Weakly-supervised Surface Reconstruction Using Floating Radial Basis Functions

Hossein Hajipour

Saarbrücken, Germany

Master's Thesis

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

May 2018

Supervisor:
Prof. Dr. Christian Theobalt, Max Planck Institute for Computer Science, Saarbrücken, Germany

Co-supervisor:
MSc Ayush Tewari, Max Planck Institute for Computer Science, Saarbrücken, Germany

Date
May 18, 2018, in Saarbrücken

Reviewers:

Prof. Dr. Christian Theobalt, Max Planck Institute for Computer Science, Saarbrücken, Germany

Dr. Michael Zollhöfer, Stanford University, Palo Alto, U.S.A.

Dean

Prof. Dr. Frank-Olaf Schreyer, Saarland University
Saarbrücken, Germany

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

(Datum/Date)	(Unterschrift/Signature)

Abstract

We propose a novel learning-based approach for monocular reconstruction of arbitrary 3D shapes. Our approach takes a single image as input and outputs a detailed signed distance function (SDF) representation that describes the geometry of the object based on its zero-th level-set. We represent SDFs efficiently based on a linear combination of floating radial basis functions, where both the supporting points as well as the blending weights are free variables. At the core of our approach is a weakly-supervised learning scheme that does not require access to the ground-truth parameters of the radial basis functions during training. Our approach efficiently combines the desirable low space complexity of approaches which only reconstruct point clouds, with the surface approximation properties of distance field-based reconstruction techniques. We perform extensive qualitative and quantitative evaluations to compare our approach with the state-of-the-art methods. These evaluations demonstrate the capabilities of our approach to reconstruct high-quality surfaces of objects, even with local details. In addition, we show that our approach produces comparable results to the state-of-the-art methods in terms of quality and accuracy of the estimated 3D shape.

Acknowledgements

I would first like to express my sincere gratitude to Professor Christian Theobalt for giving me the opportunity to write my thesis under his supervision in the ”Graphics, Vision and Video” group, and for his support during my research.

Especial thanks to Dr. Michael Zollhöfer and Ayush Tewari for their invaluable guidance and support. Their guidance helped me throughout the research and writing of this thesis. Whenever I had a question about my research or needed someone to discuss an idea, they took their time to discuss the problem and helped me find the solutions.

I also would like to thank Abhimitra Meka, Edgar Tretschk, and Jiayi Wang for proofreading the thesis.

At last, I would like to thank my family, especially my parents for supporting me spiritually throughout my life.

Contents

List of Figures

List of Abbreviations

1D	One dimensional.
2D	Two dimensional.
3D	Three dimensional.
CNN	Convolutional neural network.
CPU	Central processing unit.
CUDA	Compute unified device architecture.
FC	Fully-connected.
GPU	Graphics processing unit.
LMA	Levenberg-Marquardt algorithm.
LSTM	Long short-term memory.
NN	Neural network.
RAM	Random-access memory.
RBF	Radial basis function.
ReLU	Rectified linear unit.
RGB	Red green blue.
SDF	Signed distance function.
SfM	Structure from motion.
SLAM	Simultaneous localization and mapping.

List of Symbols

Ψ	A radial basis function.
P	3-variate linear polynomial.
N_c	Number of center points used to define the radial basis function.
N_s	Number of sample points.
α_j	j-th blending weight of the radial basis function.
β_k	k-th parameter of the 3-variate linear polynomial.
$\mathbf{c}_j$	j-th center point of the radial basis function.
d_i	Closest signed distance of the i-th off-surface point to the sample points.
λ	A parameter for controlling the width of the Gaussian function.
$\mathcal{X}$	Set of free variables.
E_{total}	Our proposed loss function.
E_{point}	Zero-crossing term.
E_{normal}	Normal term.
E_{empty}	Empty-space term.
w_{point}	Weight for zero-crossing term.
w_{normal}	Weight for normal term.
w_{empty}	Weight for empty-space term.
$\mathbf{s}_i$	i-th sample point.
$\mathbf{n}_i$	Normal vector of the i-th sample point.
$\mathcal{R}$	Set of random points with their closest distance to the sample points.
$\mathbf{r}$	A random point in a unit cube.
lr	Learning rate.

Chapter 1

Introduction

3D object reconstruction from a single monocular image has many applications including augmented reality, virtual reality, e-commerce, and industrial design. Inferring the 3D structure of the underlying scene in the 2D image is an easy task for humans, but it is a challenging and ill-posed problem for a machine. This difficulty is mainly because a 2D image could be the projection of many 3D shapes (Figure 1.1).

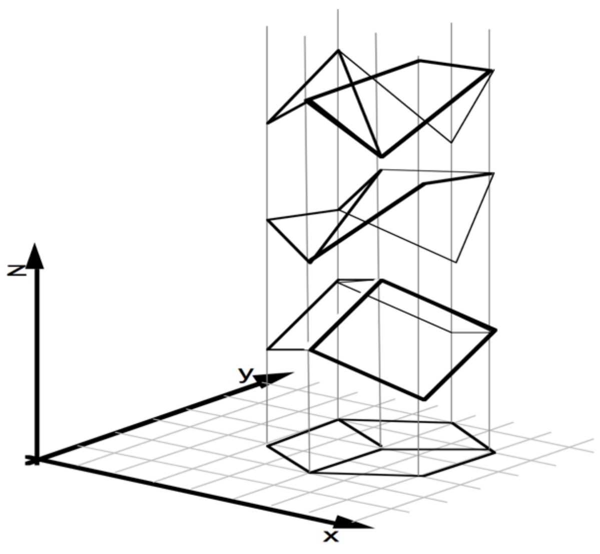

Figure 1.1: A 2D image could be the projection of many different 3D structures. (Sinha & Adelson (1993)).

Recently, this task has been tackled by learning-based techniques that exploit priors of the considered object classes. Most of these approaches use a convolutional neural network (CNN) to map the input monocular image to the 3D geometry of the object.

Significant progress has been made in deep-learning-based approaches to learn an efficient and effective representation of the 2D images (Krizhevsky *et al.* (2012); Oquab *et al.* (2014); Szegedy *et al.* (2016); Zeiler & Fergus (2014)). Using current deep-learning-based approaches, we can solve different types of tasks in computer vision (Dong *et al.* (2016); Girshick *et al.* (2014); Long *et al.* (2015)) and computer graphics (Gryka *et al.* (2015); Kalantari *et al.* (2015); Wang *et al.* (2017)). However, learning a model to reconstruct the 3D shape of a single monocular image remains a challenging task.

To tackle the reconstruction of 3D shapes from 2D images, different deep-learning-based methods employ different 3D representations. One type of these methods uses occupancy grids to represent the 3D geometry of a given 2D image (Choy *et al.* (2016); Häne *et al.* (2017)). With such an approach we can represent the arbitrary topologies of different categories of shapes. However, due to the cubic space complexity of the volume, we can reconstruct only a coarse representation of the 3D shape. Most of these methods commonly use a $32\times32\times32$ volume to represent the 3D geometry of the shapes. Another type of deep-learning-based methods uses a point cloud-based representation to reconstruct the on-surface points of different shapes (Fan *et al.* (2017)). The point cloud representation is an efficient and sparse representation of the 3D shapes. However, it cannot capture the surface normals and fine-details of the objects. To train these deep-learning-based approaches, usually ShapeNet (Chang *et al.* (2015)) or ModelNet (Wu *et al.* (2015)) datasets are used to prepare a large set of 2D images with their corresponding 3D information. Figure 1.2 shows an example of the voxel-based (left) and the point-based (right) representations.

Voxel-based approaches usually train a deep convolutional neural network that maps an input image to an occupancy grid. To train these models, most of these methods use pairs of 2D images with their corresponding occupancy grids. As mentioned before, due to the space and computational complexity, these approaches only predict a coarse 3D shape of the object. Moreover, by increasing the resolution of the volume, the ratio between occupied and non-occupied voxels becomes smaller and smaller.

Point-based approaches also employ a deep convolutional neural network to predict the 3D geometry from a 2D image. In these approaches instead of voxel-grids, the point clouds are used to represent the 3D shape of the object, which is a

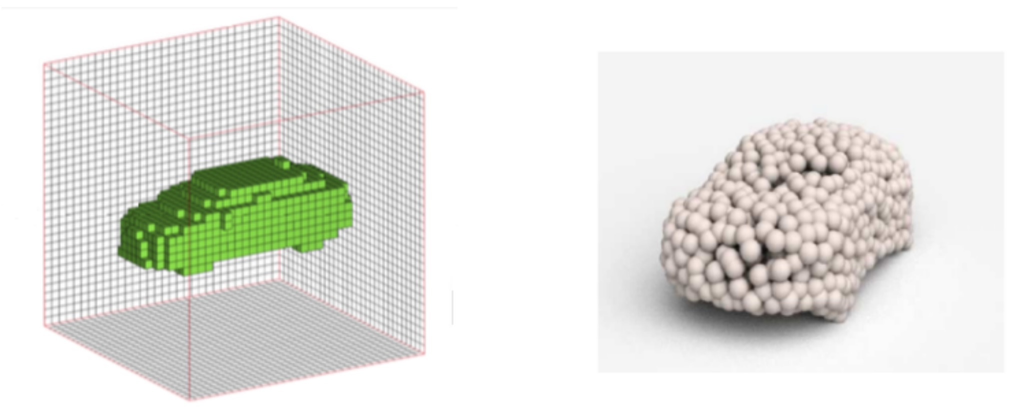

Figure 1.2: An example of the voxel-based (left)(Yumer & Mitra (2016)) and the point-based representation (right) (Fan *et al.* (2017)) of a car.

more efficient way of 3D representation in comparison to the voxel-grid representation. This efficiency is because of predicting only the position of the points in 3D space without considering the non-occupied regions. We can train these models using pairs of 2D images with their corresponding point clouds. Even though a point cloud is an efficient 3D representation, however, it cannot be represented as a CAD model or even a simple mesh.

In this thesis, we propose a deep-learning-based approach which tries to combine the advantages of these two worlds. We seek for the space complexity of the point-based techniques, while still maintaining a surface representation. Our proposed learning-based model maps the given single monocular image to the detailed signed distance function (SDF) representation of its 3D shape. With this representation, we can describe the geometry of the object based on the zero-th level-set of the signed distance function. To represent the signed distance functions efficiently, we use a linear combination of radial basis functions (RBFs). Using radial basis functions, we can represent complex shapes based on a small number of basis functions (Turk & O'brien (1999)).

In our approach, we employ a deep convolutional encoder to represent the 3D shape of the monocular input image as an implicit surface representation. Using this encoder, we can extract the features of the input image and map them to the set of parameters of the RBF. In fact, in this thesis instead of directly regressing points on the shape, we regress a set of basis functions and blending coefficients of the RBF that describe the underlying distance field (Figure 1.3).

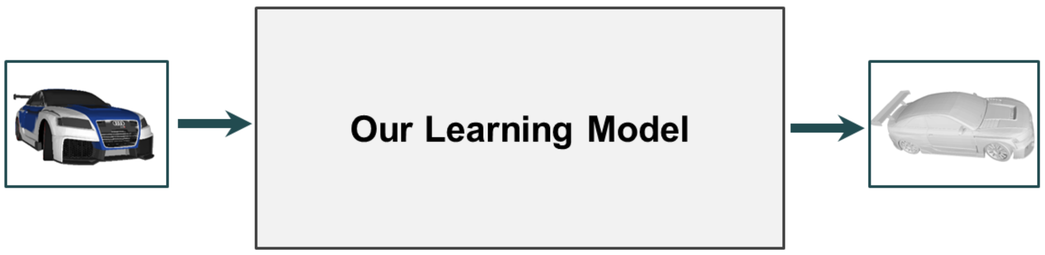

Figure 1.3: Given a single monocular image, our learning model outputs a detailed signed distance function which represents the underlying geometry of the object.

Since there is no dataset that provides the ground-truth distance fields, we propose a weakly-supervised training approach that uses point clouds for optimizing the best fitting distance field approximation. In this thesis our three main contributions are:

- An end-to-end approach for monocular reconstruction of 3D shapes,
- A weakly-supervised training approach that does not require ground-truth distance fields during training,
- An efficient GPU implementation of the proposed weakly-supervised loss function.

The remainder of the thesis is organized as follows: In the next chapter (Chapter 2), we give an overview of the related optimization-based and learning-based approaches for 3D reconstruction. We compare different types of 3D representations and specify their advantages and limitations. We introduce all of the ingredients which we will use to devise our approach in Chapter 3. In this chapter, we provide a brief introduction about, implicit surface representations, convolutional encoders, and GPU programming. In Chapter 4, we provide the details of the proposed method and explain all the terms of the weakly-supervised loss. After describing all aspects of the proposed learning-based model, we illustrate the training procedure (Chapter 5). In Chapter 6, we provide the implementation details and the results of our approach on different categories. We discuss the potential future works and possible extensions to our proposed model in Chapter 7. Finally, we summarize the proposed work in Chapter 8.

Chapter 2

Related Work

2.1 Point Cloud Reconstruction

3D reconstruction from point clouds has a long history in computer vision and computer graphics. This problem has been mostly tackled by optimization based methods which fit a surface to the scattered data points. Recently, with the availability of large-scale 3D data, a few deep-learning-based methods have been proposed to tackle this problem. In the rest of this section, we review some optimization-based and learning-based methods which have been proposed to reconstruct the surface from a given point cloud.

2.1.1 Optimization-based Reconstruction

We can classify optimization-based methods, which have been proposed for solving the point cloud reconstruction problem, in many different ways. Süßmuth *et al.* (2010) classified these methods into Delaunay based methods and implicit methods.

Delaunay based methods use Voronoi diagrams or the Delaunay triangulation of the point cloud to reconstruct the surface of the object (Cazals & Giesen (2006)). In Boissonnat (1984), which is one of the first Delaunay based methods, the reconstruction problem is reduced to the computation of the local reconstruction based on the tangent plane at the sample points. The final shape is reconstructed by patching these local reconstructions together. Gopi *et al.* (2000), proposed an extension of the Boissonnat (1984) work which uses normals and tangent planes to compute the Delaunay neighborhood. The main benefit of many of these methods is that under certain sampling conditions they guarantee the correctness of the output. However, as these methods usually try to interpolate the input points, they are very susceptible to noise. For more details about these methods, we refer to Cazals & Giesen (2006).

Implicit methods approximate the input points as the zero-th level-set of an implicit function. This approximation could be represented by different functions f. Hoppe *et al.* (1992), compute the function f as a piece-wise linear function. This function is computed based on the signed distance to the tangent plane of the closest point. Ohtake *et al.* (2003) propose an implicit surface representation using the multi-level partition of unity. They reconstruct the surface locally using quadratic polynomials and blend the local functions to reconstruct the surface globally. There is a powerful class of implicit functions that reconstruct the surface based on radial basis functions. The value of these functions depends only on the distance of the points from some other points which are called center points. In Turk & O'brien (1999), they use radial basis functions by generating artificial off-surface points to approximate the surface. They generate these off-surface points based on the normals directions of the point cloud. In this work, as they use one center point per each point, the reconstruction of the surface with a large set of points would be computationally expensive or even infeasible. In Carr *et al.* (2001), they use a greedy algorithm to reduce the number of center points which leads to a fast approximation method for fitting the RBF. Süßmuth *et al.* (2010) introduce the concept of floating center points to obtain a better approximation while using a smaller number of center points. They reconstruct the surface hierarchically using floating radial basis functions. Figure 2.1 show an example of reconstructing a surface using this method.

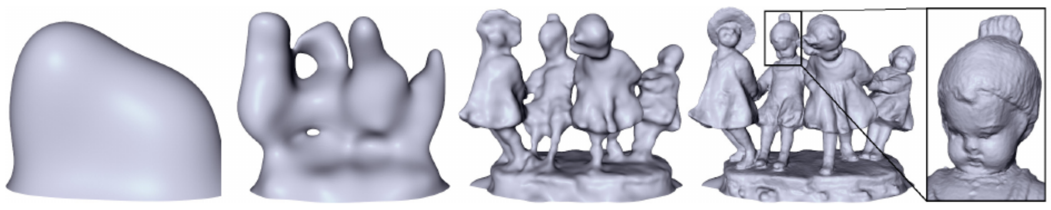

Figure 2.1: An example of hierarchical reconstruction using floating radial basis functions. (Süßmuth *et al.* (2010)) .

2.1.2 Learning-based Reconstruction

A few deep-learning-based approaches are proposed to reconstruct the surface of the object from a point cloud. To tackle this problem using deep-learning-based methods we need a large set of 3D data. ShapeNet (Chang *et al.* (2015)), and ModelNet (Wu *et al.* (2015)) provide large-scale 3D datasets which contain different categories of CAD models. Using such a large-scale data set of 3D data, AtlasNet (Groueix *et al.* (2018)) and Deep Marching Cubes (Liao *et al.* (2018)) proposed deep-learning-based methods to reconstruct the surface of the given point cloud.

In AtlasNet (Groueix *et al.* (2018)), to reconstruct the surface from a point cloud, they train a deep neural network model. This model gets the point cloud as input and approximates the target surface by learning the parameterization and a 2D embedding of the shape. They also propose a deep learning model to reconstruct the surface from a given RGB image, which we review in Section 2.3. In our method, we also represent the surface as a set of parameters of a function. However, to reconstruct the surface of the object we do not need to learn the embedding information, as in our representation the zero-th level-set of the reconstructed function represents the surface.

Liao *et al.* (2018) propose Deep Marching Cubes to reconstruct the surface from a point cloud using a learning-based model. They propose a modified differentiable representation of the marching cubes algorithm (Lorensen & Cline (1987)). This modified marching cubes algorithm is used as the final layer to reconstruct the surface of the given data points. This approach is capable of reconstructing a surface that corresponds to a point cloud with arbitrary topology in a volume. However, due to the growth of the volume, this model can not reconstruct surfaces with high resolution. In contrast, in our proposed method, as we represent the surface of the shape as an implicit function, to capture more details of the shapes, we only have to increase the number of parameters, which is in order of $O(n)$.

2.2 Multi-view Reconstruction

3D Reconstruction based on multi-view geometry is a well-researched (Hartley & Zisserman (2003)) topic in graphics and vision. Traditionally this problem is solved by optimization-based methods, where different viewpoints of the objects are used to reconstruct the underlying 3D information. Recently, deep-learning-based methods have also been proposed to tackle the 3D reconstruction problem using multi-view information. Most of these learning-based methods use synthetically rendered data as training data, where the underlying 3D information is available for all of the images. To provide more information about this category of methods we review some of the related optimization-based and deep-learning based methods in the following subsections (Subsections 2.2.1, and 2.2.2).

2.2.1 Optimization-based Reconstruction

Structure from Motion (SfM) (Özyeşil *et al.* (2017)) and Simultaneous Localization and Mapping (SLAM) are two major directions in optimization-based 3D reconstruction using multi-view geometry. In the SfM-based methods, usually, the extracted features and the estimated camera positions of the images are used to reconstruct the 3D structure (Schonberger & Frahm (2016)). Using the obtained features and estimated motions makes the problem solvable by minimizing the total reprojection error. SLAM is another type of method which uses multi-view images to reconstruct the 3D information (Fuentes-Pacheco *et al.* (2015). SLAM methods are used to establish the position of an agent in an environment and extract the 3D information of the explored zone.

These methods are very successful in reconstruction and navigation. However, they have two main restrictions. First, they require multiple views of the object to reconstruct the 3D structure. In fact, as these approaches can not reconstruct the unseen parts of the object or scene, they usually require a large number of different views of the object to obtain a reconstruction. Another restriction of these methods is related to the appearance of the objects and the scene. One of the ways for dealing with such cases is using shape priors (Bao *et al.* (2013); Dame *et al.* (2013); Hane *et al.* (2014)). However such a prior cannot cover all possible cases. These restrictions lead to the current trend of deep-learning-based methods, which we cover in the following subsection (Subsection 2.2.2).

2.2.2 Learning-based Reconstruction

Recently, a lot of deep-learning-based methods have been proposed to reconstruct 3D information using multi-view images. To train these methods usually a large set of synthetically rendered images is used. Most of these methods only use multi-view images in the training procedure, and after that they just require a single-view of the object to reconstruct its 3D geometry. We review these methods in Subsection 2.3.2. A few of these deep-learning-based approaches use multi-view images in both the training and testing procedure. In this subsection we cover this type of deep-learning-based methods.

In Choy *et al.* (2016) and Kar *et al.* (2017), they used a recurrent neural network to reconstruct 3D geometry of the shapes from multi-view and single-view images. Choy *et al.* (2016) use Long Short-Term Memory (LSTM) (Schmidhuber & Hochreiter (1997)) to map the image(s) to the 3D geometry of the shape in a voxel-grid. This approach takes one or more images with different viewpoints and predicts the 3D shape as an occupancy grid (Figure 2.2). In Kar *et al.* (2017), they propose a learning-based system which reconstructs the 3D geometry of the object using feature projection and unprojection from 2D to 3D and vice versa. In this work, they reconstruct the 3D geometry of the multi-view images as voxel occupancy grids or per-view depth maps. Their method is also capable of reconstructing 3D geometry from a single-view image as a voxel-grid.

