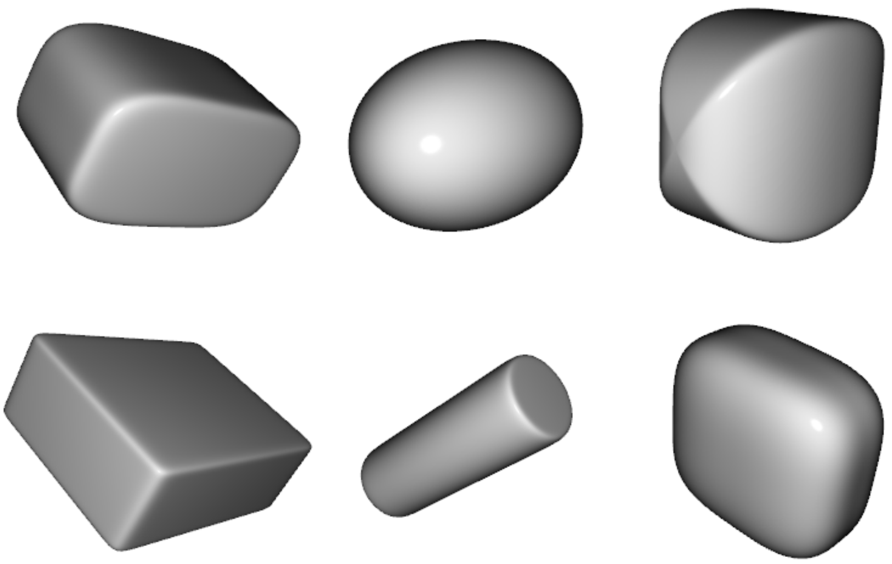


Figure 1.1: Some examples of superquadrics; With their flexible surface equation, they can represent a wide variety of different 3D shapes.

A good example of geometric primitives are superquadrics [2]. These are parametric models, capable of forming a wide array of different shapes. Another major contribution is the amount of information needed to describe them. With only five internal parameters for size (a_1, a_2, a_3) and shape (ϵ_1, ϵ_2), we can represent shapes ranging from quadrics to ellipsoids or something in between, such as cylinders. By extending the model with additional seven external parameters for translation (t_1, t_2, t_3) and rotation (q_i, q_j, q_k, q_w), we can place them within the context of a 3D environment and estimate the shape of any 3D object within it. Some examples are shown in Figure 1.1.

A reconstructed scene can be used inform an autonomous agent of it's surroundings and allow it to learn or take actions. The choice of object representation in relation to a specific task is an important decision to be made when designing a solution. With collision avoidance, maneuvering or grasping tasks, it is usually instrumental to only know and process a subset

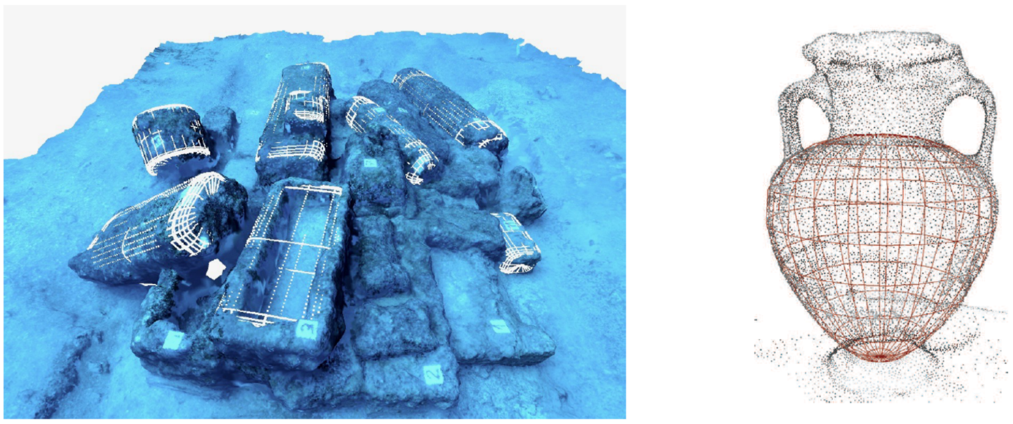


Figure 1.2: Superquadrics can be used to model complex real-world objects; On the left are modelled stone blocks and sarcophagi from a sunken Roman ship (courtesy of [4]). On the right is an amphora, modelled by a deformable superquadric (courtesy of [5]).

of possible features of an object. An exact representation, e.g., a 3D model of an object, would often be redundant and would lead to more data needed to be processed in order to infer a decision in a situation. In such tasks, superquadrics are a fitting choice for object representation. Their usefulness was already proven in practical applications, specially in various robot grasping tasks, where the shape and position if objects is undeterministic, for example, when handling mail pieces [3]. Another example is the usage in digital heritage, shown in Figure 1.2.

On the topic of sensing, depth imaging is now a relatively inexpensive method of gathering visual data from the 3D environment. Depth images are often being labeled as 2.5D data. They are essentially two dimensional images, but they encode depth, rather than color information for each pixel. If needed, they can be easily transformed into 3D data, e.g., a point cloud (a collection of points in 3D space), usually scattered across object's surface. Because of their compactness, the usage of depth images is also memory efficient. While depth imaging nowadays is inexpensive in terms of hardware requirements, the process of creating a large dataset, suitable for training deep neural networks, would be a very time consuming process. To alleviate this, synthetic datasets can be constructed to prototype on, expand or even replace a real dataset. For example, 3D mesh data or parametric shapes can be easily rendered only with the depth buffer, or in case of ray tracing, by calculating distances to nearest intersection of ray with the environment.

Successful methods for recovery of superquadric parameters already exist in literature, but are often of iterative nature and consequently, slow. The execution time can be vary depending on how many object there are in the scene or how big they are. With the recent advancements of deep learning techniques, our aim is to revisit the problem of superquadric recovery and make use of Convolutional Neural Networks (CNNs) [6, 7] to significantly speed up the recovery process, while maintaining the high accuracy of existing methods.

1.1 Challenges

While object appearance is approximated in object-space using the superquadric model, we also fit them into the context of world-space using the same external parameters. This is considered to be an issue of pose estimation. This is a complex problem due to the vagueness in describing spatial relations and especially, orientation. While a simple CNN regression model would work for other parameters, the real challenge lies in determining the rotation of the superquadric. It is an ambiguous representation of the state of an object, since multiple rotations can describe a superquadric that appears the same and holds the exact same volume as it's rotated counterpart. One example would be, if we rotate a cylinder along it's major axis. Other situation might involve a superquadric with switched axis sizes, but same rotation, thus appearing different. These examples can be seen in Figure 1.3.

Because of these symmetrical properties of superquadrics, we believe it is important to try to determine it's parameters using a more geometrically-

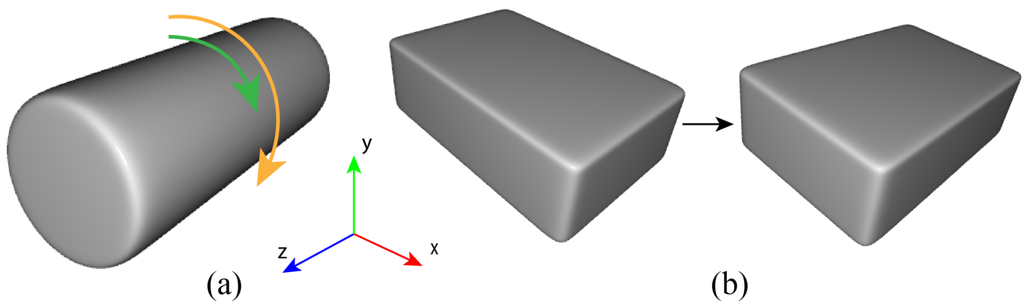


Figure 1.3: Describing objects in general position can be ambiguous; If we rotate the cylinder (a) along the z axis, the actual appearance of it wouldn't change. Both, the green and yellow rotations would yield the same result. For object (b), it is impossible to determine how it was transformed from pose on the left to pose on the right. We could either rotate it by 90° around y axis or just swap its dimensions in x and z axes.

aware learning criterion. Instead of comparing predicted parameters directly, we could instead compare fully rendered superquadrics. To achieve this, we can help ourselves by using the superquadric surface equation, which has nice characteristics and could allow us to design a new loss function. The optimization process would then be based much more on the 3D properties of the superquadrics, rather than just comparing raw numbers.

There are also challenges related to data acquisition. When capturing monocular depth images, i.e., imaging from a single perspective, we are effectively dealing with partial data. Objects are viewed only from one side and information is lost at the moment of capture. Another potential problem would be the availability of labeled data. It is not hard to generate a synthetic dataset of depth images of virtual objects. If we possess ability to render these objects, we obviously already have their geometric description in some form or another. This makes is easy to construct ground-truth labels for individual examples. However, when dealing with real-world data, we either have to make manual measurements of real-world object or that information is not available to us at all. In such situations, we could benefit greatly by using unsupervised learning, meaning there is no need for labeled data in the learning process. This is a complex problem, since we would need a differentiable loss function, that is not only capable of estimating parameters, but also reconstruction of the original input data.

1.2 Contribution

Within the scope of this Master's thesis, we make the following contributions:

- We introduce a new and improved CNN regressor, able to predict 12 parameters of a single superquadric in general position from a depth image. This is the direct upgrade to an existing model, only capable to estimate parameters of unrotated superquadrics.
- We propose a geometrically-aware loss function, which is able to train a CNN regressor to recover superquadric parameters in a supervised manner by evaluating the superquadric inside-outside function.
- We propose two additional loss functions, which are able to train a CNN regressor to recover superquadric parameters in an unsupervised manner with unlabeled data.
- We present the results and follow with an extensive analysis. We compare our methods to each other and to the state-of-the-art in superquadric parameter recovery.

1.3 Thesis structure

This work has a total of eight chapters. In Chapter 2, we begin with an overview of the related work. A brief history of reconstruction and classical methods of parametric recovery are presented first, followed by some of the current work which already makes use of CNNs. In Chapter 3, we present the mathematical background behind superquadrics, needed for further definitions and derivation of our loss functions. In Chapter 4, we make a short overview of neural networks. We define what the core building blocks are and what are some of the procedures, involved in training a neural network. We then present our experimental setup in 5. This includes describing the methodology, shared by all the experiments. We define the CNN architecture, training procedure, as well as present our dataset and metrics. In Chapters 6 and 7, we present the methodology behind our main contributions. Both approaches, supervised and unsupervised, are described in detail and for each, we also show the results and end with a discussion. Finally, we conclude this thesis with closing remarks in Chapter 8.

Chapter 2

Related work

In this section, we first cover a brief history of 3D reconstruction and volumetric models. Then, we outline the main superquadric parameter recovery approaches, achieved by classical computer vision methods. Finally, we survey work within the contemporary machine learning solutions and how they apply to computer vision tasks. Specifically, we are interested into pose-estimation and 3D recovery, with a priority on estimating objects in general position.

2.1 Early volumetric reconstruction

The idea of a generalized bottom-up reconstruction originally resulted from the advancements in perceptual psychology and neuroscience in 1960's and 70's. During that time, the visual system was being studied extensively and the ideas were transferred to the emerging field of computer vision. A mathematical model for object representation was needed and the first volumetric primitive surfaced in the 1971, called the generalized cylinder or sometimes also generalized cone [8]. These are volumes, created by parametrizing an arbitrary curve in 3D space. The parametrization would come in different forms, for example sweeping a 2D shape along a curve in 3D space.

In that period, the first theoretical vision system was introduced by David

Marr [1], sometimes called the "Marr's paradigm". He proposed a process to recover 3D information from 2D images by first extracting depth cues from the image as an interim step. Individual part-level models would then be approximated by hierarchically by using a suitable shape representation. He defined the criteria for representation effectiveness within the context of object recognition:

1. **Accessibility** – having an efficient way to compute the representation from an image.
2. **Scope and uniqueness** – all possible shape priors for a given task should be accounted for, while still maintaining the uniqueness of individual shapes.
3. **Stability and sensitivity** – having the ability to capture more general properties of a shape, but distinguishing smaller variations between different shapes.

With these guidelines in mind, Brooks presented the first implementation of Marr's theoretical system, called ACRONYM [9]. It used generalized cylinders as volumetric models and the shapes were computed directly from edge-based image features, without the intermediate step of extracting depth information. The implementation, however, showed some weaknesses of Marr's system, in practice being a lot more restrictive than the theoretical model.

The next breakthrough came in mid 80's, specifically Biederman [10] argued, that human object recognition works by assembling volumetric primitives, called geons, into larger constellations, forming complex models of the environment. Geons are a set of 36 simple 3D shapes, such as cubes, cylinders or cones, and the process was called Recognition-by-components (RBC). This was considered an upgrade to Marr's system, offering a viewpoint-invariant object recognition by analyzing curvature properties of geons. RBC-based systems were often only evaluated by interpreting hand-drawn lines. The processing of real-word images, outside of a controlled lab environment, lead to a combinatorial explosion when interpreting image features [11].

The generalized cylinder remained relevant as a volumetric representation well into 1990's. Efforts to segment and extract them from intensity [12] or depth images [13] were initially successful, however, the parametric model was complex which hindered further advancements. To find an alternative to generalized cylinders and geons, a search for better volumetric models was well under way. Some examples of these proposals include implicit polynomials [14], blobby models [15], symmetry-seeking models [16], Fourier surfaces [17] and finally, superquadrics [18] and various derivatives, such as hyperquadrics [19]. For more extensive overview of volumetric-based reconstruction the reader is referred to [20, 11].

2.2 Superquadric parameter recovery

The formulation of superquadrics can be traced back to 1910, when mathematician Gabriel Lamé [21] first described parametric curves, now known as Lamé curves. A subset of these were superellipses, a generalization of the ellipse, capable of modeling many symmetric shapes, ranging from rectangles, ellipses, rhomboids to various concave 4-armed stars. In 1981, Barr created an extension to quadric surfaces and parametric patches used commonly for computer graphics in that period. He introduced superquadrics [2], 3D parametric objects, capable of modeling many desired shapes only with a few interactive parameters. Pentland first brought superquadrics to the attention of computer vision community [18]. He proposed them as the volumetric model of choice due to their simplicity and universality. He devised an analytical approach, where the relationship between surface normal and the texture/contour of the object in intensity image is considered. Except for some synthetic images, the approach was not successful [20]. Pentland also proposed a brute-force search of the parameter space [22] using parallel computing.

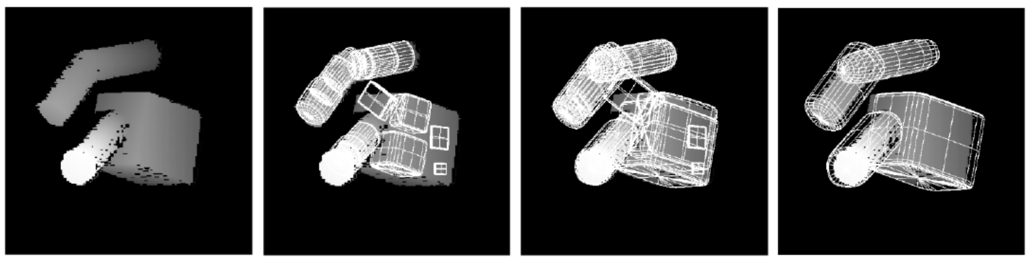


Figure 2.1: The joint segmentation and recovery of multiple superquadrics, devised by [29]. Here shown is the fitting process; On the left is the original depth image. Initial seed superquadrics are then gradually expanded until an MDL hypothesis is reached on the right.

In 1987, Bajcsy and Solina [23] proposed a least-squares minimization process for depth images, which they formulated in detail a few years later [24]. This is is still considered as the basis for current state-of-the-art superquadric recovery solutions. Boult and Gross [25, 26] have proposed using the superquadric radial distance as the fitting function, however, the resulting recovered superquadrics were visually indifferent to those, recovered by the inside-outside function. Additionally, more computation is needed to calculate the radial distance, so the inside-outside function was used more extensively by researchers. Others have approached the recovery procedure from another perspective, for example, by using genetic algorithms [27]. Various extensions to superquadrics were also proposed [19, 28], but ultimately still use the same iterative process, which constrained further development in this direction. The original method from Solina and Bajcsy [24] was later expanded by Leonardis and Solina [29], where they achieved a joint segmentation and recovery of multiple superquadrics on the basis of Minimum Description Length (MDL) principle [30]. The fitting process is shown in Figure 2.1. The fitting function, however, remained of iterative nature. An extensive survey of the field was also covered by [20].

2.3 Deep learning in 3D

In the last decade, a new machine learning paradigm emerged, which enabled various fields of computer science to advance further than before. Deep learning revitalized many areas, where research stalled or was for a time devoid of fresh ideas. Especially, the field of computer vision gained significant ground in numerous recognition, detection, semantic segmentation and reconstruction efforts because of the performance of CNNs. Different from the classical approaches described above, we outline here the main contributions of deep learning models in 3D and how they might be useful for the task of superquadric parameter recovery.

2.3.1 Structure and volumetric representation

Neural networks for 3D reconstruction are employed in various configurations and use different building blocks and shape representations. What these networks have in common, however, is handling of 3D data in some form or another. Most of the 3D reconstruction research relies heavily on the use of encoder-decoders, a type of deep neural architecture, which first encodes information in a latent vector, then uses an up-sampling decoder for data reconstruction.

Wu *et al.* [31] were the first to introduce the idea of using discretized volumetric grids for spatial representation. Their encoder network consists of 3D convolutional layers and takes as input a voxelized depth image. The output is a latent vector, used for object recognition and to determine the next best view. Another alternative is to encode 2D or 2.5D data directly using 2D convolutional layers, which is the standard way of processing images with CNNs. For example, MarrNet [32] takes as input only a single RGB image and then as an intermediary step estimates surface normals, depth and silhouette of the object. This is considered to be a 2.5D sketch according to the Marr paradigm [1]. The intermediate representations are then encoded into a latent vector, which is followed by a 3D decoder that outputs an occupancy grid of size 128^3. It is obvious, that 3D encoders have a far greater impact on memory consumption and performance than 2D encoders. An increase of the resolution of the voxelized space results in cubic growth of required resources. Nevertheless, volumetric grids allow for storage of true 3D data of arbitrary resolution. In contrast, 2D images only contain a single perspective, leading to object self-occlusion and loss of information.

Two main versions of volumetric grids exist in relation to 3D reconstruction: occupancy and distance grids. Occupancy grids can be divided into binary and probabilistic counterparts. In binary grids, we mark individual voxels whether they are a part of target shape or not [32, 33]. With probabilistic grids, a probability for each voxel is given instead. For example, in [34], a sigmoid is applied element-wise after a 3D decoder processes the latent vector. Distance grids can be thought of as discretized 4D functions. Each voxel encodes the information about the distance from it's position to a reference point. One example are signed distance functions (SDF), which are positive when a point lies outside of the object and negative if it lies inside. In contrast, Park proposed DeepSDF [35], a network which learns a spatial classifier, enabling continuous estimation of the SDF. This is different to the volumetric approach. Rather than regressing values for each element of a discretized distance space, the authors learn a parametric spatial boundary classifier, which returns the value of SDF at a specific continuous point in space.

Objects can also be represented by an unordered set of 3D points, most often sampled from their surface. This is an efficient way of representation, since only the object boundary is used in computation and not the whole 3D space. The main issue, however, is the structure of data, which can't be processed nicely with a convolutional layer. To store point-clouds, grids of size $N \times 3$ or $H \times W \times 3$ are used. Using a orthographic view, a depth image could also be considered a point-cloud with the Z coordinate encoded in pixel intensity. Point-based architectures only recently started to emerge with some specialized architectures. Qi *et al.* [36] were the first to create a geometrically-aware architecture, capable ob encoding unordered 3D points. They use it to segment and classify individual object in the scene. Alternatively, [37] proposed a point-based decoder to predict point-clouds from single RGB images.

2.3.2 Pose estimation

Pose estimation is the task of determining the rotation and position of a particular object in relation to world center. It can either be a standalone task or often, an auxiliary estimation in a larger reconstruction system. This is not a trivial task due to the ambiguous and periodic nature of rotation description. Zhu *et al.* [38] use a standard encoder-decoder architecture to reconstruct volumetric data, however, they simultaneously train a pose regressor. Miao *et al.* [39] uses Mean Squared Error (MSE) to train six separate regressors for all parameters of a general pose; position and rotation. They did, however, use non-complex medical X-ray images, generally taken in a constrained environment. Methods where the loss function is based more on geometric representation are generally more successful. For example, Xiang *et al.* [40] minimize the distance between points on the surface of rotated objects to predict rotation. They use depth cues and semantic segmentation information from RGB to segment and retrieve individual objects. To calculate the difference between ground-truth and predicted pose, surface point-set-based distances are often used, such as Chamfer distance [37] or Earth Mover's distance [41].

Most commonly, pose estimators are regression models, which output continuous values as pose descriptors [39, 40]. This presents no issues for translation parameters, however, rotation description has some intrinsic limitations. First, rotation is periodical, e.g. by increasing the angle of rotation in one axis, we eventually end up in the initial pose. This means that a hypothetical loss function, comparing ground-truth and predicted pose, would have infinite global optimums. The choice of global optimum, to which the model converges, is then dependant on the initialization of model parameters.

Another limitation is the description itself. Euler angles are known to suffer from gimbal lock [42]; a loss of one degree of freedom. Rotation quaternions have the unit norm constraint, which makes regression non-trivial [38]. To ensure better initial estimation for a regressor, various activation functions are used to limit the prediction in a specific range. For example *sigmoid* and *tanh* activations put the predicted value in ranges $[0, 1]$ and $[-1, 1]$, respectively [38]. Vector normalization is used as a non-linearity for quaternions [43, 44].

2.3.3 Recovery of volumetric primitives

Work was also done on the bottom-up concept by recovering individual volumetric primitives from 3D data, which relates to our goal the most. In 2017, Tulsiani *et al.* [43] proposed a method to learn shape abstractions using primitive shapes. They used cuboids as their geometric primitive of choice and fitted them to triangle meshes. The result was a joint segmentation-recovery pipeline, where multiple primitives were successfully fitted on parts of the mesh at once. In a recent paper by Paschalidou *et al.* [44], the authors adapted this pipeline to use superquadrics instead of cuboids, achieving a significantly smaller fitting error due to the wide range of shapes superquadrics can approximate. A later expansion to this work [45] proposes a system for hierarchical unsupervised recovery of superquadrics. Both authors used ShapeNet [46] as their dataset and thus trained only on it's limited set of object categories. We think a more generalized approach is needed in order to model objects in a truly unconstrained fashion. Superquardics were also being recovered from point clouds by Slabanja *et al.* [47].

