UPPSALA
UNIVERSITET

UPTEC F 20006
Examensarbete 30 hp
Mars 2020

Unsupervised Deep Learning for Primitive-Based Shape Abstraction

Vilhelm Urpi Hedbjörk

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Unsupervised Deep Learning for Primitive-Based Shape Abstraction

Vilhelm Urpi Hedbjörk

The ability to give simple but representative descriptions of complex shapes, known as shape abstraction, is a useful tool in computer science. An unsupervised deep neural network that predicts a set of primitive shapes to describe an arbitrary three-dimensional input shape is constructed and trained on airplane and chair models. The network is based on the research paper by D. Paschalidou et al., with a few modifications to rectify unsatisfactory results. The network is able to predict sets of primitives that resemble the shape of the input. However, the sets of primitives are not parsimonious, i.e. several primitives may be used to describe the same part of an object. A remedy for this problem is proposed through an addition of a punishing function for overlapping primitives, but fail to produce the desired effect. Results from a direct implementation of the network as well as with the modifications are presented and discussed.

Handledare: Alexandros Papachristou
Ämnesgranskare: Anders Hast
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F 20006
Tryckt av: Uppsala

1 Populärvetenskaplig Sammanfattning

Artificiell intelligens, eller "AI", är en term som väcker både nyfikenhet och rädsla hos folk. AI används i allt från marknadsföring och chattbotar, till ansiktsigenkänning och robotar med mänskliga egenskaper. Den bakomliggande algoritmen för en AI kan egentligen vara allt från en simpel if-sats till stora komplexa neuronnät. För att lösa mer komplexa problem med AI så krävs sannolikt mer komplexa algoritmer. Detta medför dock också ett krav på mer datorkraft och tillgänglig data att träna AI:n på. De senaste 10 årens framsteg inom både hårdvara och forskning har möjligjort utvecklingen av komplicerade neruronnätsbaserade AI. Samlingsnamnet för den här typen av artificell intelligens är "deep learning" och syftar på de många lager av sammankopplade funktioner som utgör algoritmen.

Att representera föremål som en sammansättning av dess beståndsdelar är sannolikt något som vi människor gör undermedvetet varje dag. T.ex. när vi ser ett bord så ser vi detta som en kombination av dess ben och bordsskiva. På samma sätt ser vi troligtvis en kaffekopp som en cylinder med botten och ett handtag. Detta gör det möjligt för oss att identifiera och klassificera föremål, och som t.ex. i fallet med en kaffekopp, även hitta lämpliga sätt att interagera med dessa. Egentligen så får ju hjärnan bara signaler som har genererats av stimuleringen av tappar och stavar i ögonen, och måste sedan tolka dessa signaler för att komma fram till slutsatsen att det står en kaffekopp på ett bord. Denna översättning av signaler är inte trivial, utan är resultatet av en kombination av hjärnans komplexa nätverk av neuroner och mycket tid och data att träna på. Hjärnans struktur har på många sätt också inspirerat forskare i utvecklingen av neuronnät, och inte bara in namngivningen av dessa. Ett neuronnät är i grund och botten ett stort nät av sammankopplade noder (neuroner) som aktiveras i respons till en inkommande signal. Genom att träna på stora datamängder över många iterationer, och modifiera hur signalerna färdas genom nätverket, kan neuronnät tränas till en mängd olika arbetsuppgifter. Denna uppsats undersöker just ett neuronnäts förmåga att m.h.a ett set av enkla former (primitiver), kunna representera mer komplexa föremål som t.ex. flygplan och stolar.

Nätverket som används i detta projekt är baserat på en forskningsartikel av D. Paschalidou et al. [12]. En direkt implementation av detta forskarlags nätverk ger inte de förväntade resultaten, varpå en del justeringar och tillägg introduceras. Resultaten efter dessa ändringar antyder att neuronnät sannolikt är ett lämpligt verktyg för att abstrahera komplexa former till en sammansättning av enklare former, men att ytterligare justeringar av nätverket är nödvändiga.

2 Acknowledgements

I want to thank Dan Song and Alexandros Papachristou at Gleechi AB for giving me the opportunity to do this master thesis, and everyone at the office for providing an awesome work environment. Special thanks to Alex for helping me with the details of the project. Thanks also to Anders Hast for helpful comments on the report.

Contents

3 Introduction

This thesis aims to answer the question whether deep neural networks can be used in the context of shape abstraction. More specifically, if it is possible to construct an unsupervised neural network that can learn to predict a sparse set of shape primitives that describe a more complex input shape. In the following sections a general introduction on the topic of shape abstraction and deep learning is provided, as well as a discussion on related work on the topic of primitive-based shape fitting.

3.1 General Introduction

Shape abstraction is the process of assigning a parsimonious description to complex shapes, while preserving the essence of the original shape. One way to do this is by representing the complex shape by a set of primitive shapes. The objective of this thesis is to create a deep neural network (based on the paper by D. Paschalidou et al. [12]) that predicts the shape and position of such a set of primitives to describe an arbitrary more complex input shape.

There are several potential application areas for primitive based shape abstraction. For instance, D. Paschalidou et al. [12] recognize that the network consistently predicts the same primitives for the same part of the input shape across an object category. That is, the same primitives are used for e.g. the wings of the planes, or the body. Thus object part discovery and segmentation can be achieved through the primitive-based shape abstraction. Furthermore, as proposed by S. Tulsiani et al. [21], the authors of a similar network, the method can also be used for image based abstraction and shape manipulation. Image based abstraction, meaning the ability of the network to infer 3D-shape abstractions from a simple 2D-inputs, could assist robots in creating good interactions (e.g. grasp planning) using a simple camera to provide input. Shape manipulation is also made possible through the ability of the network to assign unique primitives to distinct parts of the input object. Each point of the original input shape can be mapped to its closest primitive and assigned a local coordinate in the frame of the primitive. By rotating or translating the primitive coordinate systems and letting the points of the input shape transform in the same manner, the original shape can be modified. For example, tilting of the head of an animal, or adjusting the pose of any object in general.

Deep neural networks (and machine learning in general) have got a resurgence in popularity in recent partly due to the increase in computing power, but also because of scientific breakthroughs in training large neural networks. The application areas of deep neural networks range from image classification [20] and segmentation [2], to interactive voice bots [17] and natural language processing [19]. The nature of shape abstraction is related to the problem of object segmentation, and thus it is likely that deep learning is a viable approach. This paper investigates the applicability of deep neural networks in the context of primitive-based shape abstraction.

3.2 Primitive-Based Shape Fitting

Many different approaches have been investigated in the context of primitive-based shape fitting and neural networks. G. Sharma et al. [18] present a recurrent neural network that given a 2D or 3D input, outputs a program that generates the shape. It uses constructive solid geometry (CSG), i.e. a set of primitive shapes combined using boolean operators (e.g. union, subtraction, intersection) applied recursively to create a more complex shape. The model uses a recurrent neural network, since these are well suited for predicting sequences [10]. Typically, recurrent neural networks are used in natural language processing due to the sequential nature of language. In the case of the CSG-net, a recurrent neural network is suitable since the set of instructions are predicted in sequence.

A generative recurrent neural network for primitive shape fitting is proposed by C. Zou et al. [22]. The network predicts the shape of cuboids and a sequence of cuboids with different height, width and depth parameters are assembled to form the final shape.

L. Li et al. [8] propose a network that takes a 3D point cloud as input and predicts per-point properties of the point cloud. It outputs point-to-primitive membership (i.e. which primitive a point is mapped to), surface normal and which type of primitive the point belongs to. They allow for 4 typed of primitives: spheres, planes, cylinders and cones, and show that the model is able to fit point clouds accurately, even for tiny segments.

The primitive shapes used to for the shape abstraction in the network presented in this paper are superquadrics (see section 4.2 for theoretical details). Superquadrics have been used previously in

research due to their ability to represent several shapes with a small set of parameters. R. Pascoal et al. [13] utilize superquadrics for both segmentation and object shape fitting. The method does not include a neural network. Instead an objective function that accounts for "the shape of the object, the distance of range points and partial occlusions" is formulated and minimized.

4 Theoretical Background

An introduction to supervised and unsupervised machine learning is provided in the following sections, as well as an explanation of the primitives used for the shape abstraction (superquadrics) and how to sample the surface of superquadrics. Additionally, quaternions and spatial rotation is described briefly as well as how to achieve uniform sampling of triangular meshes. The optimization algorithm ADAM used to train the neural network is also presented.

4.1 Machine Learning

Machine learning can be divided into two main categories: supervised and unsupervised machine learning. The difference lies in how the algorithms are trained. Supervised training in the context of neural networks, involves having a set of correctly labeled outputs that the network is trying to predict from a given input. Image classification is a classic example where supervised training is typically used. The network will train on a set of labeled images and some loss function will punish the network for predicting the incorrect label to an image. An obvious downside of supervised training is the need for labeled input data. Generally, the more input data available, the better the training. However the process of labeling data is costly and time-consuming. There are some tricks to reduce the need for large training data sets. A common approach is to reuse the trained weights of a network trained on a large set of input, and only train the last couple of layers with the additional smaller available input data. Another workaround is to augment the existing labeled data (such as rotating or adding noise) to artificially increase the size of the training data.