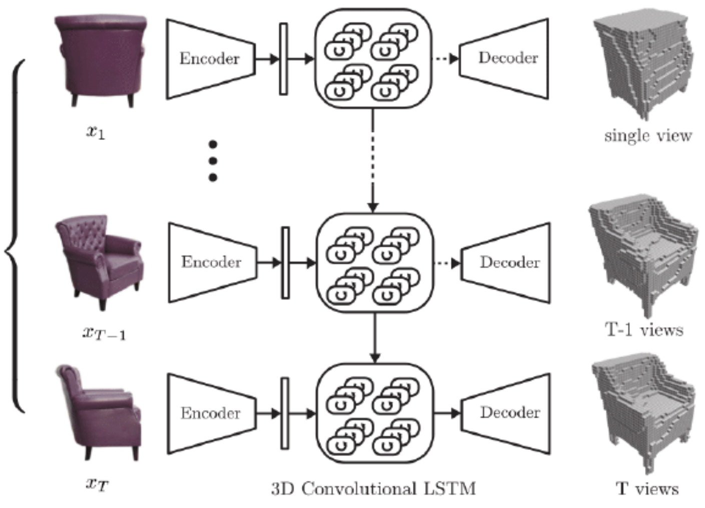

Figure 2.2: An overview of the 3DR2N2 model (Choy *et al.* (2016)) .

The main drawback of these two works is the inefficient representation of the 3D geometry of the shapes. In fact, due to the cubic space complexity of the voxel-grid, these methods cannot predict results with high-quality details. In addition to this as we mentioned in Chapter 1, increasing the size of the volume leads to decreasing the ratio between occupied and non-occupied voxels. In general, inefficient representation is the main drawback of all 3D reconstruction methods which use a voxel-grid to represent the 3D geometry of the shapes. In contrast, our proposed approach uses an efficient representation, where the complexity relies on a set of blending weights and center points. We use these parameters to construct the function f which represents the surface of the given object.

2.3 Single-view Reconstruction

While most of the research is focused on reconstructing the 3D information using point clouds and multi-view images, a more desirable solution would be reconstruction based on a single input image. This is a challenging and ill-posed problem. Due to the difficulty of this problem, there are only a few optimization-based methods which have tried to reconstruct 3D shapes using single-view images. This problem has been mainly tackled by learning-based approaches. Recently, lots of different deep-learning-based approaches have been proposed to reconstruct the 3D shape from single-view images. By training these learning models using a large set of data, these methods learn to directly map from the RBG image to the 3D shape of the object. In the rest of this section, we review some of the optimization-based and learning-based methods which have been proposed to reconstruct the 3D geometry of the shape from a single image.

2.3.1 Optimization-based Reconstruction

ShapeFromX is the type of early work which tries to reconstruct 3D information based on single-view images (Remondino & El-Hakim (2006)). Most of these methods, such as shape from shading (Horn & Brooks (1989)) and shape from texture (Aloimonos (1988)) make a strong assumption about the environment lighting condition and the property of the shapes. In Horn & Brooks (1989), to reconstruct the 3D shape of the object, it is assumed that the object's reflectance is Lambertian. However, many real-world cases do not follow this model. Aloimonos (1988) also made a strong assumption on the texture distribution to reconstruct 3D information from single-view images. In this work, they assume that natural textures have a uniform density, and based on this assumption they devise a method to recover the orientation of the surface under perspective projection.

2.3.2 Learning-based Reconstruction

There are a lot of deep-learning-based methods which have been proposed to tackle the 3D reconstruction problem from a single-view image. These deep-

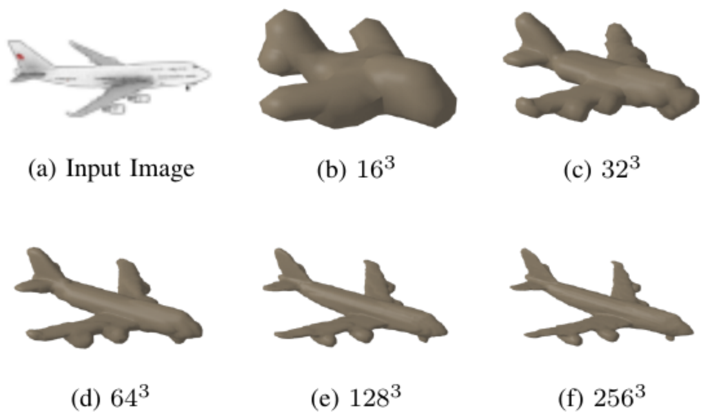

Figure 2.3: An example of hierarchical surface prediction based on a single image (Häne *et al.* (2017)).

learning approaches try to map the input RGB image to the 3D geometry of the object using a deep neural network architecture. These methods usually predict the 3D geometry of the object as an occupancy voxel-grid or a point cloud.

The method of Choy *et al.* (2016), which we reviewed in Subsection 2.2.2, is also capable of predicting the occupancy grid in a volume based on a single-view image. Girdhar *et al.* (2016) propose a novel architecture that combines a generative model for 3D data and a 2D embedding network for reconstructing the 3D shapes from single-view images. They also represent the 3D shape as a 3D voxel-grid. In Yan *et al.* (2016), they propose a novel projection loss based on the perspective transformation to reconstruct the 3D geometry of the object, which is represented as a voxel-grid. This projection loss uses multi-view information as ground-truth to reconstruct the 3D geometry of the object.

In Subsection 2.2.2 we described that due to the cubic space complexity of the uniform grid we can not reconstruct a fine-detailed 3D data in a voxel-grid. Häne *et al.* (2017) try to facilitate the high-resolution prediction of the voxel-grids using a hierarchical prediction method. In addition to predicting free and occupied voxels, they predict the boundary space, which helps in predicting only the occupied space around the boundary. By using this information they hierarchically predict voxels from a coarse-to-fine resolution in an octree. Figure 2.3 show an example of the predicted voxel-grids from coarse to fine resolution.

Fan *et al.* (2017) use the point cloud representation to avoid cubic spatial growth. They employ a deep neural network architecture to map the input RGB images to the point clouds which represent the 3D shapes of the objects. Figure

2.4 shows an example of point cloud reconstruction of the input image. In this work, the generated point cloud does not provide enough information to recover the 3D mesh of the shape.

In AtlasNet (Groueix *et al.* (2018)), in addition to mapping the input point clouds to the surfaces, a deep neural network model is proposed to map the given RGB images to the surfaces of the 3D shapes. To train this model they used a set of RGB images of the objects with their corresponding ground-truth point clouds. They reconstruct the surface of a shape by jointly learning a set of 2D embeddings and their parameterization. This method is capable of reconstructing the surface and capturing the local details. However, AtlasNet represents the surfaces of the objects as a group of patches, where these patches could be locally disconnected. In contrast, our proposed method represents the surface of the object as an implicit function. Using this representation, we can capture fine-details of the objects and reconstruct surfaces with high quality which are consistent throughout the objects.

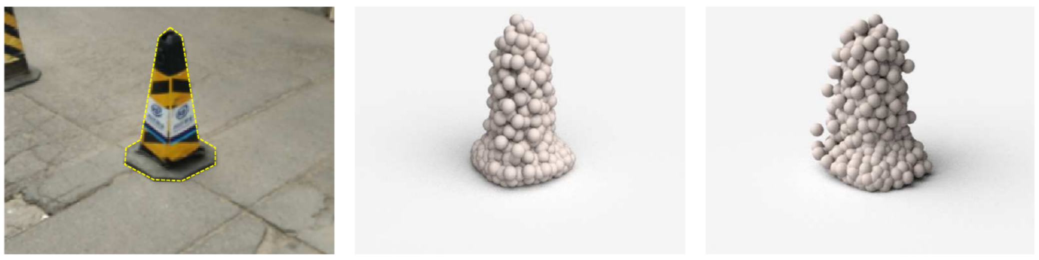
Figure 2.4: A point cloud reconstruction of a given 2D image visualized from two different views (Fan *et al.* (2017)) .

Chapter 3

Background

In Chapter 1, we mentioned that in this thesis a deep convolutional neural network is employed to map the given single monocular image to the implicit surface representation. In this chapter, we first introduce the implicit surface representation and provide the mathematical details of this representation (Section 3.1), and then in Section 3.2, we introduce the neural networks and describe the convolutional neural networks in details. At the end of this chapter, we provide information about GPU programming (Section 3.3), which we use to parallelize the implementation of our proposed loss function.

3.1 Implicit Surface Representation

We can represent a surface in $\mathbb{R}^3$ implicitly using a function f, where the value of this function on the surface is equal to a constant c. For example, we can represent a unit sphere implicitly using the function $f(\mathbf{x}) = \|\mathbf{x}\| - 1$, where $\mathbf{x} = (\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z)$. This function implicitly represents regions inside, outside and on the sphere. In other words, for every arbitrary point $\mathbf{x} \in \mathbb{R}^3$, we can determine if the point is on the surface or not. If $f(\mathbf{x}) = 0$, this point is on the surface of the sphere, if $f(\mathbf{x}) < 0$, it is inside the surface, otherwise it is outside. Note that in this thesis we consider $\|.\| = \|.\|_2$, and $\|.\|^2 = \|.\|_2^2$. In the rest of this section we provide mathematical details of implicit surface representation (Subsections 3.1.1, and 3.1.2), and in Subsections 3.1.3, and 3.1.4, we describe how we can use this representation to reconstruct the surface of an object.

3.1.1 Level-set Methods

Level-set methods are a computational framework to perform numerical computation on curves and surfaces based on the level-set models. Osher & Sethian (1988) for the first time used level-sets to compute and analyze the motion of 2D or 3D interfaces. They devised a computational framework based on a simple observation that a curve in $\mathbb{R}^2$ or a surface in $\mathbb{R}^3$ can be represented as the level-set of a function in a higher dimensional space (Osher & Fedkiw (2001)). For example, a curve γ in $\mathbb{R}^2$ can be represented as a c-th level-set of a real-valued function f in $\mathbb{R}^3$ as follow:

$$\gamma(x,y) = \{(x,y,c) \in \mathbb{R}^3, f(x,y) = c\} \ .$$

The generalization of representing n-dimensional hyper-surfaces using the level-set method is straightforward. For example, an n-dimensional hyper-surface Γ in $\mathbb{R}^n$ can be represented as c-th level-set of a real valued function f in $\mathbb{R}^{n+1}$ as follows:

$$\Gamma(x_1,x_2,...,x_n) = \{(x_1,x_2,...,x_n,c) \in \mathbb{R}^{n+1}, f(x_1,x_2,...,x_n) = c\} \ .$$

Representing a circle as a level-set of a cone is a well-known example. We can represent a circle with radius r as the r-th level-set of a cone: $f(x,y) = r$, where $f(x,y) = \sqrt{x^2+y^2}$, or the zero-th level-set of a cone: $f(x,y) = 0$, where $f(x,y) = \sqrt{x^2+y^2} - r$.

Each c-th level-set of a function $f(\mathbf{x})$ where $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$ has the following properties:

$$\begin{aligned} f(\mathbf{x}) &> c \ \ for \ \mathbf{x} \in \Omega^+ \\ f(\mathbf{x}) &= c \ \ for \ \mathbf{x} \in \partial\Omega \\ f(\mathbf{x}) &< c \ \ for \ \mathbf{x} \in \Omega^-, \end{aligned}$$

where $\partial\Omega$ is on the interface (e.g., on the surface in the 3D case). Ω^+ is outside, and Ω^- is inside the interface (Osher & Fedkiw (2001)). In general, with these level-set functions, we can specify if a point is on an interface or not, but the absolute value of the function f does not necessarily give us the shortest distance to the interface.

Another valuable property of these methods is related to the gradient of the real-valued function f. If this function is differentiable at each point, the gradient of f is either zero or perpendicular to the corresponding level-set of that point in f (Osher & Sethian (1988)).

Intuitively speaking, we can consider the function f as a mountain and a circular road around f as c-th level-set of f, and imagine two hikers A, and B on this mountain. Hiker A goes in a direction with the steepest slope which is the direction of the gradient of the function f (mountain), and hiker B is just walking on the c-th level-set of the function f (mountain) which has a constant height everywhere. These two hikers will see each other on the same point with two perpendicular directions.

3.1.2 Signed Distance Functions

Signed distance functions are a sub-set of level-set functions which share all of the properties discussed in 3.1.1. In addition to all of these properties, a signed distance function gives us the shortest distance to the interface. In other words, If the signed distance function f is differentiable, then:

$$\|\nabla f(\mathbf{x})\| = 1 \ .$$

The intuition behind this condition is as follow: in Section 3.1.1 we mentioned that the gradient of a level-set function f is perpendicular to the interface, so on the interface the gradient of f gives the normal of the interface. The normal vector direction points to the positive side of the interface. As the magnitude of the gradient is equal to unity, for each step in the normal direction or in the opposite direction with unit length, the value of the signed distance function increases or decreases by one respectively.

Based on this description we can define a signed distance function $f(\mathbf{x})$ where $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2 ... \mathbf{x}_n)$ as follows:

$$f(\mathbf{x}) = \begin{cases} d(\mathbf{x}, \partial\Omega) & \text{if } \mathbf{x} \in \ \Omega^+ \\ 0 & \text{if } \mathbf{x} \in \ \partial\Omega \\ -d(\mathbf{x}, \partial\Omega) & \text{if } \mathbf{x} \in \ \Omega^- \ . \end{cases}$$

Where $d(\mathbf{x}, \partial\Omega)$ is the distance of $\mathbf{x}$ to its closest point on the interface. In the 3D case, it is the distance to the closest point on the surface.

3.1.3 Surface Reconstruction using Radial Basis Functions

Given n sample points $\mathcal{P} = \{\mathbf{x}_1, ..., \mathbf{x}_n\} \in \mathbb{R}^3$ of a surface, we can fit an implicit function f to these points to reconstruct the surface approximately. To represent this surface implicitly as the zero-th level-set of a function f, we need to find an implicit function f which satisfies the following equation for all of the sample points $\mathbf{x}_i$:

$$f(\mathbf{x}_i) = 0, \quad i = 1, ..., n \ . \tag{3.1}$$

In addition to this condition, to avoid the trivial solution that f is zero everywhere, we can add off-surface points to the input sample points which are given non-zero distance value. With these off-surface points, the implicit function f should satisfy the following equations:

$$f(\mathbf{x}_i) = 0, \quad i = 1, ..., n$$

$$f(\mathbf{x}_i) = d_i \neq 0, \quad i = n+1, ..., N \ .$$

Where $|d_i|$ is the shortest Euclidean distance to the on-surface points, and N is the number of points, including on-surface and off-surface points. To enforce $|f(\mathbf{x}_i)|$ to be equal to the shortest Euclidean distance of the $\mathbf{x}_i$ to the on-surface points, we have to represent the approximated function f as a signed distance function. In this implicit representation, we can assign points outside of the surface to be positive and inside of the surface to be negative or vice versa.

To generate the off-surface points similar to Turk & O'brien (1999), we can move the on-surface points along their corresponding normals. These points could be inside or outside of the surface. With these off-surface and on-surface points, we can construct the signed distance function f which represents the surface implicitly. Figure 3.1 shows an example of generating the off-surface points.

Given a set of on-surface and off-surface points of an object, we can implicitly reconstruct the surface of this object using a signed distance function f. This function should be flexible and smooth enough to represent surfaces with arbitrary complexity. Radial basis functions (RBFs) are a good candidate to represent the surface implicitly as they can approximate any complex function with the sum of simple translated and scaled basis functions.

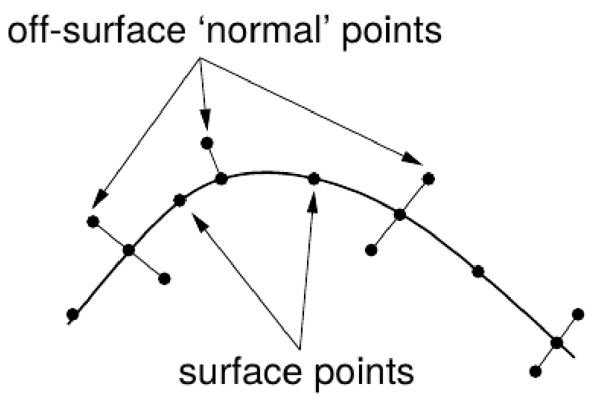

Figure 3.1: Generating off-surface points using the normals of the surface points Carr *et al.* (2001).

In the case of using RBFs to represent the surface, we can define the function f as the sum of N_c scaled and translated radial basis functions $\Psi : \mathbb{R}^3 \rightarrow \mathbb{R}$ with a polynomial of low degree:

$$f(\mathbf{x}) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{x}) + P(\mathbf{x}) \quad . \tag{3.2}$$

Where in this equation (Equation 3.2) N_c is the number of center points $\mathbf{c}_j \in \mathbb{R}^3$, and $\alpha_j \in \mathbb{R}$ are the corresponding blending weights, and $P(\mathbf{x}) = \sum_{k=1}^{4} \beta_k b_k(\mathbf{x})$, where $\beta_k \in \mathbb{R}$, and $b(\mathbf{x}) = (\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z, 1)$. Note that, $(\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z)$ are the elements of the 3D vector $\mathbf{x}$.

In general, the radial basis function Ψ could be any function which satisfies the following property :

$$\Psi(\mathbf{x}) = \Psi(\|\mathbf{x}\|) \quad .$$

There are a lot of different functions which satisfy this property. In the following we provide some of the functions which are commonly used in practice to reconstruct the surface of an object:

- Gaussian:
$$\Psi(\mathbf{c}, \mathbf{x}) = e^{-\lambda||\mathbf{c}-\mathbf{x}||_2^2} \quad .$$

- Inverse quadratic:
$$\Psi(\mathbf{c}, \mathbf{x}) = \frac{1}{1 + \epsilon||\mathbf{c} - \mathbf{x}||_2^2} \quad .$$

- Triharmonic splines:
$$\Psi(\mathbf{c},\mathbf{x}) = (||\mathbf{c}-\mathbf{x}||_2^2)^3 \ .$$

Note that in Gaussian function $\lambda > 0$, and in Inverse quadratic function $\epsilon > 0$. To fit a surface to the given point cloud using RBFs, usually we assume that there is one basis function located at each on-surface point. In this case, the number of center points N_c is equal to the number of sample points, and each center point $\mathbf{c}_j = \mathbf{x}_i$, where $i = j$. To construct the implicit function f, we have to find the parameters $\alpha = \alpha_1, ..., \alpha_{N_c}$ and $\beta = \beta_1, ..., \beta_4$ which satisfy the condition of Equation 3.1, and $P^T\alpha = 0$ (see Carr *et al.* (2001) for more details and derivation). We can find the parameters of the function f by solving the following linear system of equations:

$$\begin{pmatrix} \Psi & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} v \\ 0 \end{pmatrix} \ .$$

Where $\Psi \in \mathbb{R}^{N\times N_c}$, for each $\Psi_{i,j} = \Psi(\mathbf{c}_j, \mathbf{x}_i)$, and $P \in \mathbb{R}^{N\times 4}$ for each $P_{i,k} = b_k(\mathbf{x}_i)$. In this linear system, $v \in \mathbb{R}^N$ for the on-surface points is equal to zero and for the off-surface point $\mathbf{x}_i$ is equal to d_i, where $|d_i|$ is the Euclidean distance of the point $\mathbf{x}_i$ to the closest points on the surface. With solving this linear system of equations, we can construct the signed distance function f, which represents the surface implicitly.

3.1.4 Surface Reconstruction using Floating Radial Basis Functions

In Subsection 3.1.3, we mentioned that to fit a surface to a point cloud using RBFs, usually, we set the number of center points equal to the number of sample points. In cases where we have a large set of sample points, this leads to a huge linear system of equations, which is computationally expensive to solve. Süßmuth *et al.* (2010), try to deal with this problem by introducing the concept of floating centers. In this work, they formulate the problem of fitting a surface to the given point cloud as minimizing an energy functional. Using this formulation, they optimize the blending weights and positions of the center points of the RBF. By optimizing the center position, they obtain a better approximation of the surface even while using fewer center points.

To explain the details of this approach, assume that we have N_c center points and N_s on-surface points, where $N_c \ll N_s$. This approach seeks the parameters of the RBF by minimizing the following energy functional:

$$E(\mathcal{X}) = \sum_{i=1}^{N_s} ||f(\mathbf{x}_i)||^2 \ . \tag{3.3}$$

Where in Equation 3.3, $\mathcal{X} = \{(\alpha_j, \mathbf{c}_j, \beta_k),\ j = 1, \ldots, N_c,\ k = 1, \ldots, 4\}$ are free variables.

Solving the fitting problem just by minimizing the energy functional in Equation 3.3 yields the trivial solution that f is zero everywhere. In Subsection 3.1.3, we stated that to avoid this trivial solution we can generate new off-surface points along the normals direction. The problem with these artificial points is that they may penetrate the surface of the object, and there is no guarantee to generate the non-on-surface points.

In Süßmuth *et al.* (2010), instead of generating new artificial points, they directly incorporate the normal vectors of the on-surface points in the energy formulation. The gradient of the signed distance function f is perpendicular to the function, or in other words, it corresponds to the normal vectors of the on-surface points. With this property of the signed distance functions, they extend the energy functional (Equation 3.3) by adding a gradient term which penalizes the deviations of ∇f from the normal vectors:

$$E(\mathcal{X}) = \theta \sum_{i=1}^{N_s} ||f(\mathbf{x}_i)||^2 + \hat{\theta} \sum_{i=1}^{N_s} ||\nabla f(\mathbf{x}_i) - \mathbf{n}_i||^2 \ . \tag{3.4}$$

In this equation (3.4), $\mathbf{n}_i$ is the normal vector of the sample point $\mathbf{x}_i$, and $\hat{\theta} = (1 - \theta)$. We can use θ to balance between fitting the on-surface points and the gradient in the energy functional. As the normal vectors have unit length, and the sample points are normalized in a unit cube $[0, 1]^3$, they set $\theta = 0.95$ to compromise between the data term and the gradient term in the energy functional.