Based on the idea of combining 3D visual data and CNNs, we recently started working towards a superquadric recovery method, which would eliminate this iterative constraint and offer a deterministic and possibly real-time execution speeds. We presented the idea in form of a preliminary study [48], where we trained a CNN regressor to estimate 8 parameters (size, shape and position) for a superquadric in an isometric view. We were successful in achieving a faster execution time and also improving parameter estimation in comparison to the classic method. We showed that a model, trained with synthetic data could be used for real-world examples. We did not, however, managed to estimate rotational parameters using this model. The goal of this thesis is thus to expand on the mentioned paper and introduce rotation prediction.

Chapter 3

Superquadrics

In this chapter, we explain what superquadrics are and why they can be used in practice. We describe some crucial definitions and present the mathematical foundation, on which we will build our methods later on. First, we introduce where the idea of superquadrics came from, then we present the implicit equation and some of it's properties, that make superquadrics a good choice for object estimation.

3.1 Superellipses

Superellipses are a subgroup of Lamé curves [21], which are defined by an implicit equation

$$\left(\frac{x}{a}\right)^m + \left(\frac{y}{b}\right)^m = 1, \tag{3.1}$$

where m is a rational number. If we add an additional constraint

$$m = \frac{2p}{2q+1}; p, q \in \mathbb{Z}^+, \tag{3.2}$$

meaning the numerator is even and denominator is odd, we get superellipses. By having an even number as the numerator, the value of each bracket becomes positive, effectively mirroring the function of the 1st (+,+) quadrant in all other quadrants of the Cartesian coordinate system. Parameters a and

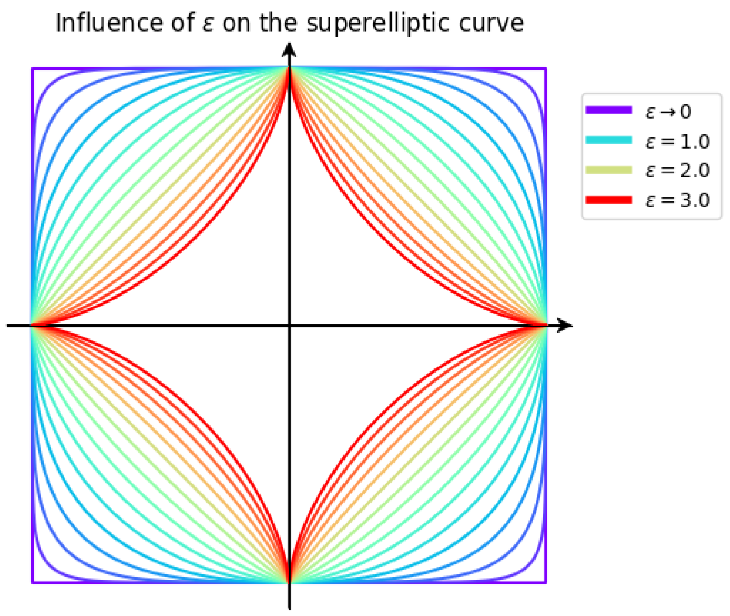


Figure 3.1: The effect of parameter ϵ on the superelliptic curve. When ϵ approaches 0, the resulting curve forms a square. By increasing it towards ∞, the curvature turns toward the coordinate system origin in forms many useful shapes in the process.

b then represent the size of the resulting connected and symmetric superelliptic curve in x and y axis, respectively. We can simplify this further by first raising the terms to the second power, and defining a new variable $\epsilon = \frac{2q+1}{p}$, which replaces the residual of m. Consequently, the parity of the new shape parameter ϵ becomes unconstrained. Superellipses can then be described by the implicit equation

$$\left(\frac{x}{a}\right)^{\frac{2}{\epsilon}} + \left(\frac{y}{b}\right)^{\frac{2}{\epsilon}} = 1, \tag{3.3}$$

where ϵ is a real positive number. In Figure 3.1, we can observe the many different shapes that parameter ϵ allows us to approximate. When increasing ϵ from near 0 to ∞, we get in order: a square ($\epsilon \to 0$), squircle ($0 < \epsilon < 1$), circle ($\epsilon = 1$), rounded rhombus ($1 < \epsilon < 2$), rhombus ($\epsilon = 2$) and then it forms into a concave star ($\epsilon > 2$), finally resembling a cross when $\epsilon \to \infty$.

3.2 Parametric equation and superellipsoids

The implicit equation is an intuitive description for the superellipse, however it only allows us to check, whether a point $[x, y]^T$ lies on it's curve. To find out which points lie on the curve directly, or rather, sample points from the curve, we can transform equation 3.3 into the parametric form

$$s(\eta) = \begin{bmatrix} a\ cos^{\epsilon}(\eta) \\ b\ sin^{\epsilon}(\eta) \end{bmatrix},\ 0 \leq \eta \leq 2\pi. \tag{3.4}$$

This way x and y coordinates of the curve are calculated for any angle η w.r.t the origin. The granularity of such a curve is controlled by setting the step size when computing for angle η.

The transition from 2D curves to 3D surfaces can be achieved by the spherical product of two curves $p(\eta, \theta) = s_1(\eta) \otimes s_2(\theta)$. In case of superellipses, the resulting equation is a parametric form of a superellipsoid surface:

$$p(\eta, \theta) = \begin{bmatrix} a_1\ cos^{\epsilon_1}(\eta)\ cos^{\epsilon_2}(\theta) \\ a_2\ cos^{\epsilon_1}(\eta)\ sin^{\epsilon_2}(\theta) \\ a_3\ sin^{\epsilon_1}(\eta) \end{bmatrix},\ 0 \leq \eta \leq 2\pi,\ 0 \leq \theta \leq \pi. \tag{3.5}$$

The size of the superellipsoid is determined by three parameters (a_1, a_2, a_3), each spanning half of it's respective axis. Shape is determined by two curvature parameters (ϵ_1, ϵ_2). As with superellipses, ϵ_1 determines the curvature of the xy plane (cross-section at $z = 0$). Similarly, ϵ_2 determines the curvature of the perpendicular plane to the xy plane, which also contains z axis (either $x = 0$ or $y = 0$). An example of such a superellipsoid can be seen in Figure 3.2. There, a combination of a circular curve with a rectangular curve results in a superellipsoid, which closely resembles a cylinder.

Finally, not all superquadrics are superellipsoids. Superquadrics by definition are a broader family of 3D surfaces, which besides superellipsoids also contain supertoroids and superhyperboloids. In this thesis, we are essentially dealing only with superellipsoids, however we are using the term superquadrics, since in general literature, superquadrics have a synonymous meaning to superellipsoids.

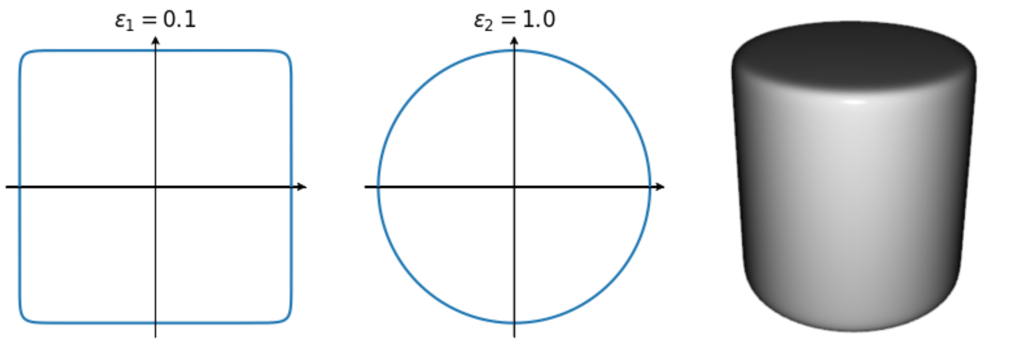


Figure 3.2: Spherical product of two superelliptic curves with parameters $\epsilon_1 = 0.1$ and $\epsilon_2 = 1.0$. The resulting superellipsoid has a cylinder-like shape.

3.3 Superquadric inside-outside function

By converting the parametric form for ellipsoids, given by Eq. (3.5) back to the implicit form, we get:

$$\left(\left(\frac{x}{a_1}\right)^{\frac{2}{\epsilon_2}} + \left(\frac{y}{a_2}\right)^{\frac{2}{\epsilon_2}}\right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{z}{a_3}\right)^{\frac{2}{\epsilon_1}} = 1. \tag{3.6}$$

This is the implicit superquadric equation in object-space. Size parameters (a_1, a_2, a_3) represent the size of the superquadric in x_s, y_s and z_s axis, respectively, while shape parameters (e_1, e_2) represent the roundness of vertical and horizontal edges. The solution of this equation is a set of points $\boldsymbol{p} = [x, y, z]^T$, which lie on the surface of the superquadric.

We can convert the implicit equation to a function $F : \mathbb{R}^3 \to \mathbb{R}^+$ and evaluate it for a specific point in space $\boldsymbol{p}$:

$$F(\boldsymbol{p}) = F(x, y, z) = \left(\left(\frac{x}{a_1}\right)^{\frac{2}{\epsilon_2}} + \left(\frac{y}{a_2}\right)^{\frac{2}{\epsilon_2}}\right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{z}{a_3}\right)^{\frac{2}{\epsilon_1}}. \tag{3.7}$$

This is also called the inside-outside function. As the name already suggests, the function can be used to easily determine, where a point lies in relation to the superquadric. The result of the inside-outside function is a non-negative real number. If the points lies inside the superquadric then $F(\boldsymbol{p}) < 1$, if it lies outside of the superquadric, then $F(\boldsymbol{p}) > 1$ and as already stated, $F(\boldsymbol{p}) = 1$ when the point lies on the surface. The value of F at the center of the superquadric is exactly 0. The function is continuous w.r.t. the distance from the center. It's value will increase exponentially the further away from superquadric center a point will be. From this point on, any mention of the "superquadric function" will refer directly to the inside-outside function.

3.4 Superquadrics in general position

Until now, we defined the superquadric function in object-space. However, in practice, objects are often transformed into a world coordinate system. This includes translation and rotation of an object. These operations are not commutative, so a specific order is needed. In our case, we first rotate the superquadric around the origin, then translate it. The superquadric function implies that the point $\boldsymbol{p} = [x, y, z]^T$ is given in relation to the object, meaning it is object-centered, however we are dealing with world-centered coordinates. This is why we first transform world-space coordinates into object-space coordinates and then calculate the superquadric function.

In general, there is a minimum of six additional external parameters for the superquadric function. We have three parameters for translation (t_1, t_2, t_3) and three for rotation (α, β, γ) of the superquadric.

3.4.1 Rotation representation

The number of rotational parameters can vary if we use a different type of representation. The simplest way to describe rotation is by using Euler angles, denoted as (α, β, γ). Usually, a reference coordinate space is rotated in a specific sequence of intrinsic rotations, where the magnitudes are given by the three Euler angle parameters. However, this method is prone to gimbal locking, a state of the system, where two axes end up in a parallel configuration, leading to loss of one of the degrees of freedom. One solution is to change the description of rotation.

In our case, we use unit quaternions $\boldsymbol{q} = [q_i, q_j, q_k, q_w]^T$, also called versors, to represent rotation. This is a compact representation, needing only one more parameter. It does not suffer from gimbal lock and multiple rotations can be more easily composed in comparison to Euler angles.

To apply a quaternion rotation, e.g. rotate a point $\boldsymbol{p}$ around it's origin, we can derive a rotation matrix $\boldsymbol{R}_q$ from a quaternion $\boldsymbol{q} = [q_i, q_j, q_k, q_w]^T$:

$$\boldsymbol{R}_q = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} 1-2(q_j^2+q_k^2) & 2(q_iq_j - q_kq_w) & 2(q_iq_k + q_jq_w) \\ 2(q_iq_j + q_kq_w) & 1-2(q_i^2+q_k^2) & 2(q_jq_k - q_iq_w) \\ 2(q_iq_k - q_jq_w) & 2(q_jq_k + q_iq_w) & 1-2(q_i^2+q_j^2) \end{bmatrix}. \tag{3.8}$$

Then, we simply calculate the product $\boldsymbol{R}_q\boldsymbol{p}$ to get the new location of point $\boldsymbol{p}$, rotated by quaternion $\boldsymbol{q}$ around it's origin. This transition from quaternion to rotation matrix is unique and so is the reverse operation. A point could also be rotated using a quaternion directly, however, this method provides a more intuitive approach.

3.4.2 Transformation matrix

The superquadric function implies superquadric centered coordinates, so to place a superquadric into the world, we need to derive the appropriate transformation matrix. We know how to transform points from object-centered coordinate system $\boldsymbol{p}_s = [x_s, y_s, z_s]^T$ into world-space points $\boldsymbol{p}_w = [x_w, y_w, z_w]^T$ with the following homogeneous matrix $\boldsymbol{Rt}$:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \boldsymbol{Rt} \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}, \quad \boldsymbol{Rt} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.9}$$

With this matrix, a homogeneous point is first rotated around the coordinate system origin and then translated. To reverse the transformation and compute $\boldsymbol{p}_s$ from $\boldsymbol{p}_w$, we can simply invert the transformation matrix by

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \boldsymbol{Rt}^{-1} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}, \quad \boldsymbol{Rt}^{-1} = \begin{bmatrix} r_{11} & r_{21} & r_{31} & -(t_1 r_{11} + t_2 r_{21} + t_3 r_{31}) \\ r_{12} & r_{22} & r_{32} & -(t_1 r_{12} + t_2 r_{22} + t_3 r_{32}) \\ r_{12} & r_{23} & r_{33} & -(t_1 r_{13} + t_2 r_{23} + t_3 r_{33}) \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.10}$$

The inverse $\boldsymbol{Rt}^{-1}$ changes the order of operations, where the point is first translated into the local coordinate system of the superquadric and then rotated around it's origin. Finally, we can evaluate the superquadric function for world-centered points $F(\boldsymbol{Rt}^{-1}\boldsymbol{p}_w)$. If we expand the new inside-outside function in general space, we get

$$\begin{aligned} F(\boldsymbol{p}_w) = \Bigg(& \left(\frac{x_w r_{11} + y_w r_{21} + z_w r_{31} - t_1 r_{11} - t_2 r_{21} - t_3 r_{31}}{a_1}\right)^{\frac{2}{\epsilon_2}} + \\ & + \left(\frac{x_w r_{12} + y_w r_{22} + z_w r_{32} - t_1 r_{12} - t_2 r_{22} - t_3 r_{32}}{a_2}\right)^{\frac{2}{\epsilon_2}} \Bigg)^{\frac{\epsilon_2}{\epsilon_1}} + \\ & + \left(\frac{x_w r_{13} + y_w r_{23} + z_w r_{33} - t_1 r_{13} - t_2 r_{23} - t_3 r_{33}}{a_3}\right)^{\frac{2}{\epsilon_1}}. \end{aligned} \tag{3.11}$$

From now on, the evaluation of the superquadric function will simply be denoted as $F(\boldsymbol{p})$, though the reader should note that world-centered points $\boldsymbol{p}_w$ are being used. In total, we use 12 parameters

$$\lambda = (a_1, a_2, a_3, \epsilon_1, \epsilon_2, t_1, t_2, t_3, q_i, q_j, q_k, q_w), \tag{3.12}$$

to evaluate the superquadric function $F(\boldsymbol{p}; \lambda)$.

3.5 Superquadric vocabulary

By choosing an appropriate parameter space, we are effectively choosing a vocabulary of shapes, that the superquadric equation can represent. This is particularly notable with shape parameters (ϵ_1, ϵ_2). For use in computer vision applications, shape is usually bounded: $0 < \epsilon_1, \epsilon_2 < 1$ [20]. This only includes convex shapes out of all possible superquadrics. For our purposes, we further limit the upper bound to 1. This range includes shapes like spheres, cuboids, cylinders and everything in between. This is shown in Figure 3.3. Note, that at $\epsilon = 0$, the inside outside function is singular and numerical instability can develop when $\epsilon \to 0$. Due to this, our final range for shape parameters becomes: $0.1 \leq \epsilon_1, \epsilon_2 \leq 1$.

The vocabulary is also determined by size parameters (a_1, a_2, a_3). By constraining these to a range $[a_{min}, a_{max}]$, we effectively limit the maximum ratio between two axes of an object to a_{max}/a_{min}. The bigger this ratio is, the more diverse set of objects can be represented.

3.6 Visualization

A superquadric and it's inside-outside function can be rendered and visualized in different ways. We first outline the process for rendering the superquadric surface. Then, we show how the evaluated inside-outside function can be visualized in 3D space.

3.6.1 3D surface rendering

To render a superquadric on a Graphics Processing Unit (GPU), we have to first create a 3D mesh. We begin by sampling points from the superquadric surface with the parametric equation 3.5. We first create a 2D array of polar coordinates by sampling equally-spaced points from ranges $0 \leq \eta \leq 2\pi$ and $0 \leq \theta \leq \pi$. We evaluate the parametric equation on sampled points, which results in a set of 3D points on superquadric surface in object-space. The exponentiation in Eq. (3.5) is in fact a signed power function:

$$x^p = sign(x)|x|^p, \tag{3.13}$$

where $sign(.)$ returns -1 if x is negative and 1 otherwise. This is done to assure, that the exponentiation does not result in complex numbers when x

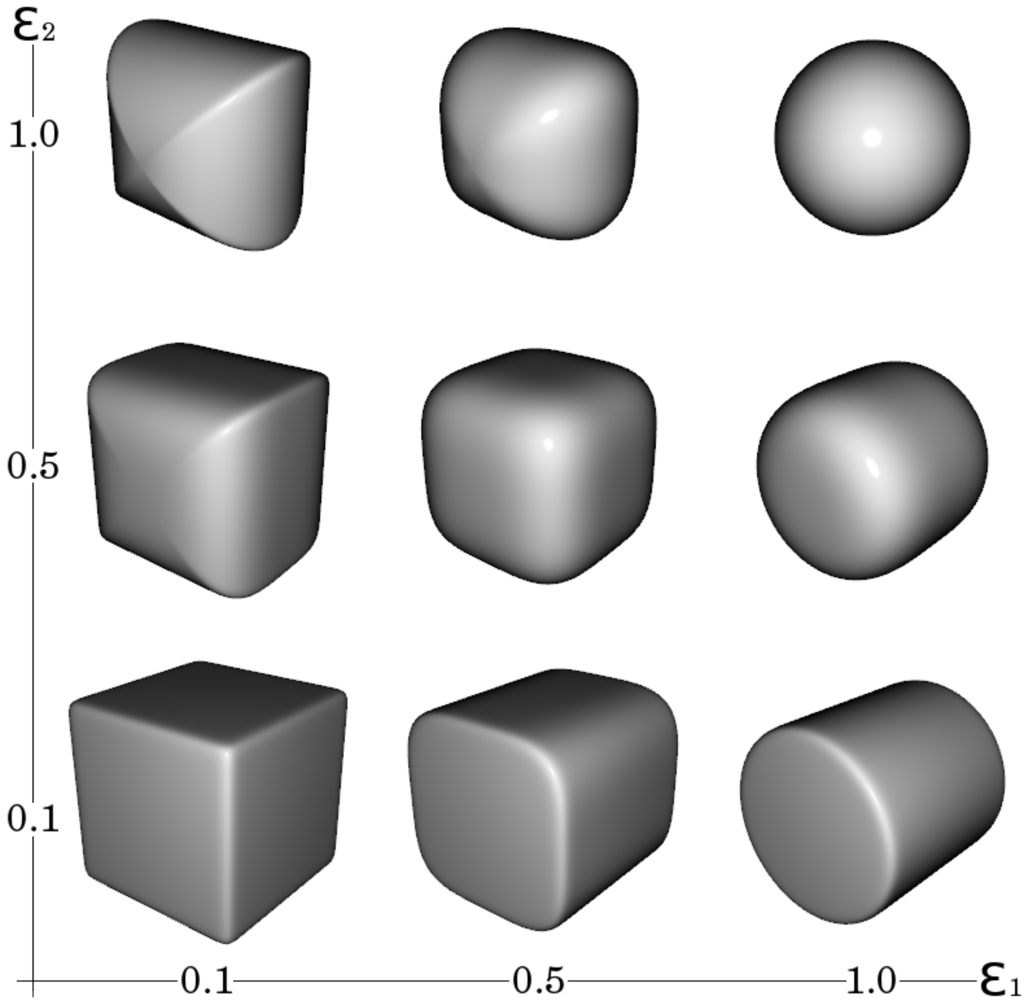
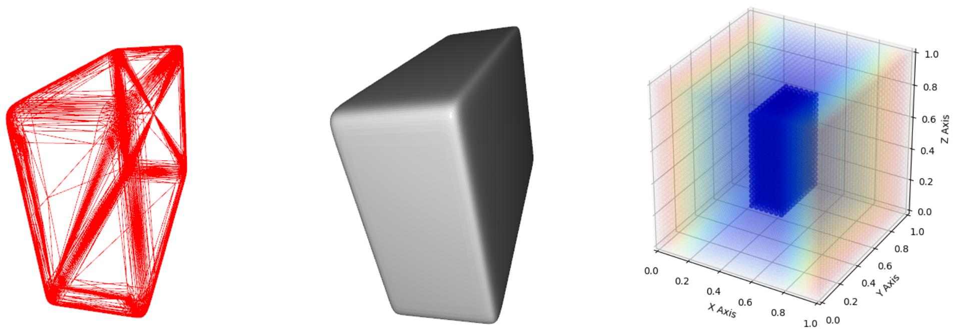


Figure 3.3: The vocabulary of superquadrics; We limit the range of shape parameters ϵ to [0.1, 1]. By doing this we avoid numerical instabilities at lower values of ϵ and at the same time the occurrence of concave shapes at higher values of ϵ.



(a) **(b)** **(c)**

Figure 3.4: Visualization of superquadric surface and the inside-outside function: (a) a wiremesh of superquadric surface, which we use to denote the ground-truth superquadric. (b) Phong lightning shaded superquadric, used to represent predicted superquadrics. (c) a scatter plot of the superquadric hypersurface. The color represents the value of inside-outside function and opacity is set to 1 for points inside the superquadric. All examples represent a superquadric with the same parameters.

is negative. To transform derived points to world-space, we multiply them with transformation matrix $\boldsymbol{Rt}$, shown in Eq. (3.9).

The next step would be to transform the point-set into a connected triangle mesh. We use an implementation of the *Qhull* algorithm [49] for quick convex hull computation, since our vocabulary of superquadrics only includes convex shapes. Alternatively, any similar algorithm, which computes mesh from points can be used.

