Monocular 3D Reconstruction of Human Heads

Björn Bastian Welker

September 8, 2022

Version: CleanThesis 0.4.1 - JGU patches

Johannes Gutenberg Universität Mainz

FB08
Institute of Computer Science
Research Group

Bachelor Thesis

Monocular 3D Reconstruction of Human Heads

Björn Bastian Welker

1. Reviewer Prof. Dr. Ulrich Schwanecke
Institute of Computer Science - Working Group
Hochschule RheinMain

2. Reviewer Prof. Dr. Elmar Schömer
Institute of Computer Science - Working Group
Johannes Gutenberg Universität Mainz

Supervisors Prof. Dr. Ulrich Schwanecke and Prof. Dr. Elmar Schömer

September 8, 2022

Björn Bastian Welker
Monocular 3D Reconstruction of Human Heads
Bachelor Thesis, September 8, 2022
Reviewers: Prof. Dr. Ulrich Schwanecke and Prof. Dr. Elmar Schömer
Supervisors: Prof. Dr. Ulrich Schwanecke and Prof. Dr. Elmar Schömer

Johannes Gutenberg Universität Mainz
Research Group
Institute of Computer Science
FB08
Staudingerweg 9
55128 Mainz

Abstract

This thesis is concerned with the estimation of 3D models of human heads from single images. The approach of the thesis is based on [4] where 3D models of human bodies (including clothing) are estimated from single images. The approach is based on the use of certain neural networks. We describe the concepts from computer vision and machine learning mostly relevant to the thesis, provide a description of the architecture underlying our approach and give implementation details. In our conclusion we point to problems that were solved on the way and exhibit paths to future work.

Abstract (Deutsch)

In dieser Arbeit beschäftigen wir uns mit der Schätzung von 3D Modellen von menschlichen Köpfen basierend auf einem einzigen Bild des Kopfes. Unser Ansatz folgt [4]. Hier wird zusätzlich die Schätzung von 3D Modellen von menschlichen Köpern (mit Kleidung) auf einer Aufnahme betrachtet. Der Ansatz dort basiert auf gewissen neuronalen Netzen. In unserer Arbeit beschreiben wir die Konzepte aus Computer Vision und Machine Learning, die für den Ansatz der Arbeit am wichtigsten sind. Weiter beschreiben wir die dem Ansatz zugrunde liegende Architektur und beschreiben Details der Implementierung.

Zum Schluß beschreiben wir, wie wir Probleme bei der Implementierung gelöst haben, und zeigen auf, wie der Ansatz in Zukunft weiter verfolgt werden könnte.

Acknowledgement

I thank Prof. Dr. Ulrich Schwanecke for the excellent supervision of the thesis and for suggesting an application of [4] to human heads as a thesis topic. My thanks also go to Prof. Dr. Elmar Schömer for many helpful suggestions and for assuming the role of a co-supervisor. Finally, I thank the research group on visual computing of Mainz University for its hospitality and friendly atmosphere.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, September 8, 2022

Björn Bastian Welker

Contents

1 Introduction

> *I'm not suggesting that neural networks are easy. You need to be an expert to make these things work. But that expertise serves you across a broader spectrum of applications. In a sense, all of the effort that previously went into feature design now goes into architecture design and loss function design and optimization scheme design. The manual labor has been raised to a higher level of abstraction.*
>
> **— Stefano Soatto**

1.1 Postcards: My Address

Björn Welker
Zimmerplatzweg 10
35274 Kirchhain
Germany

1.2 Motivation and Problem Statement

Reconstructing a 3D object from a single RGB image seems to be an impossible task. Mathematically it is indeed an impossible task, since it requires to estimate an object described by a high-dimensional parameter space from data in a space whose parameter dimension is much smaller. Nevertheless, due to the fact that reasonable approximations of the 3D object already have numerous important practical applications there has been and is very active research in this direction. Especially application in the fields of virtual and artificial reality appear to be enticing. As a consequence, in computer vision and machine learning the problem of

estimating 3D geometries from a single 2D image only has been a hot research topic for some years. Much of this research was driven by the use of neural networks.