With this energy formulation, they minimize the least-square error by jointly optimizing for the blending weights and the center positions. As optimizing the set of center positions to minimize the least-square error no longer boils down to solving a linear system, they use the Levenberg-Marquardt algorithm (LMA) (Marquardt (1963)) to optimize the parameters of the RBF which minimize $E(\mathcal{X})$ in Equation 3.4. Figure 3.2 shows the effect of optimizing the center locations.

In this figure, we can see that with optimizing the center points, the final result captures more details of the object.

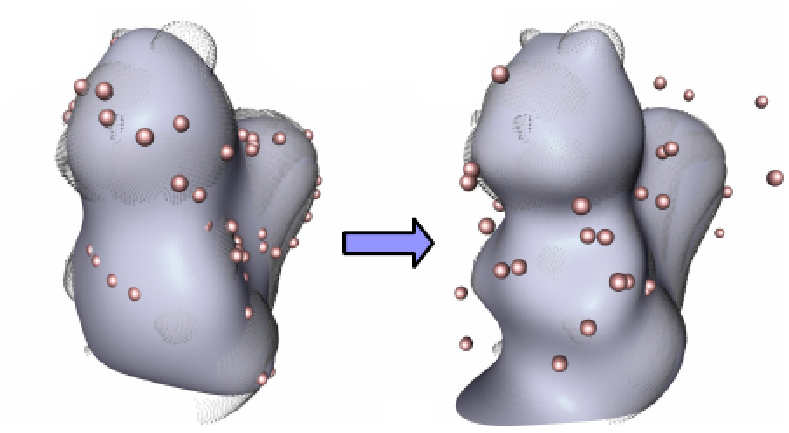

Figure 3.2: Reconstruction of an object with 90 center points, before (left) and after (right) optimizing center locations (Süßmuth *et al.* (2010)).

3.1.5 Marching Cubes Algorithm

The marching cubes algorithm is a simple approach for extracting a polygonal mesh of an implicit function proposed by Lorensen & Cline (1987). Given an implicit function f, we can obtain the underlying polygonal mesh of the function by marching over a uniform grid in a volume and evaluating the function at the corners of each cube. We can specify the resolution of the sampling grid in the volume depending on the desired quality. By increasing the resolution, marching cubes algorithm can generate meshes with higher-quality.

The marching cubes algorithm evaluates all the corners of a cube based on the given function f. If all of the corners of a cube are positive the cube is entirely outside of the surface, or if all of them are negative then the cube is entirely inside of the surface, or vice versa depending on the definition of the function. Otherwise, part of the cube is inside the surface and another part is outside of it. The surface of the function f intersects this cube's edges where one of the corners is inside, and another one is outside of the surface. For each cube, we have 2^8 possible situations where a surface can intersect an edge, however many of these are equivalent and there are only 15 unique cases which are depicted in Figure 3.3. The marching cubes algorithm reconstructs the polygonal mesh of the function f by interpolating the surface intersection along the edges.

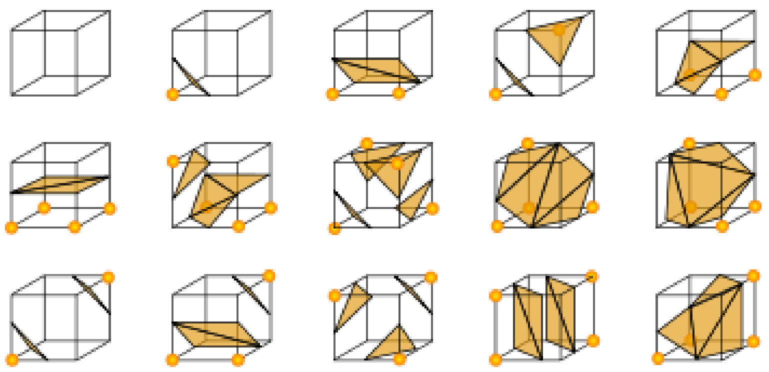

Figure 3.3: 15 unique patterns of cube configurations. (Lorensen & Cline (1987))

3.2 Neural Networks

A neural network (NN) is an information processing framework, which is composed of a group of neurons. These neurons can receive input data and produce an output using an activation function. The network architecture is formed by arranging these neurons in layers. In a simple architecture, the output of each layer is connected to the input of the next layer with a set of weights. In other words, the neuron forms a directed weighted graph, which processes the information through the layers to produce the output. We can train a neural network model using a set of training data to solve a specific task such as classification, recognition, etc. The training process consists of providing the input data and the target output to the network. For every input, the network outputs a prediction. Using a loss function we can measure the error of the network's prediction, and update the weights to minimize the error. We provide more details about the training procedure in the next subsections.

In Chapter 1, we mentioned that in our proposed method, we use a convolutional neural network (CNN) as an encoder to encode the input image to the 3D geometry of the shape. In the rest of this section, we first explain how we can encode and decode input data using a neural network model (Subsection 3.2.1), and then we describe the CNN architecture and its training procedure in details (Subsection 3.2.2).

3.2.1 Auto-encoders

An Auto-encoder is a neural network, which encode the input data to the latent space representation and try to reconstruct the input data using the encoded representation of the data. Intuitively speaking, we can see the auto-encoders as a learning-based compression method, which encodes the input data such as an image to a small vector of real numbers.

This type of neural networks is also similar in spirit to principal component analysis (PCA). It creates a representation of the input data in an unsupervised fashion where just the information essential to the task is preserved, and the rest of the information is removed.

An auto-encoder usually contains two main components, an encoder part which encodes the input data to a latent representation, and a decoder part which receives the output of the encoder as input and tries to decode it to obtain the input data. Figure 3.4 shows a simple architecture of an auto-encoder. We explain each part of the auto-encoder with an example in the following:

- **Encoder:** To encode a vector $\mathbf{x} \in \mathbb{R}^n$, the encoder takes $\mathbf{x}$ as input and encodes it to the latent space representation $\mathbf{z} \in \mathbb{R}^m$ where $m \ll n$.

- **Decoder:** This part of the network takes the output of the encoder which is $\mathbf{z} \in \mathbb{R}^m$ as input, and tries to reconstruct the input data to the network approximately as $\mathbf{x}' \in \mathbb{R}^n$.

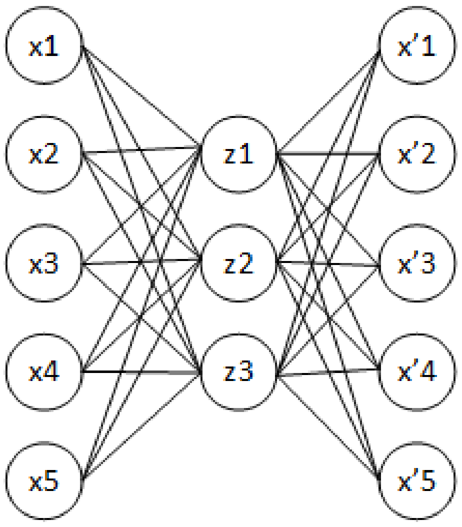

Figure 3.4: A simple auto-encoder network (Wang (2016)).

3.2.2 Deep Convolutional Neural Networks

Convolutional neural networks (CNNs) are a special type of neural networks for processing data with grid-like topology. These data could be a time-series data in a 1D grid, or a RGB image in a 2D grid. The main difference between the CNN architecture and a typical NN architecture is using convolutional operations instead of the general matrix multiplication at least in one of the layers. Using a stack of convolutional operations followed by activation functions and pooling layers makes this type of NN a unique architecture to solve image recognition, classification, segmentation and many other vision and graphics tasks.

Figure 3.5 shows the architecture of a simple convolutional neural network to solve an image classification task. As we can see in Figure 3.5, usually a CNN architecture has the following layers:

- Convolutional Layer
- Non-linearity Layer
- Pooling Layer

Each of these layers has a key role in the CNN architecture. In the rest of this section, we explain how these layers work and how a deep stack of these layers makes a CNN a powerful architecture to work with the grid-like data.

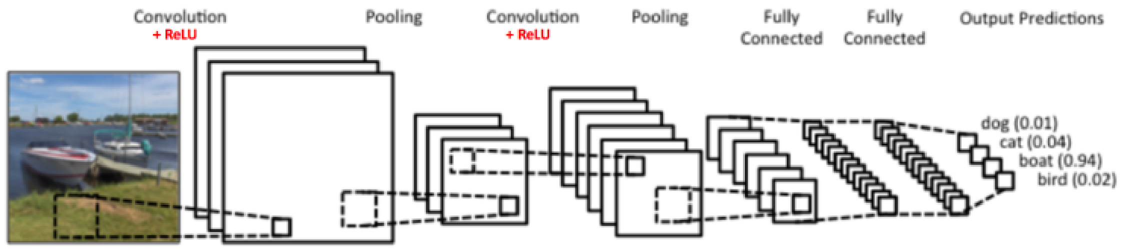

Figure 3.5: A simple convolutional neural network (Karn (2016)).

3.2.2.1 Convolutional Layer

A convolutional layer is a layer which performs a convolution operation using one or more filters on an input tensor. Convolving each of these filters with the input tensor extracts a specific feature map of the input, and these extracted feature maps will be the input tensor of the following layer in the network. This input tensor could be the input RGB image with size $w \times h \times 3$ or the output of a hidden layer with size $w \times h \times d$, where w and h refer to the width and height of the tensor respectively, and d refers to its depth. The size of the filters which is used in the convolutional layer is smaller than the input tensor. We provide more details about this layer in the following example:

- **Example** : Consider that we have input RGB images with size $32 \times 32 \times 3$, and two filters with size $5 \times 5 \times 3$ which have different values. To perform convolution on the input image, we have to slide the filters over the image, and in each step, we multiply the values in the filter with the corresponding pixel values of the image and at the end, we sum up the result. In this example, as we have filters of size $5 \times 5 \times 3$, in each step we have to perform 75 multiplications in total. Every step of sliding the filter over the images produces a number. In this case after sliding a filter with size $5 \times 5 \times 3$ over an image with size $32 \times 32 \times 3$ we will have 28×28 numbers which forms a feature map. After sliding two filters on the input image we will have two extracted feature maps. Figure 3.6 shows a visualization of sliding one filter of size 5×5 over an input data.

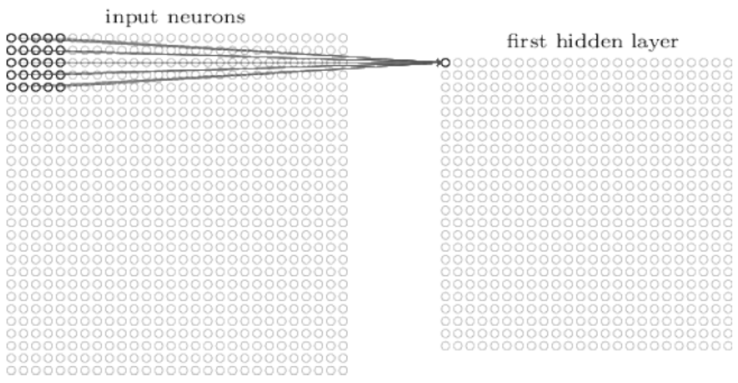

Figure 3.6: Visualization of a 5×5 filter convolving on an input data (Nielsen (2015)).

3.2.2.2 Non-linearity Layer

In the CNN architecture, usually, each convolutional layer is followed by a non-linearity layer. This layer applies an element-wise activation function on the extracted feature map. The Rectified linear unit (ReLU) (Nair & Hinton (2010)) which is depicted in Figure 3.7 is one of the recommended activation functions that is used in most of the feed-forward neural network architectures. This operation is defined by the following function:

$$f(x) = max(0, x) \ .$$

With applying this activation on the extracted feature maps, all of the negative values will be replaced by zero. This operation yields a non-linear transformation on the output of the convolutional layer which performs a linear transformation on the input data. We can also use the tanh function or sigmoid function as the activation function, however, ReLU performs better in most situations (Goodfellow *et al.* (2016)).

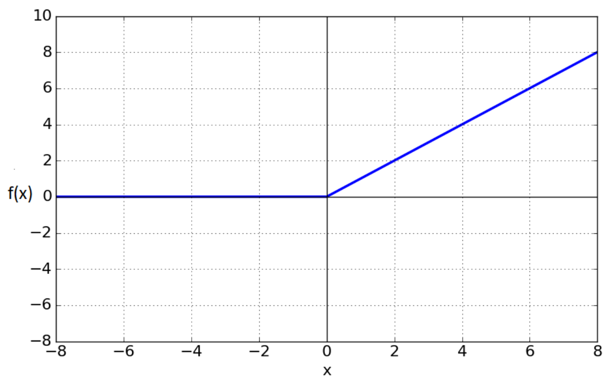

Figure 3.7: Rectified linear activation function.

3.2.2.3 Pooling Layer

It is common to use pooling layers after each non-linearity layer. This layer down-samples the extracted features of the previous layer by retaining the most important information of each feature map. In this layer, we can use different pooling operators to down-sample the input feature maps, e.g., Max pooling, Average pooling, Sum pooling, etc.

To apply the pooling operator, we define a spatial neighborhood window and slide it over the width and height of the outputs of the previous layer. In case of using max pooling with window size 2×2, every max operation takes a max over 4 numbers in the 2×2 region. Figure 3.8 show an example of this layer with the max pooling operator.

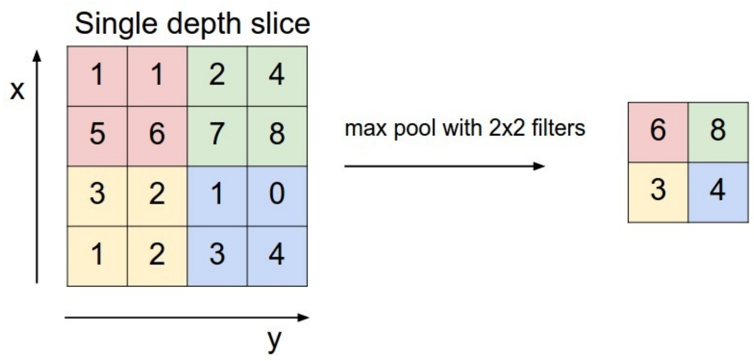

Figure 3.8: Max pooling operation over a slice of an input (Karpathy (2016)).

The pooling layer reduces the number of parameters and computation in the network by down-sampling the extracted features of the previous layer. Also, this layer helps the network to extract features of the input data which are invariant to small transformations, translations, and distortions.

In the CNN architecture, we usually have a stack of convolutional layers followed by a non-linearity layer and pooling layer. By using pooling layers at different levels of the network we can extract an almost scale-invariant representation of the input data.

3.2.2.4 Feature Learning

In a CNN architecture, consider a convolutional block as a component which contains at least a convolutional layer, non-linearity layer(s), and pooling layer(s). In general, in the network architecture, we have a stack of these blocks on top of each other. With more blocks, the network learns to extract more complicated and informative features of the input data. For example, in an image classification task, in the first convolutional block, filters learn to detect simple features such as edges and blobs. In the next convolutional blocks, as we down-sample the extracted features, filters learn to detect higher-level features of the input data. Figure 3.9 shows an example where filters learn to detect a different level of features in different layers of the network. Note that Figure 3.9 is just an example from Convolutional Deep Belief Network (Lee *et al.* (2009)) to demonstrate the idea, and in fact, convolutional filters may detect features which have no meaning for humans.

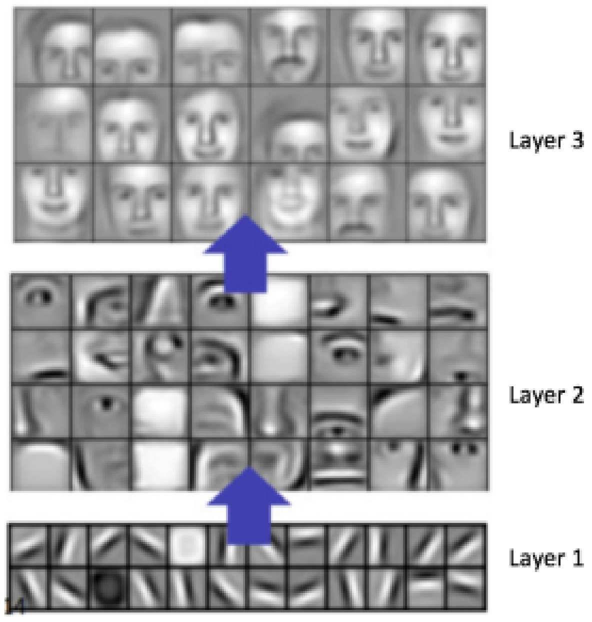

Figure 3.9: Learning hierarchical representations of the data using the Convolutional Deep Belief Network (Lee *et al.* (2009)).

3.2.3 Training Procedure

Consider that we have a stack of convolutional blocks followed by one or more fully-connected layers. For the sake of explanation, assume that we want to solve an image classification task. To train a classification model, we need a set of training pairs of images and their corresponding class scores. Figure 3.5 shows a four-class classification task, in this task a binary vector is assigned to each input data. For example, in Figure 3.5 for the four-class classification task, the ground-truth score of the boat image would be $(0, 0, 1, 0)$. In general, with a fixed architecture and pairs of training data to train a CNN model, we can do the following steps:

1. **Initialization:** Initialize all of the filters' weights and parameters of the network randomly.

2. **Forward pass:** Feed the training data to the network. These data go through the convolutional blocks to extract the features of the input data. In the case of a classification task, the network outputs the probabilities for each class. For example, if we feed the boat image for the first time, as we

have initialized the parameters randomly, the output probability could be $(0.3, 0.1, 0.2, 0.4)$.

3. **Error calculation:** For each training input, the network outputs a prediction. We calculate the difference between this prediction and the ground-truth label.

4. **Backward pass:** Using the backpropagation algorithm, which is commonly used to train deep neural network, we can calculate the gradient of the loss function with respect to all of the network's parameters. With the calculated gradient, we can update the parameters of the network using an optimization algorithm to minimize the output error. For example, with updating the weights after one pass, the output probabilities for the boat image might be $(0.05, 0.1, 0.65, 0.2)$, which is closer to the ground-truth. By updating the parameters based on the output error, the network learns to classify images such that it minimizes the prediction error.

3.2.4 Transfer Learning

In Subsection 3.2.3, we mentioned that to start the training procedure we have to initialize all of the filters' weights and parameters randomly, however this is not necessarily an optimal way to initialize the parameters of the network. In fact, randomly initializing the parameters is more like reinventing the wheel as we usually need more general filters in the very first levels of the network such as edge and blob detectors.

A more efficient and common solution is to use the parameters of a pre-trained model as an initialization for the parameters of the network. To initialize the parameters in this way, we can train a network on a very large dataset such as ImageNet Deng *et al.* (2009), which contains 1.2 million images of 1000 different categories. After training the model with a very large dataset, we can use the trained parameters as the initialization or a feature extractor for a specific task. In other words, we transfer the knowledge which is learned by a network to another network to solve a different task. In general, we have two major scenarios for Transfer learning:

1. **Feature extractor:** In this scenario, we first train the network with a very large dataset such as ImageNet to solve the classification problem. After that, we remove the last or all of the fully-connected layers after the

convolutional blocks and treat the rest of the network as a fixed feature extractor for another dataset or even another task. The extracted features can be used to solve another classification task with a different dataset or a different task such as an image segmentation or object detection problem. We can use the extracted features as an input of a fully-connected layer which is initialized randomly, and train this FC layer using the new dataset.

2. **Fine-tuning:** Another scenario is to re-train the pre-trained model for the new task, by fine-tuning the parameters using the backpropagation algorithm. We can update the parameters in all of the levels of the network or freeze some of the very first layers and only fine-tune the parameters of the higher-level part of the network. Usually, we can fine-tune just the higher-level part of the network as the first levels of the network extract more generic features, and the later layers extract more specific features based on the training data.

3.3 GPU Programming

CUDA is a parallel computing platform and programming model which was invented by NVIDIA. This platform lets programmers develop a high-performance algorithm by running thousands of parallel threads on a graphics processing unit (GPU) (NVIDIA (2012)). Nowadays, almost all the deep-learning libraries and frameworks use this platform to accelerate the computation time of different learning tasks.

3.3.1 CUDA Programming Model

The CUDA platform provides a heterogeneous model to use both the CPU and GPU. In CUDA programming, we refer to the CPU and its memory as host, and to the GPU and its memory as device. We can prepare data on the host and perform the computation on the device by executing many threads in parallel. In general, to develop a program with the CUDA programming model, we can follow these steps (Cheng *et al.* (2014)):

1. Define and allocate the host's and device's variables.
2. Copy data from the host to the device.
3. Execute device's functions (kernels) to do the computation.

4. Copy the results from the device to the host.

In CUDA, we can parallelize the device's functions (kernel) by executing them with many threads in parallel. CUDA organizes these threads in a group which is called a warp. Each warp contains 32 threads, a set of warps is considered as a thread block and a set of these blocks is considered as a grid. Different thread blocks can be executed concurrently using many parallel processors which are grouped into Streaming Multiprocessors (SM). The number of SMs on a GPU depends on the GPU model. However, the number of threads and blocks should be specified by the programmer during the kernel launch. Figure 3.10 shows an example of 2D thread blocks in a grid.