We then render the resulting mesh on the GPU in two ways. One is by rendering a wire-frame model, which only renders triangle lines. This mode is useful, when we want to see the density of sampled points on the surface. The other mode is achieved by rendering and shading the entire mesh including triangle faces. In this case, we make use of Phong lightning, which can be computed using surface normals. Both modes of rendering are shown in Figure 3.4.

3.6.2 Visualization of the inside-outside function

The inside-outside function is a hypersurface, which makes it difficult to visualize. It's a 4D function which we have to visualize in 3D space. One way is by using color coding. We know how to visualize points in 3D space by using a scatter plot. Each point in the plot is then colored according to the value of inside-outside function at that position.

We can further increase the visibility of such a visualization by controlling the opacity of individual points. We set opacity of points inside the superquadric to 100% and the opacity of points outside to 15%. This configuration provides us with a clear view of the superquadric and at the same time the increasing values of the inside-outside function around it. This can be observed in Figure 3.4.

Chapter 4

Neural networks

In this chapter, we give a coarse overview of artificial neural networks, particularly for the purpose of recovering superquadric parameters. In accordance to the main theme, we mostly cover convolutional neural networks and provide the reader with some fundamental definitions, on which we build our method later on. We first briefly present the general term of ANNs, and then define learning. Last, we take a look at CNNs which are used predominantly in the field of computer vision.

4.1 Artificial neural networks

The study of neurons and neural networks is a broad scientific endeavour, a multidisciplinary research which combines sciences, such as biology, psychology and neuroscience. By trying to mimic these processes, the domain of machine learning adapted many concepts found in real life and a new subgroup was founded. Artificial neural network is a general term, used to describe a computing system, which is loosely analogous to the neural structure and operations, found in our brains. This paradigm combines different data structures, algorithms and other conventions and now forms a well-established way of learning statistical data representations.

4.1.1 Neuron

The artificial neuron was conceived in response to new neuroscientific achievements in 1940' [50]. It tries to mimic the concepts of a biological neuron. Human ability to remember and recall was at that time contributed the the neural structure in our brain. This is the reason that a mathematical formulation was interesting for the field of artificial intelligence. We can find different kinds of biological neurons in different parts of our brain, nevertheless, they all share the same structure:

1. **Dendrites** act as an input to the neuron and transmit incoming signal to the nucleus.
2. **Axon** transmits electrical signal from the neuron towards synapses.
3. **Synapses** then transfer the output signal to other neurons.

The natural neuron has a so-called all-or-none response, meaning that the neuron gets excited only when the stimulus crosses a certain threshold. There is no magnitude of excitement based on the potential input, it's either excited or it's not. These findings contributed to formulation of a mathematical model of the neuron: the artificial neuron. It is defined as a weighted sum of inputs:

$$\boldsymbol{y} = g(\sum_i x_i w_i + b_0), \tag{4.1}$$

where w_i are weights, x_i are inputs, b_0 is the bias and g is an activation function. If we model the all-or-none activation, similarly to a biological neuron, then g is a step function:

$$g(x) = \begin{cases} 1 & x \geq t \\ 0 & x < t \end{cases}, \tag{4.2}$$

where t is a specific threshold. It has to be noted, that this model is abstract, simplified and only loosely based on the real biological neuron. Any mentions of neurons or neural networks from this point on will refer to artificial neurons and artificial neural networks (ANNs), respectively, except if stated otherwise.

4.1.2 Network topology

Any configuration of arbitrarily connected neurons is by definition a neural network. These can be further classified, depending on certain properties of the topology. Two of the major groups are Recurrent Neural Networks (RNN) [51, 52] and Feed-Forward Neural Networks (FFNN) [53, 54]. RNNs are connected with cyclic directed graphs, where neurons can propagate data forwards, backwards or have loop-back connections. In contemporary research, these are mostly used to learn the distribution of sequenced data, e.g., text, speech, time-series, etc. By allowing cyclic connections, temporally-dependant information can be memorized by the network.

FFNNs are more constrained, since neurons can only distribute data forward. They are usually grouped into layers, where each layer has an arbitrary number of neurons. A generalized type of FFNN is called a **perceptron** [53]. These can be divided into two groups:

- **Two-layer perceptron** only has an input layer and an output layer. The input layer only propagates the values forward to the output layer and does not compute anything, so some also call it the single-layer perceptron. The output layer then calculates a weighted sum of inputs and passes the result through an activation function.

- If we add one or more hidden layers between the input and output layers, we create a **multi-layer perceptron** (MLP). By having multiple layers of nonlinear neurons stacked together, MLP has the ability to model highly nonlinear data. In theory, such a network can approximate any nonlinear function. If the MLP has three or more hidden layers, the network is considered as a deep neural network. Alternatively, the network is shallow.

In general, we define a neural network as a function $y = f(x; \theta)$, where x is the input, y is the output of the neural network, f is a composition of functions $f = f_1 \circ f_2 \circ \cdots \circ f_n$, where each function represents an individual layer, and θ are the weights of individual layers, also called network parameters.

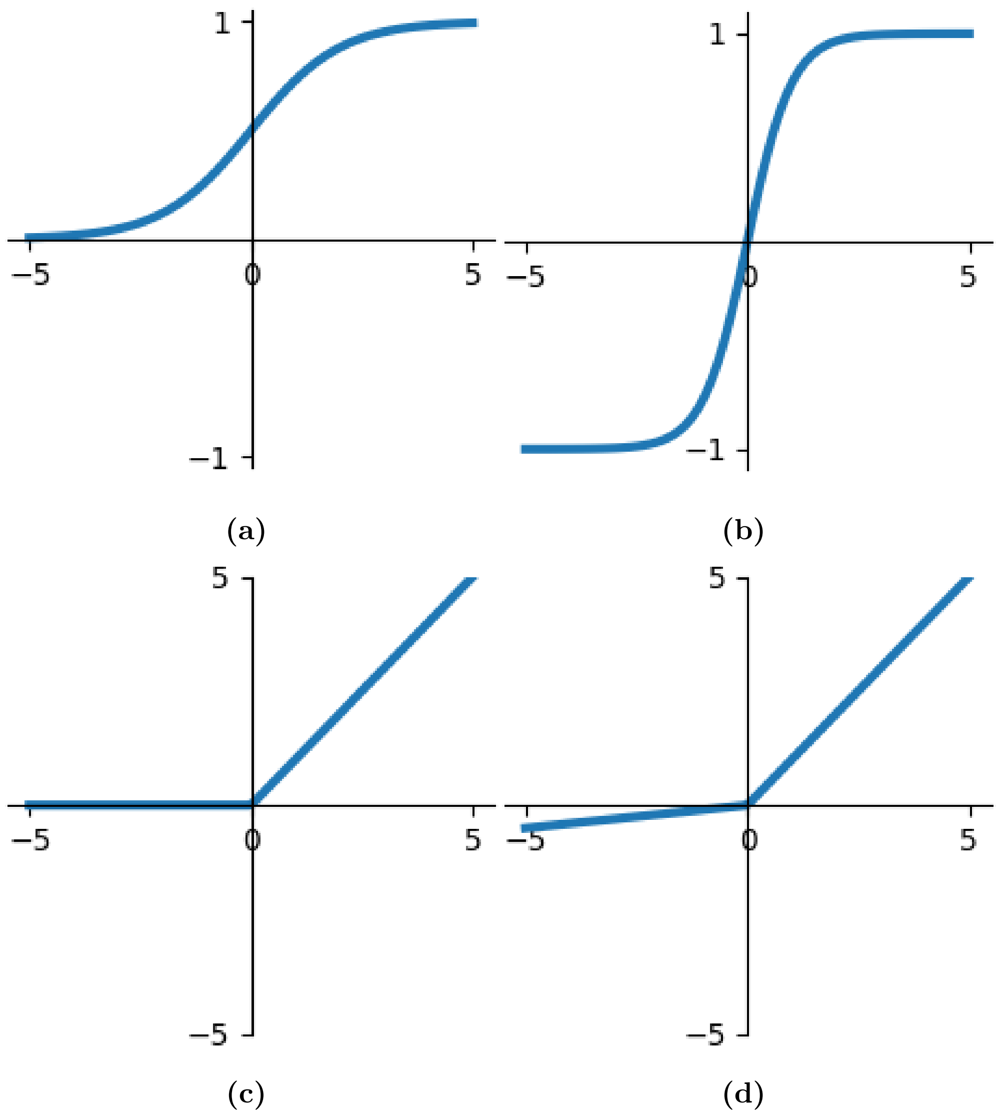


Figure 4.1: The visualization of various activation functions; (a) sigmoid, (b) tanh, (c) ReLU and (d) Leaky RelU.

4.1.3 Activation functions

With neural networks, the activation function is the main reason behind the ability to model highly nonlinear data. There are many options we can choose, but each has it's advantages and uses. These functions have to be designed with multiple criteria in mind. An activation function has to be differentiable to compute gradients and update network weights. This mechanism will be further discussed in Section 4.2. It is also advantageous to have a function, which is not expensive to compute. We group them accordingly to their popular functionality in the contemporary literature. We also visualize them in Figure 4.1.

Hidden layer activation

For hidden layers, we can use the *Rectified Linear Unit* (*ReLU*) [55] function , which is defined as

$$ReLU(x) = max(x, 0). \tag{4.3}$$

The rectifier is a simple, piece-wise function, which can be efficiently computed. For this reason it is suitable for usage in hidden layers, since hidden layers usually represent the majority of neurons in a deep neural network. It is also differentiable, however, the derivative when $x < 0$ is 0, which leads to the problem of **vanishing gradient** in a neural network. When this happens, some of the neurons become inactive and don't contribute to the learning anymore. Consequentially, the model looses capacity. One of the possible solutions is to use a **Leaky ReLU** [56] function:

$$LeakyReLU(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}, \tag{4.4}$$

where α is a small constant. This enables a small gradient even when $x < 0$, preventing neuron shutdown, while retaining the nonlinear properties.

Output activation

Activation in the output layer are different depending on the type of output we want to predict. While the type of the function influences output range, it is also dependant on the type of data distribution we want to model. The justification for these functions is based on the maximum likelihood estimation (MLE) method, used in statistics for estimating distribution parameters. For this reason the following function are frequently used in regression tasks.

A linear activation is simply an identity mapping:

$$Linear(x) = x. \tag{4.5}$$

It's range is $[-\infty, \infty]$, which makes it suitable for regressing arbitrary continuous parameters. Other useful functions are sigmoid functions. The *logistic sigmoid* is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4.6}$$

It's range lies in $(0, 1)$, which makes it suitable for modeling a Bernoulli distribution, for example in binary classification tasks. An improvement to the logistic sigmoid is a hyperbolic tangent, also called *tanh*:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1. \tag{4.7}$$

This is a shifted and scaled version of the logistic sigmoid and offers bigger gradients around $x = 0$, since it's range is in $(-1, 1)$, leading to faster convergence during training. Nevertheless, both of these functions suffer from the saturation of gradients when the output is large (in negative or positive direction).

4.2 Learning

Learning is an abstract concept within the context of neural networks. It is the process of acquiring knowledge, in our case, by analysing a distribution of examples. These experiences can then be used to make assumptions about unseen examples from the same distribution. We first define what is an optimization problem, then explain what means to learn and finally, look at the different types of learning.

4.2.1 Definition of learning

Training a neural network is an optimization problem P, which we can define as a tuple $P = (X, Y, f, \mathit{opt}, \mathcal{L})$, where:

- $\boldsymbol{X}$ is a set of training examples. These are passed as input to the first layer of the neural network.
- $\boldsymbol{Y}$ is a set of target values.
- f is a neural network.
- *opt* is either *minimization* or *maximization* of the criterion.
- $\mathcal{L}$ is the criterion function. For a pair $(y, f(x)); y \in \boldsymbol{Y}, x \in \boldsymbol{X}$, the criterion function results in a measure of cost (minimization) or fitness (maximization).

To solve this optimization problem, we are looking for a neural network $f : \boldsymbol{X} \to \boldsymbol{Y}$, which will yield the smallest cost given criterion $\mathcal{L}$. Specifically, we have to find suitable network parameters, which will minimize the cost. The goal of training a neural network can then be defined as

$$\arg\min_{\theta} \mathcal{L}(f(\boldsymbol{X}; \theta), \boldsymbol{Y}). \tag{4.8}$$

4.2.2 Gradient descent and backpropagation

To find suitable parameters $\boldsymbol{\theta}$ for neural network f, we need to somehow search the parameter space. Because of high dimensionality of the parameter space, brute-force methods like grid search or random search are not suitable methods to solve the optimization problem. We can instead use

gradient descent. This is an iterative optimization algorithm, which traverses the parameter space by calculating the derivative of the function being minimized. At each iteration, we move by a small step in the direction of the negative gradient. We define this update rule as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \gamma \nabla \mathcal{L}_\theta, \tag{4.9}$$

where γ is the learning rate parameter, which defined the step size, and $\nabla \mathcal{L}_\theta$ is the gradient of loss function $\mathcal{L}$ w.r.t. parameters $\boldsymbol{\theta}$. There are three versions of the basic gradient descent algorithm:

- Batch gradient descent computes the gradient on the whole dataset $(\boldsymbol{X}, \boldsymbol{Y})$. This results in accurate update steps and the algorithm usually finds the closest path to a local minimum. However, the training is slow, since the whole dataset needs to be processed for a single step.

- Stochastic gradient descent (SGD) computes the gradient only on a single pair (x_i, y_i) from the dataset. This significantly speeds up the training process. This induces some noisiness in the optimization path, but doesn't normally influence the resulting optimum.

- Mini-batch gradient descent uses a subset of data for each gradient update. This is a compromise between stochastic and batch gradient descent algorithms. By only a small subset of examples, for example 32, we can already make a more generalized guess of the gradient, while still maintaining much faster computation time, in comparison to updating for the whole dataset. This approach is currently most used in practice.

What is left is the computation of the gradient $\nabla \mathcal{L}_\theta$, which can be efficiently computed with **backpropagation**. This algorithm follows the chain rule of differentiation and composes derivatives of individual layers to calculate the gradient of the whole network. It starts at the output of the network and then propagates the partial derivatives back until it reaches the first layer. This is very efficient in comparison to the naive method of calculating derivatives for each network parameter individually. In practice, this is implemented with automatic differentiation techniques.

4.2.3 Supervision

When training a neural network, the amount of supervision can be controlled. This decision can be based on different factors, like the type of task or the availability of data. There are three degrees of supervision:

- **Supervised learning**. Each training example $x \in \boldsymbol{X}$ has a matching label $y \in \boldsymbol{Y}$. A label is the target value we wish our network to predict, given x.
- **Semi-supervised learning**. During training, some of the examples may have labels attached to them. This approach can be useful for when labeling data is costly or if we want to expand our dataset automatically.
- **Unsupervised learning**. No training example has a label associated with it. The result is usually a generative model - a statistical representation of the training data itself.

4.3 Convolutional neural networks

Like ANNs themselves, convolutional neural networks (CNNs) are inspired by some biological processes. Hubel and Wiesel [57] were the first to discover how specific organization of neurons in the visual cortex enable complex processing of visual data by combining simple, but numerous stimuli. Based on their findings, Fukushima [58] later proposed a computational model. This was the first CNN, however, backpropagation was still not used at the time. First one to combine CNNs and backpropagation was LeCun [59]. Since then, many improvements were proposed for this computational model, but the ability to train neural networks on a GPU had the biggest impact on the field. Since then, CNNs have dominated many computer vision challenges.

4.3.1 Convolutional layer

The main building block of CNNs is the convolutional layer. A convolution is defined as an operation between two functions $f * g$. In the context of CNNs, these functions represent the n-dimensional arguments. For 2D data, the first would be an input image I and the second a learnable filter K, also called a feature. The filter then gets convolved across the image which forms a response map S. This process can be defined as

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n), \tag{4.10}$$

where $S(i,j)$ is a single pixel in the resulting response map. In other words, for every pixel in the input image I, we calculate a dot product with it's surrounding neighborhood and the kernel K. Each convolutional layer then has multiple kernels, each acting as a separate, independently trained neuron.

This is a simplified notation of convolution. In practice, cross-correlation is actually used instead of convolution. These operations are similar, the only difference being that convolution flips the filter before processing. In terms of network weights, this does not matter, since only weight indices are changed. We therefore use the term convolution for any of these operations. When constructing a CNN architecture, we use a more generalized form, which also allows processing n-dimensional data and supports additional hyper-parameters, such as stride and size.

4.3.2 Architecture

The topology of a neural network, often called the architecture, defines how data is processed and how the representation is shaped as a result. The design of CNNs differentiates from the design of fully-connected FFNNs. Since the first proposal of a CNN [58], the general idea of the architecture was retained to this day. With it's cascading structure, it is loosely based on neural pathways inside the visual cortex. CNNs are usually composed of the following building blocks:

- **Convolutional layers** receive image data as input and convolve it with trainable filters.
- **Pooling layers** reduce the dimensionality of data by grouping and reducing it's input with one of the reduction functions (min, max, mean, etc.).
- **Activation layers** induce nonlinearity into the model. Most commonly used is the $ReLU$ function.
- **Batch normalization layers** normalize the data across mini-batches by having mean and bias as trainable parameters.
- **Fully-connected layers** are usually last in the pipeline. These make high-level assertions about features, gathered from convolutional layers.

One or more of these elements are then grouped into more abstract structures, called blocks. Each block consists of two steps. The first block is feature extraction with one or more convolutional layers, each followed by an activation layer. The data is then down-sampled, either by selecting a bigger stride on the last convolutional layer or by using pooling operations. At the same time, the number of trainable filters is increased. By using such a configuration, we effectively enlarge the **receptive field**. This allows for a hierarchical representation inside the neural network, where initial layers focus on a smaller set of simple features, whereas deeper layers combine these to form more numerous complex representations.

In the end, fully-connected layers encode features from convolutional layers into a latent vector. This is the representation of input date, which had it's dimensionality reduced. Depending on the task, the activation function of the last layer is then set accordingly.

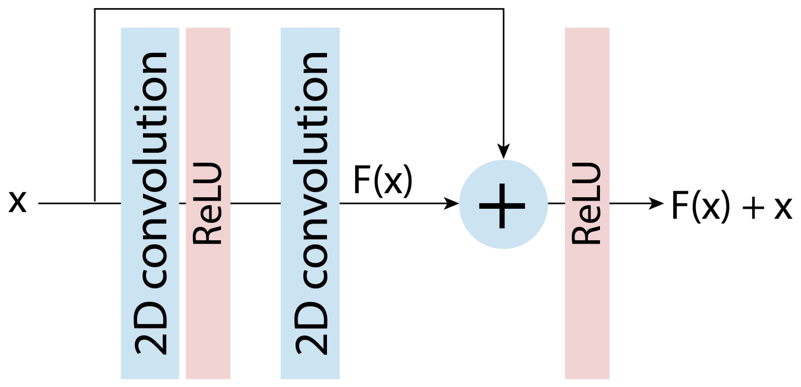


Figure 4.2: The shortcut connection in ResNet [6]; The input x is joined by the results of the block $F(x)$. This identity mapping ensures that the gradients are propagated all the way to the beginning of the network.

CNNs are also **invariant to translation**. Since all input elements (pixels in case of an image) share the same trainable filters, each filter learns features independently of their position.

4.3.3 ResNet

We describe a well established convolutional architecture called ResNet [6]. This is currently considered as the state-of-the-art architecture and since it inception in 2016, no significant breakthroughs in the field were achieved. Due to this, it is considered as the go-to convolutional encoder for many different tasks.

ResNet architecture tries to solve a problem, which was very prominent with deep neural networks in general. It is the problem of **vanishing gradients**. With gradient based training algorithms, such as gradient descent, the weight of the network are updated based on the partial derivatives of loss function w.r.t. the weights. This is calculated with backpropagation, by utilizing the chain rule. This means that the gradient of early layers is a long chain of products and the direct influence of their parameters gets diminishing returns with deeper networks. This leads to slow learning of initial layers, compared to final layers of the network. In [6], the authors propose residual connections. These connections are defined as

$$y = f(x) + x, \tag{4.11}$$

where f is a functional block, composed of multiple convolutional layers, x is the input data and y the result of the connection. In other words, an identity mapping is performed by joining the output of the block with it's input. This is shown in Figure 4.2. With this mechanism, the contributions of initial layers can be propagated deeper into the network. The gradient can thus be estimated better, since the derivative of an identity mapping is equal to 1. These "shortcut connections" do not add any new parameters or computational complexity. ResNets can be arbitrarily deep by composing together individual blocks. A few versions exist, numbered by the number of convolutional layers: ResNet-18, 101, etc.

Chapter 5

Experimental setup

In this chapter we present our experimental setup, which is shared for all experiments. We first describe our synthetic dataset and some properties of the examples in it. We also discuss the choice of CNN architecture, the choice of different hyper-parameters and then describe the training procedure. Finally, we define performance metrics, which we use to evaluate the model.

5.1 Dataset

We use a similar synthetic dataset as in our preliminary study [48]. It contains artificial depth images, where a single superquadric is placed in a scene and rendered in a orthographic projection using a custom ray-tracing renderer. The renderer works by following a ray in discrete steps along the z axis and then detects when the ray intersects the superquadric surface. When that happens, the value of distance from the viewport is written to the respective pixel. The result is an image of size 256×256. This is convenient, since raw grayscale pixels are usually stored with 8 bits of information, which yields a total of 256 possible pixel values. In this way, the dept image is represented by a 3D space of size 256^3.