In this thesis we consider the restriction of the problem to estimating the human body or parts of it. This special case is indeed central in many of the above mentioned potential applications to virtual or artificial reality. Even in that more narrow field there have been many different [34, 35, 3, 10, 16, 19, 22, 25, 27, 37, 47, 5, 32, 45, 4] approaches which have tried and still try to solve this problem. In this thesis we will focus on a single approach based the work from [4].

In [4] a so called ”U-net” (see Section 3.7) is employed to estimate 3D models of humans wearing clothing from a single image. The results from the paper are quite stunning and provide convinging evidence for the power of the principal method. The idea behind this thesis is to adapt the approach from [4] to the to the estimation of human heads from single images.

For this goal we start from scratch and develop the software for our purpose either completely ourselves or adapt existing modules to our specific needs. As a main ingredient we use the ”PyTorch” framework [29]. Our software [40] can be found on GitHub under

https://github.com/br0wnbeer/3Dhumanheads.

In the next section we describe the structure of the thesis.

1.3 Thesis Structure

Chapter 1

After an introduction to the topic of the thesis in Section 1.2 in Chapter 1 we also give an overview of the structure and content of the thesis in Section 1.3.

Chapter 2

In Chapter 2 ”Related Work” we review some of the research on estimation of images from single, very few images or short video sequences. We confine ourselves to work done on images of human bodies and images of heads. For human bodies we mostly follow the review from [4], for human heads we rely on our own literature search.

Chapter 3

In Chapter 3 ”Concepts” we introduce the basic concepts from computer vision and machine learning that are applied in this thesis. On the side of computer vision we define signed distance functions in Section 3.1, image filters and convolution in Section 3.2, the block marching / marching squares algorithm in Section 3.3, octree sampling in Section 3.4 and sphere tracing in Section 3.5. On the side of machine learning we introduce convolutional neural networks in Section 3.6 and the U-net in Section 3.7.

Chapter 4

In Chapter 4 ”Data Set” we describe the data set which was supposed to be used for training the adaption of the methods from [4] to human heads. We explain the generation, the data format and the Blender API capable of reading the given format.

Chapter 5

In Chapter 5 ”Architecture” we define the principal setting and architecture of our approach. In Section 5.1 a rough global sketch of the approach is given. The components of the approach are then detailed in the following section. In Section 5.2 an extension of the signed distance function from Section 3.1 is defined, in Section 5.3 we explain the positional encoding, in Section 5.4 we say how the feature extraction is done and in Section 5.5 we explain the shading. In Section 5.6 we then also provide details on the loss functions used in [4].

Chapter 6

In Chapter 6 ”Implementation” we describe the details of our implementation of the architecture described in Chapter 5. In Section 6.1 the choice of the principle framework ”PyTorch” is explained and justified. Then in Section 6.2 the implemetnation of the network and its components Feature Extractor, SDF with Color Component, Shading Network and Positional is detailed. In Section 6.3 the specifics of the implementation of the sphere tracing are explained. Finally, in Section 6.4 we discuss the implementation of the losses.

Chapter 7

In Chapter 7 ”Conclusion” we draw a few conclusions from the experiences and results gained while working on this thesis and the corresponding code. We discuss the general approach as well as issues with the loss functions and the data set. For this we draw a picture of what remains to be done to a complete the adaption of the approach from [4] and talk about potential extensions.

Appendix A

We reproduce in Section A.1 an e-mail exchange [2] between us and one of the authors of [4].

2 Related Work

In the field of estimating 3D models of human bodies or parts of human bodies from images or videos there exist a wide range of approaches. In this thesis we follow the approach from [4] which performs the estimation based on a single image. In this chapter we compare this approach with preceding and parallel work.

The SMPL [22] and GHUM & GHUML [42] frameworks for example produce results on 3D model estimation as a byproduct of human pose estimation based on priors in statistical models of the human body. Their results fall short of the goals of [4], since clothing and hair are not included in their results. This goal is achieved in [3, 8] where images are produced from short video sequences using techniques from mathematical optimization. Taking advantage of newly developed neural net methods the approaches taken in [3, 6] are based on very few images.