Unsupervised training, on the other hand, does not involve any labeled training data. Although this eliminates the need for labeled input data, the challenge is to formulate a loss function that forces the network to learn to produce the desired output. D. J. Rezende et al. [16] show that it is possible to create an unsupervised network that accurately infer 3D representations from 2D observation. The network presented in this paper is also an unsupervised neural network, largely based on the paper by D. Paschalidou et al. [12], with some modifications.

4.2 Superquadrics

Ideally, the choice of primitives should allow for a variety of shapes while at the same time being described by only a few parameters. Superquadrics possess both of these qualities. Superquadrics are the spherical product of two superellipses, parameterized by two angles η and ω, given by the equation

$$\mathbf{r}(\eta,\omega) = \begin{bmatrix} \alpha_1 \cos^{\epsilon_1}\eta \, \cos^{\epsilon_2}\omega \\ \alpha_2 \cos^{\epsilon_1}\eta \, \sin^{\epsilon_2}\omega \\ \alpha_3 \sin^{\epsilon_1}\eta \end{bmatrix} \quad \begin{matrix} -\pi 2 \leq \eta \leq \pi 2 \\ -\pi \leq \omega \leq \pi, \end{matrix} \tag{1}$$

where α scales the superquadric in the three spatial dimensions and ϵ determines the degree of non-linearity of the equation (visually interpreted as the shape of the superquadric). By restricting the shape parameters ϵ to the range $0.1 - 1.9$ (to avoid numerical issues with backpropagation) superquadrics can take on the shapes of cubes, octahedrons or spheres in a continuous parameter space. Figure 1 shows the parameter space w.r.t. the shape paremeters ϵ. The network will learn the shape parameters ϵ_1 and ϵ_2 and the scaling parameters α_1, α_2 and α_3 as well as the transformation parameters t_1, t_2 and t_3 and the rotation parameters q_1, q_2, q_3, q_4. Additionally, an existence probability parameter γ will be learned to make the network able to choose whether a primitive should be included in the shape representation or not. The loss function includes punishing terms for superfluous primitives (e.g. overlapping primitives) which enables the network to assign a low probability of existence to primitives that does not improve the shape representation.

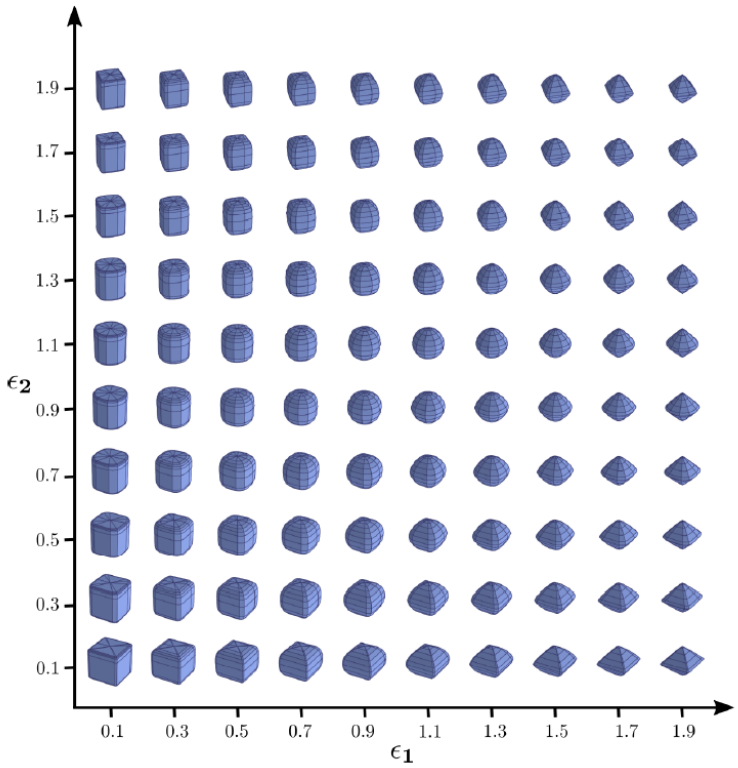


Figure 1: The parameter space of superquadrics w.r.t shape parameters ϵ_1 and ϵ_2. (Image borrowed from D. Paschalidou et al. [12])

4.3 Superquadric Sampling

The network will predict size (α) and shape (ϵ) parameters and the surface of the superquadrics has to be sampled to be used in the training of the network. However, achieving a uniform sampling of the surface is not trivial, due to the non-linear nature of the equation defining the superquadrics 1. The sampling becomes especially problematic when ϵ is small (e.g. 0.1). To overcome this issue the method for sampling superellipses proposed by M. Pilu et al. [15] is extended to three dimensions. A superellipse is defined by the equation

$$\mathbf{x}(\theta) = \begin{bmatrix} a_1 \cos(\theta)^\epsilon \\ a_2 \sin(\theta)^\epsilon \end{bmatrix} - \pi \leq \theta \leq \pi, \tag{2}$$

with $0 \leq \epsilon \leq 1$. The arclength between two close points of the superellipse $\mathbf{x}(\theta)$ and $\mathbf{x}(\theta + \Delta_\theta(\theta))$ can be approximated by the straight line

$$\mathbf{D}(\theta)^2 = |\mathbf{x}(\theta + \Delta_\theta(\theta)) - \mathbf{x}(\theta)|^2. \tag{3}$$

For small $\Delta_\theta(\theta)$ a first order approximation of the r.h.s. of 3 can be done

$$\mathbf{D}(\theta)^2 = \left(\frac{\partial}{\partial \theta} (a_1 \cos(\theta)^\epsilon) \Delta_\theta(\theta) \right)^2 + \left(\frac{\partial}{\partial \theta} (a_2 \sin(\theta)^\epsilon) \Delta_\theta(\theta) \right)^2. \tag{4}$$

Solving for the sampling angle $\Delta_\theta(\theta)$ in 4 yields

$$\Delta_\theta(\theta) = \frac{\mathbf{D}(\theta)}{\epsilon} \sqrt{\frac{\cos(\theta)^2 \sin(\theta)^2}{a_1^2 (\cos(\theta)^\epsilon)^2 \sin(\theta)^4 + a_2^2 (\sin(\theta)^\epsilon)^2 \cos(\theta)^4}}. \tag{5}$$

The set of angles achieve the uniform sampling is obtained by setting the approximate arclength $\mathbf{D}(\theta)$ in 5 to a constant and iterating

$$\theta_i = \theta_{i-1} + \Delta_\theta(\theta_i). \tag{6}$$

Since a superellipse is symmetric, the sampling only needs to be done from 0 to $\pi/2$ and the remaining points can be obtained by mirroring the coordinates of the sampled quadrant across the coordinate axes. To deal with the singularities in 5 occurring near $\theta = 0$ and $\theta = \pi/2$, M. Pilu et al. [15] make the simplification of 2

$$\mathbf{x}(\theta) = \begin{bmatrix} a_1 \\ a_2\theta^\epsilon \end{bmatrix}, \tag{7}$$

by approximating $\cos(\theta) \approx 1$ and $\sin(\theta) \approx \theta$ for small θ. The distance between two points becomes

$$\mathbf{D}(\theta) = y(\theta + \Delta_\theta(\theta)) - y(\theta) = a_2(\theta) + \Delta_\theta(\theta))^\epsilon - a_2\theta^\epsilon. \tag{8}$$

Again, solving for the sampling angle $\Delta_\theta(\theta)$ gives

$$\Delta_\theta(\theta) = \left(\frac{\mathbf{D}(\theta)}{a_2} + \theta^\epsilon\right)^{\frac{1}{\epsilon}} - \theta. \tag{9}$$

To sample near $\pi/2$, a_2 is substituted with a_1 in 9 and sampled for small θ.

The extension to three dimensions is achieved by first sampling η, setting $a_1 = 1$, $a_2 = \alpha_3$ and $\epsilon = \epsilon_1$, and then sampling ω with $a_1 = \alpha_1$, $a_2 = \alpha_2$ and $\epsilon = \epsilon_2$ according to Equations 5 and 9. The reason for this particular substitution of variables is that the superquadric (Eq. 1) is the spherical product of two superellipses with $a_1 = 1$, $a_2 = \alpha_3$ and $\epsilon = \epsilon_1$, and $a_1 = \alpha_1$, $a_2 = \alpha_2$ and $\epsilon = \epsilon_2$ respectively. Since the superquadric is symmetric around the coordinate axes, only one octant needs to be sampled and the remaining points are obtained by mirroring the coordinates of the first octant around the axes.

4.4 Quaternions and Rotation

Quaternions are a number system that extends the complex numbers, introduced by Hamilton in 1843. One application of quaternions is to represent spatial rotation in a compact way, while at the same time avoiding the gimbal lock [24] problem encountered when using Euler angles to represent rotation. D. Pavllo et al. [14] also recognize the advantage of using quaternions in their quaternion-based neural network aimed at modeling human motion.