In comparison to [48], we updated the dataset in two ways. First, we

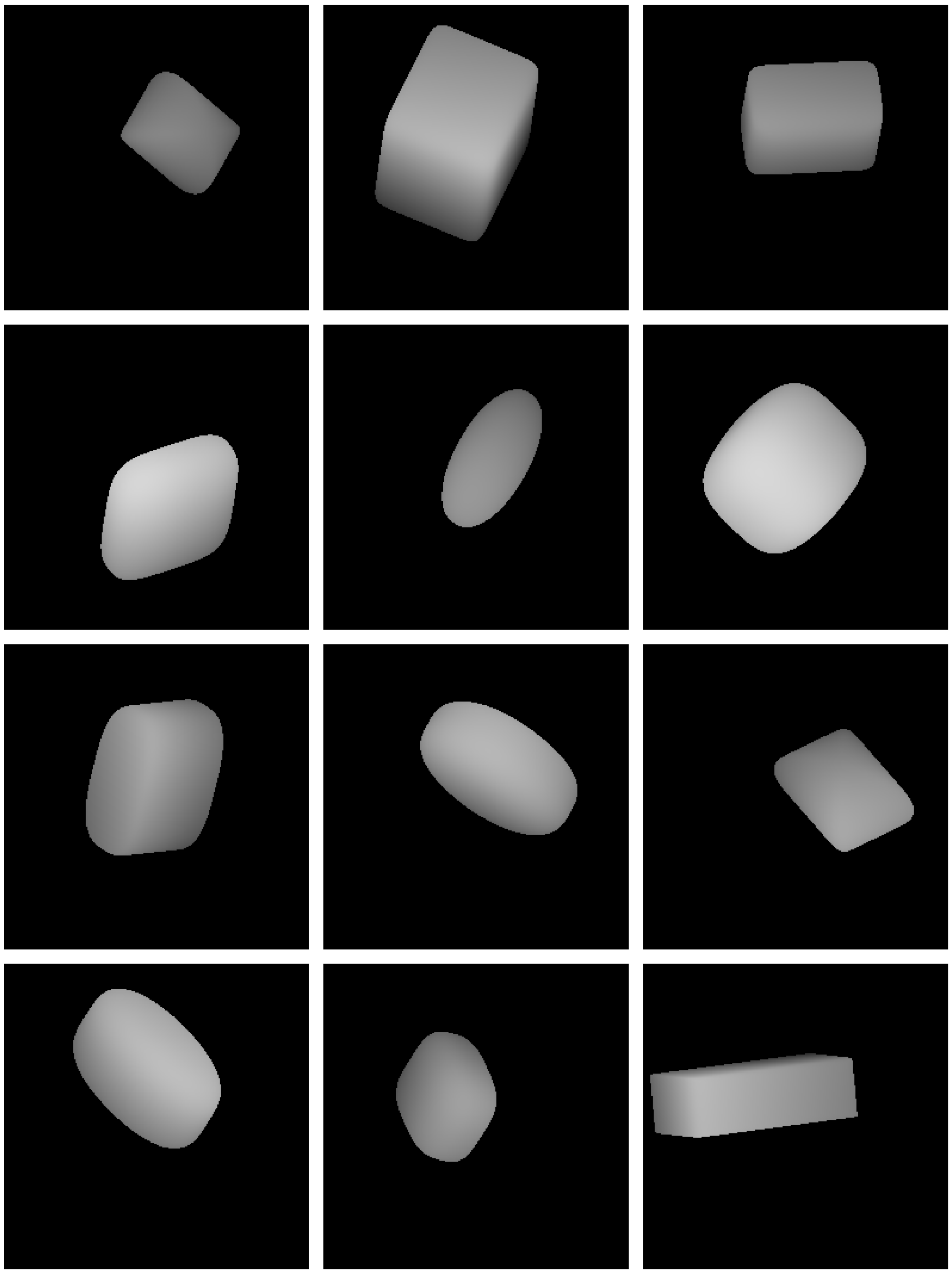


Figure 5.1: Examples from our synthetic dataset. These are depth images of size 256 × 256, where a a single superquadric in rendered inside the 3D space. The projection of the camera is orthographic.

changed the parameter range for shape parameters ($0.1 \leq \epsilon_1, \epsilon_2 \leq 1$). Previously the minimum value was 0.01. As suggested in [20], we decided to increase the lower bound to minimize the possibility of numerical instability, that might occur when evaluating the superquadric equation at small numbers of ϵ. When ϵ approaches 0, the inside-outside function becomes highly exponential. A point, far from the superquadric surface would then have a large value, which might be hard to represent using single or even double precision floating point format, which would lead to overflow errors. The reduction in precision only occurs with sharp-edged objects and is perceptually barely noticeable. We also increased the number of examples to a total of 150,000 depth images for training. Each image is annotated with all 12 parameters. For the test set, we generated additional 20,000 examples, on which we make the final evaluation of performance and compare the different models. Some examples from the dataset are shown in Figure 5.1.

5.2 Model architecture

In our preliminary study [48], we used a custom architecture, similar to VGG [7]. It's main characteristics are in line with the original ideas of CNNs: the network is deep and has a widening receptive field with deeper layers. While such an architecture was proven to be suitable for our task, we can still improve this by using an even more contemporary architecture, with additional advantages. We therefore decided to use a network with residual connections, called ResNet [6]. It's modular nature allows us to easily choose the depth of the network, which influences the model capacity. For tasks involving real-world images, ResNet with 50 or more layers is often used to model the real-world complexity. In contrast, we use a version with 18 convolutional layers, which should be enough to extract simpler features from our depth images. Another improvement are the shortcut connections between consecutive blocks. Any overhead in model capacity can be mitigated by simply learning an identity mapping, like ResNets do. This helps us, when

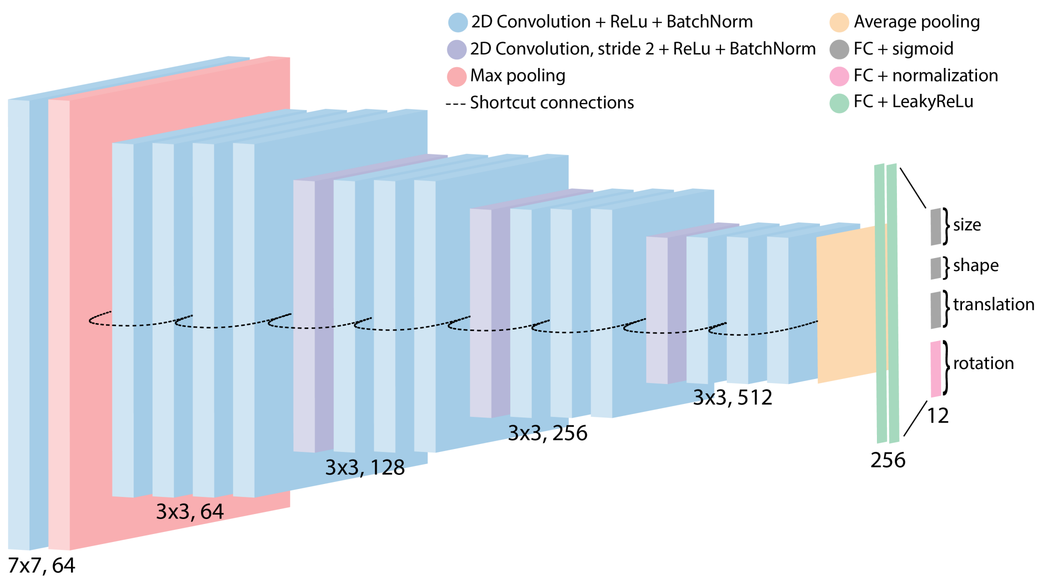


Figure 5.2: We use a modified ResNet-18 [6] as the architecture for our CNN model. Instead of a large fully connected output used for classification, we add two fully connected layers with 256 neurons each and four regression heads, each for a specific superquadric parameter group.

searching for the optimal network depth, since any redundant deeper layers can be skipped. These mappings also enable faster learning of initial layers, since the gradient is back-propagated better.

The final architecture is similar to the reference ResNet. It is shown in Figure 5.2. The first convolutional layer has a filter of size 7 to capture a bigger initial receptive field, since our depth images mostly consist of low-frequency information. From that point on, we use a filter of size 3. By setting the stride of convolution to 2 every 3 convolutional layers, the data is pooled, which widens the receptive field of convolutional filters. In the end, we modify the network slightly to change it from a classification to a regression model. The existing final layer, used for classification, is removed and two new fully-connected layers are added, with each having 256 neurons. This part of the network is the regressor, which mixes and processes features, received from the convolutional layers. The network output is then split into four groups, one for each parameter type. Size, shape and translation parameter groups each have a fully-connected layer with 3, 2 and 3 outputs, respectively. We also attach a final sigmoid activation, which is done so the initial predictions result in values close to "0.5", making the result of superquadric inside-outside function stable and suitable for backpropagation. Rotation parameters have a final fully-connected layer with 4 outputs. Since we use versors to describe rotation, we use L_2-based normalization as the final activation function.

To speed up the training process, we use transfer learning. Specifically, we use pre-trained weights for ResNet-18, trained on the ImageNet [60] dataset. These are loaded into the model before any modifications to the network. The pre-trained model was trained on color images with 3 color channels, however, our depth images only have a single depth channel. To fix this misalignment, we sum the weights of the first pre-trained convolutional layer along the channel axis. This way, we convert the input shape of the pre-trained model to accept our depth images without loosing the pre-trained weights.

5.3 Training and hyper-parameters

The dataset is first split into two parts: 10% for the validation and 90% for the training set. We define a custom generator, which yields batches of data during training. Each epoch, we first iterate through the whole training dataset and then evaluate the performance on the validation set. The data is also shuffled each pass to ensure a representative distribution of the whole dataset in a single batch. For the optimization algorithm, we use Adam [61], for which we set an initial learning rate of 1e$-$4. We set the batch size to 32, which is commonly used in practice for this algorithm.

When evaluating the performance on the validation dataset, we also calculate the accuracy metric, defined in Chapter 5.4. Both, validation loss and accuracy are stored and used as control values for the training procedure. We create a callback function, which monitors the validation loss and decreases the learning rate by a factor of 10, when the validation loss stagnates for 10 epochs. This is done to stabilize gradient descent around the local minimum and to ensure convergence of the loss function. When a new minimum validation loss is achieved, we store the parameters of the whole network to disk. The training is complete, when validation loss stagnates for 20 epochs.

5.4 Metrics

To evaluate the performance of our model, we use volumetric intersection-over-union (IoU) inside a 3D space. We use the binary occupancy function 6.5 to generate superquadric voxel grid $V_{B,\lambda}$. All points that lie outside of the superquadric have a value of 0 and all others a value of 1. The IoU error is then calculated between binarized voxel grids of predicted and ground-truth superquadric parameters:

$$IoU(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{\sum^{|V|} \boldsymbol{V}_{B,\hat{\boldsymbol{y}}} \cap \boldsymbol{V}_{\boldsymbol{B},\boldsymbol{y}}}{\sum^{|V|} \boldsymbol{V}_{B,\hat{\boldsymbol{y}}} \cup \boldsymbol{V}_{B,\boldsymbol{y}}} \tag{5.1}$$

where $|V| = r^3$ and represents the size of the discretized space, $\mathbf{y}$ are ground-truth parameters and $\hat{\mathbf{y}}$ are the predicted parameters. Our goal is then to maximise this performance measure, which represents the ratio of coverage between the generated and true superquadrics. A value of 0 means that there is no overlap and a value of 1 means that the superquadrics overlap perfectly.

Next, we define some metrics to compare the quantitative properties of predicted parameters. We can't compare these directly, e.g. predicted $\hat{a_1}$ with ground-truth a_1. The order of size parameters a can be arbitrary, since the network is not restricted in this sense. For size parameters, we calculate the average relative difference between volumes of ground-truth and predicted superquadrics:

$$\Delta A = \frac{1}{N} \sum_{i=1}^{N} \frac{\hat{a_1}\hat{a_2}\hat{a_3} - a_1 a_2 a_3}{a_1 a_2 a_3}, \tag{5.2}$$

where N is the number of examples in the test set, $a_{1,2,3}$ are the ground-truth size parameters and $\hat{a_{1,2,3}}$ are the predicted size parameters of individual examples. The result is a relative error, which tells us by how many percent the predicted superquadrics differ in volume from the ground-truth superquadrics. For example, $\Delta A = -4\%$ would mean, that the predicted superquadrics have on average a 4% smaller volume.

In a similar fashion, we evaluate shape parameters ϵ. The term $(\epsilon_1 + \epsilon_2)/2$ represents the average value of parameters ϵ for a given superquadric. We then simply calculate the relative difference in average value of ϵ for all superquadrics by

$$\Delta E = \frac{1}{N} \sum_{i=1}^{N} \frac{(\hat{\epsilon_1} + \hat{\epsilon_2}) - (\epsilon_1 + \epsilon_2)}{\epsilon_1 + \epsilon_2}, \tag{5.3}$$

where N is the number of examples in the test set, $e_{1,2}$ are the ground-truth shape parameters and $\hat{e_{1,2}}$ are the predicted shape parameters of individual examples. The number 2 in the denominator can be dropped, since all terms include it.

Translation parameters t are the only group, where the order is deterministic, so we can evaluate each parameter individually. 3D information in depth images is encoded differently for z axis in comparison to x and y. The ability to analyze individual translation parameters can therefore be useful to check, how the selected data representation influences model bias. The error is defined as

$$\Delta T_k = \frac{1}{N} \sum_{i=1}^{N} \hat{t_k} - t_k; k \in \{1, 2, 3\}, \tag{5.4}$$

where N is the number of examples in the test set, t_t the ground-truth translation parameter and $\hat{t_k}$ is the predicted translation parameter. ΔT then forms a tuple $(\Delta T_1, \Delta T_3, \Delta T_3)$ for all translation parameters t_1, t_2 and t_3, respectively.

We do not compare the difference in predicted and ground-truth rotation-It is difficult to determine a suitable metric for this kind of comparison. Even if we used a distance metric between two rotations, the predictions of rotation can be arbitrary, since other superquadric parameters can be adjusted to fit the shape properly. This wouldn't be practical, so we rather determine the accuracy of rotation indirectly from the volumetric IoU.

Chapter 6

Supervised learning of superquadric parameter recovery

In this chapter, we present a new method for superquadric parameter recovery from depth images. We first formulate the problem for this task and present the motivation behind our approach.

6.1 Motivation

In [48], we trained a regression model using a very simple objective function, by minimizing the squared error between predicted and ground-truth parameters. As evident, this interaction was not complex enough to capture the ambiguous nature of rotation. By comparing only the regressed values themselves, we reduce the parameter space of quaternion coefficients to have only a single global optima, regardless of object symmetry, and despite orientation being a periodic description. The estimated parameters should instead be compared in geometric terms. Ideally, the error of each parameter group should correspond exactly to the error between ground-truth and estimated superquadrics, constructed using respective parameters. For example, if two superquadrics were exactly the same, but have different positions, only the translation parameters would need to contribute to the error the most.

The goal is therefore to design a new training objective, i.e. a loss function, which would in a geometric and object-aware fashion guide the optimization procedure of a convolutional neural network for the purpose of superquadric parameter recovery. There should be some consideration when designing such an objective:

1. The characteristics of each parameter group need to be accounted for. For example, a superquadric being more rounded that it should be is in general considered less severe, than an error in it's position. Per-parameter importance needs to be reflected in the magnitude of error.
2. The shape of the superquadric needs to be considered. This particularly important when considering rotation and symmetry. Two superquadrics could look the same, but have different parameters associated with them. A trivial example would be rotating a superquadric along any axis by 180 degrees, which would to an observer yield the same 3D shape, because of its symmetry.
3. The objective function needs to be differentiable in order to support backpropagation of gradients and to allow proper training of the neural network.
4. It should also be effectively computed and numerically stable.

When considering possible solutions, the usage of superquadric inside-outside function from Eq. (3.11) seems tempting. It was, after all, used in the original least-squares minimization method, proposed by Solina and Bajcsy [24]. The function is positive and increases continuously from the center the superquadric towards infinity. Lastly, the function is differentiable w.r.t. the parameters, which enables the computation of the gradient.

The main question is: Can we compare two superquadric functions? By mapping 3D coordinates to a real value, the inside-outside is by definition a

4D function, also called a hypersurface in R^4. We can measure the difference between two curves in R^2 or two surfaces in R^3, for example, by sampling equally-distanced points and calculating the mean squared error (MSE) between the values at those points. Of course, MSE can be changed for any other metric. If the metric is differentiable, we can create a cost function that enables us to fit the parameters of a curve or a surface using gradient descent optimization. We can do the same for two hypersurfaces in R^4. This is the main idea behind our approach.

6.2 Problem formulation

The goal is to create a predictor, which recovers the parameters λ of a single superquadric. The predictor can be defined as a function:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{X}), \tag{6.1}$$

where $\boldsymbol{X}$ is the input in form of a depth image. The function then returns all 12 estimated superquadric parameters $\hat{\boldsymbol{y}} = [\hat{a_1}, \hat{a_2}, \hat{a_3}, \hat{\epsilon_1}, \hat{\epsilon_2}, \hat{t_1}, \hat{t_2}, \hat{t_3}, \hat{q_i}, \hat{q_j}, \hat{q_k}, \hat{q_w}]$. Since superquadric parameters λ are continuous real values, we formulate this task as a regression. For the predictor function, we use a convolutional neural network

$$\hat{\boldsymbol{y}} = f_{CNN}(\boldsymbol{X}; \boldsymbol{\theta}), \tag{6.2}$$

where $\boldsymbol{\theta}$ are learnable parameters of the network and $\hat{\boldsymbol{y}}$ is the output. In the case of a regression model, the outputs are continuous values $\hat{\boldsymbol{y}} \in \mathbb{R}^{12}$. To train the neural network, we have to minimize a cost function

$$\mathcal{L}\left(\mathbf{y}, \hat{\mathbf{y}}\right), \tag{6.3}$$

where $\hat{\boldsymbol{y}}$ are the predicted superquadric parameters and $\boldsymbol{y}$ are the ground-truth superquadric parameters. The design of this cost function is described in Chapter 6.3.

6.3 Occupancy loss

Here we first define all the intermediate volumetric representations and then formulate our geometrically-aware cost function.

6.3.1 Adjusted inside-outside function

The inside-outside function has an unwanted property, that violates one of the considerations, listed in Chapter 6.1. The problem arises when comparing superquadrics with different shape parameters (ϵ_1, ϵ_2). In this case, the difference in value of these inside-outside functions, evaluated at the same point would be disproportionately attributed only to the shape parameters, since they control the exponentiality of the function. A point in space would yield a far bigger value for a sharper superquadric, even if all other parameters match. To adjust the exponentiality of the inside-outside function, we can raise it to the power of ϵ_1, as suggested in [24]:

$$F^{\epsilon_1}(x, y, z) = \left(\left(\left(\frac{x}{a_1} \right)^{\frac{2}{\epsilon_2}} + \left(\frac{y}{a_2} \right)^{\frac{2}{\epsilon_2}} \right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{z}{a_3} \right)^{\frac{2}{\epsilon_1}} \right)^{\epsilon_1} . \tag{6.4}$$

6.3.2 Occupancy grid

Even though the inside-outside function is now adjusted for shape parameters, the biggest differences still occurs outside of the superquadric where $F^{\epsilon_1}(x, y, z) > 1$. We want to have a greater focus on differences in close proximity to the superquadric. To achieve this, we can further transform the inside-outside into an occupancy function. The trivial solution would be a simple binarization function $B : \mathbb{R}^3 \rightarrow \{0, 1\}$:

$$B(x, y, z) = \begin{cases} 1 & F^{\epsilon_1}(x, y, z) \leq 1 \\ 0 & F^{\epsilon_1}(x, y, z) > 1 \end{cases} , \tag{6.5}$$

however, this operation is not differentiable. Instead, we follow the proposal of [45] and calculate probabilistic occupancy with function

$$G(x, y, z) = \sigma(s(1 - F^{\epsilon_1}(x, y, z)), \tag{6.6}$$

where $G : \mathbb{R}^3 \to (0, 1)$ and s is a scaling factor, which controls the sharpness at the border of the superquadric. This function returns a value close to 1 if a point is inside the superquadric, close to 0 if it is outside and 0.5 of the point is directly on the surface of the superquadric. The function is also continuous and therefore differentiable.

6.3.3 Sampling of the inside-outside function

The probabilistic occupancy function 6.6 is evaluated for a specific world-centered point $\boldsymbol{p} = [x, y, z]^T$ in the Cartesian coordinate system. If we evaluate the function for all possible points $\boldsymbol{p} \in \mathbb{R}^3$, we end up with a continuous hypersurface in R^4, where each coordinate encodes the value of the given function.

We can calculate an approximation of this hypersurface by first discretizing the coordinate system into a set of fixed, equally-distanced points. The discretization procedure is controlled by resolution parameter r and minimum and maximum bounds b_{min} and b_{max} for each of the axis. In a linear manner, we sample r points in each axis from b_{min} to b_{max}, which results in a 3D grid of discretized points $\boldsymbol{p}_d$. For each of these points, the occupancy function is evaluated and stored into a volumetric grid:

$$\boldsymbol{V}_{G,\lambda} = G(x, y, z; \lambda), \quad \forall [x, y, z]^T \in \boldsymbol{p}_d. \tag{6.7}$$

By doing this we get a discrete approximation $\boldsymbol{V}_{G,\lambda}$ of occupancy function in 3D space, given parameters λ. In other terms, we create a voxel grid, where each voxel encodes the value of the occupancy function at that location. The size of the grid corresponds to the selected resolution: $|\boldsymbol{V}| = r^3$.

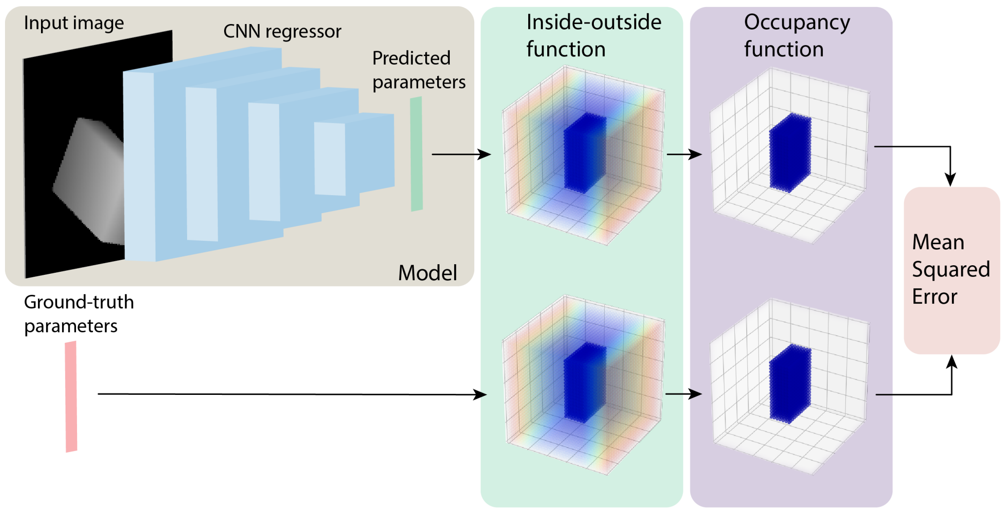


Figure 6.1: The visualization of the occupancy loss function; Both, predicted and ground-truth parameters are first used to calculate the inside-outside function for a discretized 3D space. Each inside-outside grid is then converted to an occupancy grid and MSE is calculated between them.