Similarly head pose estimation based on single images and statistical models [11, 39] yield in an analogous fashion results directly pointing towards the goal of this thesis. Indeed, more recent work in this direction is even based on convolutions neuronal networks [5, 32, 45] just as the approach taken in this thesis.

Still the literature on estimating full human bodies from single images is far more rich then the one on estimating heads. The methods are plentiful and can be divided into several categories according to the chosen presentation of the shape. The approaches from [19, 37, 47] are voxel based and have a high memory consumption. Other presentations have shown to be more useful (see [10, 13, 16, 25, 27, 38, 34, 35]). These are based on visual hulls or moulded front and back depth maps of implicit function networks.

In the following table Firgure 2.1 whose entries are taken from [4] the authors compare their own framework PHORHUM with with PIFu from [34], PIFu-HD from [35], both using IFNs to produce pixel aligned feature, GeoPIFu [16] further developing this approach, Arch [18] and Arch++ from [17] both using pixel aligned feature vectors from statistical models.

In Table 2.1 by end-to-end training we mean that the whole network can be trained from input to output in one pass, systems with no check in that column often iteratively improve the output and hence cannot be trained in one pass. The

	end-to-end trainable	signed distances	returns color	returns albedo	returns shading	true surface normals	returns shading
PIFu			✓			✓	
PIFuHD						✓	✓
GeoPIFu						✓	
Arch			✓				
Arch++			✓				
PHORHUM	✓	✓	✓	✓	✓	✓	✓

Tab. 2.1.: Comparison of single image human body estimation methods (see [4, Table 1])

other concepts appearing in Table 2.1 will either be explained later, such as signed distances in Section 3.1 and Section 5.2 or are self explanatory such as color, albedo and shading. Note that in the table true surface normals are not checked when estimated surface normals are used.

3 Concepts

In this chapter we will introduce the reader to the fundamental concepts and algorithms used in the project. These include different methods, models and other objects from different areas in the fields of computer vision and deep learning as well as some of their mathematical aspects. We describe the principal structure of the concepts and algorithms and provide more detail on how they are used in our setting in Chapters 5 and 6. We start with the concepts from computer vision and then describe the ones from machine learning.

3.1 Signed Distance Function

Let $S \subseteq X$ be an object in a metric space X with metric d. In our applications X will always be the Euclidean 3-dimensional space and d the Euclidean metric. We assume that there is a well defined concept of a ”surface” of S, which we assume to behave almost like a not too wild 2-dimensional manifold. A ”Signed Distance Function”, also known as an SDF of S is a function $f : X \to \mathbb{R}$ that describes the distance of a point p in space to the surface of the object S. The sign of the output values is chosen so that it tells if p is inside or outside of the object. ”Inside” points will produce a negative sign, ”outside” points will produce a positive sign and points on the surface yield 0.

For example if S is the ball of radius r in 3-dimensional Euclidean space, centered around (0,0,0) X then:

$$f(p) = \sqrt{p_x^2 + p_y^2 + p_z^2} - r$$

is the signed distance function of the point $p = (p_x, p_y, p_z)$. Note that for this S the surface is the sphere or radius r. Clearly, in general an SDF will not be expressible by an analytic term but must be approximated by suitable methods.

3.2 Image Filter/ Image Kernel

In the field of image processing a kernel / image filter / convolution is a linear or non-linear operation on a field (e.g., array, tensor) of data usually performed locally. For us linear and Gaussian filters are the most relevant. Linear filters are defined by a so called mask/kernel which can be described by matrix of usually smaller size than the field of data, here an array, itself. The operation defined by the matrix is different from matrix operations such as addition or multiplication. Rather around an data element a matrix neighborhood of the same size as the mask is cut out. Then the target value is computed as the dot product of mask and neighborhood, i.e., the corresponding matrix entries are multiplied and then the products are then summed up. Thus the local neighbours of a data point are weighted by the elements of the kernel / mask. An example for this is described in Figure 3.1. Here the mask is the 3×3 matrix on the upper left multiplied by $\frac{1}{9}$. Then around the data value 39 another 3×3 matrix is cut out from the data and the final value is computed as

$$\frac{1}{9}(1 \cdot 61 + 1 \cdot 17 + 1 \cdot 13 + 1 \cdot 51 + (-1) \cdot 38 + 1 \cdot 8 + 1 \cdot 13 + 1 \cdot 13 + 1 \cdot 59) = 43.$$

In cases where the mask is not fully contained in the data array, we will use filler values to overcome this problem.