To define a rotation around an arbitrary axis in 3D, the minimum amount of numbers required are four. Three numbers to define an axis of rotation and one number to define the amount of rotation around this axis. A quaternion consists of four elements in the form $a + bi + cj + dk$, where a, b, c, d are real numbers and i, j, k are the quaternion units. The quaternion representing spatial rotation about a unit axis $\overline{u} = (x, y, z)$ is given by

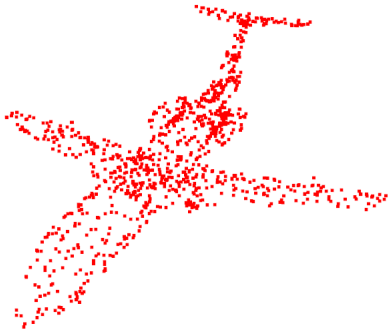
$$\overline{q} = \big(cos(\theta/2), xsin(\theta/2), ysin(\theta/2), zsin(\theta/2)\big), \tag{10}$$

where θ is the angle of rotation. See [1] for more details on quaternions and rotation. The network predicts the value of these four numbers (see section 5.1) and thus is able to predict any spatial rotation of the superquadrics.

4.5 Uniform Sampling of Input Mesh

During training, the network requires a point cloud sampled from the surface of the input shape. To accurately represent the input shape by a set of points, the sampling method of the input needs to achieve a uniform distribution of points on the input surface. The shapes provided by ShapeNet come in triangular meshes (in Wavefront .obj file format). The sampling is done by first choosing a triangle with a probability based on its area relative to the total area of all triangles of the mesh. Once a triangle is chosen, a point on the triangle needs to be sampled. Any point on the surface of a triangle p can be described by a convex combination of its vertices, i.e. $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$, where α are weighting factors that satisfy the equation $\alpha_1 + \alpha_2 + \alpha_3 = 1$. There are two degrees of freedom (since the weighting factors are dependent), and thus two numbers need to be randomly generated to specify a random point on the triangle. So α_1 and α_2 are randomly generated a value between 0 and 1. To ensure that the equality $\alpha_1 + \alpha_2 + \alpha_3 = 1$ holds, if $\alpha_1 + \alpha_2 > 1$, α_1 and α_2 are set to: $\alpha_1 = 1 - \alpha_1$ and $\alpha_2 = 1 - \alpha_2$. This procedure is then repeated N times to achieve a point cloud of N uniformly sampled points. The sampling method is based on an article by David de la Iglesia [5]. Figure 2 shows a rendered Wavefront .obj file and and a sampling of this input using the described sampling method with 1000 sampled points.

(a) Rendered input from a Wavefront .obj file.



(b) Uniform sampling of input mesh with 1000 points.

Figure 2: A comparison of input data (triangular mesh) and a uniform sampling of the input using 1000 points.

4.6 Training Neural Networks

To successfully train a neural network and within an acceptable timeframe, a number of techniques are used. Batch normalization and Rectified Linear Units, or ReLUs (explained below), are both part of the encoder of the network. Batch normalization [6] normalizes all the activations of a layer, both reducing the risk of saturating non-linear activation gates (like sigmoid gates) and removes the problem of layers constantly having to optimize for different distributions every iteration. ReLU gates are defined by $f(x) = \max(0, x)$, and are used in the encoder part of the network. One of the big advantages of ReLU gates over non-linear gates (such as sigmoid gates) is that they do not have a saturation region where the gradient dies during training. ReLUs are the most commonly used activation function today [9]. X. Glorot et al. [3] show the advantages of using ReLUs over other common activation gates. Standard ReLU gates can "die" when the input is less than zero since this results in a gradient of zero. However this network uses leaky ReLUs defined by

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{else,} \end{cases} \tag{11}$$

with $\alpha = 0.2$ which alleviates the problem of dead neurons.

The network is trained using batch gradient descent (i.e. using the averaged first order derivative of a subset of the input to find the local minimum of a function) with the ADAM optimization algorithm. The idea behind using batch gradient descent is to balance the advantages and problems with gradient descent using only a single data per iteration and using the whole training set. The advantage of using single data is that a lot less data needs to be stored on the device at any given time, but the problem is the noisy gradient it results in. The reason is that any single input data is likely a bad representation of the whole training set. By using the whole training set each iteration and calculating the average gradient, a very good approximation of the gradient is likely obtained. However this is infeasible due to the large amount of data having to be stored on the device at the same time. Batch gradient descent is the trade off, where mini-batches of e.g. 32 elements at the time is used to calculate the gradient each iteration.

The ADAM optimization algorithm [7] is the currently most favored optimization algorithm used in neural network training. It updates each parameter individually, taking the momentum of the gradient into account (i.e. the speed at which the gradient has changed in previous iterations) as well as ensuring reasonable initial stepsize. The advantage of using momentum is that the optimization is more robust to noisy data, since large fluctuations in the gradient has less impact on the stepsize and direction. Ensuring reasonable initial stepsize prevents the algorithm from stepping too far away from the local minima that is close to (a hopefully) reasonable initialization of weights. See Figure 3 for an example of the resulting set of superquadrics after the network is initialized, before any learning has taken place. The learning rate is set to 0.001 which is the same

that is used in the network that this paper is based on (D. Paschalidou et al. [12]). The other parameters (described in detail in [7]) are set to the recommended values, i.e. $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$.

5 Method

The objective of the network is to represent a 3D input shape by a parsimonious set of primitives. The choice of primitives and loss functions heavily determine the success of the network. The network is constrained to the primitive shapes it is able to predict and there is no correct solution (like e.g. labeled image classification), but only the loss functions to guide the training of the network. This network uses superquadrics as primitives and a set of loss functions that punish both inaccurate predictions of primitive position and shape, as well as the use of a superfluous number of primitives. The details of the implementation are described in the following sections.

5.1 Network

The network consists of an encoder that takes a 32x32x32 voxelized input data and, regressors that outputs the five parameters that uniquely defines a superquadric and the loss function. Using a voxelized input allows the network to use 3D convolutional layers in the encoder. Convolutional neural networks (CNNs) have proved to be more successful and computationally efficient compared to traditional neural networks with only fully connected layers [4].

The encoder takes the voxelized input through a series of layer combinations and outputs a single channel $1x64$ feature vector. The layer combinations are comprised of a 3D-convolutional layer with 1 voxel padding and a stride of 1, followed by a batch normalization layer and a leaky ReLU gate. The output of the ReLU gate is then fed into a max pool layer with a kernel size of $2x2x2$ with a stride of 2, resulting in a reduction in the length of each of the spatial dimensions by half. This sequence of layers is repeated five times until the one-dimensional feature vector is obtained. The feature vector is then passed through four fully connected layers and eventually fed into one of the five regressors that determines the final value of the predicted parameters. Each of the five regressors begin with a fully connected layer. The last layer of the size α, the shape ϵ and the probability of existence γ regressors are sigmoid gates, since the output of a sigmoid gate is between 0 and 1. A small constant is added to the sigmoid gate of the alpha regressor to avoid singularities and the output of the fully connected layer in the epsilon regressor is added and scaled to fit within the desired values of ϵ. The range used for ϵ in this paper is $0.4 - 1.1$. The probability of existence variable γ determines the probability of a primitive m being part of the shape abstraction according to the Bernoulli distribution $p(z_m) = \gamma_m^{z_m}(1 - \gamma_m)^{1-z_m}$ with the binary random variable $z_m \in 0, 1$. The translation regressor ends with a tanh-gate, that outputs a value between -1 and 1. The regressor that determines the value of the quaternion that gives the rotation of the superquadric only consists of the fully connected layer. Figure 4 shows an illustration of the network layout.

Proper initialization of the weights and biases of the network is important to allow for proper training. The weights and biases are initialized according to a gaussian distribution with mean and variance fine-tuned by trial and error through visual inspection of the initialized primitives. Figure 3 shows an example of the resulting set of primitives after initialization.

5.2 Input Data

The input data consists of both voxelized data with dimensions $32x32x32$ in Binvox format [11] that is fed to the encoder network, as well as a pointcloud sampled from the surface of the input data (see section 4.5) that is used in the loss functions. The data used in this paper is the airplane category from the ShapeNet [27] dataset. It consists of over 10000 different airplane models in Wavefront .obj file format. The files are parsed, voxelized and sampled. The data is then shuffled randomly and split into training, test and validation sets. The training set is the data used to train the network (i.e. the data from where mini-batches are fetched and fed to the network during training). The test set is used to evaluate the performance of the model in order to make decisions about how to change different hyperparameters (such as learning rate). The validation set is purely to test the performance of the model and must not be used in any way during the training or fine-tuning of the network.

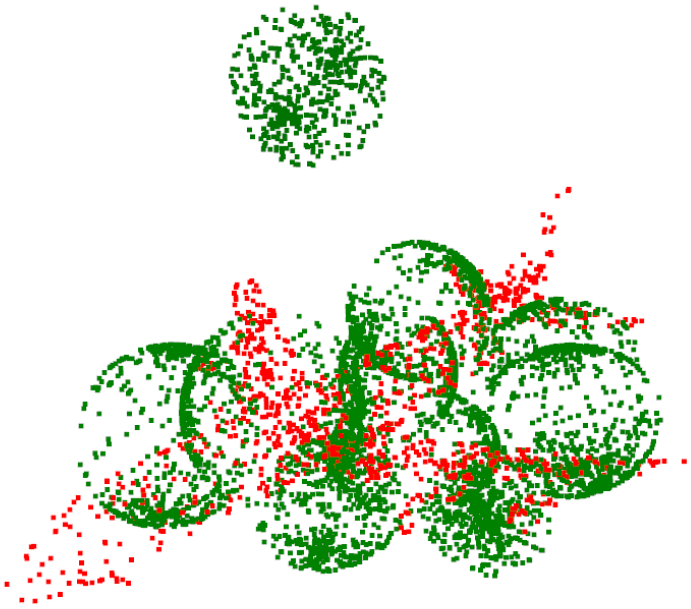


Figure 3: Set of superquadrics (green points) after initialization with maximum number of primitives $M = 10$ against input shape (red points).