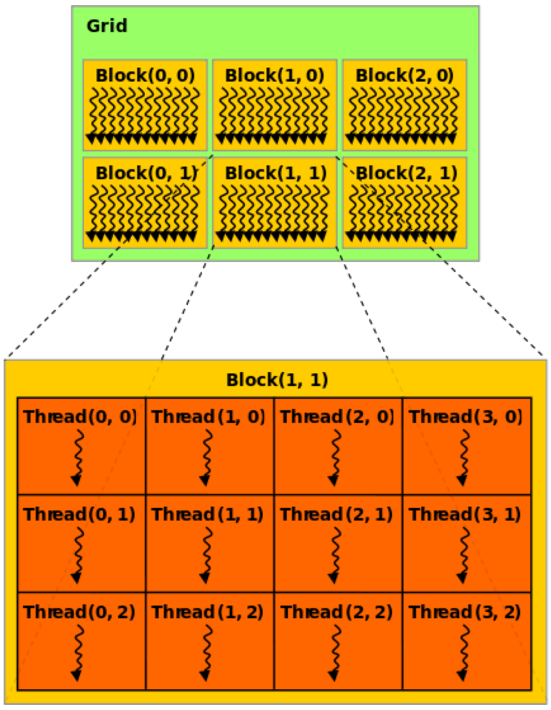

Figure 3.10: An example of a grid of thread blocks. (NVIDIA (2012)) .

3.3.2 Memory Hierarchy

GPU architectures, in general, have three main memory types. Global memory, shared memory and registers. In the CUDA platform, threads can have access to the data from different memory spaces. In this platform, each thread has its local memory which can be one or more registers. The access time to the register is faster than other types of memories. Shared memory is accessible by thread blocks, this memory is visible to all of the threads in a block, and it is slightly slower than registers. Threads in a block can cooperate with each other by sharing data through shared memory. Global memory is the slowest memory which is visible to all of the threads even in different blocks. Accessing this memory takes hundreds of cycles. Figure 3.11 show an overview of the GPU memory hierarchy.

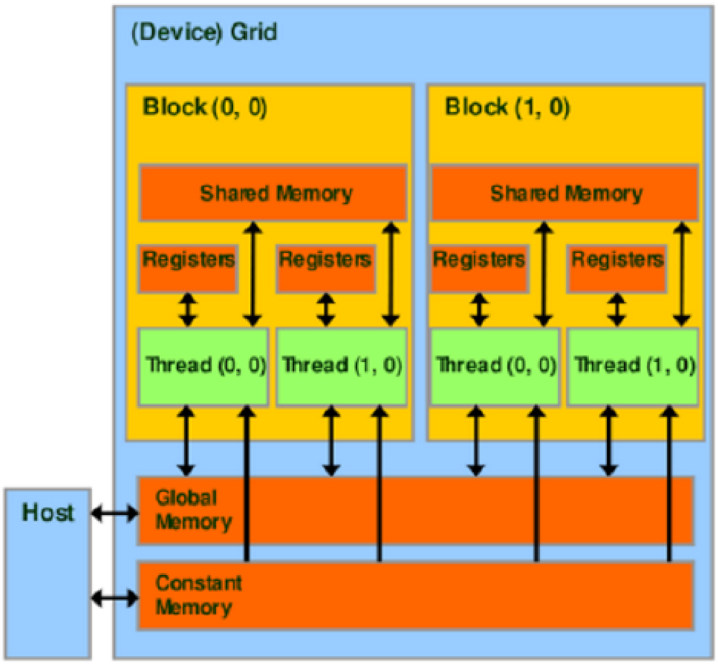

Figure 3.11: An abstract view of the GPU memory hierarchy (NVIDIA (2012)).

3.3.3 Parallel Reduction

Parallel reduction is a tree-based approach to efficiently compute the aggregations using cooperation among threads in parallel. This method provides an efficient way to apply an associative and commutative operation on the elements of an array. For example, consider an array with a large number of elements. One way of applying an associative and commutative operation on the elements of this array is to sequentially compute the operation in $O(n)$, where n is the size of the array. However, we can also compute this operation in a parallel way by executing many threads. Theoretically, we can compute such operations using the parallel reduction approach in $O(\log(n))$. In general, to do an operation on a large array, we can perform parallel reductions in the following way (Cheng *et al.* (2014)) :

1. Partition the array into some chunks, and consider a group of threads per each chunk.
2. Compute the operation for each chunk in parallel.
3. Compute the final results based on the computed partial results of each thread.

To implement an algorithm in CUDA with the parallel reduction approach, Harris (2007) provide efficient strategies. However, with the new features which are in the Kepler GPU architecture, we can make reductions on the elements of the arrays or matrices even faster. In this architecture using shuffle instructions, the threads can directly read a register from another thread in the same warp. With this property, we don't need to use shared memory to exchange the data between threads within the same warp.

Using shuffle instructions, we can make parallel reductions over the warp. To do this, we can simply use one of the shuffle instructions (based on the operation) iteratively to calculate the results for each warp. For example, Figure 3.12 shows an example of a parallel reduction in a warp using one of the shuffle instructions. Using this solution, we can also apply reductions over the entire thread block. To do this after applying reduction over warps, we can first write the partial result of each warp into an array in the shared memory. After synchronizing the threads within the block, atomic operations on the shared memory can be used to calculate the aggregation for the block.

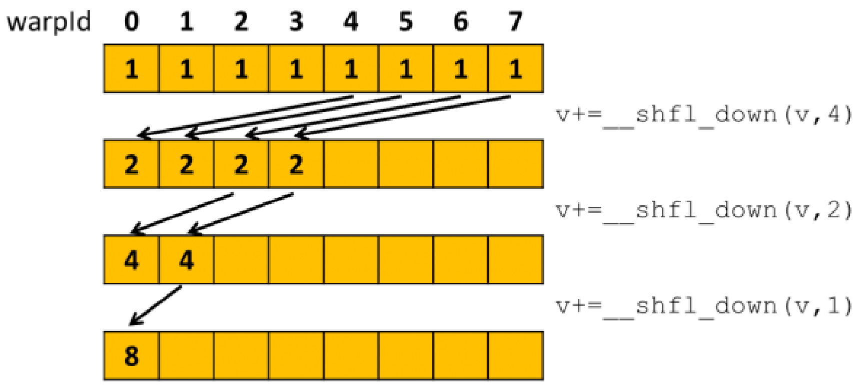

Figure 3.12: Parallel reduction using shuffle instructions to compute the summation. (Luitjens (2014)) .

Chapter 4

Method

In this section, we describe the contributions of this thesis in detail. We propose a learning-based approach to reconstruct the 3D geometry of an object based on a single monocular image. Given a single 2D image, this model outputs a detailed signed distance function (SDF) representation that describes the geometry of the object based on its zero-th level-set. We represent SDFs efficiently based on a linear combination of floating radial basis functions, where both the supporting points as well as the blending weights are free variables. We employ deep convolutional encoders to encode the monocular input image to the latent space that spans the set of radial basis function parameters. In the decoder part of our model, we use a radial basis functions (RBF) decoder, which decodes the extracted parameters to the detailed SDF representation of the object's surface. Since no dataset provides ground-truth distance fields, we propose a weakly-supervised training approach that uses sampled points on the ground-truth surfaces as weakly-supervised labels to reconstruct the surface of an object. In this training approach using the sampled points, we train a regressor to optimize for the best parameters of the radial basis functions. To develop this training approach, we propose a new differentiable loss function, which we implement efficiently to perform the forward and backward passes on the GPU. Figure 4.1 shows an overview of our proposed method.

4.1 Surface Representation

We implicitly represent the 3D surface of an object as a signed distance function f. The function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ is constructed in a way that points $\mathbf{x} \in \mathbb{R}^3$ outside the surface are assigned a positive distance, points inside the object are mapped

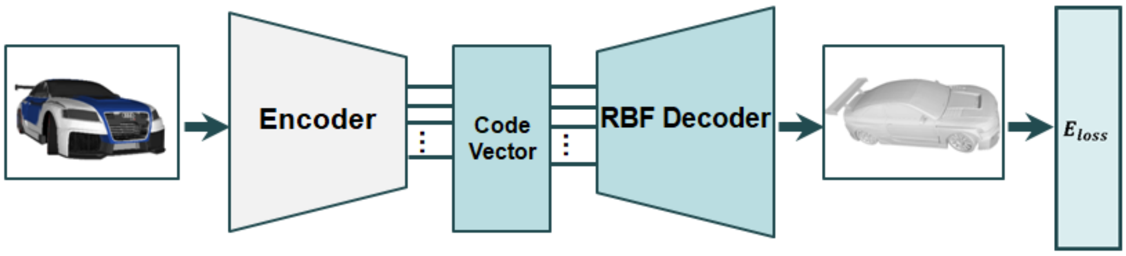

Figure 4.1: An overview of the proposed method.

to a negative distance value, and the zero-th level-set of this function represents the surface.

To parameterize the signed distance function f, we employ a mixture of radial basis functions with a 3-variate linear polynomial:

$$f(\mathbf{x}) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{x}) + P(\mathbf{x}) \ . \tag{4.1}$$

Here, the N_c center points $\mathbf{c}_j \in \mathbb{R}^3$ define the locations of the basis functions, $\alpha_j \in \mathbb{R}$ are the corresponding blending weights, and $P(\mathbf{x}) = \sum_{k=1}^{4} \beta_k b_k(\mathbf{x})$, where $\beta_k \in \mathbb{R}$, and $b(\mathbf{x}) = (\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z, 1)$. Note that, $(\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z)$ are the elements of the vector $\mathbf{x}$. We use the following radial kernels:

$$\Psi(\mathbf{c}_j, \mathbf{x}) = e^{-\lambda ||\mathbf{c}_j - \mathbf{x}||_2^2} \ . \tag{4.2}$$

Where $\lambda > 0$, and controls the width of the Gaussian function. Normally, the number of center points is small $N_c \approx 6000$. We use the floating radial basis functions (Süßmuth *et al.* (2010)), where the center points $\mathbf{c}_i$ as well as the blending weights α_i and the linear polynomial's parameters β_k are considered to be free variables. This enables the centers to 'move', which leads to higher-quality surface approximation results, especially if the center points are initialized randomly, like in our case.

Thus the set of free variables is:

$$\mathcal{X} = \{(\alpha_j, \mathbf{c}_j, \beta_k), \ j = 1, \dots, N_c, \ k = 1, \dots, 4\} \ . \tag{4.3}$$

Where in the training process of our method, we set α_j and β_k for all of the j and k to zero, and initialize $\mathbf{c}_j$ for each center point uniformly at random in a unit cube $[0, 1]^3$. We provide more detail about the initialization in Section 5.2

4.2 Deep Convolutional Encoder

To represent the 3D geometry of the given 2D image as a radial basis function f, we have to estimate the parameters of this function based on the input 2D image. We do this by employing a deep convolutional encoder to transform the input 2D image to the parameters of the function which represents the surface of the object implicitly. Our goal is to estimate the set of free variables $\mathcal{X}$ (Equation 4.3) of the radial basis function f (Equation 4.1). More specifically, using an encoder, we extract the features of the input 2D image and map them to the parameters of the function f. During the training procedure, the network learns to use the specific characteristics of different classes of 2D input images in order to distribute the center points more optimally. For example, to reconstruct a chair, it requires more center points close to the legs of the chair in comparison to the other parts.

4.2.1 Encoder Architecture

To encode the input 2D image to the parameters of the radial basis function we use a pre-trained VGG16 model (Simonyan & Zisserman (2014)) that was trained on the ImageNet dataset (Deng *et al.* (2009)). This model contains a stack of convolutional blocks followed by three fully-connected layers. We remove the fully-connected layers, and only use the convolutional blocks to extract the features of the 2D input images. All of these convolutional blocks contain two or three convolutional layers followed by rectification non-linearities (Nair & Hinton (2010)), and each convolutional block is followed by a max-pooling layer, and the output of VGG16 is connected the two branches of the fully-connected layers (see Figure 4.2). We provide more details about each part of the encoder in the following:

- **Input**: The encoder takes a $137 \times 137 \times 3$ image as input. Both the width and height of this image are 137 and it has three color channels RGB.
- **Convolutional Block**: In each convolutional block, there are two or three convolutional layers, all of these convolutional layers are followed by the rectification non-linearity layer (ReLU). The output of each convolutional block is the output of the last ReLU layer of that block.

  - Convolutional Layers: Each convolutional layer applies a 3×3 filter to the input. This input could be an input images (e.g volume of size $137 \times 137 \times 3$) or the output of the previous convolutional layer (e.g. volume of size $16 \times 16 \times 256$).
  - Rectification Non-linearity (ReLU) Layer: This layer is an activation function $f(x) = max(0, x)$ which will be applied to each element of the output of the convolutional layers.

- **Max-pooling Layers**: Each convolutional block is followed by a Max-pooling layer. All max-pooling layers use a 2×2 pixel window with a stride length of 2.

- **Fully-connected (FC) Layers** : The output of the VGG16 model is connected to the two branches of FC layers. One of these branches which contains two connected FC layers is used to transfer the extracted features to all of the α_j and β_k variables. Another branch which also has two connected FC layers is used to transfer the extracted features to estimate the center points ($\mathbf{c}_j$). Note that, as these two groups of parameters could be in different spaces, we use separate branches of FC layers to estimate them.

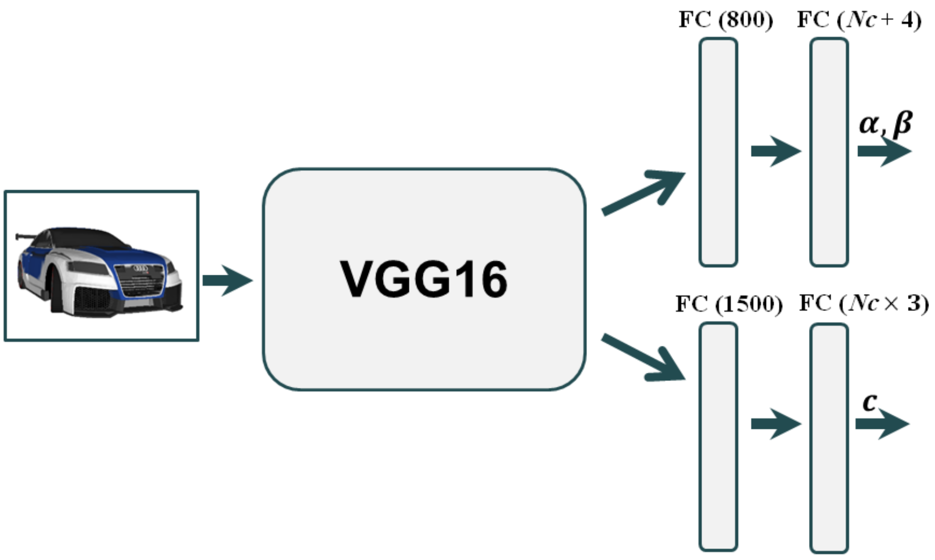

Figure 4.2: An overview of the encoder architecture. Note that, in this architecture, we only use convolutional blocks of the VGG16 model.

The output of the deep convolutional encoder spans a set of RBF parameters $\mathcal{X}$ as defined in Equation 4.3. Using these parameters we can reconstruct a function f which represent the surface of the object implicitly.

4.3 Radial Basis Function Decoder

As we already mentioned using the deep convolutional encoder, we transfer the extracted features of the input 2D image to the parameters of the radial basis function f. To reconstruct the function f we use a decoder which decodes the estimated parameters to a RBF function f. Equation 4.1 and 4.2 show the function f and the radial kernel of this function which we use in our proposed method. Since Equation 4.1 and 4.2 are differentiable with respect to the parameters, the parameters can be learned using the backpropagation algorithm.

4.4 Weakly-supervised Loss

To train our proposed model, we employ a weakly-supervised loss (Equation 4.4). Using this loss function, we measure how well the regressed signed distance field, which is parameterized by a set of variables $\mathcal{X}$, represents the surface of the object.

$$E_{\text{total}}(\mathcal{X}) = E_{\text{point}}(\mathcal{X}) + E_{\text{normal}}(\mathcal{X}) + E_{\text{empty}}(\mathcal{X}) \quad . \tag{4.4}$$

In this loss function, we employ the first two-terms ($E_{\text{point}}(\mathcal{X})$ and $E_{\text{normal}}(\mathcal{X})$) from the floating RBFs work (Süßmuth *et al.* (2010)) to apply the zero-crossing constraint and the normal constraint. These two terms only specify where the object should be in the 3D space, but they do not provide any information about the empty part of the 3D space. Without having information about empty part of the space, the model can reconstruct a surface of a shape with spurious surfaces around it. With the third term in the loss function, we try to provide information about the empty part of the space to avoid having spurious surfaces in the reconstructions. In the following, we explain all of these terms in details.

4.4.1 Zero-crossing Constraint

To apply the zero-crossing constraint, we use the following term:

$$E_{\text{point}}(\mathcal{X}) = w_{point} \sum_{i=1}^{N_s} ||f(\mathbf{s}_i)||^2 \quad . \tag{4.5}$$

This term (Equation 4.5) enforces the zero-th level-set of the function f to be on the surface of the 3D object. We do this by sampling points from the 3D surface and penalizing the deviation of the function value from zero at these points.

In Equation 4.5, w_{point} is the trade-off factor that controls the effect of this term on the whole loss function, and N_s is the number of sample points.

4.4.2 Normal Constraint

In Subsection 3.1.3 we explained that using only the zero-crossing constraint yields the trivial solution where f is zero everywhere. In this subsection (Subsection 3.1.3) we described that one of the solutions to avoid this trivial result could be adding more constraints by generating off-surface points along the normal direction. However, these artificial points may penetrate the surface of the object and lead to a poor reconstruction. Instead of generating new artificial points, we employ the normal constraint term from Süßmuth *et al.* (2010). With this term, we enforce the gradients of the function f at the sample points to be close to the corresponding normal vectors:

$$E_{\text{normal}}(\mathcal{X}) = w_{normal} \sum_{i=1}^{N_s} ||\nabla f(\mathbf{s}_i) - \mathbf{n}_i||^2 \ . \tag{4.6}$$

In Equation 4.6, w_{normal} is the trade-off factor that controls the effect of this term on the whole loss function, N_s is the number of sample points, and $\mathbf{n}_i$ is the corresponding normal vector of the sample point $\mathbf{s_i}$.

4.4.3 Empty-space Constraint

In Section 4.4 we mentioned that with the first two terms, we specify where the surface should be in the 3D space, but these terms do not provide any information about the empty part of the space. In other words, the zero-crossing and normal constraints do not penalize for occupying the empty space. This could lead the model to construct a function f which represents some spurious surfaces that are not part of the ground-truth surface.

To deal with this problem, we add an empty-space constraint to our loss function. In this term (Equation 4.7), $\forall(\mathbf{r}, d) \in \mathcal{R}$, $\mathbf{r}$ is a random point in a unit cube $[0, 1]^3$, and $|d|$ is the closest Euclidean distance of that random point to the sample points, while the sign of d is positive if $\mathbf{r}$ is outside of the surface, and negative if it is inside the surface. From this point, we call d as signed Euclidean distance.

$$E_{\text{empty}}(\mathcal{X}) = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} ||f(\mathbf{r}) - d||^2 \ . \tag{4.7}$$

To apply the empty-space constraint, we generate $|\mathcal{R}|$ random points in the unit cube, and enforce the function f at these points to be close to the signed Euclidean distance d. More specifically, with this term, we try to specify in which part of 3D space the estimated function f should not be zero. Figure 4.3 shows the effect of using this constraint in the loss function.

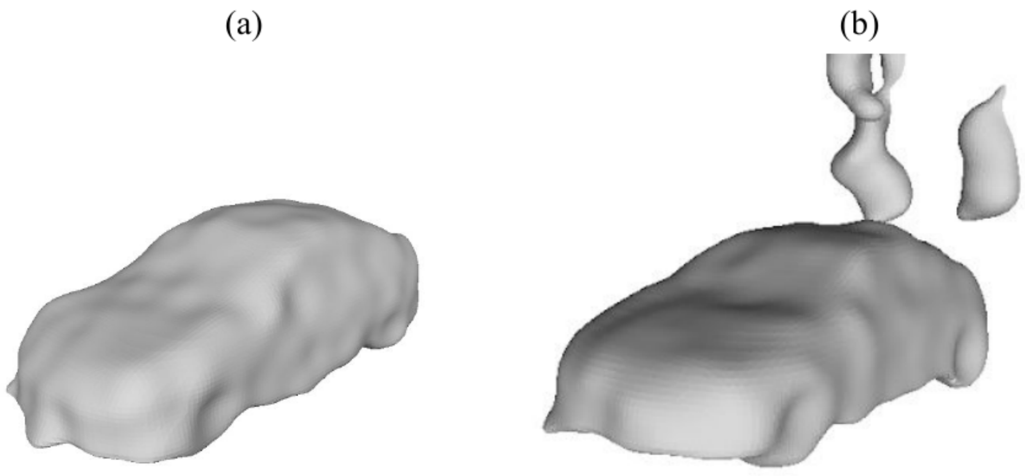

Figure 4.3: An example of reconstructing the surface of a car. (a) With using the empty-space term, (b) Without using empty-space term

In Equation 4.7, we use w_{empty} as a trade-off factor to control the effect of this term with respect to the other terms.

In this term (Equation 4.7) the generated random point $\mathbf{r}$ could be inside or outside of the surface. In our SDF representation, we want to assign a positive distance value to points outside of the surface, and a negative distance value to points inside the surface. To do this, we specify the sign of the distance d by computing the distance to the tangent plane of the closest sample point in terms of Euclidean distance. We can compute the distance $\mathbf{r}$ to the tangent plane of point $\mathbf{s}$ by using $h(\mathbf{r}) = (\mathbf{r} - \mathbf{s})^T \cdot \mathbf{n}$, where $\mathbf{s}$ is the closest sample point to the random point $\mathbf{r}$, and $\mathbf{n}$ is the corresponding normal vector of that sample point

(Figure 4.4). Using the sign of the function $h(\mathbf{r})$ we can determine the sign of the distance d.