In convolutional neuronal nets exactly these filters will be learned as part of the optimization process.

Gaussian filters for array data use a convolution function which uses as a kernel the density function of a two dimensional normal probability distribution with suitably chosen mean and standard deviation. Since the density function is supported on all of $\mathbb{R}^2$ suitable windows have to be chosen for evaluation and in addition the function has to be approximated.

3.3 Marching Cubes

The ”Block Marching” or ”Marching Cubes” algorithm, first described in [23], is used to determine the surface of a 3D object. It is the 3D version of the ”Marching Squares” algorithm used to determine the contours in a 2D image. In our case the Block Marching algorithm will be used to construct a 3D mesh and its normals from a given SDF. The algorithm can be described the following way. We are given a signed distance function $f : \mathbb{R}^3 \to \mathbb{R}$ corresponding to an object S inside a bounding

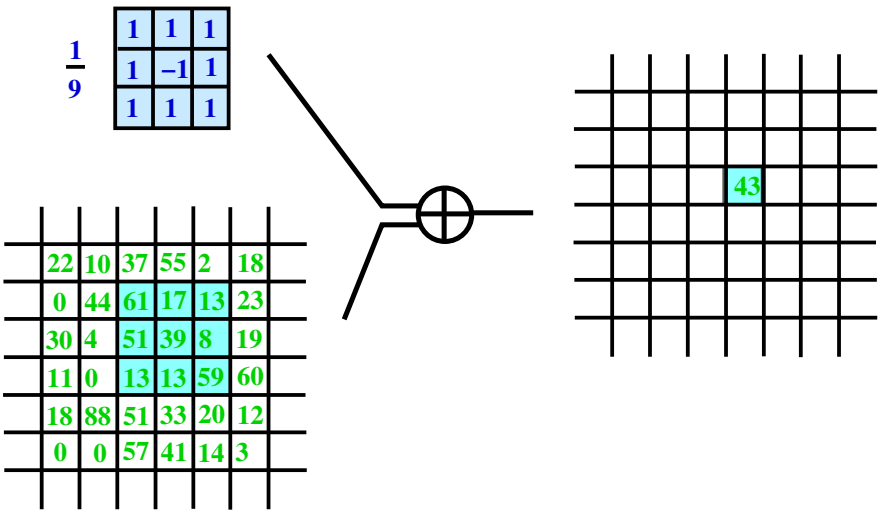


Fig. 3.1.: Example of a Filter (Linear Filter)

box in $\mathbb{R}^3$. This bounding box then will be subdivided into an $M \times M \times M$ cubical grid for some chosen parameter M. In general, the algorithm will determine a triangulated surface which approximates the isosurface $\{x \in \mathbb{R}^d \mid f(x) = c\}$ for a certain c. In our setting we are interested in determining the surface of a given object S given its SDF f. In order words we want to determine the isosurface for $c = 0$. Indeed after a possible translation of the SDF this is the generic case of the algorithm. The basis idea behind the algorithm is that if two points have an SDF value with different sign then the surface must intersect the ray between the two points. The algorithm passes through the cubes of the grid and assigns the sign of the SDF to each vertex of the cube. Since there are 8 vertices there are 2^8 possible configurations of sign patterns. If all signs are positive or negative then it is assumed that the surface does not intersect this cube and nothing has to be done. If there is a sign change along an edge then as mentioned before the surface must intersect the edge. The actual point of intersection is then approximated using numerical methods. Then according to the sign pattern of the vertices of the cube a triangular mesh approximating the intersection of the surface inside the cube is calculated. Note that if the original mesh is too coarse then a intersection of the surface with the cube having two connected components may not be detected at all or taken for a single intersection (see Section 3.4 for a method trying to solve this issues and an example for the situation in Figure 3.4).