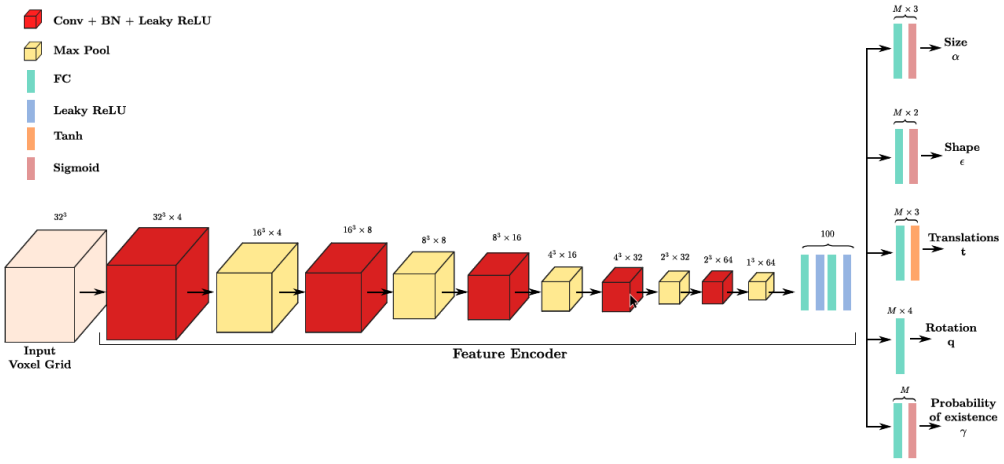


Figure 4: A visualization of the convolutional neural network (encoder) and the layers that predict the five parameters (regressors). (Image borrowed from D. Paschalidou et al. [12])

5.3 Loss Function

The function that the network will try to optimize, i.e. find the local minimum of through gradient descent, is called the loss function. It is a crucial part of designing an unsupervised neural network, since there is no right or wrong solution. Therefore if the loss function is poorly formulated, a small loss could be achieved but the results could be far from the desired outcome. The network proposed by D. Paschalidou et al. [12] uses three loss functions. The primitive-to-pointcloud loss forces the network to keep the surfaces of the predicted primitives close the surface of the input shape. The pointcloud-to-primitive loss ensures that every part of the surface of the input is described by at least one primitive. The parsimony loss both punishes the trivial solution of the the other two loss function (i.e. predict an existence probability of zero to all primitives) and also punishes the use of multiple primitives to reward the network to use a parsimonious representation of the input shape if possible. An additional loss function is proposed by the author of this paper with the objective to punish overlapping primitives, in the attempt to make each primitive represent a unique part of the input shape.

5.3.1 Primitive-to-Pointcloud Loss

The primitive-to-pointcloud loss ensures that the surface of the primitives (superquadrics) are close to the surface of the input shape. Given an input point cloud $\mathbf{X} = \{x_i\}_{i=1}^{N}$, the shortest distance between every point k on superquadric m, $\mathbf{Y}_m = \{y_k^m\}_{k=1}^{K}$, to the point cloud is calculated, i.e.

$$\Delta_k^m = \min_{i=1,..,N} ||x_i - \mathcal{T}_m(y_k^m)||_2, \tag{12}$$

where $\mathcal{T}_m(\cdot)$ is the transformation defined by the rotation and translation of superquadric m. The shortest difference for every point on the superquadric is then averaged across all points of the superquadric to give the primitive-to-pointcloud loss for superquadric m

$$\mathcal{L}_{P\rightarrow X}^m(\mathbf{P}, \mathbf{X}) = \frac{1}{K}\sum_{k=1}^{K}\Delta_k^m \tag{13}$$

Since the existence variables $\mathbf{z}$ are assumed to be independent, i.e. $p(\mathbf{z}) = \prod_m p(z_m)$, the expected value of the total primitive-to-pointcloud loss for all primitives is given by

$$\mathcal{L}_{P\rightarrow X}(\mathbf{P}, \mathbf{X}) = E_{p(\mathbf{z})}\left[\sum_{m=1}^{M}\mathcal{L}_{P\rightarrow X}^m(\mathbf{P}, \mathbf{X})\right] \tag{14}$$

$$= \sum_{m=1}^{M}\gamma_m\mathcal{L}_{P\rightarrow X}^m(\mathbf{P}, \mathbf{X}) \tag{15}$$

5.3.2 Pointcloud-to-Primitive Loss

The pointcloud-to-primitive loss makes sure that every part of the input shape surface is represented by at least one primitive. Similarly to the calculation of the primitive-to-pointcloud loss, a minimal distance is calculated, but now instead from every point on the point cloud to the closest point on a primitive m

$$\Delta_i^m = \min_{k=1,..,K} ||x_i - \mathcal{T}_m(y_k^m)||_2, \tag{16}$$

where again $\mathcal{T}_m(\cdot)$ is the transformation defined by the rotation and translation of superquadric m. The sum of the distance to the closest primitive that exists ($z_m = 1$) for all points in the pointcloud is calculated and the expected value is given by

$$\mathcal{L}_{X\rightarrow P}(\mathbf{X}, \mathbf{P}) = E_{p(\mathbf{z})}\left[\sum_{x_i\in\mathbf{X}}\min_{m|z_m=1}\Delta_i^m\right]. \tag{17}$$

The problem is that the time complexity of naively evaluating the expression inside the brackets is exponential with M, i.e. 2^M, since every additional primitive doubles the amount of possible configurations of existing primitives. D. Paschalidou et al. [12] present a way to make the time

complexity linear in M instead. By sorting the minimal pointcloud-to-primitive distances Δ_i^m in ascending order so that all Δ_i^1, is the smallest and Δ_i^M is the largest an alternative formulation can be derived

$$\mathcal{L}_{X\to P}(\mathbf{X},\mathbf{P}) = \sum_{x_i\in\mathbf{X}}\sum_{m=1}^{M}\Delta_i^m\gamma_m\prod_{\tilde{m}=1}^{m-1}(1-\gamma_{\tilde{m}}), \tag{18}$$

where $\tilde{m}$ denotes primitives that are closer to the current point x_i than m and $\gamma_{\tilde{m}}$ is the existence probability of a closer primitive. For details of the derivation of the alternative formulation equation see the paper by D. Paschalidou et al. [12].

5.3.3 Parsimony Loss

The purpose of the parsimony loss is twofold. Firstly, it deals with the trivial solution that the primitive-to-pointcloud and pointcloud-to-primitive losses suffer from, i.e. the existence probability $\gamma = 0$ for all primitives. Secondly it aims to punish the use of superfluous primitives. The parsimony loss is defined as follows

$$\mathcal{L}_\gamma(\mathbf{P}) = \max\left(\alpha - \alpha\sum_{m-1}^{M}\gamma_m, 0\right) + \beta\sqrt{\sum_{m=1}^{M}\gamma_m}. \tag{19}$$

The left term punishes the network if the sum of the existence probabilities of all superquadrics is less than one. The right term punishes the use of more superquadrics, with the goal of preventing the use of e.g. almost identical primitives with dispersed existence probabilities between them. The idea is that neither the primitive-to-pointcloud loss, nor the pointcloud-to-primitive loss will punish this configuration, and part of the objective is to create a sparse representation of the input shape. The parameters α and β are set to 1 and 10^{-3} respectively as suggested by D. Paschalidou et al. [12].

5.3.4 Overlapping Loss

The fourth loss function is an extension proposed by the author of this paper. The idea is to add a penalty to primitives that overlap in order to achieve a final result where each primitive describes a unique part of the input shape. The overlapping loss function makes use of the implicit equation for superquadrics, also called the inside-outside function [25] (since a value less than one indicates that a given point is inside the superquadric and a value larger than one indicates that the point is outside the superquadric). The inside-outside function for superquadrics is

$$\left(\left(\frac{x}{\alpha_1}\right)^{\frac{2}{\epsilon_2}} + \left(\frac{y}{\alpha_2}\right)^{\frac{2}{\epsilon_2}}\right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{z}{\alpha_3}\right)^{\frac{2}{\epsilon_1}} = 1, \tag{20}$$

where x, y and z are the coordinates of an arbitrary point in space and α and ϵ are the size and shape parameters described in section 4.2. Additionally, a small number (10^{-6}) is added to the x, y and z variables to avoid singularities during backpropagation. The problem arises when any of these variables are exactly zero, since the $\frac{d}{da}a^b = a^b\log(a)$, and hence if $a = 0$ the gradient is undefined.