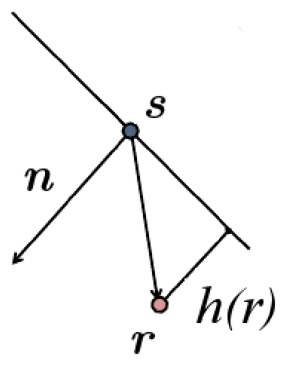

Figure 4.4: An example of computing the distance to the tangent plane.

Chapter 5

Training

In this chapter, we first describe the details of the training data and the dataset which we use to prepare this training set (Section 5.1). We show an example of the training pair of the data which consists of 2D image with its corresponding 3D information. In addition to this, we also provide details of rendering the 2D image from 3D CAD data, and sampling the points and their corresponding normals on the 3D data. In Section 5.2, we explain each step of the training procedure in details. We describe how we use pairs of 2D images and corresponding 3D information (points and the normal vectors) to train a model which predicts the 3D geometry of the 2D input as an implicit function.

5.1 Training Data

To train our model, we need pairs of 2D images with their corresponding 3D information. In our case, as we want to predict the 3D geometry of the object from 2D, we use sampled points and their corresponding normal vectors of the ground-truth mesh as the 3D information. To produce these pairs of data, we use a dataset of 3D CAD models.

5.1.1 Dataset

The ShapeNet dataset (Chang *et al.* (2015)) is used to prepare the training and testing data for our model. This dataset is a collection of 3D CAD models which is organized based on the WordNet taxonomy (Miller (1995)). ShapeNet is made of two different subsets, ShapeNetCore which contains 3D models of 55 different categories, and ShapeNetSem which contains 3D models with more dense annotation. We use a subset of ShapeNetCore with more than 15,000 models of 4 different categories. To have comparable results with other models, for splitting data into train and test set, we follow Choy *et al.* (2016). We use 4/5 of the shapes for training the model and the remaining shapes for testing the model. Figure 5.1 shows some examples of the 3D shapes from ShapeNetCore.

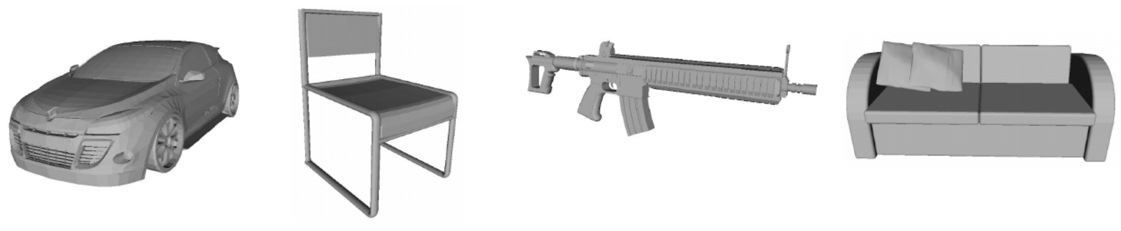

Figure 5.1: 3D shapes from four different categories of ShapeNetCore (Chang *et al.* (2015)).

5.1.2 Training Pairs

To train our model, we need 2D images as the input data and their corresponding 3D geometry information. For the 2D objects, we use the rendered 2D images which were provided by Choy *et al.* (2016). They rendered 24 RGB images per object in different views. Figure 5.2 shows an example of a pair of training data.

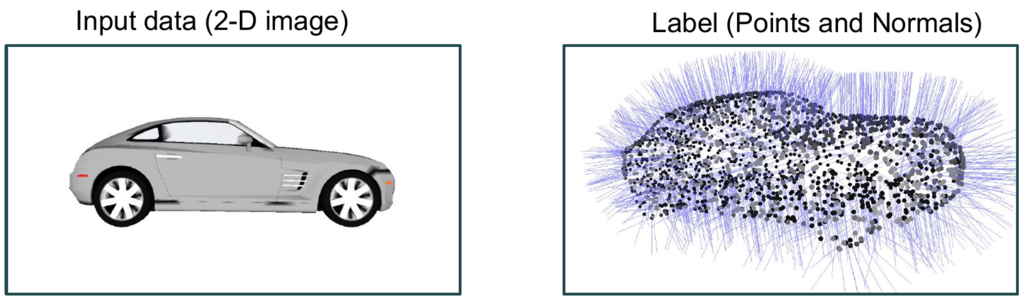

Figure 5.2: RGB image of the object (left), sample points and the corresponding normal vectors of the ground-truth mesh (right).

To provide the 3D information, we have to use 3D CAD models of the ShapeNetCore dataset. This dataset contains lots of meshes with artifacts, such as overlapping triangles, flipped normals, and non-manifold structures. To compute the sample points and their correct normal information of the ground-truth meshes, we follow Wang *et al.* (2017). They use a ray shooting algorithm to densely sample points with their corresponding normal vectors. To do this, they suggest to use 14 virtual cameras, and uniformly shoot 16k parallel rays from each direction. We can calculate the intersection of these rays with the surface, and the orientation of the normal vectors for each point. Figure 5.3 shows an example of densely sampled points using the ray shooting algorithm computed by Wang *et al.* (2017). After densely sampling the points on the meshes, for the training data, per each object we sub-sample N_s points uniformly.

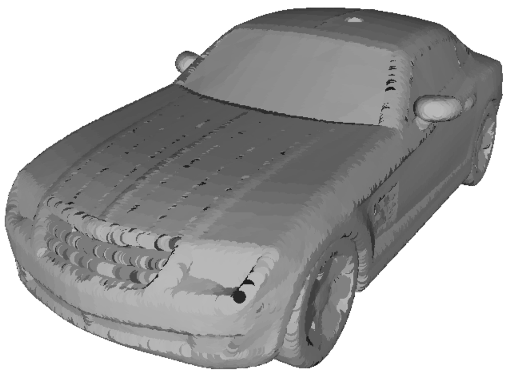

Figure 5.3: An example of the densely sampled points using the ray shooting algorithm provided by (Wang *et al.* (2017)). In this example, we sample points on a 3D shape of the car category from ShapeNet (Chang *et al.* (2015)).

5.2 Training Procedure

Our goal is to train a model which gets a 2D image as input and outputs a detailed singed distance representation. To do this, as we explained before, we use a deep convolutional encoder followed by two branches of the fully-connected layers. We use this convolutional network to encode the input 2D object to the parameters of the signed distance function. To train this model, we use our proposed loss function (Equation 4.4) as we explained in Section 4.4. As we mentioned in Subsection 5.1.2, we use pairs of 2D objects with their corresponding sample points and normal vectors as our training data. In the training procedure, we feed the 2D object to the encoder to transfer it to the parameters of the radial basis function. With the extracted parameters in the decoder part of the model, we construct the radial basis function f. To evaluate the constructed function f we use our proposed weakly-supervised loss function. After evaluation, we use the backpropagation algorithm to update the parameters of the model using the chain rule. Below, we explain each step of the training procedure of our model in more detail.

1. **Initialization:** As we stated before, to initialize the convolutional blocks of VGG16 (Simonyan & Zisserman (2014)), we employ the pre-trained VGG16 model which was trained on the ImageNet dataset (Deng *et al.* (2009)). Note that, in our network architecture, we initialize the weights of the fully-connected layers such that α_j and β_k for all of the j and k are equal to zero, and $\mathbf{c}_j$ for each center point are random vectors in a unit cube $[0, 1]^3$.

2. **Forward pass:** In this step, we feed the 2D images of the objects to the network. These images go through the convolutional blocks of the encoder to extract the features of the input data and transform the input to the parameters of the signed distance function. We encode the extracted features to the blending weights, linear polynomial parameters, and center points of the radial basis function. In the decoder part, we construct the RBF function using the estimated parameters. This function represents the 3D geometry of the object implicitly.

3. **Error calculation:** To evaluate the constructed function, we use our weakly-supervised loss function which we described in Section 4.4. Remember that, in our training data per each 2D image, we have sample points and their corresponding normal vectors of the ground-truth mesh. In addition to this, during the training procedure, we randomly sample $|\mathcal{R}|$ points in the unit cube $[0, 1]^3$. In the following, we explain how we evaluate each term of the loss function using these data.

   - **Zero-crossing constraint**: We use the sample points of the ground-truth mesh to evaluate this term. By evaluating this term, we measure how far the function f is from representing the surface as the zero-th level-set. In other words, for the sample point s_i, how close $f(s_i)$ is to zero.

   - **Normal constraint**: To evaluate this term, we use the sample points and their corresponding normal vectors. This term calculates the deviation of $\nabla f(s_i)$ from the normal vector $\mathbf{n}_i$.

- **Empty constraint**: We use this constraint to provide information about the empty part of the 3D space. As we already mentioned, to evaluate this term, per each input, we randomly sample $|\mathcal{R}|$ points in the unit cube $[0,1]^3$. With this term, we evaluate the estimated function f at each random point $\mathbf{r}$, and measure how far it is from the distance d, which is the signed Euclidean distance of $\mathbf{r}$ to its closest sample point.

4. **Backward pass:** As our proposed loss is differentiable, we can use the backpropagation algorithm to update the parameters of the network. We calculate the gradients with respect to the blending weights and center points for each term of the loss function and update all parameters of the network using the chain rule. To optimize these parameters we use the ADAM optimization algorithm Kingma & Ba (2014).

Chapter 6

Experiments

6.1 Implementation

We implement our network architecture and loss function using the Keras framework (Chollet *et al.* (2015)) with the Tesorflow (Abadi *et al.* (2015)) back-end. Keras is a high-level neural networks API which can be used on top of different frameworks such as TensorFlow, CNTK (Seide & Agarwal (2016)), or Theano (Theano Development Team (2016)). The details of the network architecture and the implementation are provided in Subsection 6.1.1. We accelerate the training procedure by using a fast parallel implementation of the proposed loss function. The implementation details of the loss function is provided in Subsection 6.1.2 .

6.1.1 Network Implementation

As we mentioned before (Section 5.2), we employ the pre-trained VGG16 model (Simonyan & Zisserman (2014)) followed by two branches of fully-connected (FC) layers to encode the input RGB images to the parameters of the RBF. To implement the pre-trained VGG16 model, we use the pre-trained weights of VGG16 which are provided in Keras. The VGG16 model was trained on ImageNet (Deng *et al.* (2009)), and we only use the convolutional blocks of this model. We connect the outputs of the VGG16 to the two branches of the FC layers. In the first branch, we have two connected FC layers that map the output of the VGG16 to the blending weights of the RBF and the parameters of the linear polynomial term. The first FC layer has 800 outputs and the size of the second one is equal to $N_c + 4$, where N_c is the number of basis functions. We have four more neurons in this FC layer to parameterize the linear polynomial part of the function f.

In another branch, we use two FC layers to map the output of VGG16 to the center points of the RBF. The first FC layer has 1500 outputs, and the size of the second layer of this branch is equal to $N_c \times 3$. Note that, as we have one center point per each basis function, so N_c is the number of basis functions and also the number of center points in RBFs. In these two branches, for the first layer of each branch we apply ReLU as the activation function, and in the second layer of each branch, we use the linear activation function $a(x) = x$.

In the decoder of our model, we construct the radial basis functions using the estimated parameters. To evaluate the estimated function, we use our proposed weakly-supervised loss function. As our loss function is differentiable, we use the backpropagation algorithm to update the parameters of the model, including the weights of the VGG16 and FC layers. To optimize the network's parameters, we use the ADAM optimization algorithm (Kingma & Ba (2014)) with a learning rate 3×10^{-4}.

6.1.2 Loss function Implementation

To accelerate the training procedure, we present a fast parallel implementation of the proposed loss function. We use the CUDA platform to parallelize the evaluation and gradient computation of this function. To parallelize this computation efficiently, we employ the parallel reduction approach.

In Equation 4.4, we can consider the evaluation of each term as the summation of all of the elements of a matrix, where the number of rows is equal to the number of sample points, and number of columns is equal to the number of center points. As all of the matrix elements are independent from each other, we can parallelize the evaluation of the loss function by launching a thread to compute each element of the matrix. For example, to evaluate the first term of the loss function, we have to compute $f(\mathbf{x}) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{x})$ for N_s different sample points. We can consider this term in the form of a $N_s \times N_c$ matrix. To parallelize the computation of this term, we specify $N_s \times N_c$ number of threads, where each of these threads computes one element of the term. To compute the summation of these elements in a parallel way we use warp reduce and block reduce approaches which we described in Subsection 3.3.3. Using the same idea, we parallelize the computation of the other two terms of the loss function. We also parallelize the computation of the gradients of our function with respect to the free variables $\mathcal{X}$ (Equation 4.3). As the gradients also take a matrix form, we use the same idea to compute the gradients of our loss function.

6.2 Results

In this section, we show qualitative and quantitative evaluations of our approach and compare it with the state-of-the-art methods. To evaluate our method, we use the dataset described in Subsection 5.1.1. To have a fair comparison with other approaches, we use the same training and validation split provided by Choy *et al.* (2016). We train separate models for four different categories of the ShapeNet-Core dataset and validate the capability of our models by providing qualitative and quantitative results. In Subsection 6.2.2, we compare our results with the ground-truth meshes qualitatively. In the next step, we compare the quality and accuracy of our reconstructions with the state-of-the-art approaches (Subsection 6.2.3). At the end of this section (Subsection 6.2.4) we analyze the quality and accuracy of the results using a different number of RBF parameters.

6.2.1 Training Data and Parameters

We validate our approach on four different categories of the ShapeNetCore dataset. We use 24 RGB images per object as input, and 15k sample points with their corresponding normals as the weakly-supervised label information. As we mentioned in Subsection 5.1.1, we sub-sample these 15k points uniformly out of all of the points on the ground-truth meshes. To train and evaluate our approach, we use 3535 3D objects of the car category, 6778 3D objects of the chair category, 2373 3D objects of the firearm category, and 3173 3D objects of the couch category. There are 24 RGB images in the different views so that we can have 24 pairs of training data per each object model. For example, in the car category, we have 84840 pairs of data, from which we use 80% of the data for training and 20% for validation.

We train our network on the training set of the four categories separately and validate the trained models using the validation set. For training the models, we use the same set of parameters and network architecture for all of the experiments. We provide details of the network architecture in Subsection 6.1.1. Each of these models is trained for 400000 iterations with a batch size of 10. We run the whole training procedure using our efficient implementation of the loss function on the GPU. The training and evaluations are performed on a NVIDIA Tesla V100 GPU with 16GB RAM. We empirically determined the parameters and weights of the loss function using the validation sets of the data. In the following, we provide all weights and parameters used in our approach:

- Weights of the loss function: $w_{point} = 90$, $w_{normal} = 0.2$, $w_{empty} = 15$.
- Number of random points for the empty-space term: $|\mathcal{R}| = 900$.
- Parameter of the ADAM optimizer (Kingma & Ba (2014)): $lr = 3 \times 10^{-4}$, $beta_1 = 0.9$, $beta_2 = 0.999$, $decay = 0.0$.
- Number of blending weights, linear polynomial parameters, and center points: $|\alpha| = 6000$, $|\beta| = 4$, $|\mathbf{c}| = 6000$.
- Parameter of the Gaussian function: $\lambda = 250$

After training the models, to validate our approach on different categories, we feed the 2D images of the test set to the trained models. Each model map the 2D input image to the parameters of the function f, which describes the underlying geometry of the object. With the extracted function f, we reconstruct the surface of the object using the marching cubes algorithms (Subsection 3.1.5). All of these surfaces are reconstructed in a unit cube with resolution $128 \times 128 \times 128$.

6.2.2 Qualitative Results

In this subsection, we show qualitative results of our approach on different categories. We visually compare the output of our method with the ground-truth meshes. In these results, we can see that our approach can reconstruct a variety of objects from different categories using only a single monocular image. Besides this, the results show that with the SDF representation, our method is capable of reconstructing a consistent surface throughout the objects with local details.

Figure 6.1 shows qualitative results of four objects from the car and chair categories. In this figure, we can see that our method can reconstruct the geometry of the cars and chairs with varying structures from different views. For example in the second row, we can see that our approach reconstructs the surface of the car only by using a single image from one side of the object. This figure also depicts the capability of our method to represent the local details of the objects. For example, in the car category we can see the four wheels of the cars are captured, or in the third row, the roundness of the chair's top rail is preserved in the reconstruction.

In another qualitative example, Figure 6.2 shows the results of four objects from the firearm and couch categories. The reconstruction results of different firearms and couches show the ability of our method to reconstruct the local details of the objects, such as the barrel of the firearms in the first two rows, and cushions of the couch in the fourth row.

Even though Figures 6.1, and 6.2 show the capabilities of our approach to estimate the surface of the different objects. We can also see some of the limitations and drawbacks of our method. For example, in Figure 6.1, the mirrors of the cars are not represented, or in the third row of this figure, the legs of the chair are not reconstructed with high-quality details.

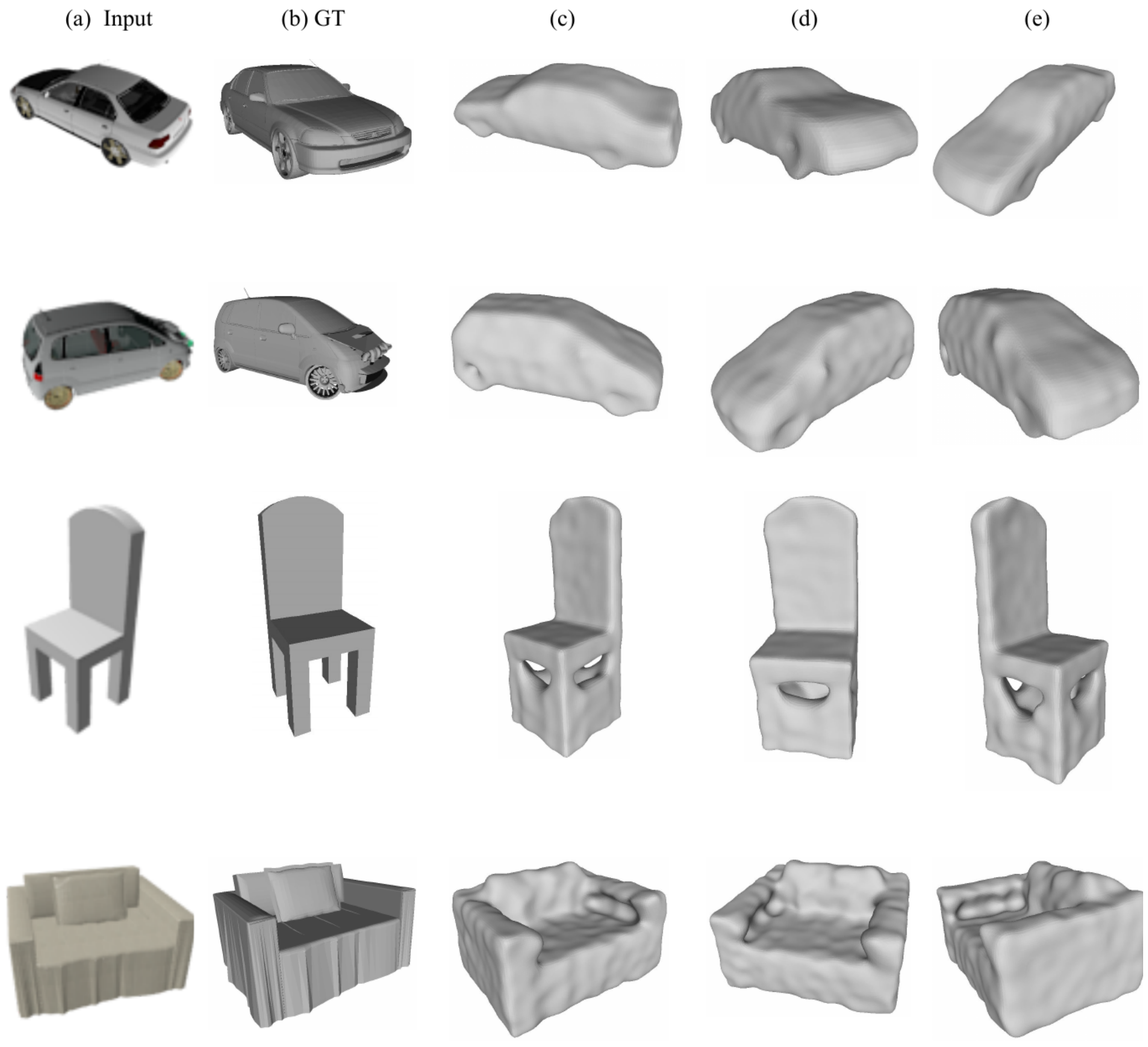

Figure 6.1: Qualitative results of the car and chair categories, (a) Input RGB image, (b) Ground-truth mesh, (c), (d), and (e) are our triangular mesh reconstructions from different views.