6.3.4 Final definition

Finally, we can define our cost function. We use MSE to calculate the difference of two voxelized occupancy spaces:

$$\mathcal{L}_{OC}\left(\boldsymbol{y}, \hat{\boldsymbol{y}}\right) = \frac{1}{|\boldsymbol{V}|} \sum_{i,j,k}^{r} (\boldsymbol{V}_{G,\boldsymbol{y}}(i,j,k) - \boldsymbol{V}_{G,\hat{\boldsymbol{y}}}(i,j,k))^2, \tag{6.8}$$

where $\boldsymbol{y}$ are ground truth parameters of the target superquadric and $\hat{\boldsymbol{y}}$ are estimated superquadric parameters. Specifically, we first sum all the squared differences between matching points and then divide this by the size of the grid $|\boldsymbol{V}|$. The visualization of this loss function can be seen in Figure 6.1.

6.4 Implementation details

The loss function as defined in Section 6.3 is relatively expensive computationally in comparison to the neural network. We have to compute the inside outside function, which is by itself not trivial due to many exponentiation functions, for every discretized point in the 3D space. The number of points rises cubically with the resolution of discretization. To compute this efficiently in practice, we use *meshgrids* to represent the discretized points. We create three separate 3D tensors, $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{Z}$, where each encodes the coordinates of the respective axis. This data structure is useful, because it allows us to parallelize the computation of the inside-outside function. The operations are independent for each point in space and computation can be done efficiently on the GPU. The meshgrid structure can also be easily transformed into a point cloud. We first stack all 3 tensors into a tensor of shape $(3, n, n, n)$, where n is the resolution of the discretized coordinate space. Then, we can reshape the tensor into $(n, n, n, 3)$ and finally flatten it, which results in a set of points of size $(n^3, 3)$.

After preparing the coordinate system tensors, we can evaluate the inside-outside function for both, ground-truth and predicted parameters. Inside the function, we first ensure that the inside-outside function will not become singular by clipping parameters to allowed ranges. These ranges are $[0.05, 1]$ for size, $[0.1, 1]$ for shape and $[0.0, 1]$ for translation parameters. The conjugated quaternion is then transformed into a rotation matrix. The coordinate system is rotated by calculating a dot product between the rotation matrix and each point in space. This is done efficiently by broadcasting the operation to all points in space in parallel. The translation vector is also rotated in the same way to transform it to object-centered coordinates. Transformed coordinates are then inserted into the inside-outside equation. Finally, we calculate the occupancy function and the resulting 3D tensors are then used to calculate MSE between them. This process is done for each pair of ground-truth and predicted parameters in a batch. We manually iterate through all examples in a batch, since we can't broadcast operations to the whole batch.

During the computation of the inside-outside function, a few numerical instabilities can arise. Most notable is the problem of undefined gradient for certain operations. For example, the exponentiation function $f(x) = x^n$ has a partial derivative $\frac{\partial f}{\partial n} = x^n \ln(n)$. When $x = 0$, the value of natural logarithm becomes undefined. The backpropagation then fails and usually corrupts the weights in the neural network. To prevent this from happening, we have to ensure, that the input to exponentiation functions is always different from 0. First, we add a small constant, e.g., 1e−4, to the initial coordinate system where a coordinate is 0. By doing this, we offset any points lying directly on the axes by a small margin. Then the inner terms in the inside-outside function are raised to the power of 2. This can result in rounding errors. For example, if we raise 1e−4 to the power of 2, the result is 1e−8, which would get rounded to 0 if we used single precision floating points. Due to this, we again add a small constant as in the previous situation. Later exponentiations don't suffer from this occurrence.

6.5 Experiments and results

In this section, we present the experiments and most importantly, the results of our methodology. First, we analyse some properties of the loss function and train a preliminary model. Then, individual experiments are described, along with the results and a short discussion.

6.5.1 Loss function analysis

Our first goal was to test if the new loss function $\mathcal{L}_{OC}$ has all the necessary properties, that would enable the model to learn rotations. To test this, we defined two superquadrics with identical shape, size and translation parameters. We then rotated one of these superquadrics along a specific axis for 360 degrees. Along the way, we calculated the error between them and then plotted a graph for each of the axes.

The result can be seen in Figure 6.2. First, we can observe the periodic nature of rotation. The loss is highest at a difference in angle of 90 degrees. At that point the superquadrics are perpendicular to each other and the loss should be highers. At 180 degrees, the loss falls back to 0, since superquadrics

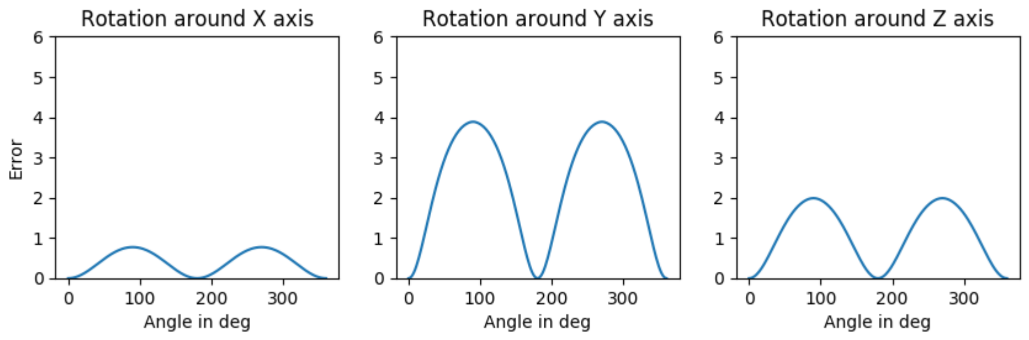


Figure 6.2: Demonstration of loss function $\mathcal{L}_{OC}$; We rotate a superquadric with parameters $\{a_1 = 0.1, a_2 = 0.2, a_3 = 0.3, \epsilon_1 = \epsilon_2 = 1\}$ for 360 degrees around each axis. Along the way we calculate the loss between it's current and unrotated pose. From the graph, we can observe the symmetric property of the loss function.

are symmetric. The same observation then repeats itself from 180 to 360 degrees. Another thing to point out are the magnitudes of responses. These differ due to the dimensions of the superquadric. In case of example in Figure 6.2, a_1 is the dimension along axis x. When we rotate it around another axis y, the magnitude is relatively large. This happens because for smaller superquadrics in a fixed space, larger values of the inside-outside function are calculated for points further from the surface on the border of space. The ratios between rotating axes are also important. Response for y axis has the biggest magnitude because the ratio between a_1 and a_3 is the biggest. Alternatively, rotation around x has the smallest magnitude, because the ratio between a_2 and a_3 is the smallest. With this experiment we confirmed the positive properties of the loss function.

6.5.2 Dual loss training

The original idea was to upgrade the model, presented in [48]. We first wanted only to expand the existing model by adding another loss function, which would optimize rotation. We created a loss function with two different criterions, where each would optimize its own group of superquadric parameters:

$$\mathcal{L}_{DUAL} = \mathcal{L}_{MSE(a,\epsilon,t)} + \mathcal{L}_{OC(q)}, \tag{6.9}$$

where $\mathcal{L}_{MSE}$ is the standard MSE loss function, calculated between parameter values:

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum^{N}(\boldsymbol{y} - \hat{\boldsymbol{y}})^2. \tag{6.10}$$

The first term would optimize only parameters for size, shape and translation and the second term would optimize rotation parameters with the new loss function. We wanted to do this for two reasons: (1) The prediction of first eight parameters (size, shape and translation) was already successful and the model was trained quickly by only calculating the MSE between predicted and ground-truth parameters directly. (2) For predicting only rotation, other parameters of $\mathcal{L}_{OC}$ can be fixed, which speeds up the otherwise relatively slow computation of the loss.

As a preliminary test, we first trained a model, which would only predict rotation of a superquadric. Ground-truth parameters were used to calculate the shape of two superquadrics. One was then rotated with the predicted quaternion and the other was rotated with ground-truth quaternion. The loss between them was then calculated using $\mathcal{L}_{OC}$. We achieved an IoU score of 75.02%. During training, the validation loss converged to local minimum, while training loss kept decreasing. This behaviour was strange and it looked like the network failed to generalize on the dataset. In comparison, a random estimator for rotation prediction would result in an IoU score of around 58%. The network did learn something, however, there was some obstacle preventing it to learn better. In Figure 6.3, we plot the distribution of shortest arc angles from predicted to ground-truth rotation. The distribution gives us the insight into why the overall accuracy is low. Many examples appear to get stuck in a local minimum during training. The peaks around 0, π and 2π radians are correctly estimated. These represent the differences of 0, 180 and 360 degrees, which all result in the same representation of a su-

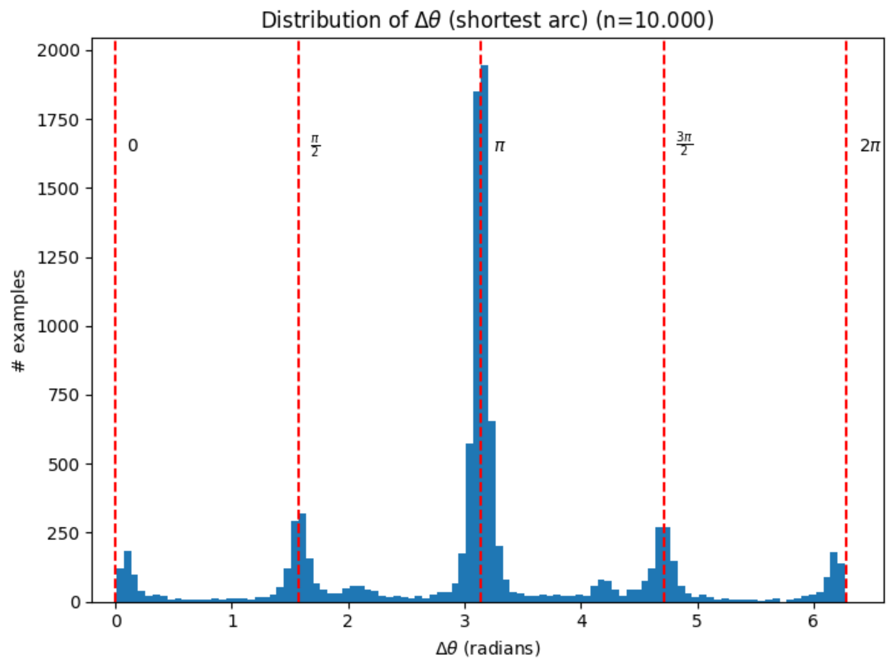


Figure 6.3: The distribution of shortest arc angles between predicted and ground-truth rotations; Note the incorrect predictions around $\frac{\pi}{2}$ and $\frac{3\pi}{2}$ radians, while rotations grouped around 0, π and 2π radians are correctly estimated. This confirms unwanted local minimum in the loss function.

perquadric, because of their symmetric nature. In contrast, peaks around $\frac{\pi}{2}$ and $\frac{3\pi}{2}$ represent rotations by 90 and 270 degrees, which results in a pose, perpendicular to the ground-truth. One example of a superquadric, which ends up in a local minima during training is shown in Figure 6.4. It's shape and size cause the opposite corners across the center to line up. Any modifications to the parameters then result in a loss, larger than previous step, causing the superquadric to get stuck.

We then trained a model, which predicts the full set of superquadric parameters and uses loss $\mathcal{L}_{DUAL}$ to calculate the error. The results are shown in Table 6.1. In this configuration, which besides rotation also predicts size, shape and translation, we achieved an IoU score of 62.76%. We know the

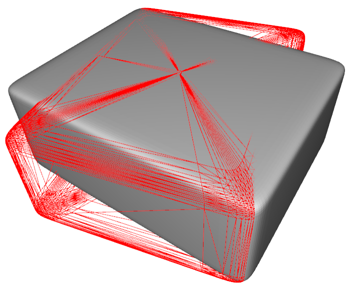


Figure 6.4: One of the local minimums in which the superquadrics get stuck when learning only rotation; The red wireframe represents the target superquadric and the predicted superquadric is shaded with gray color. In this situation, any change of rotation would yield a larger loss, compared to current step.

Table 6.1: Performance comparison of both our supervised models, which predict all 12 superquadric parameters.

Method	IoU
Model 1, $\mathcal{L}_{DUAL}$	$62.76 \pm 11.34\%$
Model 2, $\mathcal{L}_{OC}$	$\mathbf{94.62 \pm 3.18\%}$

upper limit is already set by performance of model 1. The model then suffers from additional reduction in IoU, since the faults in both loss function are compounded and not nullified.

6.5.3 Full recovery with occupancy loss

One possible solution to the shortcomings of $\mathcal{L}_{DUAL}$ was to only use a single loss function for all parameters. We therefore wanted to use $\mathcal{L}_{OC}$ to predict all parameters, not only rotation. This was intended from the beginning, however, it proved to be difficult to implement. The most common issue was numerical instability. In some parts, the parameter space would have large gradients, which would cause the optimization algorithm to make large steps

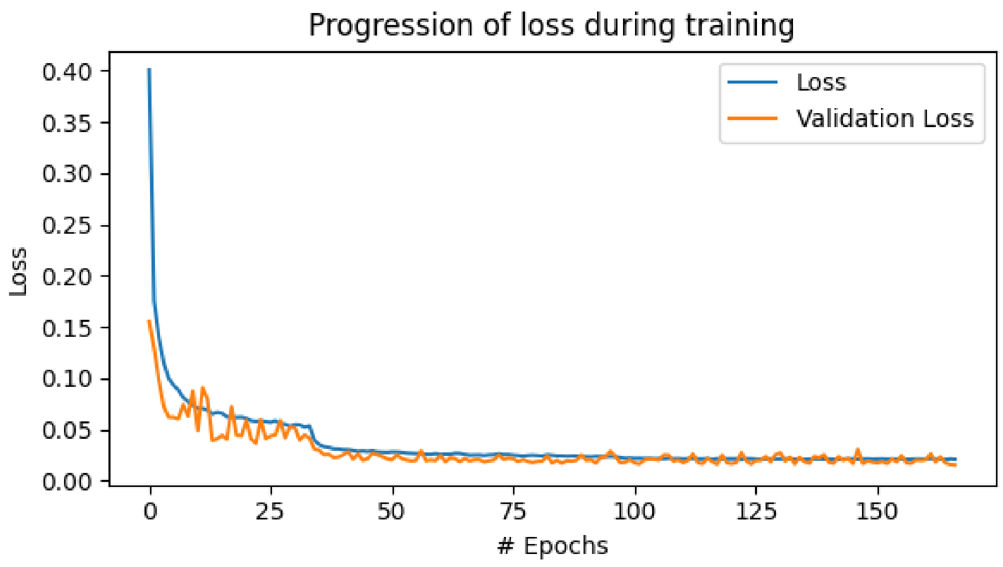


Figure 6.5: The value of loss and validation loss during training with loss function $\mathcal{L}_{OC}$ with all parameters. The step at around 30 epochs is caused by the reduction of learning rate.

and miss the local minimum completely, but at the same time be slow in other areas. We sometimes experienced numerical overflow, so we changed number representation from single-precision to double-precision floating point. Some of the functions, used in implementation if the inside-outside function, have undefined gradient for some inputs, which caused trouble during training and corrupted the gradients. We solved this by adding small constant numbers in certain places. This is described in more detail in Chapter 6.4. After these fixes were applied, we successfully implemented the loss function and then trained a model. The progression of loss and validation loss during training can be observed in Figure 6.5.

The performance of both models, which predict all superquadric parameters, is listed in Table 6.1. By training our model with the loss function $\mathcal{L}_{OC}$, we reach a IoU score of 94.62%, which is a significant increase from the model, trained using the dual loss $\mathcal{L}_{DUAL}$, presented in the previous experiment. By calculating the error on all parameters, we effectively expanded

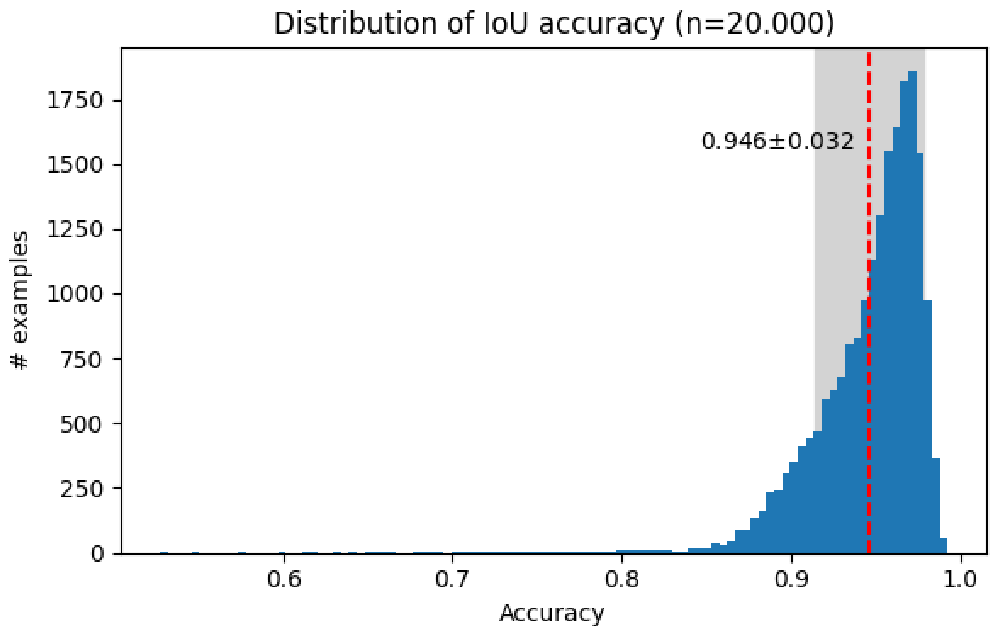


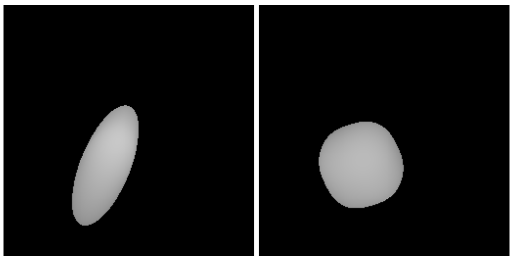
Figure 6.6: The distribution of IoU accuracy for model 3 on the test set; 99.06% of examples have an IoU greater than 85%.

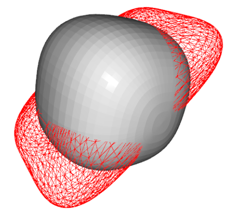
the dimensionality of parameter space, which enabled the optimization algorithm to escape from local minimum, shown in Figure 6.4. The issue appears to be completely resolved, since both, validation and training loss, decrease evenly and no overfitting occurs. We also managed to reduce the standard deviation from 11.35% to 3.18%.

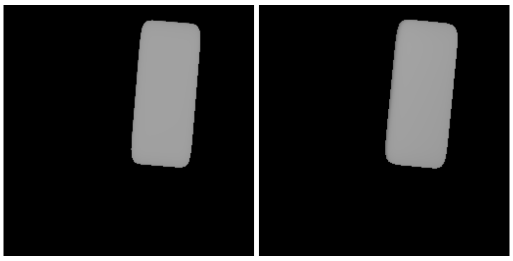
During training, different parameter groups are learned quicker than others, due to the nature of the loss function $\mathcal{L}_{OC}$. Translation parameters are fitted first and the superquadrics are centered after only two epochs. Then, size and rotation parameters are fitted at a similar pace and need about 30 epochs to converge. At this point, the average IoU is already at around 85%. After that, shape parameters are fitted slowly for more than 100 epochs, until the training as a whole converges. This is because change in ϵ has a relatively small effect on the loss function, when the superquadric is already in the target pose. In Figure 6.6, we can see the distribution of all IoU accuracy scores across the test set. The minimum IoU achieved for an individual example is 52.71% and the maximum is 99.24%. From all of the examples, 99.06% have an IoU greater than 85%.

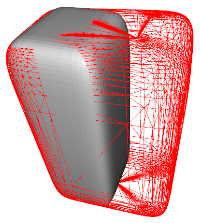
For examples with a low IoU, we can visualize, analyze and then determine what is the reason behind bad performance. We take 4 examples with less than 70% IoU and show their depth images as well as 3D renders in Figure 6.7. Each row represents a superquadric. From top to bottom, the IoU scores are: 52%, 54%, 61% and 68%. True and predicted depth images are shown in the left column and renders are in the right column. To analyze the examples, we built an iterative minimization tool, which allows us to manually search the superquadric parameter space. By calculating the gradient of the loss function w.r.t. predicted parameters in iteration i, we can then update predicted parameters in iteration $i + 1$. We are then able to determine, if it is possible to minimize the predictions even further from their current position in the parameter space.

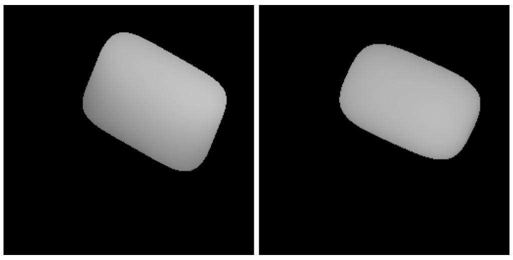
Examples in Figure 6.7 can be grouped into 2 categories, which also applies to other examples with IoU $< 70\%$. In the first group are superquadrics from first and third row; Their parameters are located inside a saddle point in the parameters space. With our manual minimization tool, we managed to find the correct parameters, however, it takes a long time to escape the saddle and find the suitable local minimum. At the saddle point, the gradients are low and there are multiple possible choices of which parameter group to update. To conclude, these kind of examples can be solved, but we would need to run the training procedure longer. This can be defined as a trade-off between the number of predicted outliers and the time it takes to train the network. The second group then contains examples, similar to those in the second and fourth row. As evident, the ground-truth and predicted depth images look almost identical to the human eye. In both cases, only one side of the superquadric if visible. This is known as the self-occlusion problem. The model can't determine the size of the occluded axis of the superquadric, so it makes a guess, representative of similar examples in the training set.

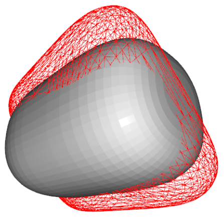

(a)

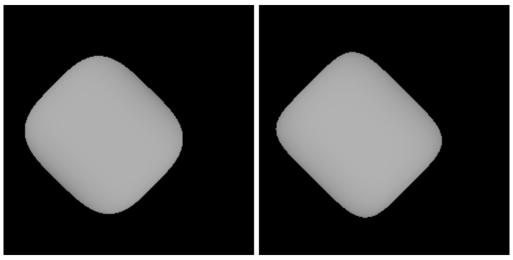

(b)


(c)


(d)


(e)


(f)


(g)

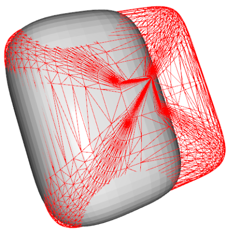

(h)

Figure 6.7: Predictions of model 3, with an IoU $< 70\%$; Each row represents a superquadric. In (a), (c), (e), and (g) from left to right, ground-truth and predicted depth images are shown. Rendered superquadrics are shown in (b), (d), (f), and (h), where red wiremesh represents the ground-truth, and gray model represents the prediction.