The algorithm loops through every superquadric. Every iteration, all the other superquadrics are rotated and translated to the coordinate system defined by the current iteration's superquadric. The inside-outside function of the current iteration's superquadric is then evaluated for all the points of the other transformed superquadrics. A flipped, shifted and scaled sigmoid gate is then applied to punish any point of a superquadric that is inside the superquadric of the current iteration (i.e. the value of the inside-outside function is less than 1). Using a sigmoid function ensures that the loss function is smooth and differentiable to allow for gradient flow. The modified sigmoid function is defined by

$$f(x) = \frac{-1}{1+e^{-20(x-1)}} + 1, \tag{21}$$

and is visualized in Figure 5. Moreover, to allow the network to take the existence probability of a particular superquadric into account, the loss is multiplied by both the existence probability of the

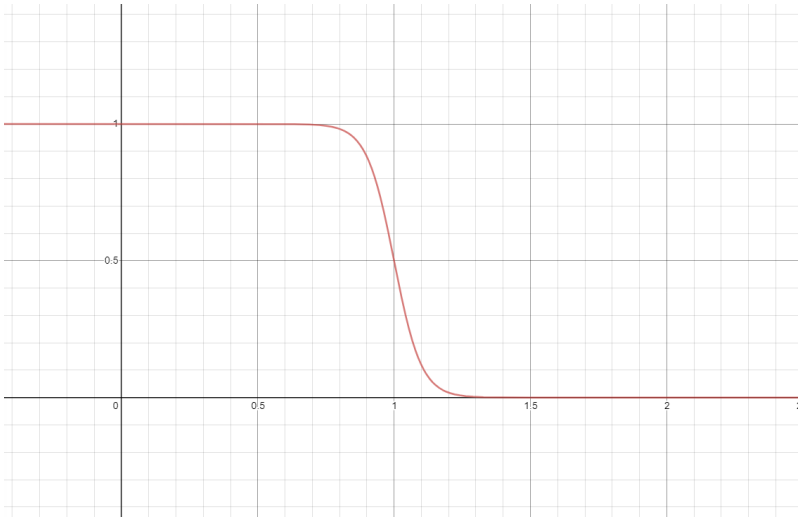


Figure 5: The modified sigmoid function that punishes values of the inside-outside function that are less than 1.

current iteration's superquadric and other transformed superquadrics. This allows the network to optimize with regards to the existence probability as well as ignore overlapping of superquadrics with a low probability of existence. Finally, to make the overlapping loss function invariant to the number of sampled points on the superquadrics, it is normalized by the number of sampled points. A weighting factor is added as a hyperparameter to scale the loss properly.

5.4 Modifications

A few modifications to the network presented in the paper are implemented due to unsatisfactory results. The first change is to make the pointcloud-to-primitive loss independent of number of sampled points on the input mesh N, by dividing the loss by N. There is no reason that the loss function should depend on the number of sampled points, and it causes the pointcloud-to-primitive loss to become several orders larger in magnitude compared to the primitive-to-pointcloud loss, depending on the number of points sampled. This in turn causes the pointcloud-to-primitive loss to dominate the gradient descent steps and results in the network converging on solutions where the primitives are not constrained to the surface of the input shape. This is likely a mistake in the paper, rather than the actual implementation by the researchers.

Another modification that is introduced is making the primitive-to-pointcloud loss independent of the existence probability variables γ. The reasoning behind this change is that there should never be a reason for the network to optimize for primitives that do not fit the input shape. Additionally, including γ in the primitive-to-pointcloud loss always punishes a high existence probability of the primitives, resulting in the network converging to solutions where the primitive representation is too sparse.

5.5 Software

The network was built using the Caffe2 [23] deep learning framework in Python. The main reason for choosing Caffe2 is that it is built on C++ and provides both a Python and a C++ API. This means that it is both fast and well suited for deployment, since you do not need to have python code in your final project compared to most other frameworks. Another advantage is the ability to construct custom operators which allows for more flexibility in the creation of neural networks.

Every operation (such as multiplication, addition, square root etc.) has a corresponding Caffe2 operator. Using these operators the network can be constructed, and through the special operator "AddGradientOperators" Caffe2 automatically adds the corresponding gradient operators that enables backward propagation (and thus training) of the network. This greatly simplifies the construction of neural networks since the programmer does not need to manually code the backward propagation.

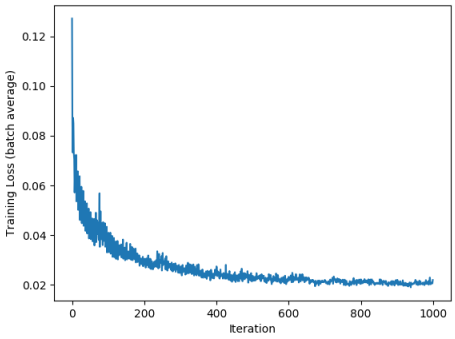
However, some issues were encountered when using Caffe2 to build the neural network. One main problem is that Caffe2 is deprecated because it is being merged with PyTorch, meaning that no development/bugfixing is currently being done and the documentation is left rather unfinished. Another obstacle with the framework is that in order for the backpropagation to work, every single tensor has to be uniquely named manually. The reason is that every tensor needs to be kept in memory for the gradient to flow through the whole network. But since this naming does not take place automatically (e.g. during loops), this makes constructing a complex network very tedious. Lastly a bug was encountered during backpropagation through two specific operators: the power operator and the square root operator. The gradient calculation through these operators requires several steps, which means that intermediate values have to be stored during the calculation. The problem arises when the same tensor is used as input to two or more of any of these operators, in which case the framework starts to autogenerate additional tensors to store these intermediate values. The framework will then incorrectly add the values of these tensors to the total gradient, which causes incorrect results. A workaround was implemented in which at any point where the same tensor was used as input to two or more of any of these operators, the tensor was first copied into new tensors with unique names, successfully avoiding the bug. However, the debugging of this particular issue was very time-consuming. In order to identify the issue, every operator had to be tested individually and the large Protobuf file [26] that defines the network had to be manually inspected.

6 Results

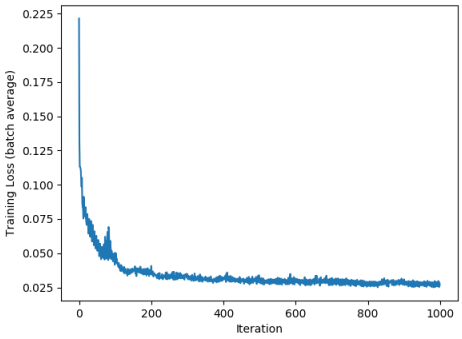
The network is able to predict shape, size, position and rotation of a set of superquadrics that resemble the input shape. The network is only trained on a small subset of the available training data (20 airplane or chair models in total). The reason is the inability to run the network on the GPU as well as performing other necessary optimizations to improve the speed of the training, which is due to the lacking documentation of the Caffe2 framework.

The network is trained with several different configurations of loss functions. All training is done on the same training set of 20 airplanes or chairs with a batch size of 2 for a total of 1000 iterations. The maximum number of primitives is first set to $M = 10$. Results from two random input shapes from the training set are presented. Figure 6a shows the average batch training loss against number of iterations when the pointcloud-to-primitive, the primitive-to-pointcloud and the parsimony loss were used (just like in the paper by D. Paschalidou et al. [12], except with the normalized pointcloud-to-primitive loss discussed in section 5.4). A visualization of the results is shown in 9, where the red dots represent the sampled input shape, and the superquadrics are shown as solid shapes. The solid shapes are created from the convex hulls of the sampled points of the superquadrics. Figure 6b shows the loss over iterations when gamma is not included in the primitive-to-pointcloud loss (see section 5.4) and 10 is the corresponding visualization. The network is run with the overlapping loss function added to the three original loss functions and the results are shown in Figures 7a and 11. Figures 7b, 12 and 13 show the results of using the non-gamma dependent primitive-to-pointcloud loss as well as the overlapping loss. The vanilla network (like the research paper D. Paschalidou et al. [12]) as well as the vanilla network with the modification that the primitive-to-pointcloud loss is made independent of gamma are also trained with the maximum number of primitives $M = 20$. The results of the vanilla network with $M = 20$ are shown in Figures 8a and 14, and the results when the primitive-to-pointcloud loss is made independent of gamma are shown in Figures 8b and 15 for airplane models and Figure 16 for chair models. Table 1 shows the average training loss of the last iteration of training compared to the test loss with all different configurations of the network.

Results on fitting the training data using the vanilla network when the primitive-to-pointcloud loss is independent of gamma are presented in Figure 17 for airplane models and Figure 18 for chair models.

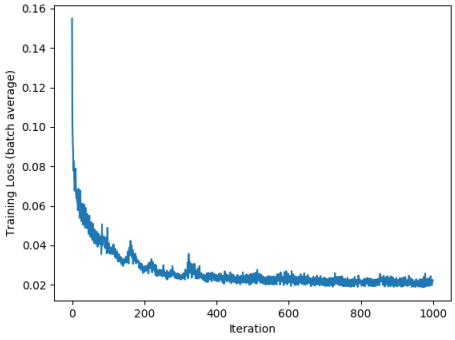


(a) Vanilla network

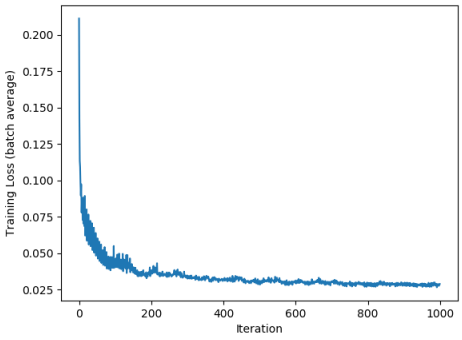


(b) Gamma-independent primitive-to-pointcloud loss

Figure 6: The average loss of the mini-batch per iteration, maximum number of primitives $M = 10$.