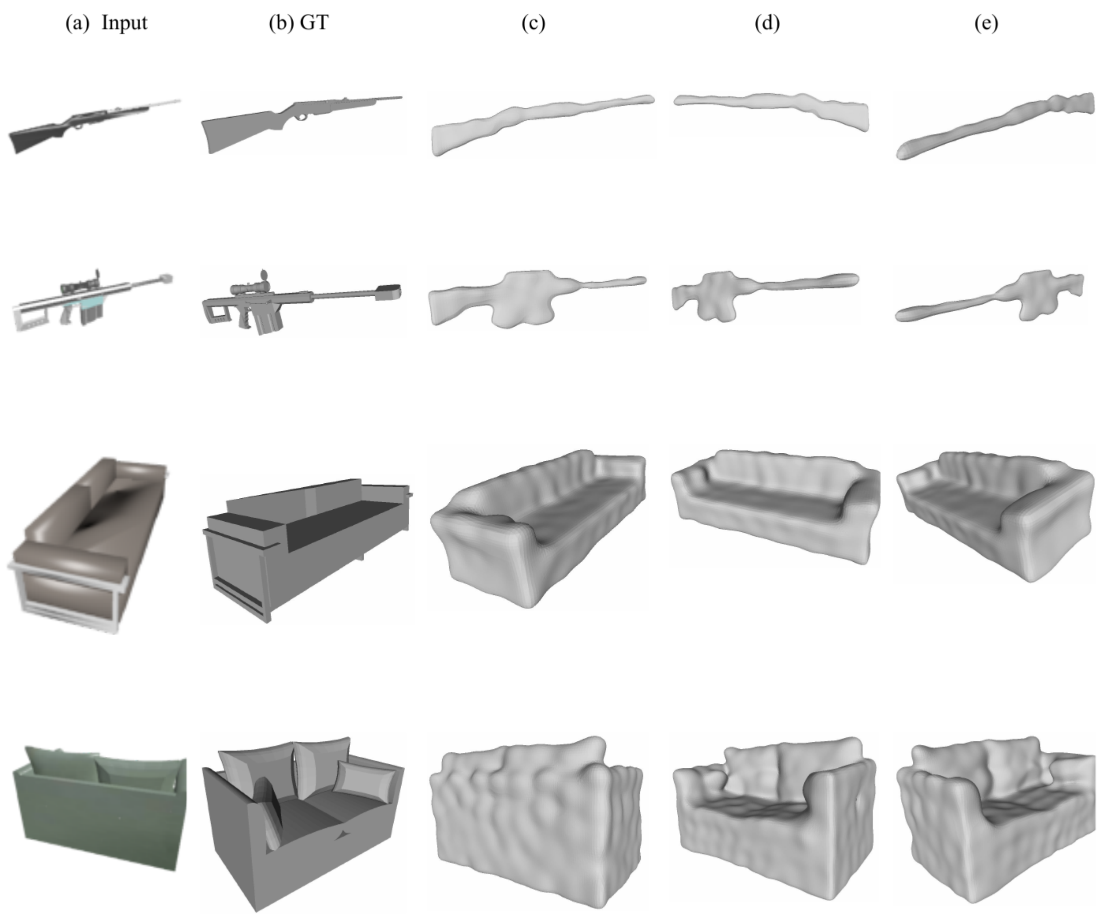

Figure 6.2: Qualitative results of the firearm and couch categories, (a) Input RGB image, (b) Ground-truth mesh, (c), (d), and (e) are our triangular mesh reconstructions from different views.

6.2.3 Comparison

We compare our proposed method with three state-of-the-art approaches, AtlasNet (Groueix *et al.* (2018)) which is concurrently done and has not been published yet, PointSetGen (Fan *et al.* (2017)), and 3D-R2N2 (Choy *et al.* (2016)). We provide qualitative comparisons of our results with these three methods and quantitatively compare the accuracy of our reconstruction with AtlasNet (Groueix *et al.* (2018)) and PointSetGen (Fan *et al.* (2017)). AtlastNet represent the surfaces of the objects using a group of patches, PointSetGen use point cloud representation to reconstruct the 3D geometry of the shape, and 3D-R2N2 use voxel-grid to represent the 3D shape of the objec. In Chapter 2 we introduced all of these approaches in more details.

To perform these comparisons, we train separate networks for different categories. However, all other methods train one model to reconstruct all of the categories. We use the trained networks provided by the authors for the evaluations [1]. Training one network for reconstructing multiple categories is left for future work. In these comparisons, the AtlasNet meshes are reconstructed with 25 patches, PointSetGen results are represented with 1024 points, and 3D-R2N2 outputs are reconstructed in a volume with resolution $32 \times 32 \times 32$.

In Figures 6.3, 6.4, 6.5, and 6.6, we qualitatively compare our approach with 3D-R2N2, PointSetGen, and AtlasNet. These results show that our method reconstructs more local details of the objects in comparison to 3D-R2N2 and PointSetGen. Besides this, as we use the SDF representation instead of a group of patches, in comparison to AtlasNet, our reconstructed surfaces are more consistent throughout the objects.

Based on the qualitative comparisons we can see that 3D-R2N2 and PointSetGen can not reconstruct many of the local details of the objects. For example, in Figure 6.3, 3D-R2N2 does not capture the wheels of the cars accurately, or in Figure 6.5, the barrel of the firearms is not reconstructed with fine-details. In Figure 6.5, in the fourth row, PointSetGen does not represent the thin structure of the firearm's stock. In contrast, our approach can represent these local details. For example, in Figure 6.3, our method represents the wheels of the cars with high-quality, and in Figure 6.5 it reconstructs the firearms' barrel and stock with more local details in comparison to the 3D-R2N2 and PointSetGen approaches.

In the qualitative results, we can also see the comparisons between our approach and AtlasNet. Figures 6.3, and 6.4 show that our method and AtlasNet can reconstruct the surface of the cars and chairs with more local details in comparison to the other methods. AtlasNet generates the surface of the objects with a group of patches, where these patches could be disconnected. In Figure 6.4, and 6.6, we can see the local discontinuities on AtlasNet's reconstructions. For example, in the first and second rows of Figure 6.4, there are disconnected patches on the arms of the chairs. In contrast, our approach can reconstruct surfaces which are more consistent throughout the objects.

In these qualitative results, there are also cases where AtlasNet reconstructs more local information of the objects in comparison to our method, such as mirrors of the cars (Figure 6.3). In Figure 6.4, we can see that AtlasNet reconstructs

[1]While it would be more fair to compare our results with other approaches in the same setting (one network per category), we do not have access to such trained models.

the thin structures of the chair such as legs and arms. However, our method is not able to reconstruct some of these thin structures correctly. For example, in the fourth row, one of the legs of the chair is not connected to the chair's seat. There are also cases where AtlasNet cannot capture the fine-details and thin structures of the objects, which shows that reconstructing these thin structures is a difficult problem for all of the methods. For example, in the fifth row of Figure 6.4, we can see that AtlasNet does not capture the four legs of the chair. However, our method represents all four legs of the chair. Even when AtlasNet represents more local details of the shapes in comparison to our method, we can see that our approach reconstructs surface which are more consistent throughout the objects.

To quantitatively compare our results with other methods, we use two criteria. First, we compare the Chamfer distance (Borgefors (1984)) of the output points set sampled from the reconstruction with the ground-truth point clouds. Using this criterion, we compare our method with PointSetGen and AtlasNet. This criterion only compare point clouds without considering the surface connectivity. To account for surface connectivity, we use the METRO tool (Cignoni *et al.* (1998)) to compute the Hausdorff distance between the outputs and the ground-truth meshes. To do these quantitative comparisons, we randomly select 20 objects from each category. Note that, all of the outputs and the ground-truths are normalized to a unit cube $[0, 1]^3$.

Table 6.1 shows the Chamfer distance (Borgefors (1984)) between the ground-truth point clouds and the outputs of PointSetGen, AtlasNet, and our method. Note that, ground-truth point clouds have 30k points, PointSetGen outputs have 1024 points, AtlastNet outputs have 30625 points, and our outputs have 10k to 500k points depending on the volume of the shape. To perform this comparison, we ran the iterative closest point (ICP) algorithm (Besl & McKay (1992)) for 30 iterations between each output and the corresponding ground-truth point cloud. We use the ICP algorithm to align the outputs of each method to the ground-truth point clouds. In this table (Table 6.1) we can see that our method outperforms PointSetGen on all of the categories and in average. This is mainly due to the tendency of PointSetGen to generate points inside the 3D object, while our approach, because of empty-space term in our loss function, generates surfaces on the 3D shape. Based on this table, AtlasNet performs better than our method and PointSetGen on all of the categories and in average. However, as we mentioned before, our method reconstruct consistent and smooth surfaces, while AtlasNet uses patches to generate the surfaces which can be locally disconnected. In Figure 6.7 we show the heat map errors of one object per each category. The

	Car	Chair	Firearm	Couch	Mean
PSG	3.122	3.108	0.945	2.586	2.440
AtlasNet	0.883	1.615	0.531	1.408	1.109
Ours	1.073	2.652	0.7199	1.773	1.554

Table 6.1: Quantitative comparison of PointSetGen (Fan *et al.* (2017)), AtlasNet (Groueix *et al.* (2018) and our method for different categories. The Chamfer distance is computed between each method and the ground-truth point cloud, after running ICP alignment with the ground-truth point cloud. The results are multiplied by 10^2, and the last column shows the category-wise mean.

	Car	Chair	Firearm	Couch	Mean
AtlasNet	1.108	1.574	1.005	1.068	1.188
Ours	1.077	1.854	0.944	1.260	1.283

Table 6.2: Quantitative comparison between AtlasNet (Groueix *et al.* (2018) and our method for different categories. The Hausdorff distance is computed between outputs and ground-truth meshes using the METRO tool (Cignoni *et al.* (1998)). The results are multiplied by 10, and the last column shows the category-wise mean.

heat maps show the Chamfer distance of each point to the corresponding ground-truth point cloud.

In Table 6.2, we report the Hausdorff distance between the ground-truth meshes and the outputs of AtlasNet and our approach. To compute the Hausdorff distance, we use the METRO tool (Cignoni *et al.* (1998)). METRO is a software to evaluate the difference between two triangular meshes, which is more meaningful for our outputs. This software considers the sampled points inside the faces of the meshes to compute the error. Based on the results in Table 6.2, our method outperforms the AtlasNet in the car and firearm categories, but in the chair and couch categories, AtlasNet performs better than our method. Even though in average, AtlasNet performs better than our approach, however, our method does not require to optimize the parameterization to generate the meshes at the test time. Beside this, AtlasNet cannot recover the orientation of the patches, and in most of the cases, some patches are flipped inside out. In contrast, in our approach because of the normal term we can correctly recover the orientation of the surface.

6.2.4 Ablation Study

In this subsection, we analyze the sensitivity of our approach to the number of blending weights and center points (RBF parameters) on the quality and accuracy of the results. We evaluate several models with different number of center points and blending weights on the car category to analyze the effect of RBF parameters on the quality of the reconstructed surfaces. Based on the qualitative and quantitative results, we show that with more RBF parameters, our approach can capture more details of the objects, and estimate more accurate surfaces.

To do this comparison, we train three models with different number of RBF parameters. To train these models, except the number of RBF parameters, we use the same parameters which we specified in Subsection 6.2.1. We provide the number of RBF parameters for each model in the following:

1. Model with **1k** RBF parameters: $|\alpha| = 1000$, $|\beta| = 4$, $|\mathbf{c}| = 1000$.
2. Model with **3k** RBF parameters: $|\alpha| = 3000$, $|\beta| = 4$, $|\mathbf{c}| = 3000$.
3. Model with **6k** RBF parameters: $|\alpha| = 6000$, $|\beta| = 4$, $|\mathbf{c}| = 6000$.

In Figure 6.8 we show the qualitative results of our approach with varying number of RBF parameters. In this figure, we can see that our model with 6k RBF parameters reconstructs surfaces with more local details and less surface noise in comparison to the other models. This figure shows that with 3k RBF parameters our reconstructed surfaces have lots of noises on the surface with spurious surfaces around the objects. We can also see that with 1k RBF parameters, our method cannot represent even a rough shape of the objects.

In Table 6.3, we provide the Hausdorff distance between ground-truth meshes and outputs of our models with different number of RBF parameters on the car category. We compute the Hausdorff distance using the METRO tool (Cignoni *et al.* (1998)). In this table, we can see that our approach improves as we increase the number of RBF parameters. In fact, this table shows that with more blending weights and center points, our approach can capture more accurate surfaces of the objects.

	1K	3K	6K
Ours	2.403	1.322	1.077

Table 6.3: Quantitative comparison among different models of our approach on car category, with 1k, 3k and 6k RBF parameters for car category. The Hausdorff distance is computed between outputs and ground-truth meshes using the METRO tool (Cignoni *et al.* (1998)). The results are multiplied by 10.

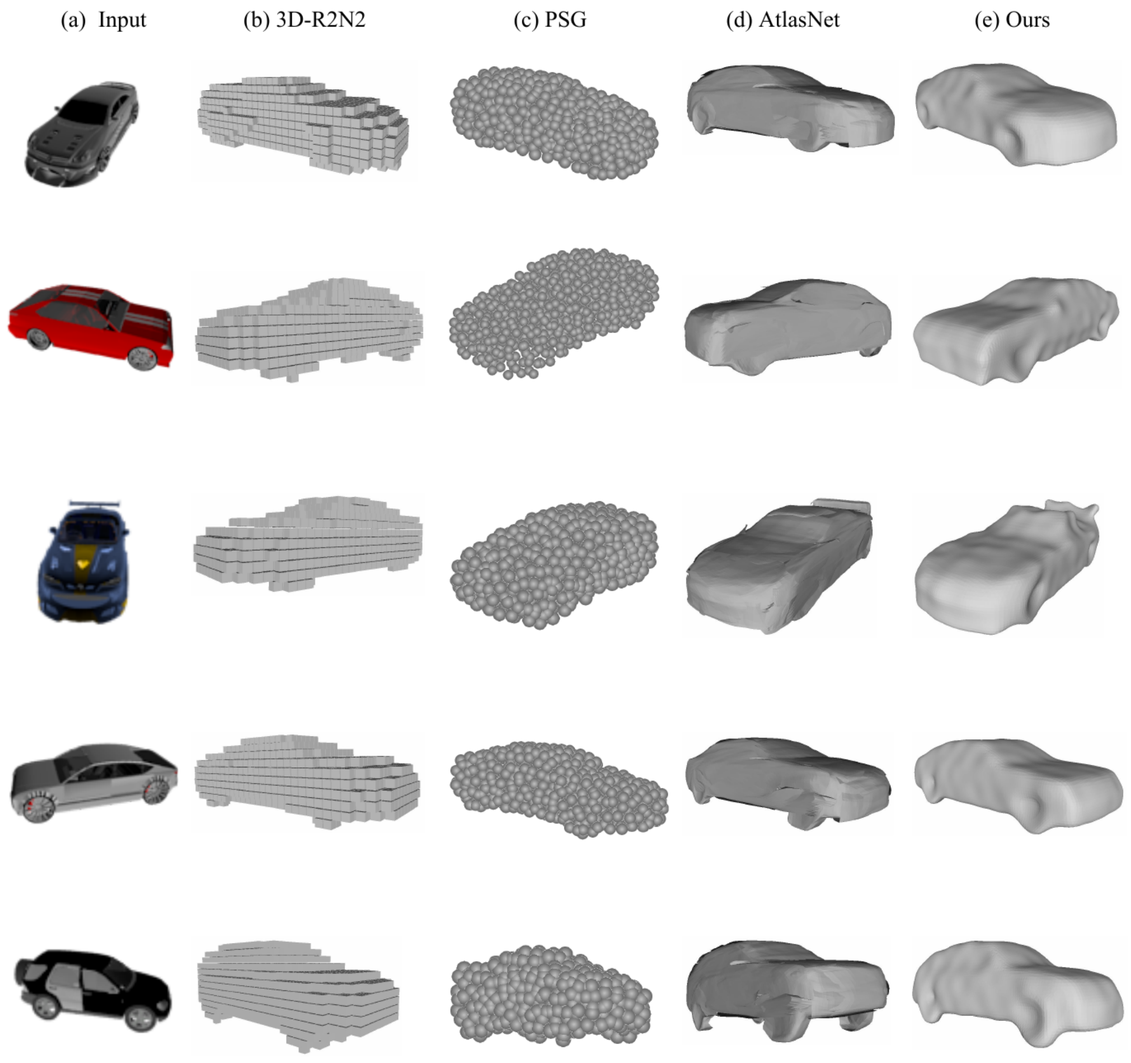

Figure 6.3: Visual comparison of the car category, (a) Input RGB image, (b) voxel-based reconstruction of 3D-R2N2 (Choy *et al.* (2016)), (c) Point cloud 3D model reconstruction of PointSetGen (Fan *et al.* (2017)), (d) Triangular mesh reconstruction of AtlasNet (Groueix *et al.* (2018), and (e) Our triangular mesh reconstruction.

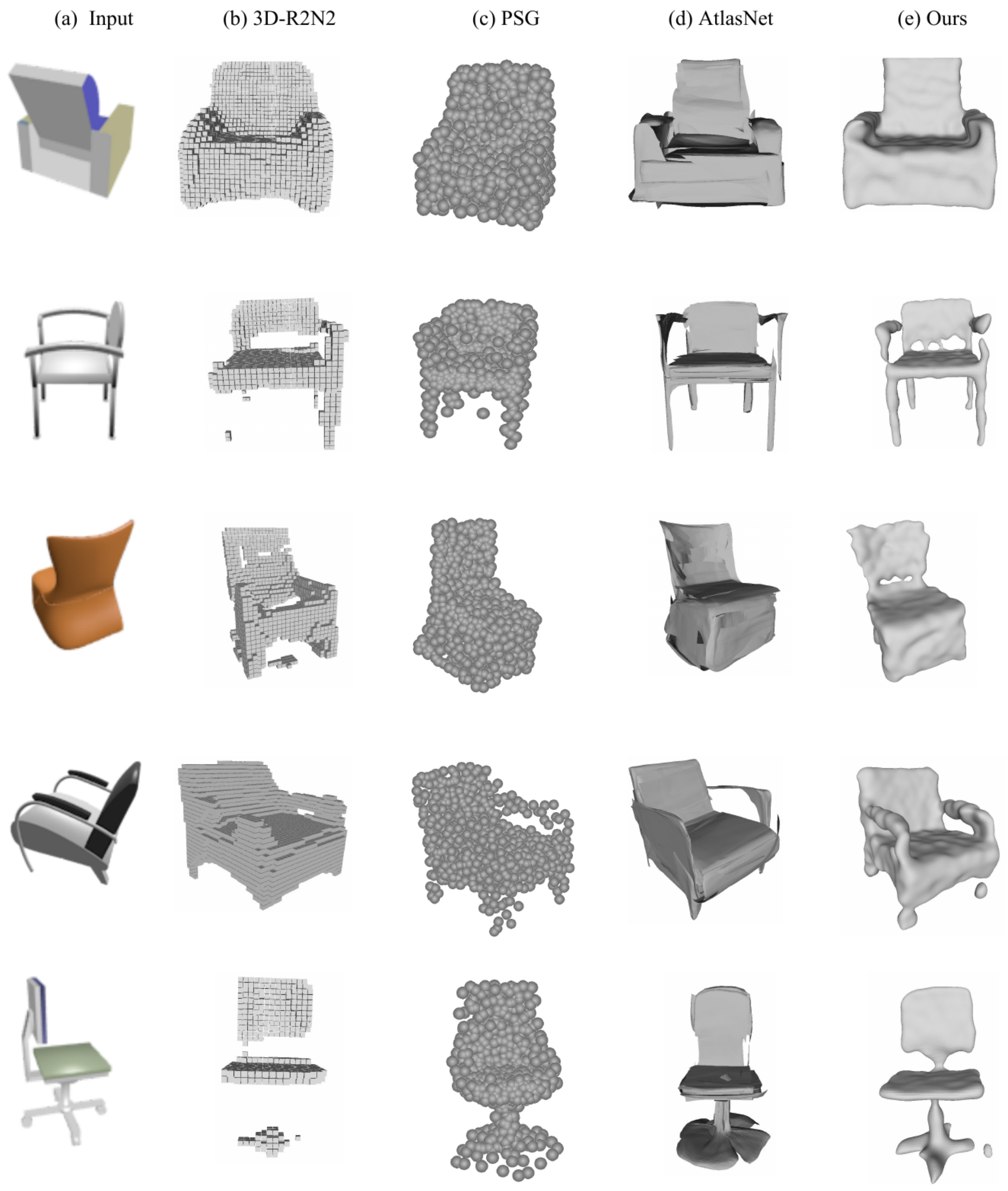

Figure 6.4: Visual comparison of the chair category, (a) Input RGB image, (b) voxel-based reconstruction of 3D-R2N2 (Choy *et al.* (2016)), (c) Point cloud 3D model reconstruction of PointSetGen (Fan *et al.* (2017)), (d) Triangular mesh reconstruction of AtlasNet (Groueix *et al.* (2018), and (e) Our triangular mesh reconstruction.