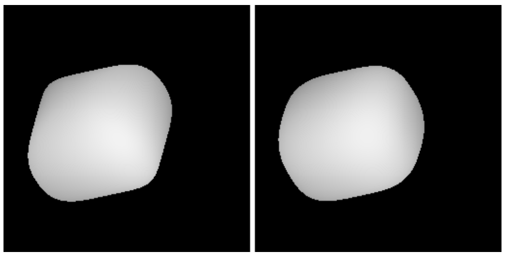
Table 6.2: Errors for each parameter group; We compare out best model with the iterative method.

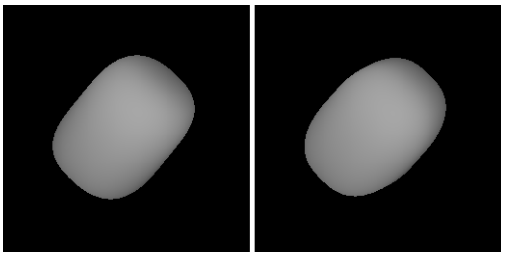
Method	ΔA	ΔE	ΔT
Iterative [24]	-10.30%	$+6.07\%$	$(\mathbf{+0.00\%}, \mathbf{+0.00\%}, +1.56\%)$
supervised, $\mathcal{L}_{OC}$	$\mathbf{+0.14\%}$	**-0.36%**	$(+0.01\%, +0.01\%,$ **-0.03%**$)$

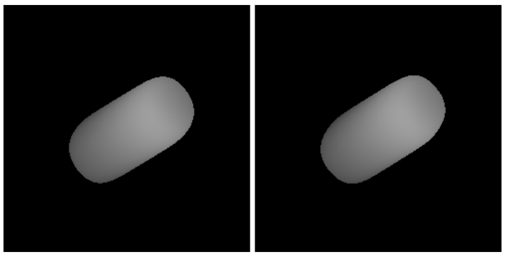
We expected this could be an issue, however, we couldn't predict what the model will do in such a case. For all such examples, the size of the occluded axis is predicted as being shorter in comparison to the ground-truth. We can not improve the performance of such examples, since the information is lost and it is up to the model to hallucinate the occluded data.

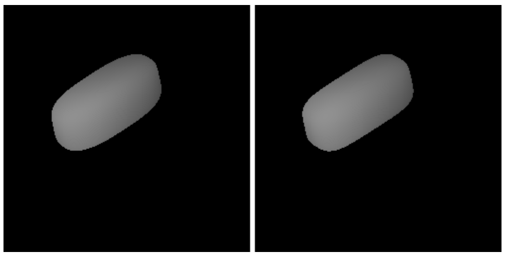
We extend the qualitative analysis to the examples around mean IoU, specifically within one standard deviation. These are presented in Figure 6.8. To the human eye, predictions with accuracy bigger than 95% are hardly distinguishable from the ground-truth. Below this point, we start to observe some differences, most notably, in object shape. Predicted superquadrics are in many cases slightly more rounded than their ground-truth counterparts. This happens because changes in shape parameters have a small impact on the loss function. Consequently, the gradients are small and learning slower. We also observe, that when the predicted superquadrics are more rounded, there is a bigger chance that size will increase as well. Size parameters have steeper gradients during training, so the neural network compensates rounded edges by increasing the volume of the superquadric.

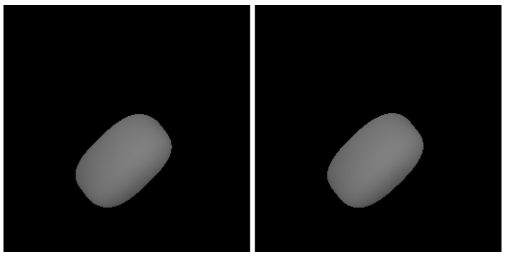
In Table 6.2, we observe our error metrics for each parameter group, predicted by the model. Overall the error are centered closely around 0. This means that no major bias was introduced into the model. The volume of the predicted superquadric is on average 0.14% larger then ground-truth, as indicated by the metric ΔA. The model predicts superquadrics as being slightly less rounded than their ground-truth counterparts, specifically, by 0.36%. The average differences of translation parameters are negligible. We

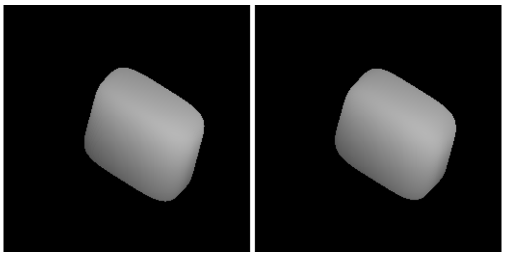

(a) 93.19% IoU

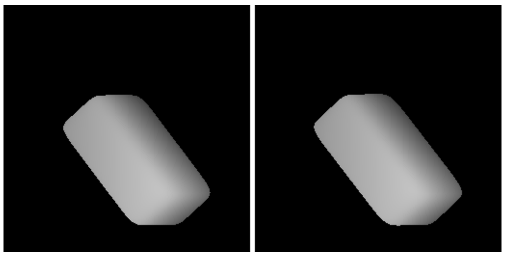

(b) 93.76% IoU


(c) 94.55% IoU


(d) 94.97% IoU


(e) 95.53% IoU


(f) 95.94% IoU


(g) 97.11% IoU

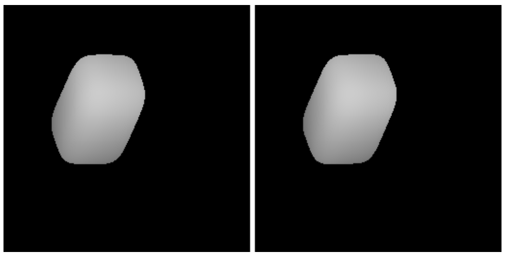

(h) 97.98% IoU

Figure 6.8: Predictions of model 3; The IoU score is within the distance of one standard deviation from mean IoU. This range represents the majority of the predictions (68.27%) on the test dataset.

Table 6.3: Comparison of our best model with the classic method from Solina and Bajcsy [24]

Method	IoU	Execution time [ms]
Iterative [24]	$84.51 \pm 8.89\%$	690.52 ± 275.62
supervised, $\mathcal{L}_{OC}$	$\mathbf{94.62 \pm 3.18\%}$	$\mathbf{2.88 \pm 0.92}$

can only observe a slightly bigger bias for z coordinate, which might be because the coordinated for z axis are encoded differently in comparison to x and y, since we use depth images as the main 3D representation.

6.5.4 Comparison to the classic method

Finally, we compare the performance of the supervised approach with the classic method from Solina and Bajcsy [24]. First noticeable in Table 6.3 is a notable increase in IoU accuracy. We managed to increase it by an absolute value of 10.09%. At the same time, we reduced the standard deviation from around 9% to around 3%. We have therefore managed to maintain a high accuracy and even to improve the performance, however, even more important is the execution time, which we reduced by a factor of 240. Since this is a feed-forward CNN the execution time is constant, which was expected. We should also consider the possibility of parallel computing. Our CNN regressor processes a mini-batch at a time, while maintaining the constant execution speed. This means that we can at once calculate parameters for multiple depth images. Similar performance was already achieved in our preliminary study [48], however, the execution time is now further reduced by $0.72ms$. This is because we use a modified ResNet-18 instead of a VGG-like network [7], and thus use less parameters than before. The classic method in comparison to ours also exhibits larger biases, as evident in Table 6.2. The volume of predictions is heavily underestimated by the iterative method and the predicted shape is more rounded than that of the ground-truth.

6.6 Discussion

We managed to create a new loss function by evaluating the inside-outside function and comparing the occupancy between two superquadrics. The original idea of only expanding the original model from our preliminary study [48] did not work; By dividing the loss into two parts, we created a simpler and faster loss function, able to recover rotations, however, we also limited the parameter space and introduced more false local minimums. Training only with loss $\mathcal{L}_{OC}$ was difficult to implement due to all numerical instabilities and other variables, which had to be accounted for. In the end, this approach proved to be successful and despite various technical problems during implementation, our perseverance was rewarded. With this model we achieved not only a faster and even real-time execution speed, but also improved the overall accuracy by a large margin in comparison to the classic iterative method.

Chapter 7

Unsupervised learning of superquadric parameter recovery

In this chapter, we present an alternative approach to training superquadric parameter recovery. This is an unsupervised version of our training procedure. We explore two different methods for loss function formulation and then compare them. We also analyze the results in comparison to the supervised version and the classic iterative method.

7.1 Motivation

While supervised learning is obviously a more reliable approach, it is often not the most efficient strategy when it comes to real-world data. Neural networks need a large dataset to properly learn the distribution of all possible inputs. As with other machine learning algorithms, the more complex the data is, the more data we need and consequently, the model needs more capacity and takes longer to train. Constructing a large annotated dataset is an expensive task, since each example needs to be labeled separately. This time consuming and often, a human expert is needed to infer labels. There are also ambiguous examples, which lie on the decision boundary or have missing features. These can be hard to annotate, even for an expert. Manual annotation of the dataset influences model bias and variance. Deriving a valid unsupervised learning method is therefore an attractive solution.

We ultimately want to recover superquadric parameters of real objects from depth images. For many of these objects, multiple superquadrics would be needed to describe the complexity of their shape. An annotator would need to manually segment objects into smaller part-objects and then try to estimate superquadric parameters of every part-object. It would be practically impossible to label real data this way. It is difficult for a human to determine the shape and general position of a superquadric, only by observing a depth image.

The classic method from Solina and Bajcsy [24] is an iterative minimization process. The idea behind it can act as an inspiration towards an unsupervised solution, possibly in form of a loss function, which would minimize the same cost function as their algorithm does. For the classic approach, the depth images are first converted into a point cloud. The algorithm then searches for solution so that optimally, the point cloud lies on the surface of the predicted superquadric. We can in the same way utilize the input data to guide the training process of a CNN.

Another side effect of recent advancements in 3D deep learning are various differentiable conversion techniques between 2D and 3D representation. For example, these methods are able to convert 2D depth images to a voxelized occupancy grid or alternatively, render a 3D occupancy grid to a 2D image. The latter can be useful for us. As we demonstrated in Chapter 6, we can calculate an occupancy grid of a superquadric, which is, evidently, a differentiable operation. If we then also implement a differentiable renderer and use it to render depth images, we can directly compare the input depth image with the rendered depth image of a predicted superquadric. Gadelha *et al.* [62] first proposed a projection operator, which calculates the silhouette of an 3D object. Later, the authors expanded on this idea by creating a depth projection operator, capable of rendering depth images of voxel grids [63]. We can use this method to adapt our loss function for supervised learning and use it to train the network in an unsupervised manner.

7.2 Problem formulation

As with our supervised learning method, we want to create a prediction model, which would recover superquadric parameters λ from depth images $\boldsymbol{X}$. We define this as a regression task, where we use a CNN as the predictor function:

$$\hat{\boldsymbol{y}} = f_{CNN}(\boldsymbol{X}; \theta), \tag{7.1}$$

where θ are learnable parameters of the network and $\hat{\boldsymbol{y}}$ is the output. To train the network, we minimize a cost function

$$\mathcal{L}\left(\boldsymbol{X}, \hat{\boldsymbol{y}}\right), \tag{7.2}$$

where $\hat{\boldsymbol{y}}$ are the predicted superquadric parameters and $\boldsymbol{X}$ is the input depth image. Note that loss is being computed in an unsupervised fashion only with the input data and predicted parameters. No ground-truth labels are used.

7.3 Least squares loss

As our fist option for a loss function, we draw inspiration from the original least squares minimization method, proposed by Solina and Bajcsy [24]. The algorithm uses the properties of the inside-outside function to fit a superquadric to the input point cloud in an iterative process. We derive our loss function by following some general guidelines for implementing the classic iterative method in [20].

7.3.1 Point cloud fitting

The main idea behind this approach is to use a property of the inside-outside function in Eq. (3.7). Specifically, we know that when $F(\boldsymbol{p}) = 1$ for a point

$\boldsymbol{p} = [x, y, z]^T$, the point $\boldsymbol{p}$ lies directly on the superquadric surface. Our input data comes in form of depth images, so we first need to transform them into a point cloud. Since our depth images are made in orthographic perspective, the transformation into a point cloud is trivial. We define an operation

$$H(\boldsymbol{X}) = \{[i, j, \boldsymbol{X}(i, j)]^T; \boldsymbol{X}(i, j) > 0\}, \quad \forall 0 \leq i < h, 0 \leq j < w, \tag{7.3}$$

where h and w are height and width of the input image X. In other terms, the z coordinate in the depth image is encoded as the depth value, so for every pixel $\boldsymbol{X}(i, j)$ in the image, we create a 3D point $[i, j, \boldsymbol{X}(i, j)]^T$. This results in a dense point cloud of size $w \times h$, which also includes background points. To get only points on object surface, we further filter the point cloud and remove all points where $z = 0$.

Ideally, the goal of the optimization problem is then to approximate superquadric parameters λ, so that

$$F(H(\boldsymbol{X}); \lambda) = 1. \tag{7.4}$$

This means we are searching for a superquadric surface, which would fit to the input points. To change this into an objective function, we can minimize the squared distance between input points and superquadric surface directly:

$$\min \sum_{i=1}^{n} (F(H(\boldsymbol{X}); \lambda) - 1)^2, \tag{7.5}$$

where n is the total number of points, gathered from the depth image of a superquadric. This is the core idea behind this approach.

7.3.2 Self-occlusion problem

The criterion in Eq. (7.5) is under-constrained for fitting partial-view data, such as depth images. Due to object self-occlusion, the resulting point cloud can only represent at most half of the object. The other part is occluded and the information is consequently lost. The missing data cannot be retrieved and assumptions cannot be made about it. We therefore have to predict

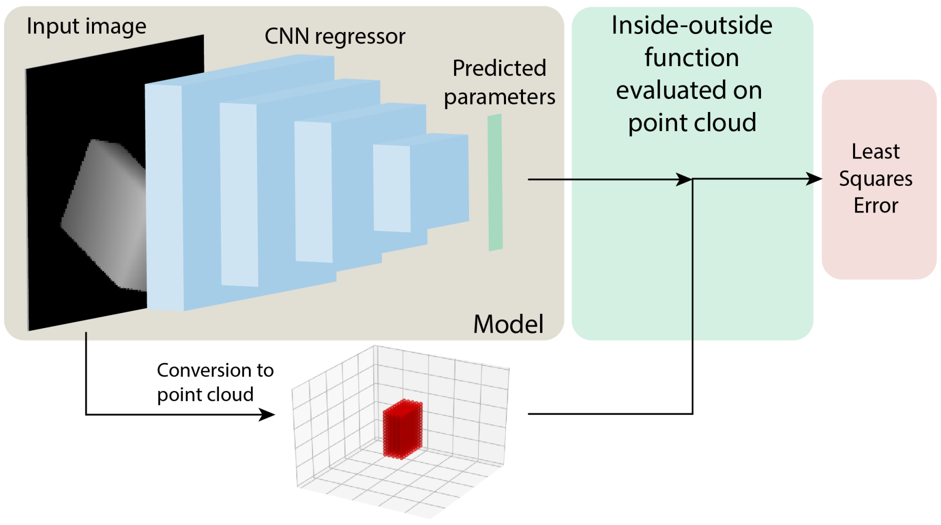

Input image
CNN regressor
Predicted parameters
Model
Inside-outside function evaluated on point cloud
Least Squares Error
Conversion to point cloud

Figure 7.1: The visualization of least squares loss. The initial depth image is converted into a point cloud. We then try to minimize the distance between these points and the surface of the predicted superquadric.

a superquadric, which fits the partial data best without redundancies. It is suggested in [20], that we multiply the objective function with the term $\sqrt{a_1 a_2 a_3}$. The term is proportional to the volume of the superquadric. By doing this, a solution with a smaller superquadric is favoured over a bigger one. The parameter space is changed, having a bigger gradient around the local minimum. With this addition, the objective function is defined as

$$\min \sum_{i=1}^{n} \sqrt{\lambda_1 \lambda_2 \lambda_3} (F(H(\boldsymbol{X})C\lambda) - 1)^2, \tag{7.6}$$

where λ_1, λ_2 and λ_3 are size parameters a_1, a_2 and a_3, respectively.

7.3.3 Final definition

Finally, we can define our loss function. We make one additional change. As with the supervised loss, we use the adjusted inside-outside function $F(p)^{\epsilon_1}$ from Eq. (6.4), so the differences in parameter space for different shapes are minimized. Combining this with the objective function from Eq. (7.6), we get a new least squares loss function

$$\mathcal{L}_{LS}(\mathbf{X}, \hat{\boldsymbol{y}}) = \sum_{i=1}^{n} \sqrt{\hat{y}_1, \hat{y}_2, \hat{y}_3}(F(H(\boldsymbol{X}); \hat{\mathbf{y}})^{\hat{y}_4} - 1)^2, \tag{7.7}$$

where $\hat{y}$ are predicted superquadric parameters and $\boldsymbol{X}$ is the input image. Again, $\hat{y}_1$, $\hat{y}_2$ and $\hat{y}_3$ are size parameters a_1, a_2 and a_3, and $\hat{y}_4$ is shape parameter ϵ_1. The visualization of the loss function can be observed in Figure 7.1.

7.4 Differentiable render loss

Our second loss function for unsupervised learning is based on a differentiable renderer. We defined and implemented the superquadric occupancy function in Chapter 6.3. Now we need to project the 3D voxel grid into a 2D depth image. We follow the procedure, described in [62] and [63] to derive the differentiable renderer. We also make some adjustments to the process, so the output matches our depth images as closely as possible. We first present the definition of a differentiable renderer and some intermediate representations and then define the loss function.

7.4.1 Silhouette projection

We begin with our occupancy voxel grid from equation 6.7. We denote the voxel grid as $\boldsymbol{V} : \mathbb{Z}^3 \to (0, 1)$. The value of a grid element is either close to 0 at indices outside of the superquadric or close to 1 at indices inside the superquadric. The sharpness of transition close to superquadric surface is determined by parameter s in Eq. (6.6). The bigger s is, more binary will the voxel grid become.

To start of, we define the silhouette projection, described in [62]. From a certain view, a silhouette of the voxel grid can be rendered to a 2D image. This is done by using a function $P : \mathbb{R}^{n^3} \to \mathbb{R}^{n^2}$:

$$P(\boldsymbol{V}) = 1 - e^{-\tau \sum_k \boldsymbol{V}(i,j,k)}, \tag{7.8}$$

where τ is another parameter. Here, $\sum_k \boldsymbol{V}(i,j,k)$ is a line integral along each line of sight, which returns the number of voxels, intersected by the line. We then raise e to the negative power of line integral, which results in a value between 0 and 1. The final result of the function $P(\boldsymbol{V})$ is therefore a silhouette, where lines, which intersect no voxels will result in value of 0, and alternatively, lines that intersect one or more voxels will result in a value that approaches 1. Parameter τ controls the sharpness of this transition; the greater it is, more binary will the transition be. The approach of using line integrals is a valid choice, since our depth images are in an orthographic projection. The projection here is done along the z axis, as evident from the equation. Indices (i,j) then represent the location of the resulting pixel in the silhouette image.

7.4.2 Depth projection

The silhouette projection function can be further expanded to also calculate depth, as described in [63]. The idea is similar, as we again make use of integral lines along z axis. However, this time, we define an intermediate function $A : \mathbb{R}^{n^3} \to \mathbb{R}^{n^3}$:

$$A(\boldsymbol{V}, i, j, k) = e^{-\tau \sum_{l=1}^{k} \boldsymbol{V}(i,j,l)}. \tag{7.9}$$

As evident, the operator is similar to the silhouette projection. The main difference lies in the inner term $\sum_{l=1}^{k} \boldsymbol{V}(i,j,l)$. Here, we calculate the cumulative sum of voxels along each line of sight for each index k. If we then raise e to the power of this cumulative value for each index, the resulting value is equal to 1 until we hit a voxel. After a voxel is hit, the value at that point approaches 0. Again, the sharpness of this transition is determined by parameter τ.

If we then want to render a depth image, we just count the values along the z axis for each line of sight. Intuitively, we count the number of voxels until the object is hit. The original proposal from [63] implies that background has infinite distance. In our case, the space is bounded into a n^3 grid, so

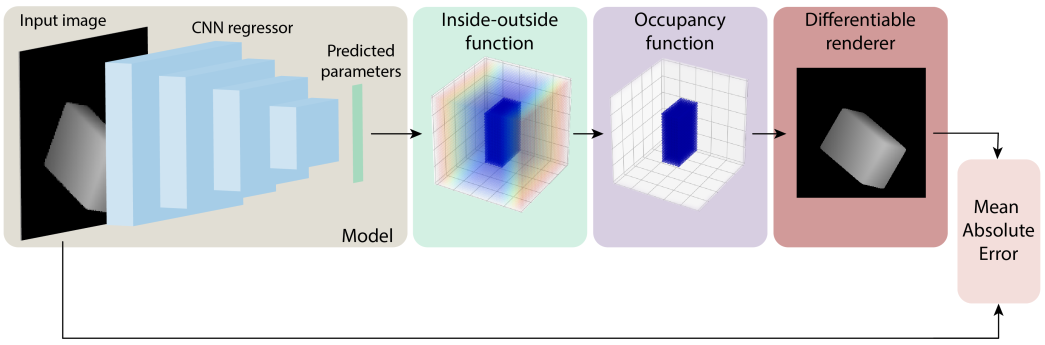


Figure 7.2: The visualization of differentiable renderer loss; We use the occupancy grid to render a depth image and then compare it with the input depth image using MAE.

we make the appropriate modification to the method. To calculate depth we use the operator $T : \mathbb{R}^{n^3} \rightarrow \mathbb{R}^{n^2}$:

$$T(\boldsymbol{V}) = 1 - \frac{1}{n} \sum_{k} A(\boldsymbol{V}, i, j, k), \tag{7.10}$$

where n is the resolution of the voxel grid in a single axis. We divide the depth by the resolution of voxel grid to normalize the value to range $[0, 1]$. A value of 1 then represents a far point and a value of 0 represents a point near to the observer. To match this with our depth images in the dataset, we flip this by subtracting from one. The resulting depth image is the near identical to images in our dataset.