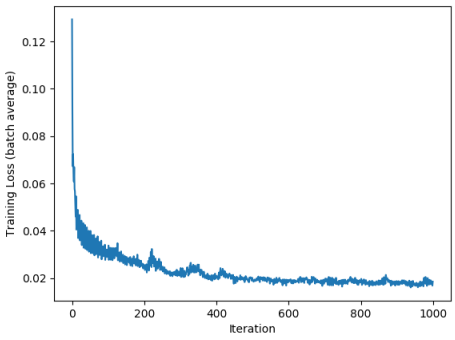


(a) Overlapping loss (with scaling factor 0.01)

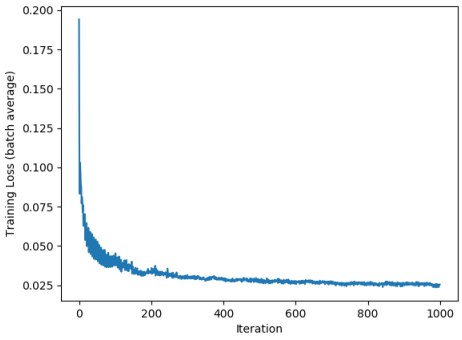


(b) Gamma-independent primitive-to-pointcloud loss and overlapping loss (with scaling factor 0.01)

Figure 7: The average loss of the mini-batch per iteration, maximum number of primitives $M = 10$.

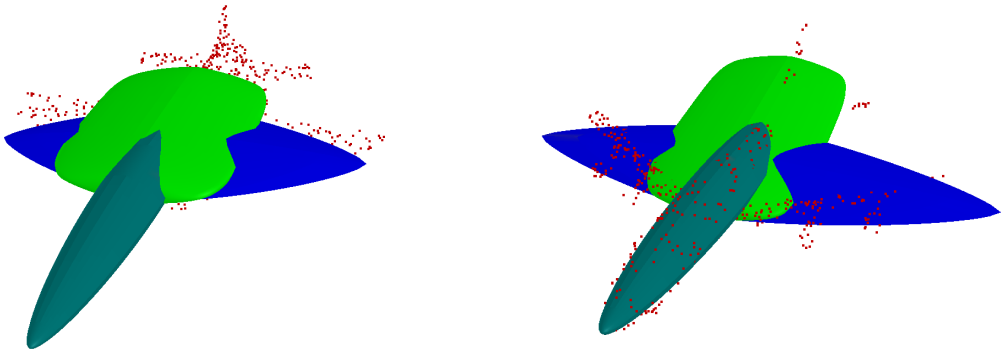


(a) Vanilla network



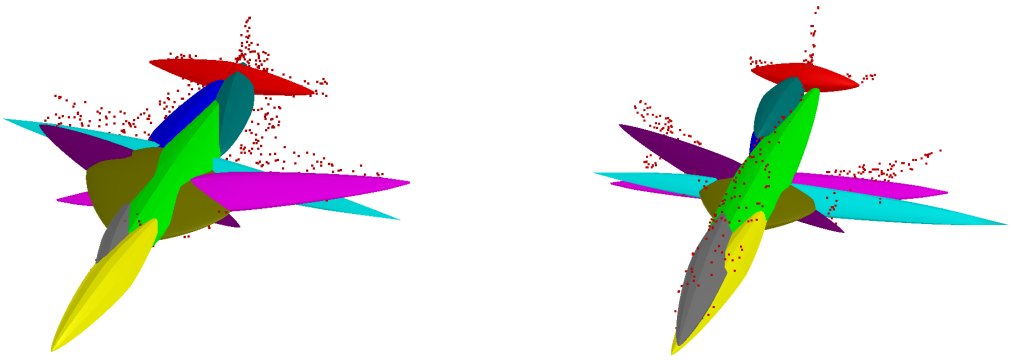
(b) Gamma-independent primitive-to-pointcloud loss

Figure 8: The average loss of the mini-batch per iteration, maximum number of primitives $M = 20$.



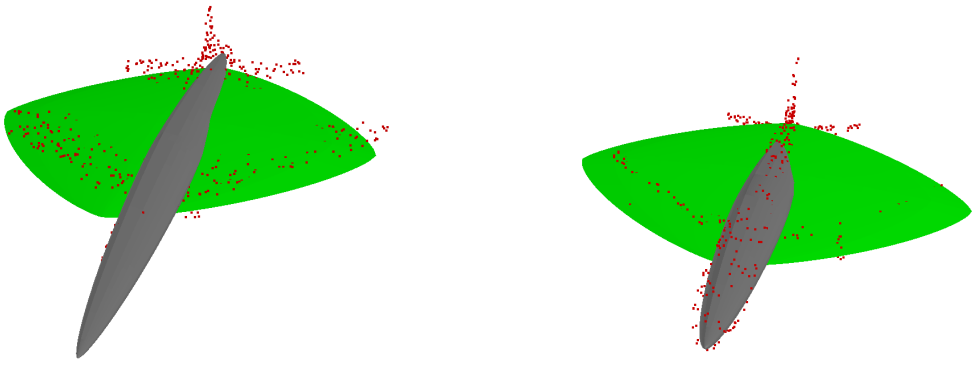
(a) (b)

Figure 9: Visualization of results with the vanilla network, maximum number of primitives $M = 10$.



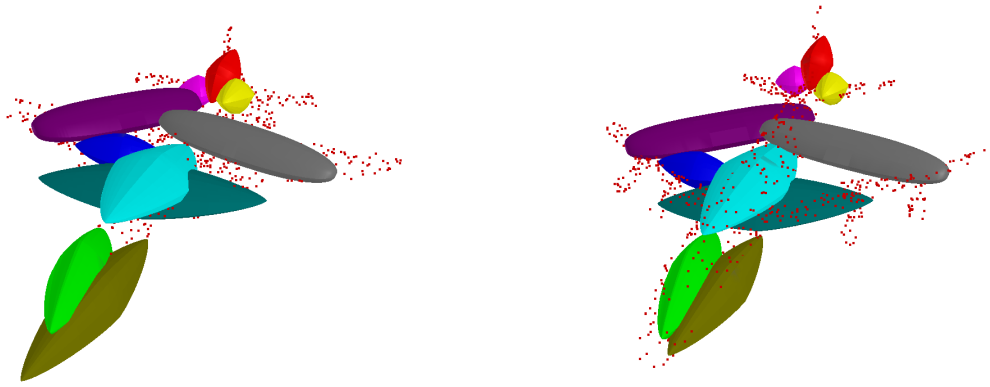
(a) (b)

Figure 10: Visualization of results with the vanilla network but no gamma dependent primitive-to-pointcloud loss, maximum number of primitives $M = 10$.



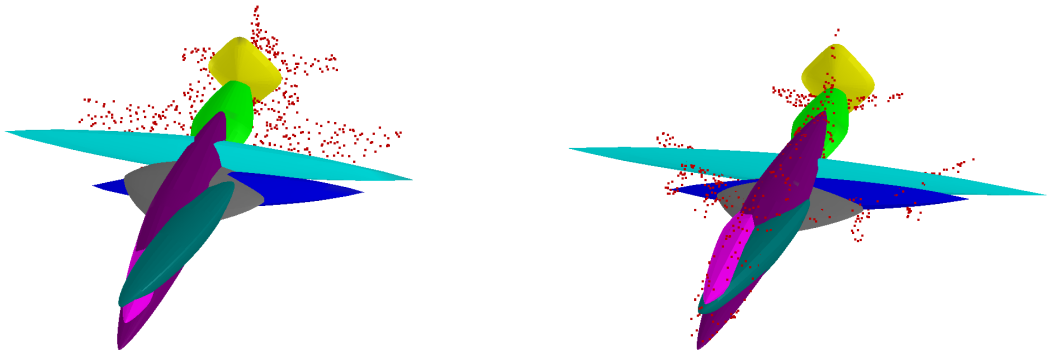
(a) (b)

Figure 11: Visualization of results with the addition of the overlapping loss (with scaling factor 0.01), maximum number of primitives $M = 10$.



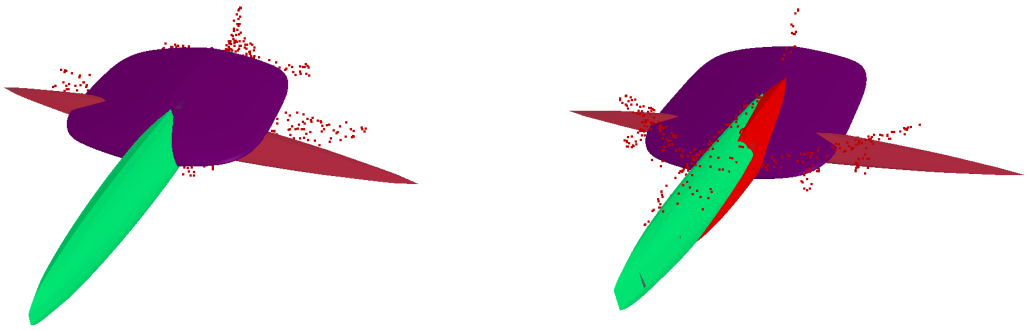
(a) (b)

Figure 12: Visualization of results with gamma independent pointcloud-to-primitive loss and the addition of the overlapping loss (with scaling factor 0.1), maximum number of primitives $M = 10$.