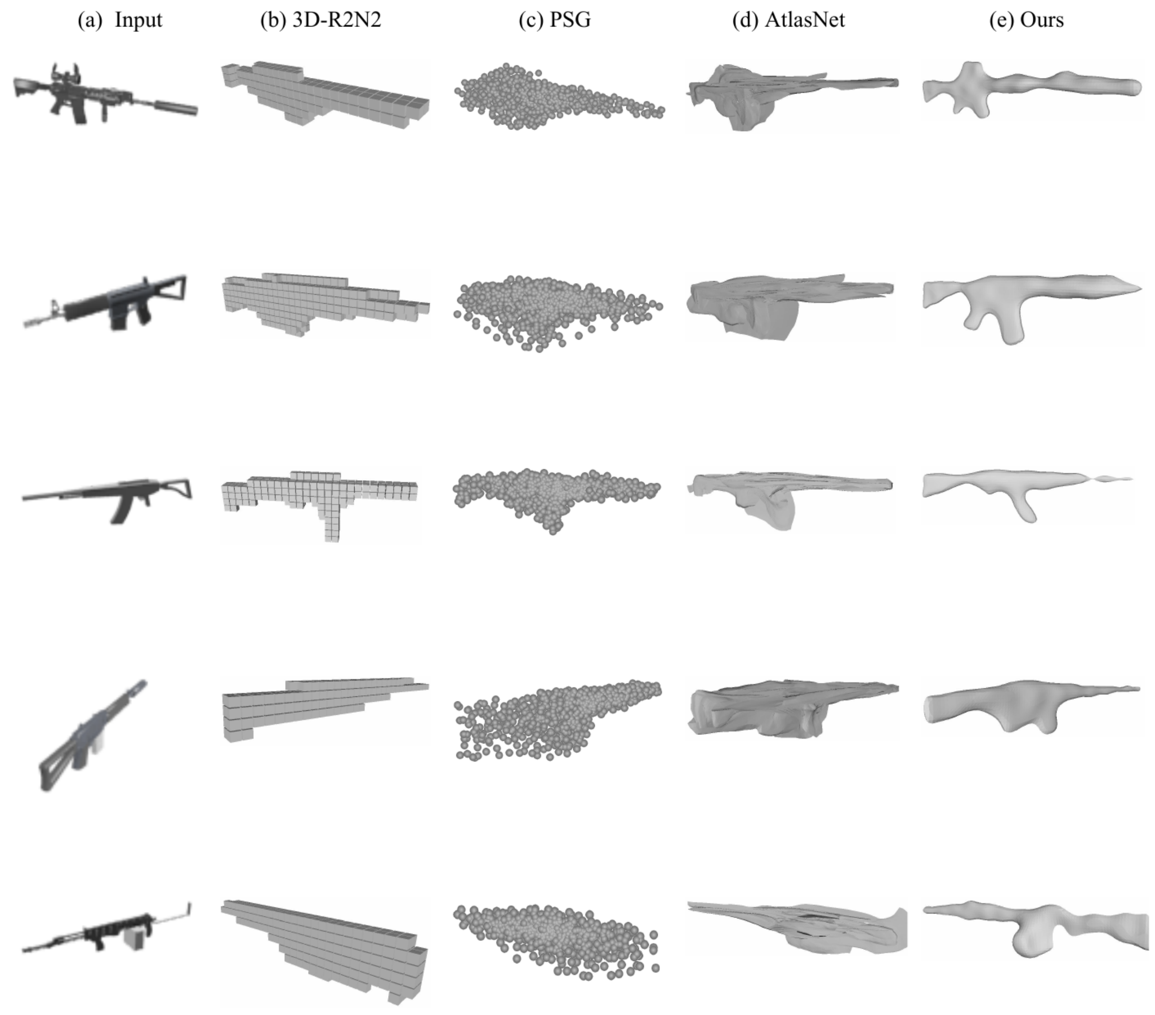

Figure 6.5: Visual comparison of the firearm category, (a) Input RGB image, (b) voxel-based reconstruction of 3D-R2N2 (Choy *et al.* (2016)), (c) Point cloud 3D model reconstruction of PointSetGen (Fan *et al.* (2017)), (d) Triangular mesh reconstruction of AtlasNet (Groueix *et al.* (2018), and (e) Our triangular mesh reconstruction.

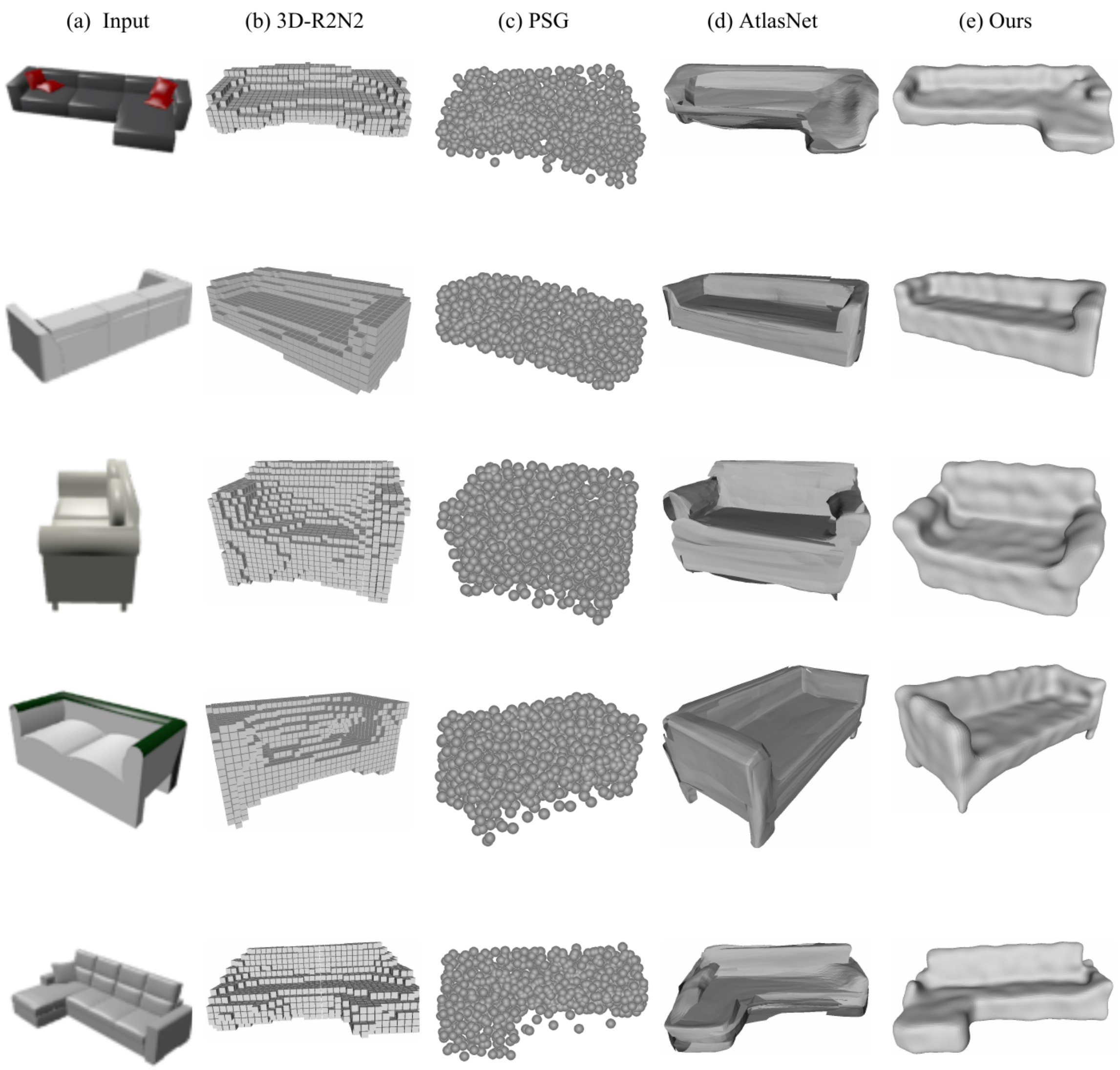

Figure 6.6: Visual comparison of the couch category, (a) Input RGB image, (b) voxel-based reconstruction of 3D-R2N2 (Choy *et al.* (2016)), (c) Point cloud 3D model reconstruction of PointSetGen (Fan *et al.* (2017)), (d) Triangular mesh reconstruction of AtlasNet (Groueix *et al.* (2018), and (e) Our triangular mesh reconstruction.

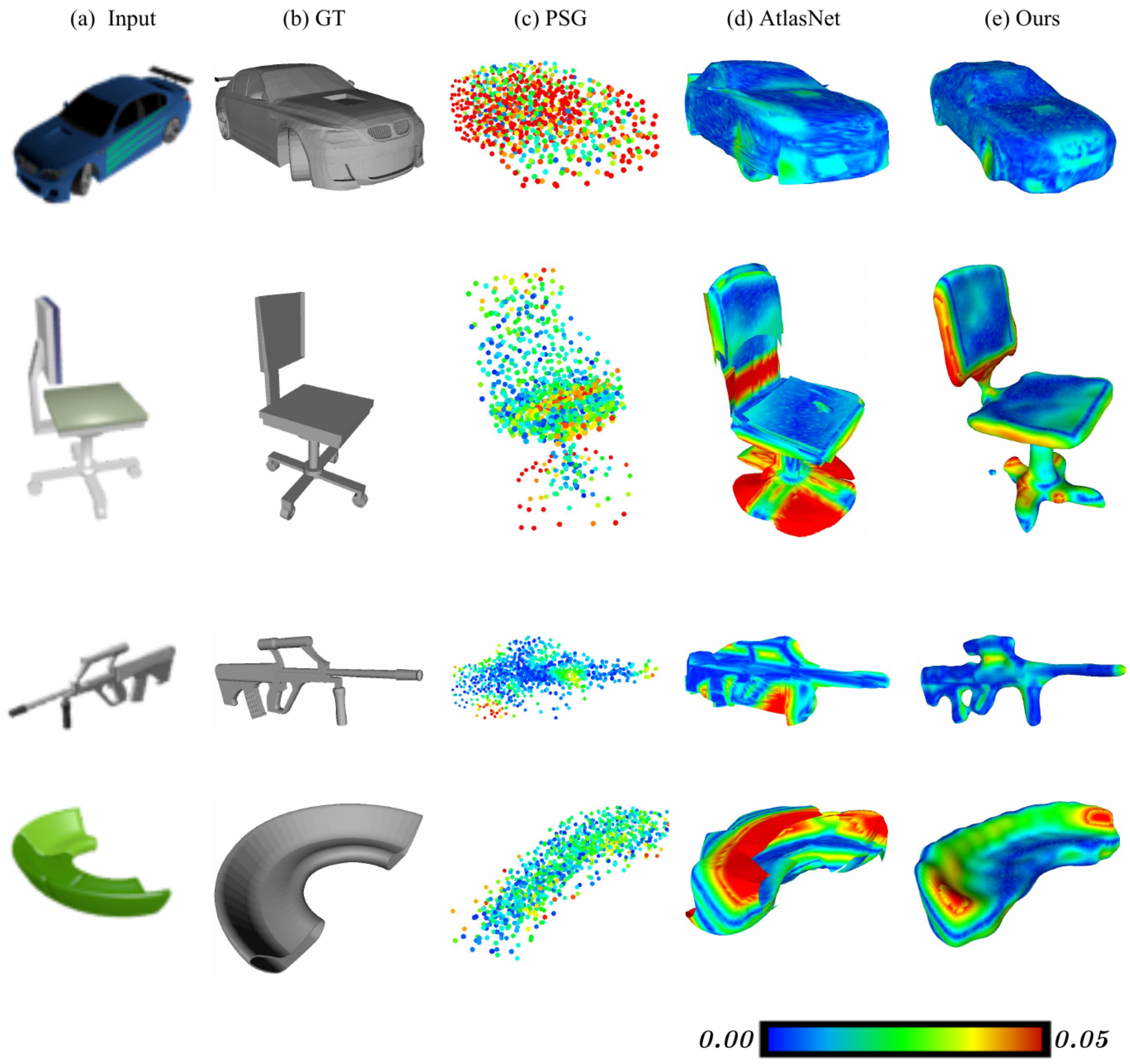

Figure 6.7: Visual comparison of the heat map errors which are computed based on Chamfer distance of the outputs to the ground-truth point clouds, (a) Input RGB image, (b) Ground-truth mesh, (c) Heat map errors of point cloud 3D model reconstruction of PointSetGen (Fan *et al.* (2017)), (d) Heat map errors of triangular mesh reconstruction of AtlasNet (Groueix *et al.* (2018), and (e) Heat map errors of our triangular mesh reconstruction. Note that to do this comparison we normalize all of the outputs and ground-truth points to a unit cube $[0, 1]^3$.

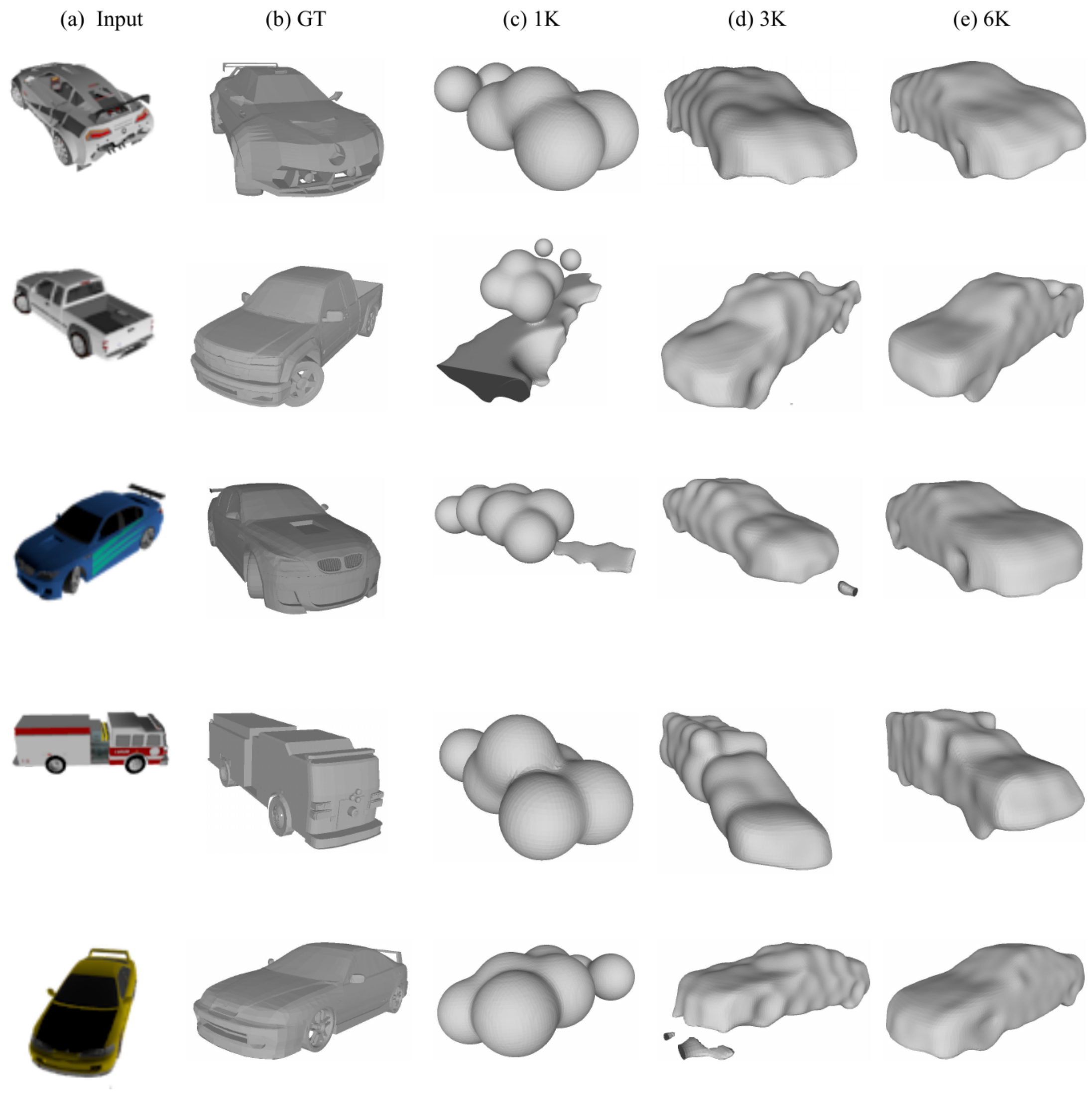

Figure 6.8: Visual comparison among different models of our approach on the car category, (a) Input RGB image, (b) Ground-truth mesh, (c) Our model with 1k RBF parameters , (d) Our model with 3k RBF parameters, and (e) Our model with 6k RBF parameters.

Chapter 7

Discussion and Future Work

In this thesis, we proposed a learning-based model to reconstruct the surface of an object depicted in a single image. Even though the proposed model can estimate an efficient and fine-detailed representation of the 2D image, this approach has its limitations and drawbacks. Beside this, there are many potential possibilities to improve and extend this method. In this chapter, we discuss the limitations of our approach and propose potential solutions to tackle them.

7.1 Limitations

In Section 6.2, we showed that our approach can reconstruct a smooth surface of the object including local details. However, there are cases where our method is not able to obtain some of the fine-details and the thin structures of the objects. For example, in the car category (Figure 6.3), for most of the cases we can not see the mirrors of the cars in the reconstructed surfaces. In the chair category (Figure 6.4), there are some cases where the legs of the chairs are not reconstructed with high-quality.

In Section 6.2, based on the qualitative and quantitative results we showed that our approach is capable of reconstructing objects of different categories. However, we have to train separate networks to reconstruct each category. In the next step of this project, we are going to extend our approach to reconstruct objects of different categories using a single trained model. To do this, we have to leverage the generalizability of our approach to capture the fine-detailed surfaces of a variety of objects. In the next section (Section 7.2), we provide some potential solutions to tackle these limitations.

7.2 Potential Solutions

Curvature-based Sampling:

In Subsection 5.1.2, we explained that to prepare the 3D information of the training data, N_s points are uniformly sampled from the ground-truth objects. By uniformly sampling the points on the objects, there will be a smaller number of points on the thin structures of the shape in comparison to other smooth structures. For example in Figure 7.1, we can see that there is a much smaller number of points on the legs in comparison to the number of points on the seat of the chair. With these sampled points, during the training procedure, our learning model mostly tries to reconstruct parts of the object which have more sample points. So, because of this sampling method, our approach can not reconstruct the fine-details and thin structure of some of the objects. One of the possible solution to remedy this problem is sampling the points based on their curvature, the points with high curvature have more chance to be sampled than the other points. With this sampling method we emphasis more on the non-flat regions of the shape, which could lead to having more sample points on thin structures.

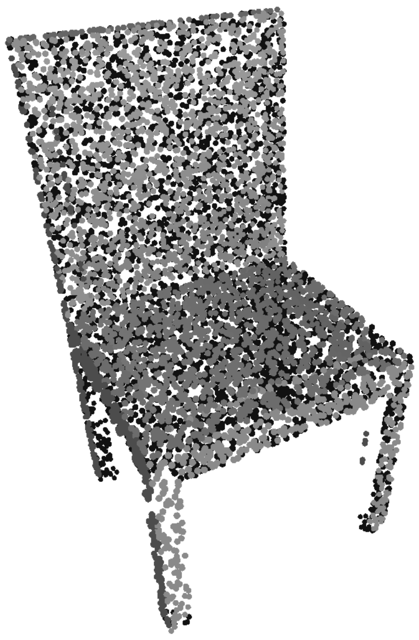

Figure 7.1: An example of a point cloud with 15000 points. These points are uniformly sampled from the points on the object.

Adaptive Basis Functions:

In our approach, the Gaussian function ($e^{-\lambda||\mathbf{c}-\mathbf{x}||_2^2}$) is used as the radial kernel to construct the radial basis functions f. In this kernel λ control the width of the function. By increasing and decreasing the width of the Gaussian function, we can control capturing the local and global details of the shapes. Currently, we use the same λ for all of the N_c basis functions, which means that to reconstruct all parts of the objects with different properties we use a Gaussian function with the same width. However, capturing the local and global details requires different ranges of λ. To have different λ for reconstructing local and global details of the objects, we can set λ as a free variable in the formulation of our method. More specifically, in addition to blending weights, linear polynomial parameters, and center points, we consider λ as a free variable which will be regressed by the network. Thus the new set of free variables will be:

$$\mathcal{X} = \{(\alpha_j, \mathbf{c}_j, \lambda_j, \beta_k),\ j = 1, \ldots, N_c,\ k = 1, \ldots, 4\}\ .$$

With this solution, we can have adaptive basis functions to represent the surfaces of the different objects. These adaptive basis function can change their supporting regions based on the local properties of the objects. In other words, with this solution we can generate more accurate surfaces with higher quality. In addition to this, using Gaussian functions with different widths we can increase the generalizability of our approach to reconstruct the objects with different features from multiple categories.

Auxiliary Classification Network:

Using an auxiliary classification network is another potential solution to increase the generalizability of our approach. This solution can facilitate reconstructing objects of different categories using a single trained model. In this auxiliary network, we can use the object class prediction as an auxiliary task for reconstructing the surface of the object. More specifically, in the training procedure of our approach we can jointly reconstruct the surface and predict the class of the objects. To implement this idea, we can use one or two fully-connected layers after the VGG16 model, in parallel to the other FC branches, to classify the input object using a cross-entropy loss (De Boer *et al.* (2005)).

Chapter 8

Conclusion

In this thesis, we proposed a novel learning-based approach for monocular reconstruction of arbitrary 3D shapes. Our approach takes a single RGB image as the input and outputs a detailed signed distance function (SDF) representation that describes the object geometry based on its zero-th level-set. We use a linear combination of floating radial basis functions to represent the SDFs efficiently, where the center points, the blending weights, and the parameters of the linear polynomial are free variables.

To map the input RGB image to the detailed signed distance function, we employ a deep convolutional encoder, which extracts the features of the input images and encodes them to the parameters of the floating radial basis function. This encoder estimates the parameters of the radial basis function to construct the SDF representation of the surface, where its zero-th level-set represents the surface of the 3D shape. Our approach efficiently combines the lower favorable space complexity of point-based approaches, with the surface approximation properties of distance field-based reconstruction techniques.