7.4.3 Final definition

The final definition of the differentiable render loss function can then be assembled. We calculate the MAE between the input depth image and the rendered depth image, constructed from predicted superquadric parameters:

$$\mathcal{L}_{DR}\left(\boldsymbol{X}, \hat{\boldsymbol{y}}\right) = \frac{1}{|X|} \sum_{i,j}^{n} |\boldsymbol{X}(i,j) - T(\boldsymbol{V}_{G,\hat{\boldsymbol{y}}})(i,j)|, \tag{7.11}$$

where $\boldsymbol{V}_{G,\hat{\boldsymbol{y}}}$ is the occupancy voxel grid from Chapter 6, n is the size of the image in one axis and (i, j) are indices of a pixel in the image. We chose MAE as the error measure, since we want outliers to have a lesser impact on the loss function. This is explained in Chapter 7.6. The visualization of the loss function can be observed in Figure 7.2.

7.5 Implementation details

Again, we describe the implementation more in detail. We first talk about specifics of the least squares loss and then the about the differentiable renderer loss.

7.5.1 Implementation of least squares loss

To implement the least squares loss function, we partially reused our implementation of the inside-outside function. The function first receives as arguments the input image and predicted parameters. In the beginning, we set a parameter r, which controls the resolution of our viewport. The intuition behind this is similar to the resolution of discretized space of the occupancy loss. The input image can be of arbitrary resolution, which then gets resized to resolution r. No interpolation must be used here, so we resize the image with nearest neighbor method. To convert the resized depth images into point clouds, we iterate over all images in the batch and gather all indices and values of non-zero elements. This results on a set of points on superquadric surface, which we use to calculate the error.

For each point, we also switch x and y coordinates and then flip the values of y coordinate again. This is done to align the coordinate system used internally by the loss function and the one used by the renderer of original depth images. Predicted parameters and points clouds are then inserted into the loss function and the error is computed.

7.5.2 Implementation of differentiable renderer

The predicted superquadric parameters and the input image are given as arguments to the loss function. This time, the input depth image is only resized in the same manner, as in least squares loss. No other modifications are made. The predicted parameters are then used to calculate the occupancy voxel grid.

The main addition here is the implementation od the depth projection operator. To calculate the integral along each line of sight, a cumulative sum function is used. This is applied to the dimension in z axis. The implementation of the rest of the loss function is then trivial. Again, we permute and flip the dimensions, as described in Chapter 7.5.1, to align the coordinate systems of the internal representation and the input depth image.

7.6 Experiments and results

Here, we present the results for models, trained without supervision. We first describe experiments, related to the least squares loss. Then, we analyse the differentiable rendering approach. We also compare both methods with the supervised model, as well as the classic iterative minimization method.

7.6.1 Least squares loss

The implementation of $\mathcal{L}_{LS}$ was heavily influenced by the classic method [24]. The main advantage of least squares objective function is its simplicity and speed. When calculating loss, we only need to minimize for points, gathered from the surface of the superquadric on a depth image. In the context of training and not inference, such a process can be faster than other alternatives. Especially the usage of 3D data representation can result in expensive loss functions. On the same machine, it takes 21 minutes to complete an epoch when using the least squares loss. In contrast, the loss function for supervised learning from Chapter 6 takes 35 minutes. For one instance of

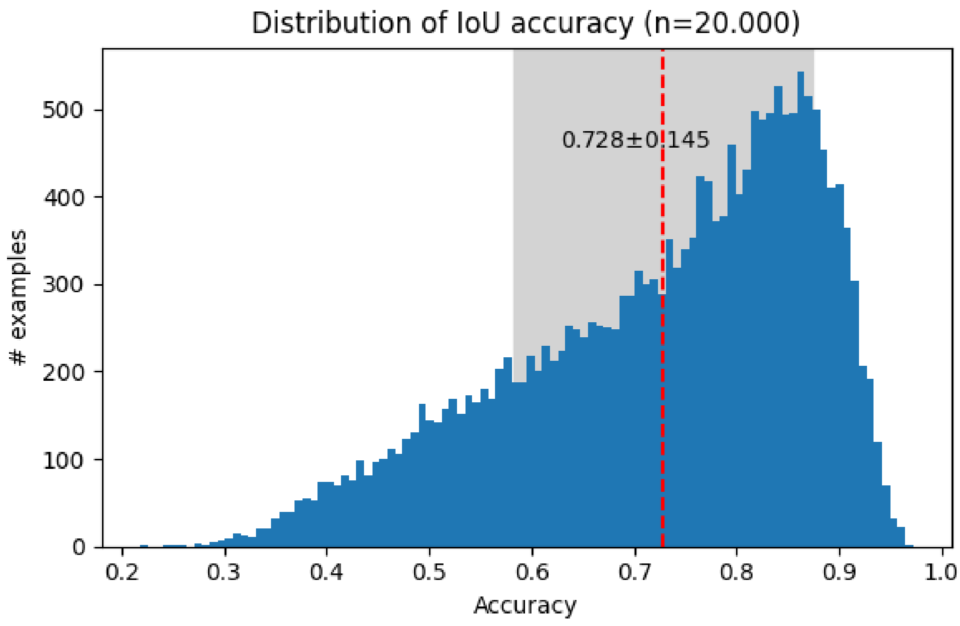
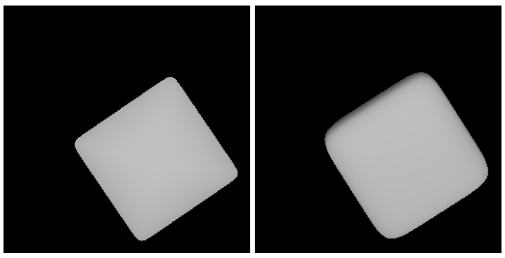


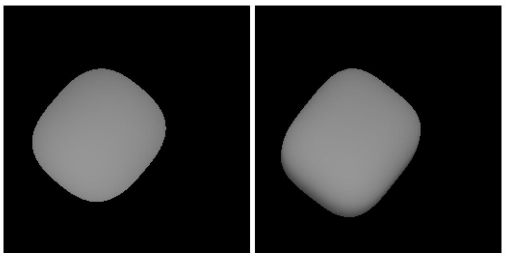
Figure 7.3: The distribution of IoU accuracy of predictions. The model was trained with loss $\mathcal{L}_{LS}$ in an unsupervised manner.

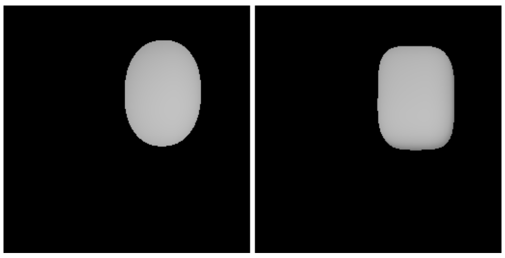
training, e.g., for 150 epochs, this can amount to more than 24 hour difference in total time.

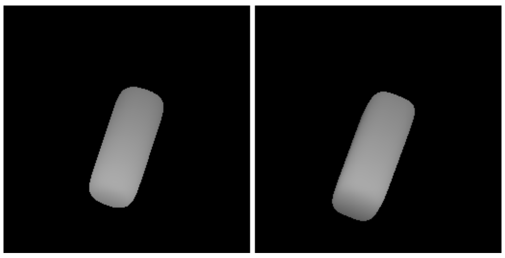
The loss converged quickly during training. On the test set, we reach an IoU accuracy of 74.82% with standard deviation of 14.5%. To get a better understanding of the performance, we look at the distribution of IoU scores in Figure 7.3. The distribution starts to increase at around 30% IoU and rises almost linearly, until reaching a peat at around 87%. It is more spread out, which is also signified with a large standard deviation, relative to the total range.

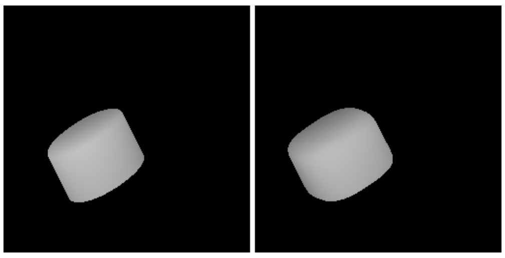
Next, we try to explain the performance of the model by looking at individual cases. In Figure 7.4, we present the qualitative results of the trained model. The first thing we notice is the similarity between ground-truth and predicted depth images. The difference is minimal and can hardly be noticed with the human eye, even in cases with low IoU. This, however, is an illusion caused by the orthographic projection of the depth images. We can

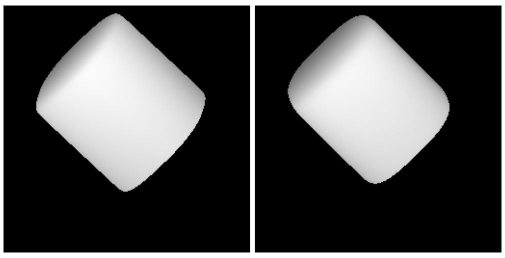

(a) 21.84% IoU

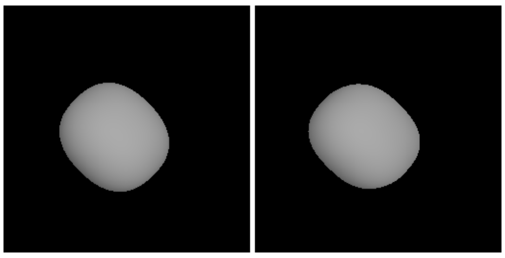

(b) 34.88% IoU


(c) 43.08% IoU


(d) 55.02% IoU


(e) 64.11% IoU


(f) 74.98% IoU


(g) 85.99% IoU

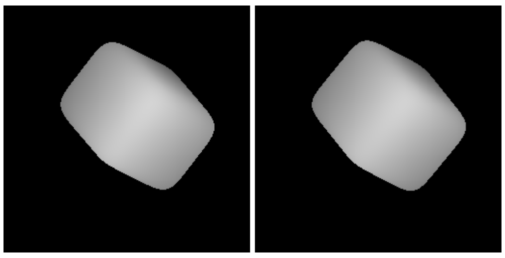
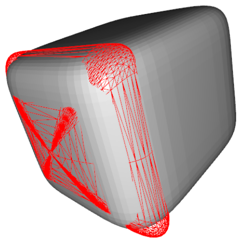
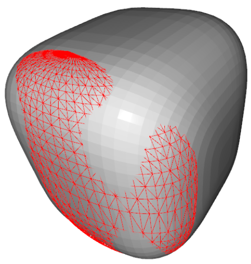
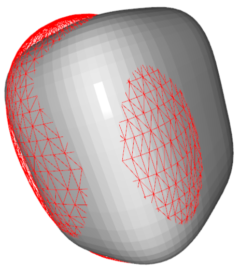
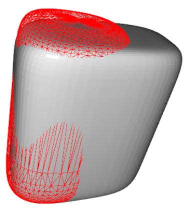

(h) 95.11% IoU

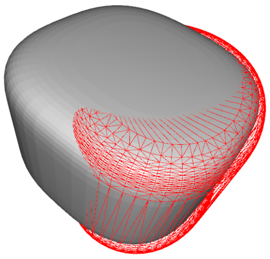
Figure 7.4: Qualitative results of the model, trained with unsupervised least squares loss $\mathcal{L}_{LS}$. The examples are sampled across the whole range. For each subfigure, the original depth image is on the left and the predicted one on the right.

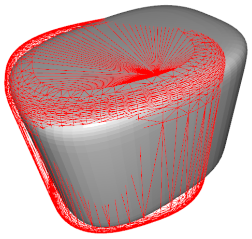

(a) 21.84% IoU

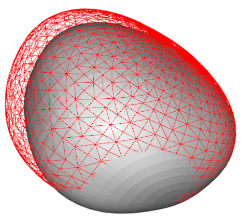

(b) 34.88% IoU


(c) 43.08% IoU


(d) 55.02% IoU


(e) 64.11% IoU


(f) 74.98% IoU


(g) 85.99% IoU

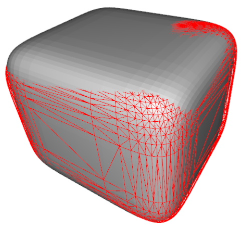

(h) 95.11% IoU

Figure 7.5: 3D renders of predictions by the model, trained without supervision with loss $\mathcal{L}_{LS}$. The examples match those in Figure 7.4. Red wiremesh represents the ground-truth superquadric and the gray object is the predicted superquadric.

see a different picture when looking at 3D renders of these same examples in Figure 7.5. The model overestimates the size of the superquadrics by a large margin. This is especially noticeable, when self-occlusion is present in the depth images. Examples with IoU $< 50\%$ are mostly images, where only one side of the superquadric is visible, similarly to examples (a), (b) and (c). This effect is magnified, when superquadrics are shorter in the direction of the occluded axis. Almost all of the low IoU scores can be attributed to this property of the model. The second contributor to this are shape parameters. The model predicts the superquadrics as having slightly sharper edges than the ground-truth examples. Otherwise, the prediction of translation and rotation is accurate.

The issue behind overestimating the volume is caused by the nature of the objective function. The loss only works one way; It guarantees that points from the depth image will be in close proximity to the surface of predicted superquadric. It does not, however, penalize a superquadric that also extends away from the points in the depth image. This results in larger predictions. This is also discussed in [24], where the authors suggest to introduce the term $\sqrt{a_1, a_2, a_3}$ into the objective function, which minimizes the overall volume of the superquadric. Despite following the suggestion and adding the term, our model is not able to minimize volume to such extent during training. Another difference is in the estimation of the initial parameters. The classic iterative method uses the depth image to infer some rough initial parameters, which are used as the starting point in the parameter space. This enables the optimization to converge to the correct local minimum. We, on the other hand, do not have such control over the initial predictions. These are dependant on network weight initialization and it wouldn't be practical to set these manually. What we can do, is to ensure, that the initial predictions don't cause singularities in the inside-outside function. This is done with a final sigmoid activation, which results in all initial predictions to be around 0.5. This inability to influence initial predictions more might be the cause why our model fails to converge to the proper local minimum.

Table 7.1: Errors for each parameter group for model, trained with differentiable render loss $\mathcal{L}_{LS}$.

ΔA	ΔE	ΔT
$+13.69\%$	-19.27%	$(+0.41\%, -0.05\%, -2.67\%)$

In Table 7.1, we observe the average error of each parameters set. Some of them are quite significant. The volume of predicted superquadrics is on average 13.69% larger than ground-truth superquadrics. This is supported by our qualitative analysis, where we found that predicted superquadrics are often over-extended, particularly when there is notable self-occlusion in the depth image. The roundness is underestimated by 19.27%. Predicted superquadrics have sharper edges, which can also be seen in most of rendered examples in Figure 7.5. The error in translation is relatively small for x and y axis, however, the error of estimating z coordinate stands out. On average, the superquadrics are predicted to be located 2.67% behind the actual position in 3D space, when viewing it along the z axis. This accounts for 6.83 units in a depth image of size 256×256.

7.6.2 Depth projection hyper-parameters

Before we implemented the differentiable render loss $\mathcal{L}_{DR}$, we first needed to implement a working differentiable renderer. This was done by using a depth projection operator, described in Chapter 7.4. We would then compare the original depth images, rendered by our dataset renderer, and predicted depth images, rendered by the new differentiable renderer. It was therefore crucial that these match as closely as possible for the same set of superquadric parameters λ. The differentiable renderer includes two parameters, which influence the resulting depth image. First is the sharpness parameter s of the occupancy function from Eq. (6.6) and the second is parameter τ in the depth projection operator from Eq. (7.9). Both of these in some way represent the rate of transition between occupant and non-occupant points

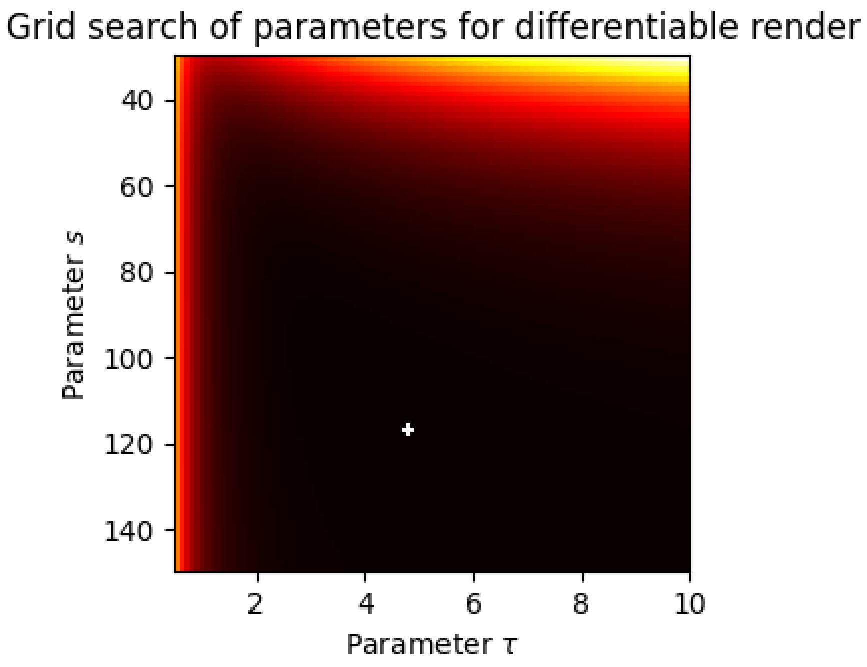


Figure 7.6: Finding optimal parameters s and τ for the differentiable renderer; We use grid search to find the parameters, where minimum MAE is achieved between original and rendered depth images. Found: $s = 117.3$ and $\tau = 4.82$ ($MAE = 0.0056$).

in space. For a depth image, low values of these parameters would result in gradual blending from background to foreground. Alternatively, we want almost binary transition, so high values are preferred.

To find optimal sharpness parameters, we take a set of superquadric parameters and render both depth images, one with dataset renderer and one with differentiable renderer. We then calculate the MAE between them. We use grid search and iterate through all the parameters to find the configuration, which results in minimum MAE. The grid search heatmap is shown in Figure 7.6. We found values $s = 117.3$ and $\tau = 4.82$ to be optimal, which results in MAE of 0.0056. Intuitively, this means that the minimum error two images will have during learning is equal to this value. This could induce

Table 7.2: Comparison of all models, trained in an unsupervised manner.

Method	IoU
unsupervised, $\mathcal{L}_{LS}$	$74.82 \pm 14.50\%$
unsupervised, $\mathcal{L}_{DR}$	$\mathbf{85.64 \pm 5.72\%}$

some bias in our model, since the optimization algorithm will try to minimize this error further.

7.6.3 Differentiable render loss

With the right hyper-parameters found for the differentiable renderer, we could incorporate it in a loss function $\mathcal{L}_{DR}$. We first tried to minimize MSE between the original and rendered images in the loss, however, this yielded poor results. The network would fit the predicted superquadric to the ground-truth superquadric only based on their 2D contours inside the depth image. The main reason behind this was MSE, where any outliers have a large impact, since the error is squared. When superquadrics were matched by contour, any perturbation of the parameters would result in contours being slightly offset. Pixels, which were a part of a superquadric in one image, but part of the background on the other, would then cause a big increase in error. The model was then stuck in this false local minimum. The issue was resolved by using MAE error, where the outliers don't carry as much weight. The network was then mostly able to escape such local minimums during training.

The results are presented in Table 7.2. As a comparison, we include the performance of model, trained using loss function $\mathcal{L}_{LS}$. We achieve an IoU accuracy of 85.64% with standard deviation 5.72%. This is a significant improvement in comparison to using $\mathcal{L}_{LS}$ as the loss function. While we increased the average IoU, we also raised the confidence of the model with lowering the standard deviation.

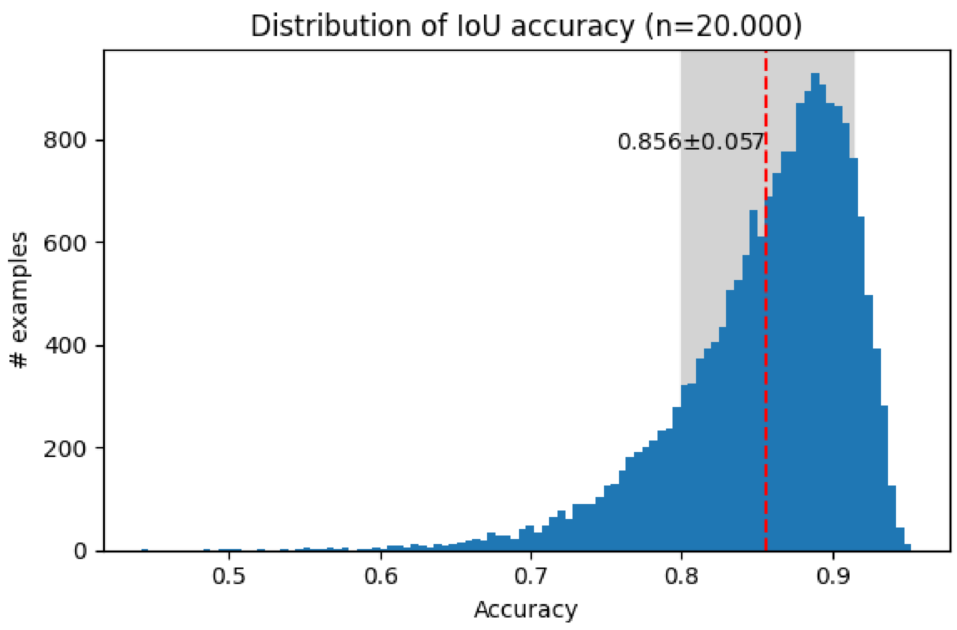
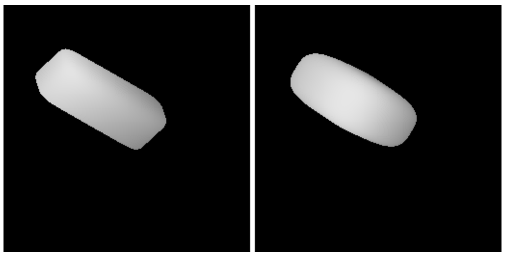


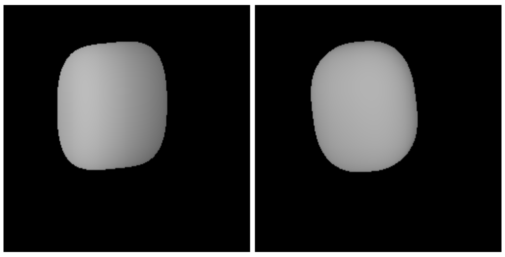
Figure 7.7: The distribution of IoU accuracy for predictions made by the unsupervised model, trained with loss function $\mathcal{L}_{DR}$.

The distribution of IoU scores can be seen in Figure 7.7. This is an overall improvement from the previous model. The minimal IoU of all examples in the test set is 44.18% and the minimal is 95.25%. The distribution starts increasing at 60% IoU and the peak is reached at around 88%. 99% of the examples in test set have an IoU bigger than 67%.