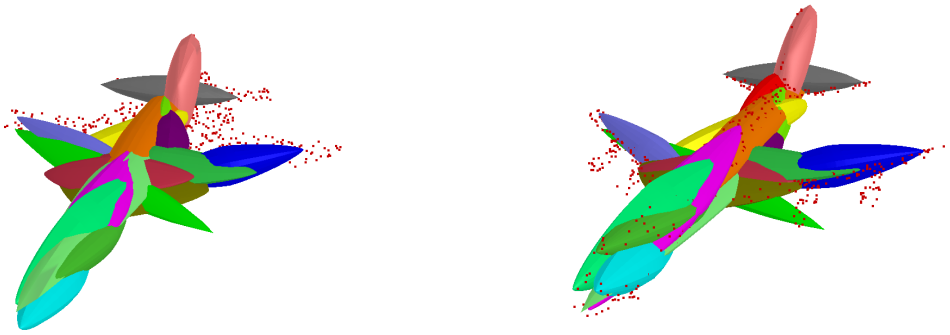
(a) (b)

Figure 13: Visualization of results with gamma independent pointcloud-to-primitive loss and the addition of the overlapping loss (with scaling factor 0.01), maximum number of primitives $M = 10$.



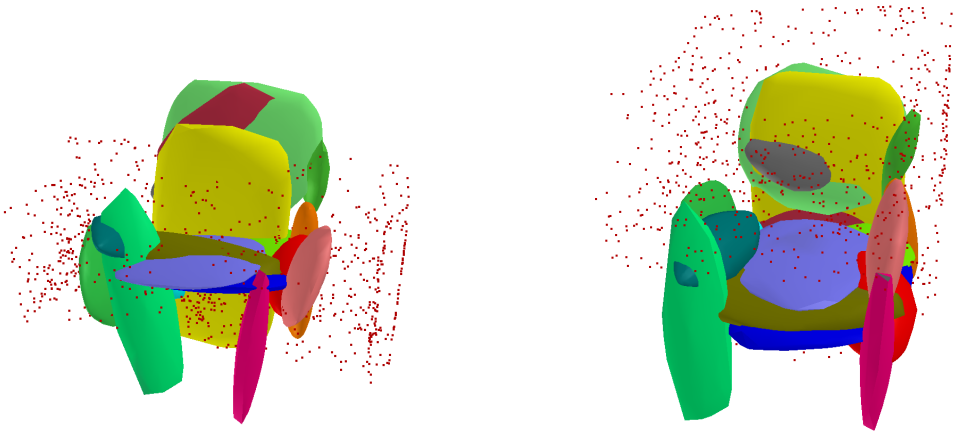
(a) (b)

Figure 14: Visualization of results with the vanilla network, maximum number of primitives $M = 20$.



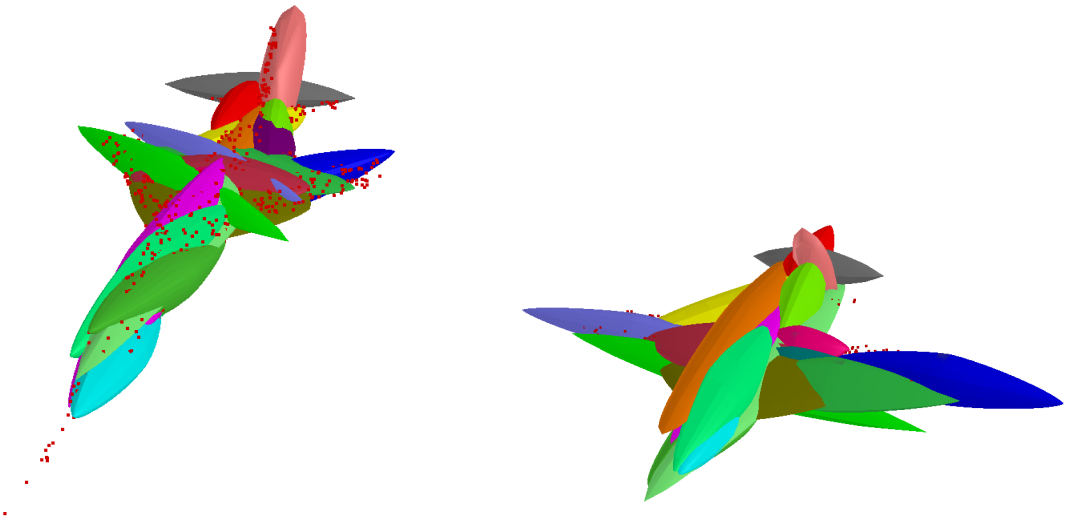
(a) (b)

Figure 15: Visualization of results with the vanilla network with gamma independent primitive-to-pointcloud loss, maximum number of primitives $M = 20$ (airplanes).



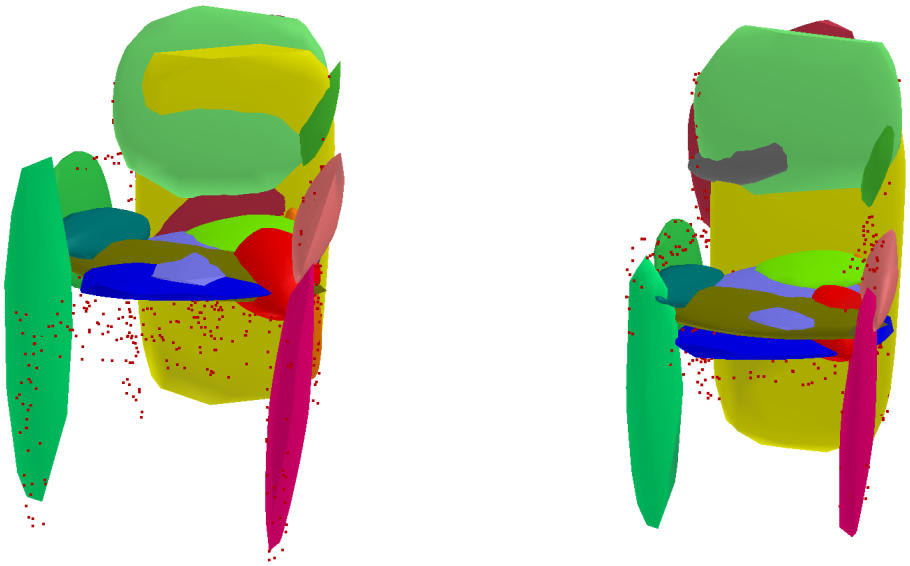
(a) (b)

Figure 16: Visualization of results with the vanilla network with gamma independent primitive-to-pointcloud loss, maximum number of primitives $M = 20$ (chairs).



(a) (b)

Figure 17: Visualization of fitting the training data with the vanilla network with gamma independent primitive-to-pointcloud loss, maximum number of primitives $M = 20$ (airplanes).



(a) (b)

Figure 18: Visualization of fitting the training data with the vanilla network with gamma independent primitive-to-pointcloud loss, maximum number of primitives $M = 20$ (chairs).

Network	Training loss	Test loss (a)	Test loss (b)
($M = 10$)	0.02194155	0.04550903	0.02346581
no γ prim-to-point ($M = 10$)	0.02691866	0.03276321	0.03369611
overlapping (0.01) ($M = 10$)	0.02215525	0.02111985	0.02142368
no γ prim-to-point, overlapping (0.1) ($M = 10$)	0.03185248	0.05870393	0.03449794
no γ prim-to-point, overlapping (0.01) ($M = 10$)	0.02884708	0.06579997	0.03238992
($M = 20$)	0.01854182	0.03630453	0.02013636
no γ prim-to-point ($M = 20$) (airplanes)	0.02560277	0.0560529	0.0275072
no γ prim-to-point ($M = 20$) (chairs)	0.05099021	0.1033478	0.12763424

Table 1: The average training loss of the last iteration of training compared to the test loss for objects (a) and (b) in Figures 9 - 16 (no network specification means the network is as described in the research paper, with the exception of the normalized pointcloud-to-primitive loss).

7 Discussion

The results show that the network is able to predict a set of primitives that resemble the shape of the input data. The network is trained on 20 input shapes and the results of 2 randomly chosen objects from the validation set are presented. However, the results are not comparable in quality to the paper that this network is based on [12], where in most cases each distinct part of e.g. the airplanes (wings, body, tail etc.) is described by only a single primitive. Additionally, a few significant alterations were introduced to give reasonably satisfactory results. An attempt to explain the discrepancy between the results of this implementation and the results presented by the researchers, as well as a comparison of the use of different loss functions are discussed. Additionally, the difference between the results on fitting the training data compared to the test data is analyzed.

The training loss over time (Figures 6, 7 and 8) behaves similarly for all different configurations of the network. A large decrease of the loss is seen initially and then the loss settles at a constant value. This indicates that the results could not be further improved by increasing the number of iterations.