Since existing datasets do not provide the ground-truth signed distance function for each object, we proposed a weakly-supervised training approach to train our model. We optimize for the best fitting distance field by computing an alignment loss function on the ground-truth point cloud of the object. Instead of directly regressing points on the shape, we regress the parameters of the radial basis functions f, such that its zero-th level-set describe the underlying geometry of the input.

We perform qualitative and quantitative evaluations to compare our approach with the state-of-the-art methods. We demonstrate that our approach is capable of reconstructing high-quality surfaces with local details. Extensive evaluations show that the results of our approach are comparable to the results of other state-of-the-art methods.

Appendix A

Gradient of the Weakly-supervised Loss

We introduce the weakly-supervised loss function in Section 4.4 (Equation 4.4). As we already explained, this loss function is differentiable, and in the training process, we can use the backpropagation algorithm to update the parameters of the model based on the gradient of the loss function. In this appendix, we analytically derive the gradient of each term of the loss function with respect to the following free variables which we mentioned before in Section 4.1:

$$\mathcal{X} = \{(\alpha_j, \mathbf{c}_j, \beta_k),\ j = 1, \ldots, N_c,\ k = 1, \ldots, 4\} \ .$$

A.1 Zero-crossing Constraint

The following equation form the zero-crossing constraint.

$$E_{\text{point}}(\mathcal{X}) = w_{point} \sum_{i=1}^{N_s} ||f(\mathbf{s}_i)||^2 \ . \tag{A.1}$$

In this equation (Equation A.1) $f(\mathbf{s}_i) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i) + \sum_{k=1}^{4} \beta_k b_k(\mathbf{s}_i)$, where $\Psi(\mathbf{c}_j, \mathbf{s}_i) = e^{-\lambda||\mathbf{c}_j - \mathbf{s}_i||_2^2}$, and $b(\mathbf{s}_i) = (\mathbf{s}_{i,x}, \mathbf{s}_{i,y}, \mathbf{s}_{i,z}, 1)$. The partial derivative of this term with respect to a blending weight α_j can be derived as follow:

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \alpha_j} = w_{point} \sum_{i=1}^{N_s} \frac{\partial ||f(\mathbf{s}_i)||^2}{\partial \alpha_j} \tag{A.2}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \alpha_j} = w_{point} \sum_{i=1}^{N_s} 2f(\mathbf{s}_i) \frac{\partial f(\mathbf{s}_i)}{\partial \alpha_j} \tag{A.3}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \alpha_j} = w_{point} \sum_{i=1}^{N_s} 2f(\mathbf{s}_i) \frac{\partial \big(\sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i) + \sum_{k=1}^{4} \beta_k b_k(\mathbf{s}_i) \big)}{\partial \alpha_j} \tag{A.4}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \alpha_j} = w_{point} \sum_{i=1}^{N_s} 2f(\mathbf{s}_i) \Psi(\mathbf{c}_j, \mathbf{s}_i) \ \ . \tag{A.5}$$

We can derive the partial derivative of the E_{point} with respect to the β_k using the same idea based on the Equations A.2 to A.5.

Center points are another free variables in our loss function. Note that, in this appendix we consider the center point $\mathbf{c}_j = (\mathbf{c}_{j,x}, \mathbf{c}_{j,y}, \mathbf{c}_{j,z})$. To derive the partial derivative of the E_{point} with respect to $\mathbf{c}_{j,x}$ we can follow the following steps:

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{point} \sum_{i=1}^{N_s} \frac{\partial ||f(\mathbf{s}_i)||^2}{\partial \mathbf{c}_{j,x}} \tag{A.6}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{point} \sum_{i=1}^{N_s} 2f(\mathbf{s}_i) \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{c}_{j,x}} \tag{A.7}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{point} \sum_{i=1}^{N_s} 2f(\mathbf{s}_i) \frac{\partial \big(\sum_{j=1}^{N_c} \alpha_j e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2} + \sum_{k=1}^{4} \beta_k b_k(\mathbf{s}_i) \big)}{\partial \mathbf{c}_{j,x}} \tag{A.8}$$

$$\frac{\partial E_{\text{point}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{point} \sum_{i=1}^{N_s} 4f(\mathbf{s}_i)\ (-\lambda)\ \alpha_j e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}\ (\mathbf{c}_{j,x} - \mathbf{s}_{i,x}) \ \ . \tag{A.9}$$

A.2 Normal Constraint

In Subsection 4.4.2, based on the following equation (Equation A.10) we define the normal constraint.

$$E_{\text{normal}}(\mathcal{X}) = w_{normal} \sum_{i=1}^{N_s} ||\nabla f(\mathbf{s}_i) - \mathbf{n}_i||^2 \ \ . \tag{A.10}$$

In this equation (Equation A.10) $f(\mathbf{s}_i) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i) + \sum_{k=1}^{4} \beta_k b_k(\mathbf{s}_i)$, where $\Psi(\mathbf{c}_j, \mathbf{s}_i) = e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}$, and $b(\mathbf{s}_i) = (\mathbf{s}_{i,x}, \mathbf{s}_{i,y}, \mathbf{s}_{i,z}, 1)$. In Equation A.10, $\nabla f(\mathbf{s}_i) = (\frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,x}}, \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,y}}, \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,z}})$. We derive each component of the $\nabla f(\mathbf{s}_i)$ in the follow:

$$v_1 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,x}} = \sum_{j=1}^{N_c} \frac{\partial \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i)}{\partial \mathbf{s}_{i,x}} + \sum_{k=1}^{4} \frac{\partial \beta_k b_k(\mathbf{s}_i)}{\partial \mathbf{s}_{i,x}} \tag{A.11}$$

$$v_1 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,x}} = \sum_{j=1}^{N_c} 2(\lambda)\ \alpha_j e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}\ (\mathbf{c}_{j,x} - \mathbf{s}_{i,x}) + \beta_1\ . \tag{A.12}$$

$$v_2 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,y}} = \sum_{j=1}^{N_c} \frac{\partial \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i)}{\partial \mathbf{s}_{i,y}} + \sum_{k=1}^{4} \frac{\partial \beta_k b_k(\mathbf{s}_i)}{\partial \mathbf{s}_{i,y}} \tag{A.13}$$

$$v_2 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,y}} = \sum_{j=1}^{N_c} 2(\lambda)\ \alpha_j e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}\ (\mathbf{c}_{j,y} - \mathbf{s}_{i,y}) + \beta_2\ . \tag{A.14}$$

$$v_3 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,z}} = \sum_{j=1}^{N_c} \frac{\partial \alpha_j \Psi(\mathbf{c}_j, \mathbf{s}_i)}{\partial \mathbf{s}_{i,z}} + \sum_{k=1}^{4} \frac{\partial \beta_k b_k(\mathbf{s}_i)}{\partial \mathbf{s}_{i,z}} \tag{A.15}$$

$$v_3 = \frac{\partial f(\mathbf{s}_i)}{\partial \mathbf{s}_{i,z}} = \sum_{j=1}^{N_c} 2(\lambda)\ \alpha_j e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}\ (\mathbf{c}_{j,z} - \mathbf{s}_{i,z}) + \beta_3\ . \tag{A.16}$$

Based on the Equations A.11 to A.16 we can derive the partial derivative of the normal constraint with respect to a blending weight α_j as follow:

$$\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \alpha_j} = w_{normal} \sum_{i=1}^{N_s} \frac{\partial ||\nabla f(\mathbf{s}_i) - \mathbf{n}_i||^2}{\partial \alpha_j} \tag{A.17}$$

$$\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \alpha_j} = w_{normal} \sum_{i=1}^{N_s} \frac{\partial ||(v1, v2, v3) - (\mathbf{n}_{i,x}, \mathbf{n}_{i,y}, \mathbf{n}_{i,z})||^2}{\partial \alpha_j} \tag{A.18}$$

$$\begin{aligned}\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \alpha_j} = w_{normal} \sum_{i=1}^{N_s} \Big(&2(v1 - \mathbf{n}_{i,x})\ 2(\lambda)\ e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}(\mathbf{c}_{j,x} - \mathbf{s}_{i,x}) \\ &+ 2(v2 - \mathbf{n}_{i,y})\ 2(\lambda)\ e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}(\mathbf{c}_{j,y} - \mathbf{s}_{i,y}) \\ &+ 2(v3 - \mathbf{n}_{i,z})\ 2(\lambda)\ e^{-\lambda ||\mathbf{c}_j - \mathbf{s}_i||_2^2}(\mathbf{c}_{j,z} - \mathbf{s}_{i,z})\Big)\end{aligned}\ . \tag{A.19}$$

We can follow the same idea of Equations A.17 to A.19 to derive the derivative of E_{normal} with respect to the β_k.

We derive the partial derivative of the term E_{normal} with respect to $\mathbf{c}_{j,x}$ as follow:

$$\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{normal} \sum_{i=1}^{N_s} \frac{\partial ||\nabla f(\mathbf{s}_i) - \mathbf{n}_i||^2}{\partial \mathbf{c}_{j,x}} \tag{A.20}$$

$$\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{normal} \sum_{i=1}^{N_s} \frac{\partial ||(v1, v2, v3) - (\mathbf{n}_{i,x}, \mathbf{n}_{i,y}, \mathbf{n}_{i,z})||^2}{\partial \mathbf{c}_{j,x}} \tag{A.21}$$

$$\begin{aligned}\frac{\partial E_{\text{normal}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{normal} \sum_{i=1}^{N_s} \Big(& 2(v1 - \mathbf{n}_{i,x})\ 4(-\lambda^2)\ \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{s}_i||_2^2}\ (\mathbf{c}_{j,x} - \mathbf{s}_{i,x})^2 + 2\lambda\ \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{s}_i||_2^2} \\ &+ 2(v2 - \mathbf{n}_{i,y})\ 4(-\lambda^2)\ \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{s}_i||_2^2}(\mathbf{c}_{j,y} - \mathbf{s}_{i,y})(\mathbf{c}_{j,x} - \mathbf{s}_{i,x}) \\ &+ 2(v3 - \mathbf{n}_{i,z})\ 4(-\lambda^2)\ \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{s}_i||_2^2}(\mathbf{c}_{j,z} - \mathbf{s}_{i,z})(\mathbf{c}_{j,x} - \mathbf{s}_{i,x})\Big) .\end{aligned} \tag{A.22}$$

A.3 Empty-space Constraint

In Subsection 4.4.3, we define the empty-space constraint as follow:

$$E_{\text{empty}}(\mathcal{X}) = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} ||f(\mathbf{r}) - d||^2 \ . \tag{A.23}$$

In this equation (Equation A.23), $\mathbf{r}$ is a random point in unit cube $[0,1]^3$, and d is the signed Euclidean distance of $\mathbf{r}$ to its closest sample points. Note that, in this equation (Equation A.1) $f(\mathbf{r}) = \sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{r}) + \sum_{k=1}^{4} \beta_k b_k(\mathbf{r})$, where $\Psi(\mathbf{c}_j, \mathbf{r}) = e^{-\lambda||\mathbf{c}_j - \mathbf{r}||_2^2}$, and $b(\mathbf{r}) = (\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z, 1)$. We can derive the partial derivative of the empty-space term with respect to a blending weight α_j based on the following steps:

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \alpha_j} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} \frac{\partial ||f(\mathbf{r}) - d||^2}{\partial \alpha_j} \tag{A.24}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \alpha_j} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big) \frac{\partial f(\mathbf{r})}{\partial \alpha_j} \tag{A.25}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \alpha_j} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big)\frac{\partial\big(\sum_{j=1}^{N_c} \alpha_j \Psi(\mathbf{c}_j, \mathbf{r}) + \sum_{k=1}^{4} \beta_k b_k(\mathbf{r})\big)}{\partial \alpha_j} \tag{A.26}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \alpha_j} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big)\Psi(\mathbf{c}_j, \mathbf{r}) \ . \tag{A.27}$$

Based on the same idea of Equations A.24 to A.27, we can derive the partial derivative of the E_{empty} with respect to the β_k.

To derive the partial derivative of the E_{empty} with respect to one element of a center point we can follow the following steps:

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} \frac{\partial ||f(\mathbf{r}) - d||^2}{\partial \mathbf{c}_{j,x}} \tag{A.28}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big)\frac{\partial f(\mathbf{r})}{\partial \mathbf{c}_{j,x}} \tag{A.29}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big)\frac{\partial\big(\sum_{j=1}^{N_c} \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{r}||_2^2} + \sum_{k=1}^{4} \beta_k b_k(\mathbf{r})\big)}{\partial \mathbf{c}_{j,x}} \tag{A.30}$$

$$\frac{\partial E_{\text{empty}}(\mathcal{X})}{\partial \mathbf{c}_{j,x}} = w_{empty} \sum_{(\mathbf{r},d)\in\mathcal{R}} 2\big(f(\mathbf{r}) - d\big)\ 2(-\lambda)\ \alpha_j e^{-\lambda||\mathbf{c}_j - \mathbf{r}||_2^2}\ (\mathbf{c}_{j,x} - \mathbf{r}_x) \ . \tag{A.31}$$

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. & Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. 49

Aloimonos, J. (1988). Shape from texture. *Biological cybernetics*, **58**, 345–360. 10

Bao, S.Y., Chandraker, M., Lin, Y. & Savarese, S. (2013). Dense object reconstruction with semantic priors. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, 1264–1271, IEEE. 8

Besl, P.J. & McKay, N.D. (1992). Method for registration of 3-d shapes. In *Sensor Fusion IV: Control Paradigms and Data Structures*, vol. 1611, 586–607, International Society for Optics and Photonics. 56

Boissonnat, J.D. (1984). Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics (TOG)*, **3**, 266–286. 5

Borgefors, G. (1984). Distance transformations in arbitrary dimensions. *Computer vision, graphics, and image processing*, **27**, 321–345. 56

Carr, J.C., Beatson, R.K., Cherrie, J.B., Mitchell, T.J., Fright, W.R., McCallum, B.C. & Evans, T.R. (2001). Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th* *annual conference on Computer graphics and interactive techniques*, 67–76, ACM. 6, 17, 18

Cazals, F. & Giesen, J. (2006). Delaunay triangulation based surface reconstruction. In *Effective computational geometry for curves and surfaces*, 231–276, Springer. 5, 6

Chang, A.X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L. & Yu, F. (2015). ShapeNet: An Information-Rich 3D Model Repository. Tech. Rep. arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago. 2, 7, 43, 44, 45

Cheng, J., Grossman, M. & McKercher, T. (2014). *Professional Cuda C Programming*. John Wiley & Sons. 29, 32

Chollet, F. *et al.* (2015). Keras. `https://keras.io`. 49

Choy, C.B., Xu, D., Gwak, J., Chen, K. & Savarese, S. (2016). 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In *European Conference on Computer Vision*, 628–644, Springer. xv, 2, 9, 11, 44, 51, 54, 60, 61, 62, 63

Cignoni, P., Rocchini, C. & Scopigno, R. (1998). Metro: Measuring error on simplified surfaces. In *Computer Graphics Forum*, vol. 17, 167–174, Wiley Online Library. 56, 57, 58, 59

Dame, A., Prisacariu, V.A., Ren, C.Y. & Reid, I. (2013). Dense reconstruction using 3d object shape priors. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, 1288–1295, IEEE. 8

De Boer, P.T., Kroese, D.P., Mannor, S. & Rubinstein, R.Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research*, **134**, 19–67. 69

Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, 248–255, IEEE. 28, 37, 46, 49

Dong, C., Loy, C.C., He, K. & Tang, X. (2016). Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, **38**, 295–307. 2

Fan, H., Su, H. & Guibas, L. (2017). A point set generation network for 3d object reconstruction from a single image. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 38. 2, 3, 11, 12, 54, 57, 60, 61, 62, 63, 64

Fuentes-Pacheco, J., Ruiz-Ascencio, J. & Rendón-Mancha, J.M. (2015). Visual simultaneous localization and mapping: a survey. *Artificial Intelligence Review*, **43**, 55–81. 8

Girdhar, R., Fouhey, D.F., Rodriguez, M. & Gupta, A. (2016). Learning a predictable and generative vector representation for objects. In *European Conference on Computer Vision*, 484–499, Springer. 11

Girshick, R., Donahue, J., Darrell, T. & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 580–587. 2

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. MIT Press, `http://www.deeplearningbook.org`. 25

Gopi, M., Krishnan, S. & Silva, C.T. (2000). Surface reconstruction based on lower dimensional localized delaunay triangulation. In *Computer Graphics Forum*, vol. 19, 467–478, Wiley Online Library. 5

Groueix, T., Fisher, M., Kim, V.G., Russell, B. & Aubry, M. (2018). AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 7, 12, 54, 57, 60, 61, 62, 63, 64

Gryka, M., Terry, M. & Brostow, G.J. (2015). Learning to remove soft shadows. *ACM Trans. Graph.*, **34**, 153:1–153:15. 2

Hane, C., Savinov, N. & Pollefeys, M. (2014). Class specific 3d object shape priors using surface normals. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 652–659. 8

Häne, C., Tulsiani, S. & Malik, J. (2017). Hierarchical surface prediction for 3d object reconstruction. In *arXiv preprint arXiv:1704.00710*. 2, 11

Harris, M. (2007). Optimizing parallel reduction in cuda. 32

Hartley, R. & Zisserman, A. (2003). *Multiple view geometry in computer vision*. Cambridge university press. 7

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. & Stuetzle, W. (1992). *Surface reconstruction from unorganized points*, vol. 26. ACM. 6

Horn, B.K. & Brooks, M.J. (1989). *Shape from shading*. MIT press. 10

Kalantari, N.K., Bako, S. & Sen, P. (2015). A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph.*, **34**, 122–1. 2

Kar, A., Häne, C. & Malik, J. (2017). Learning a multi-view stereo machine. In *Advances in Neural Information Processing Systems*, 364–375. 9

Karn, U. (2016). An intuitive explanation of convolutional neural networks. xv, 23

Karpathy, A. (2016). Convolutional neural networks. 26

Kingma, D.P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 47, 50, 52

Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105. 2

Lee, H., Grosse, R., Ranganath, R. & Ng, A.Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, 609–616, ACM. 26, 27

Liao, Y., Donne, S. & Geiger, A. (2018). Deep marching cubes: Learning explicit surface representations. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. 7

Long, J., Shelhamer, E. & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 3431–3440. 2

Lorensen, W.E. & Cline, H.E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, vol. 21, 163–169, ACM. xv, 7, 20, 21

Luitjens, J. (2014). Faster parallel reductions on kepler. 33

Marquardt, D.W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, **11**, 431–441. 19

Miller, G.A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, **38**, 39–41. 43

Nair, V. & Hinton, G.E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, 807–814. 24, 37

Nielsen, M.A. (2015). *Neural networks and deep learning*. Determination Press. 24

NVIDIA, C. (2012). Programming guide. nvidia corporation. xv, 29, 30, 31

Ohtake, Y., Belyaev, A., Alexa, M., Turk, G. & Seidel, H.P. (2003). Multi-level partition of unity implicits. In *ACM Transactions on Graphics (TOG)*, vol. 22, 463–470, ACM. 6

Oquab, M., Bottou, L., Laptev, I. & Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, 1717–1724, IEEE. 2

Osher, S. & Fedkiw, R.P. (2001). Level set methods: an overview and some recent results. *Journal of Computational physics*, **169**, 463–502. 14

Osher, S. & Sethian, J.A. (1988). Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *Journal of computational physics*, **79**, 12–49. 14, 15

Özyeşil, O., Voroninski, V., Basri, R. & Singer, A. (2017). A survey of structure from motion*. *Acta Numerica*, **26**, 305–364. 8

Remondino, F. & El-Hakim, S. (2006). Image-based 3d modelling: a review. *The Photogrammetric Record*, **21**, 269–291. 10

Schmidhuber, J. & Hochreiter, S. (1997). Long short-term memory. *Neural Comput*, **9**, 1735–1780. 9

Schonberger, J.L. & Frahm, J.M. (2016). Structure-from-motion revisited. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4104–4113. 8

Seide, F. & Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2135–2135, ACM. 49

Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. 37, 46, 49

Sinha, P. & Adelson, E. (1993). Recovering reflectance and illumination in a world of painted polyhedra. In *Computer Vision, 1993. Proceedings., Fourth International Conference on*, 156–163, IEEE. 1

Süssmuth, J., Meyer, Q. & Greiner, G. (2010). Surface reconstruction based on hierarchical floating radial basis functions. In *Computer Graphics Forum*, vol. 29, 1854–1864, Wiley Online Library. 5, 6, 18, 19, 20, 36, 39, 40

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–2826. 2

Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, **abs/1605.02688**. 49

Turk, G. & O'brien, J.F. (1999). Variational implicit surfaces. Tech. rep., Georgia Institute of Technology. 3, 6, 16

Wang, L. (2016). Recognition of human activities using continuous autoencoders with wearable sensors. *Sensors*, **16**, 189. xv, 22

Wang, P.S., Liu, Y., Guo, Y.X., Sun, C.Y. & Tong, X. (2017). O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *ACM Transactions on Graphics (TOG)*, **36**, 72. 2, 45

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X. & Xiao, J. (2015). 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1912–1920. 2, 7

Yan, X., Yang, J., Yumer, E., Guo, Y. & Lee, H. (2016). Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision. In *Advances in Neural Information Processing Systems*, 1696–1704. 11

Yumer, M.E. & Mitra, N.J. (2016). Learning semantic deformation flows with 3d convolutional networks. In *European Conference on Computer Vision (ECCV 2016)*, –, Springer. 3

Zeiler, M.D. & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, 818–833, Springer. 2