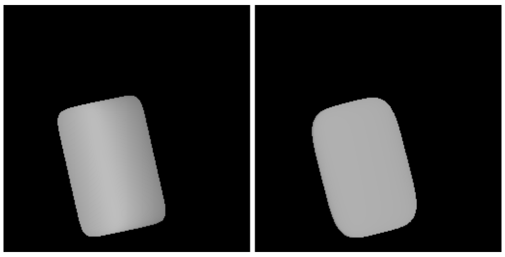
In Figure 7.8, we present the qualitative results along the range of IoU scores in increments of around 10%. Similarly to the least squares model, the predicted examples look quite similar to the ground-truth, although the differences are more noticeable here. We can see the slight difference in depth values. For reference, the rendered superquadrics can be observed in Figure 7.9. Examples with an IoU lower than 70% have one thing in common; The 2D contour between predicted and ground-truth images matches, however, the actual 3D shapes don't. The contour matching effect was already reduced by changing the final error measure in loss function $\mathcal{L}_{DR}$ from MSE to MAE, although some examples of this apparently persist. Less than 1%



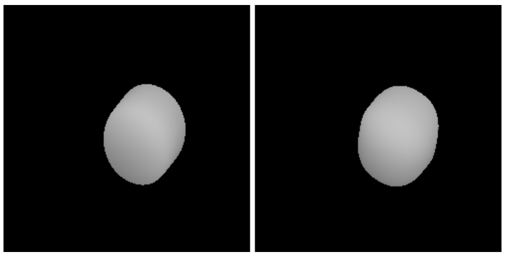
(a) 44.17% IoU



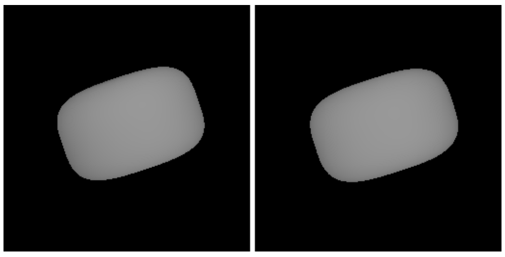
(b) 54.55% IoU



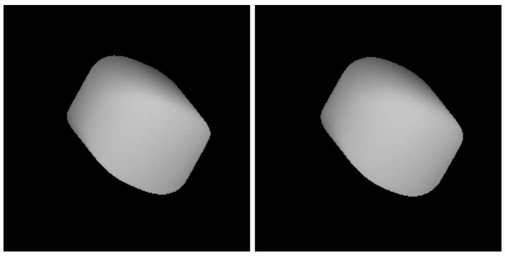
(c) 65.26% IoU



(d) 75.11% IoU

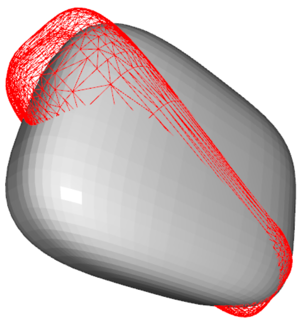


(e) 85.63% IoU

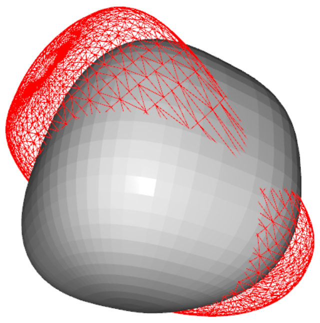


(f) 93.65% IoU

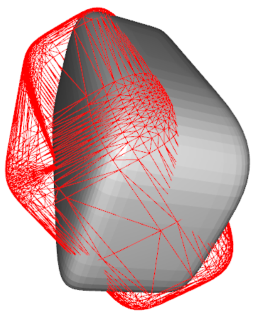
Figure 7.8: Qualitative results for model, trained with loss $\mathcal{L}_{DR}$. The examples are sampled across the whole range. For each subfigure, the original depth image is on the left and the predicted depth image is on the right.



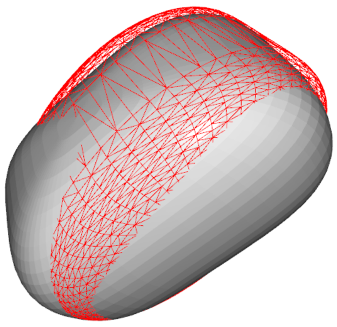
(a) 44.17% IoU



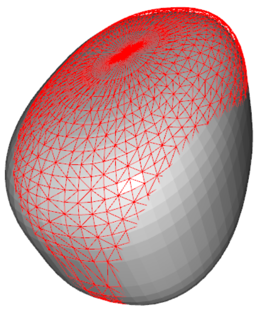
(b) 54.55% IoU



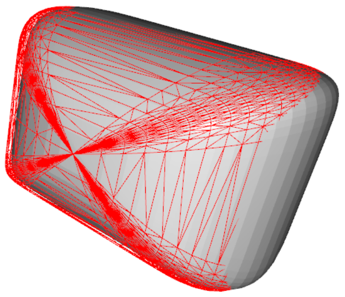
(c) 65.26% IoU



(d) 75.11% IoU



(e) 85.63% IoU



(f) 93.65% IoU

Figure 7.9: 3D renders of examples from Figure 7.8. Red wiremesh represents the ground-truth superquadric and the gray object is the predicted superquadric.

Table 7.3: Errors for each parameter group for all our models, supervised and unsupervised, and the iterative methods [24].

Method	ΔA	ΔE	ΔT
Iterative [24]	-10.30%	$+6.07\%$	$(+0.00\%, +0.00\%, +1.56\%)$
sup., $\mathcal{L}_{OC}$	$+0.14\%$	-0.36%	$(+0.01\%, +0.01\%, -0.03\%)$
unsup.,$\mathcal{L}_{LS}$	$+13.69\%$	-19.27%	$(+0.41\%, -0.05\%, -2.67\%)$
unsup.,$\mathcal{L}_{DR}$	$+8.22\%$	$+4.01\%$	$(+0.76\%, -0.63\%, -0.12\%)$

of examples suffer from this, so this issue only effects a small minority. This is the side effect of directly comparing whole images. By doing this, we also include the background in the computation of error. Ideally, we would only compare points on the superquadric and calculate the difference between them, but there is a trade-off: Any attempts to mask the background would result in overestimation of superquadric volume, similarly to how the least squares model behaves. We either focus only on the points from the original depth images and possibly have predictions with larger volumes or we compare whole images, which prevents over extension, but gives precedence to contour fitting.

Properties of model predictions can be further examined by looking at the errors in Table 7.3. The average volume of predicted superquadrics is larger by 8.22% from the ground-truth superquadrics. The model overestimates the size parameters slightly. Also increased by 4.21% is the average roundness, which causes the predicted superquadrics to have smoother edges. The error of translation parameters is relatively small when compared to the whole 3D space. Error is similar for all three exes and accounts for 0.76%, 0.63% and 0.11% in x, y and z axis, respectively. In a 256×256 image, these values would account for 1.94, 1.61 and 0.28 pixels. The error for parameter t_3 somewhat stands out from the rest and is centered closer to 0. This means than the model better estimates the depth of the input image.

Table 7.4: Comparison of our unsupervised models. Also included are the classic iterative method [24] and our model from Chapter 6, trained with supervision.

Method	IoU	Execution time [ms]
Iterative [24]	$84.51 \pm 8.89\%$	690.52 ± 275.62
supervised, $\mathcal{L}_{OC}$	$\mathbf{94.62 \pm 3.18\%}$	2.82 ± 0.76
unsupervised, $\mathcal{L}_{LS}$	$74.82 \pm 14.50\%$	2.83 ± 0.22
unsupervised, $\mathcal{L}_{DR}$	$85.64 \pm 5.72\%$	$\mathbf{2.77 \pm 0.16}$

7.6.4 Overall comparison

Table 7.4 shows the comparison between all our supervised and unsupervised models, as well as the iterative method [24]. First thing we observe is that not all our models perform better than the iterative minimization method. Unsupervised learning with the least squares loss function $\mathcal{L}_{LS}$ achieves an IoU smaller by about 10% in comparison. We believe this is due to the overextending volumes and sharper edges, as supported by parameter metrics in Table 7.3. The supervised approach is obviously superior in terms of IoU accuracy. With self-supervision, the neural network is dependant on the interpretation of the input data and at the same time the transformation from superquadric parameters to 3D data. In this regard, the complexity of recovery problem is shared between unsupervised and iterative methods, since there is a possibility of self-occlusion. The unsupervised approach by using a differentiable renderer in loss $\mathcal{L}_{DR}$ outperforms the iterative method by an absolute value of 1.13%. The average IoU is similar, however, our model also offers improved confidence with the smaller standard deviation. The execution time is similar with all CNN models, since the actual architecture is not changed. All models offer a similar speedup of around 240×. We compare the metrics in Table 7.3. Both unsupervised methods result in predictions with a slightly larger volume of the superquadrics, which is opposite of the iterative method. The large bias happens due to self-occlusion, but the iterative method then underestimates the occluded volume, rather than the other way around. We can't explain this difference in behaviour. It looks like the iterative method somehow puts a bigger focus on minimizing volume in comparison to our methods, despite implementing the same objective function. Our methods are therefore comparable when predicting superquadric shape and size, but have an edge when predicting its general pose.

7.7 Discussion

The possibility of unsupervised approach presents a large step towards large-scale learning on real-world datasets with actual objects. The process of labeling 3D data is expensive and sometimes practically impossible. To have such a method enables us to not restrict a hypothetical practical application only with laboratory-grade conditions, but also to solve real problems in an unconstrained environment. We presented here two new loss functions. The creation of first was heavily inspired by one of the more established methods of superquadric recovery. The second one was the result of new ideas and advancements in deep learning. In the end, both models, trained by these loss functions, managed to outperform the classic iterative method. We presented the advantages and disadvantages of using both loss functions. The supervised approach, though unsurprisingly, still offers a more stable and accurate model. Nevertheless, it is crucial that we have the ability to choose the appropriate method based on the availability of data, time, or other resources.

Chapter 8

Conclusion

This thesis presents a novel framework for superquadric recovery using deep learning techniques. It foremost provides the possible approaches of how to deal with 3D representation and parametric models in the context of neural networks. By recovering a single superquadric from a depth image, it builds the necessary fundamental knowledge and acts as a preliminary step towards methods, capable of interpreting more complex scenes with more superquadrics.

8.1 Summary

We divided our methodology into two parts; supervised and unsupervised. With supervised learning, we created a model which outperforms the classic approach of parametric recovery and established a new state-of-the-art for recovery of single superquadrics. We achieve an IoU of 94.62% and a speedup of the execution time by a factor of 240. In comparison to the 68.31% IoU of the iterative method, this is a significant improvement. We managed to solve the original goal of a geometrically-aware loss function, which is able to interpret superquadrics in general position. With unsupervised learning, we introduced a methodology to be used in an unconstrained environment with unknown a priori shapes. The model, trained on the least squares loss achieved an IoU accuracy of 74.82%. The main issue was the under-constrained objective function, which sometimes resulted in overextension of the predicted superquadrics. The other model was trained using a differentiable renderer and achieved an IoU of 85.64%, which outperforms the classic iterative method by a small margin, however, due to self-occlusion, cannot compete with the supervised approach directly.

8.2 Future work

As this framework only presents solutions to recover single superquadrics from depth images, there are many possible improvements to be made. Especially, the ability to process multiple superquadrics at once would be crucial to understand more complex scenes. This could come in form of a serial pipeline, where a segmentation step would first divide the input image into multiple part-level sections, followed by one of our models, which could predict parameters for each of those sections. The ultimate goal, though, is an end-to-end solution. We would need to build a model with a supporting architecture, which would predict a variable number of superquadrics. This approach would also require a loss function, which would fit the data according to MDL principle, but would also avoid trivial solutions. Many improvements can be made to how we prepare the training data, which could be augmented in some way. For example, we could add noise to mimic real-world depth images, which are usually noisy due to reflection and refraction of light rays. For real-world data, various 3D scanners are now ready to capture also RGB data. We could then train a model with combined color and depth image channels. The methods, proposed in this thesis, offer a promising new direction for research of superquadric recovery with deep learning and at the same time the flexibility to apply them to many different scenarios.

Bibliography

[1] D. Marr, Vision: A Computational Investigation into the Human Representation and Processing of Visual Information, Henry Holt and Co., Inc., 1982.

[2] A. H. Barr, Superquadrics and angle-preserving transformations, IEEE Computer Graphics and Applications 1 (1981) 11–23.

[3] F. Solina, R. Bajcsy, Range image interpretation of mail pieces with superquadrics, in: Conference on Artificial Intelligence, 1987, pp. 733–737.

[4] A. Jaklič, M. Erič, I. Mihajlović, Ž. Stopinšek, F. Solina, Volumetric models from 3D point clouds: The case study of sarcophagi cargo from a 2nd/3rd century AD Roman shipwreck near Sutivan on island Brač, Croatia, Journal of Archaeological Science 62 (2015) 143–152.

[5] Ž. Stopinšek, F. Solina, 3D modeliranje podvodnih posnetkov, in: Si robotika, 2017, pp. 103–114.

[6] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[7] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: International Conference on Learning Representations, 2015.

[8] I. Binford, Visual perception by computer, in: Conference of Systems and Control, 1971.

[9] R. A. Brooks, R. Creiner, T. O. Binford, The ACRONYM model-based vision system, in: International Joint Conference on Artificial Intelligence, 1979, pp. 105–113.

[10] I. Biederman, Recognition-by-components: a theory of human image understanding., Psychological Review 94 (1987) 115–147.

[11] R. C. Munck-Fairwood, L. Du, Shape using volumetric primitives, Image and Vision Computing 11 (1993) 364–371.

[12] M. Zerroug, R. Nevatia, Segmentation and 3-D recovery of curved-axis generalized cylinders from an intensity image, in: International Conference on Pattern Recognition, 1994, pp. 678–681.

[13] D. Dion, D. Laurendeau, R. Bergevin, Generalized cylinders extraction in a range image, in: International Conference on Recent Advances in 3-D Digital Imaging and Modeling, 1997, pp. 141–147.

[14] D. Keren, D. Cooper, J. Subrahmonia, Describing complicated objects by implicit polynomials, IEEE Transactions on Pattern Analysis and Machine Intelligence 16 (1994) 38–53.

[15] S. Muraki, Volumetric shape description of range data using "blobby model", in: Computer Graphics and Interactive Techniques, 1991, pp. 227–235.

[16] D. Terzopoulos, A. Witkin, M. Kass, Symmetry-seeking models and 3D object reconstruction, International Journal of Computer Vision 1 (1988) 211–221.

[17] L. H. Staib, J. S. Duncan, Model-based deformable surface finding for medical images, IEEE Transactions on Medical Imaging 15 (1996) 720–731.

[18] A. P. Pentland, Perceptual organization and the representation of natural form, in: Readings in Computer Vision, 1987, pp. 680–699.

[19] A. J. Hanson, Hyperquadrics: smoothly deformable shapes with convex polyhedral bounds, Computer Vision, Graphics, and Image Processing 44 (1988) 191–210.

[20] A. Jaklič, A. Leonardis, F. Solina, Segmentation and recovery of superquadrics, Kluwer, 2000.

[21] G. Lamé, Examen des différentes méthodes employées pour résoudre les problèmes de géométrie, V. Courcier, 1818.

[22] A. P. Pentland, Recognition by parts, in: International Conference on Computer Vision, 1987, pp. 612–620.

[23] R. Bajcsy, F. Solina, Three dimensional object representation revisited, in: International Conference on Computer Vision, 1987, pp. 231–240.

[24] F. Solina, R. Bajcsy, Recovery of parametric models from range images: The case for superquadrics with global deformations, IEEE Transactions on Pattern Analysis and Machine Intelligence 12 (1990) 131–147.

[25] T. E. Boult, A. D. Gross, Recovery of superquadrics from depth information, in: Workshop on Spatial Reasoning and Multi-Sensor Fusion, 1987, pp. 128–137.

[26] A. D. Gross, T. E. Boult, Error of fit measures for recovering parametric solids, in: International Conference on Computer Vision, 1988, pp. 690–694.

[27] S. Voisin, M. A. Abidi, S. Foufou, F. Truchetet, Genetic algorithms for 3D reconstruction with supershapes, in: International Conference on Image Processing, 2009, pp. 529–532.

[28] D. Terzopoulos, D. Metaxas, Dynamic 3D models with local and global deformations: deformable superquadrics, IEEE Transactions on Pattern Analysis and Machine Intelligence 13 (1991) 703–714.

[29] A. Leonardis, A. Jaklic, F. Solina, Superquadrics for segmenting and modeling range data, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (1997) 1289–1295.

[30] J. Rissanen, Modeling by shortest data description, Automatica 14 (1978) 465–471.

[31] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, J. Xiao, 3D Shapenets: A deep representation for volumetric shapes, in: Conference on Computer Vision and Pattern Recognition, 2015, pp. 1912–1920.

[32] J. Wu, Y. Wang, T. Xue, X. Sun, B. Freeman, J. Tenenbaum, MarrNet: 3D shape reconstruction via 2.5D sketches, in: Advances in Neural Information Processing Systems, 2017, pp. 540–550.

[33] A. Dai, C. Ruizhongtai Qi, M. Nießner, Shape completion using 3D-encoder-predictor CNNs and shape synthesis, in: Conference on Computer Vision and Pattern Recognition, 2017, pp. 5868–5877.

[34] S. Liu, L. Giles, A. Ororbia, Learning a hierarchical latent-variable model of 3D shapes, in: International Conference on 3D Vision, 2018, pp. 542–551.

[35] J. J. Park, P. Florence, J. Straub, R. Newcombe, S. Lovegrove, DeepSDF: Learning continuous signed distance functions for shape representation, in: Conference on Computer Vision and Pattern Recognition, 2019, pp. 165–174.

[36] C. R. Qi, H. Su, K. Mo, L. J. Guibas, PoinNet: Deep learning on point sets for 3D classification and segmentation, in: Conference on Computer Vision and Pattern Recognition, 2017, pp. 652–660.

[37] H. Fan, H. Su, L. J. Guibas, A point set generation network for 3D object reconstruction from a single image, in: Conference on Computer Vision and Pattern Recognition, 2017, pp. 605–613.

[38] R. Zhu, H. Kiani Galoogahi, C. Wang, S. Lucey, Rethinking reprojection: Closing the loop for pose-aware shape reconstruction from a single image, in: International Conference on Computer Vision, 2017, pp. 57–65.

[39] S. Miao, Z. J. Wang, R. Liao, A CNN regression approach for real-time 2D/3D registration, IEEE Transactions on Medical Imaging 35 (2016) 1352–1363.

[40] Y. Xiang, T. Schmidt, V. Narayanan, D. Fox, PoseCNN: A convolutional neural network for 6D object pose estimation in cluttered scenes, in: Robotics: Science and Systems, 2018.

[41] Y. Rubner, C. Tomasi, L. J. Guibas, The earth mover's distance as a metric for image retrieval, International Journal of Computer Vision 40 (2000) 99–121.

[42] X. Zhu, X. Liu, Z. Lei, S. Z. Li, Face alignment in full pose range: A 3D total solution, IEEE Transactions on Pattern Analysis and Machine Intelligence 41 (2017) 78–92.

[43] S. Tulsiani, H. Su, L. J. Guibas, A. A. Efros, J. Malik, Learning shape abstractions by assembling volumetric primitives, in: Conference on Computer Vision and Pattern Recognition, 2017, pp. 2635–2643.

[44] D. Paschalidou, A. O. Ulusoy, A. Geiger, Superquadrics revisited: Learning 3D shape parsing beyond cuboids, in: Conference on Computer Vision and Pattern Recognition, 2019, pp. 10344–10353.

[45] D. Paschalidou, L. V. Gool, A. Geiger, Learning unsupervised hierarchical part decomposition of 3D objects from a single RGB image, in:

Conference on Computer Vision and Pattern Recognition, 2020, pp. 1060–1070.

[46] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, et al., Shapenet: An information-rich 3d model repository, arXiv e-prints (2015). `arXiv:1512.03012`.

[47] J. Slabanja, B. Meden, P. Peer, A. Jaklič, F. Solina, Segmentation and reconstruction of 3D models from a point cloud with deep neural networks, in: Conference on Information and Communication Technology Convergence, 2018, pp. 118–123.

[48] T. Oblak, K. Grm, A. Jaklič, P. Peer, V. Štruc, F. Solina, Recovery of superquadrics from range images using deep learning: a preliminary study, in: International Work Conference on Bioinspired Intelligence, 2019, pp. 45–52.

[49] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, The Quickhull algorithm for convex hulls, ACM Transactions on Mathematical Software 22 (1996) 469–483.

[50] W. S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, The Bulletin of Mathematical Biophysics 5 (1943) 115–133.

[51] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–1780.

[52] A. Cleeremans, D. Servan-Schreiber, J. L. McClelland, Finite state automata and simple recurrent networks, Neural computation 1 (1989) 372–381.

[53] D. E. Rumelhart, Learning internal representations by error propagation, Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1 (1986) 318–362.

[54] F. Rosenblatt, Principles of neurodynamics: perceptrons and the theory of brain mechanisms, in: Brain theory, 1986, pp. 245–248.

[55] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: International Conference on Machine Learning, 2010, p. 807–814.

[56] A. L. Maas, A. Y. Hannun, A. Y. Ng, Rectifier nonlinearities improve neural network acoustic models, in: ICML Workshop on Deep Learning for Audio, Speech, and Language Processing, 2013.

[57] D. H. Hubel, T. N. Wiesel, Receptive fields of single neurones in the cat's striate cortex, The Journal of Physiology 148 (1959) 574.

[58] K. Fukushima, S. Miyake, Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition, in: Competition and Cooperation in Neural Nets, 1982, pp. 267–285.

[59] Y. LeCun, P. Haffner, L. Bottou, Y. Bengio, Object recognition with gradient-based learning, in: Shape, Contour and Grouping in Computer Vision, 1999, pp. 319–345.

[60] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database, in: Computer Vision and Pattern Recognition, 2009, pp. 248–255.

[61] D. P. Kingma, J. L. Ba, Adam: A method for stochastic gradient descent, in: International Conference on Learning Representations, 2015.

[62] M. Gadelha, S. Maji, R. Wang, 3D shape induction from 2D views of multiple objects, in: International Conference on 3D Vision, 2017, pp. 402–411.

[63] M. Gadelha, R. Wang, S. Maji, Shape reconstruction using differentiable projections and deep priors, in: International Conference on Computer Vision, 2019, pp. 22–30.