As discussed in section 5.4, the pointcloud-to-primitive loss 5.3.2 is always normalized by the number of sampled points on the input shape. Figures 9 and 14 show the results of using the network as described by the researchers (except the normalization of the pointcloud-to-primitive loss) with maximum number of primitives $M = 10$ and $M = 20$ respectively. As evident, the set of primitives becomes too sparse (two and three repsectively) to accurately describe the input shapes. The reason for this is likely that the primitive-to-pointcloud loss 5.3.1 directly punishes the use of more superquadrics, due to the dependency on the existence probability variable γ. Results from running the network without the gamma-dependency of the primitive-to-pointcloud loss (excluding the multiplication with γ in Eq. 15) are presented in Figures 10 with $M = 10$, and 15 and 16 with $M = 20$ for airplanes and chairs. The results of this modification yields superior results in terms of representing the input shape. However, the set of primitives is not as sparse which goes against the second goal of the network: to create a parsimonious representation of the input shape. To address this issue of using too many primitives in the shape abstraction, some modification of the parsimony loss could be investigated.

Figures 12 and 13 show the results of using the overlapping loss (see section 5.3.4 for details) in addition to the modified primitive-to-pointcloud loss. The objective is to decrease the amount of overlapping of primitives and hence make the network converge towards solutions where each primitive represents a unique part of the input shape. However, with too high scaling of the overlapping loss the results are unsatisfactory (Figure 12) and too low scaling causes little effect (Figure 13). The inability to find a good scaling of the overlapping loss may be due to the shape of the modified sigmoid function (Figure 5). The steepness of the function near 1 may cause problems during backpropagation of the network. However making the slope flatter would add a penalty to primitives that are not overlapping, defeating the purpose of the function.

The ability of the network to fit the training data is visualized in Figures 17 and 18 for airplanes and chairs respectively. Compared to the results on the test data for the same network configuration (Figures 15 and 16), the ability to fit the input cloud of the training data is better. Table 1 shows the average training loss of the last iteration of training compared to the test loss for objects (a) and (b) in Figures 9 - 16. Comparing the loss between different configurations of the network is

not useful, since the different loss functions and/or scaling of the loss functions may decrease the value of the loss without necessarily improving the shape abstraction. Only the network versions where the primitive-to-pointcloud loss is independent of gamma (since visually these give the best primitive fitting) will be considered in the evaluation of the test and training performance. It is clear from the difference between the training loss and the test loss that object (a) in the airplane category and both objects in the chair category are poorly predicted by the network. This can be confirmed by looking at the corresponding visualizations (Figures 10, 12, 13, 15 and 16). The reason for this is likely that the data set used for the training is small. The network will have trouble to fit any test data that diverges from the training data, and it is likely that the small training set is not representative of the whole dataset. For example, the two input shapes from the test data of the chair category (red point clouds in Figure 16) do not look like generic chairs with four legs, a seat and a backrest. However, this is likely the shape of the majority of the training data, as evident from the visualization of fitting the training data shown in Figure 18. It is also possible that over-training of the network contributed to poor test results in some of the cases. More training with different number of iterations and on larger data sets would be needed to determine the cause of the problem.

8 Conclusion

The network is able to predict primitives that resemble the shape of the input data (Figure 15), as long as the input data does not diverge significantly from the training data. However, this is only achieved after modifying (see section 5.4) the network proposed by D. Paschalidou et al. [12] and the resulting set of primitives is not parsimonious. An attempt to rectify the superfluous prediction of primitives is introduced by the overlapping loss function (section 5.3.4), but the results are not improved (Figures 12 and 13). The reason for the discrepancy between the results by D. Paschalidou et al. [12] and the results presented in this paper is unclear. Nonetheless, the ability for the network to predict a configuration of primitives that resemble an input shape shows that the approach of using deep neural network for shape abstraction is viable. Given correctly tuned and scaled loss functions, it is likely that also the second goal of predicting a sparse set of primitives is achievable. This would potentially allow for many of the possible application areas, such as object part identification, shape manipulation and 3D-scene parsing etc.. And equally as important, due to the unsupervised nature of the network, without the need for large and costly human-labeled datasets.

References

[1] M. Ben-Ari (2014-2017) *A Tutorial on Euler Angles and Quaternions*, Department of Science Teaching, Weizmann Institute of Science, https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/softwareAndLearningMaterials/quaternion-tutorial-2-0-1.pdf

[2] S. Ghosh, N. Das, I. Das, U. Maulik, (2019), *Understanding Deep Learning Techniques for Image Segmentation*, CoRR, https://arxiv.org/abs/1907.06119

[3] X. Glorot, A. Bordes, Y. Bengio, (2010), *Deep Sparse Rectifier Neural Networks*, Journal of Machine Learning Research, 15, https://www.researchgate.net/publication/215616967_Deep_Sparse_Rectifier_Neural_Networks

[4] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang (2015) *Recent Advances in Convolutional Neural Networks* CoRR https://arxiv.org/abs/1512.07108

[5] David de la Iglesia. https://medium.com/@daviddelaiglesiacastro/3d-point-cloud-generation-from-3d-triangular-mesh-bbb602ecf238

[6] Sergey Ioffe, Christian Szegedy (2015), *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. CoRR https://arxiv.org/abs/1502.03167

[7] Diederik P. Kingma, Jimmy Ba (2014), *Adam: A Method for Stochastic Optimization*. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015 https://arxiv.org/abs/1502.03167

[8] L. Li, M. Sung, A. Dubrovina, L. Yi, L. Guibas, (2018), *Supervised Fitting of Geometric Primitives to 3D Point Clouds*, CoRR, https://arxiv.org/abs/1811.089880

[9] F. Li, J. Johnson, S. Yeung, (2019), *CS231n: Convolutional Neural Networks for Visual Recognition*, Stanford University, http://cs231n.stanford.edu/

[10] Z. C. Lipton, J. Berkowitz, C. Elkan, (2015), *A Critical Review of Recurrent Neural Networks for Sequence Learning*, CoRR, https://arxiv.org/abs/1506.00019

[11] Patrick Min https://www.patrickmin.com/binvox/ based on Fakir S. Nooruddin and Greg Turk (2003), *Simplification and Repair of Polygonal Models Using Volumetric Techniques*. IEEE Transactions on Visualization and Computer Graphics, Vol. 9, No. 2

[12] Paschalidou, Despoina and Ulusoy, Ali Osman and Geiger, Andreas (2019). *Superquadrics Revisited: Learning 3D Shape Parsing beyond Cuboids*. Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). https://arxiv.org/abs/1904.09970

[13] R. Pascoal, V. Santos, C. Premebida, U. Nunes, (2014), *Simultaneous Segmentation and Superquadrics Fitting in Laser-Range Data*, IEEE, https://ieeexplore.ieee.org/document/6810150

[14] D. Pavllo, C. Feichtenhofer, M. Auli, D. Grangier, (2019) *Modeling Human Motion with Quaternion-based Neural Networks*, CoRR https://arxiv.org/abs/1901.07677

[15] Maurizio Pilu, Robert B. Fisher (1995), *Equal-Distance Sampling of Superellipse Models*. BMVC https://pdfs.semanticscholar.org/3e6f/f812b392f9eb70915b3c16e7bfbd57df379d.pdf?_ga=2.238817344.1337014346.1578930007-684869428.1578930007

[16] D. J. Rezende, S. M. A. Eslami, S. Mohamed, P. Battaglia, M. Jaderberg, N. Heess, (2016), *Unsupervised Learning of 3D Structure from Images*, CoRR, https://arxiv.org/abs/1607.00662

[17] I. V. Serban, C. Sankar, M. Germain, S. Zhang, Z. Lin, S. Subramanian, T. Kim, M. Pieper, S. Chandar, N. R. Ke, S. Rajeshwar, A. Brebisson, J. M. R. Sotelo, D. Suhubdy, V. Michalski, A. Nguyen, J. Pineau, Y. Bengio, (2017), *A Deep Reinforcement Learning Chatbot*, CoRR, https://arxiv.org/abs/1709.02349

[18] G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, S. Maji, (2018), *CSGNet: Neural Shape Parser for Constructive Solid Geometry*, CoRR, https://arxiv.org/abs/1712.082900

[19] R. Socher, (2017), *CS224d: Deep Learning for Natural Language Processing*, Stanford University, https://cs224d.stanford.edu/

[20] F. Sultana, A. Sufian, P. Dutta, (2019), *Advancements in Image Classification using Convolutional Neural Network*, CoRR, https://arxiv.org/abs/1905.03288

[21] S. Tulsiani, H. Su , L. J. Guibas, A. A. Efros, J. Malik (2016), *Learning Shape Abstractions by Assembling Volumetric Primitives*. CoRR https://arxiv.org/abs/1612.00404

[22] C. Zou, E. Yumer, J. Yang, D. Ceylan, D. Hoiem, (2017), *3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks*, CoRR https://arxiv.org/abs/1708.01648

[23] Caffe2 https://caffe2.ai/

[24] Gimbal lock http://www.chrobotics.com/library/understanding-euler-angles

[25] Implicit equation superquadrics https://cse.buffalo.edu/~jryde/cse673/files/superquadrics.pdf

[26] Protocol buffers https://developers.google.com/protocol-buffers

[27] Shapenet https://www.shapenet.org/