Due to the high number of cases and the complexity of visualization we will not illustrate the Marching Blocks algorithm here. Indeed the number of cases can be reduced by using the symmetry group of the cube but the visualization problem remains. For that reason, we do the visualization for the little sibling of the Block Marching algorithm: the Marching Squares algorithm. It proceeds through the squares of an $M \times M$ grid in the same way the Marching Blocks algorithm does for the cubes in the $M \times M \times M$ grid. As the square has only 4 vertices there are only $2^4 = 16$ cases (not taking advantage of symmetry) to consider and the triangulation of a cube indeed can be replaced by a single edge – the latter in no way reflects the complexity faced in 3D. In Figure 3.2 the different cases are listed. The vertices of the squares are labeled with the sign of the SDF and the green edge is the estimated contour line. One immediately sees that there are many symmetries within the images and indeed both in 2D and 3D these symmetries are usually employed in implementations to reduce the computational complexity. We can see that there are only 4 general cases in the block marching algorithm that can be rotated to create all possible configurations of the lines. All of these can be seen in the second line of Figure 3.2.

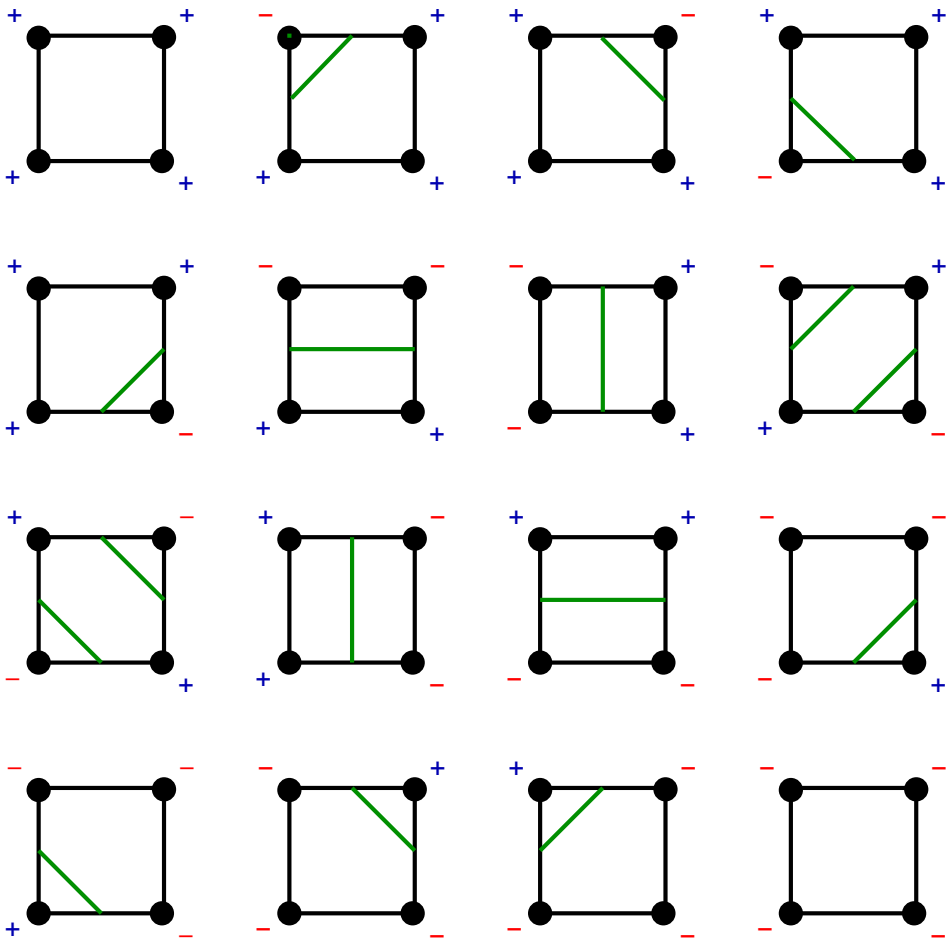


Fig. 3.2.: Cases for the Marching Squares algorithm

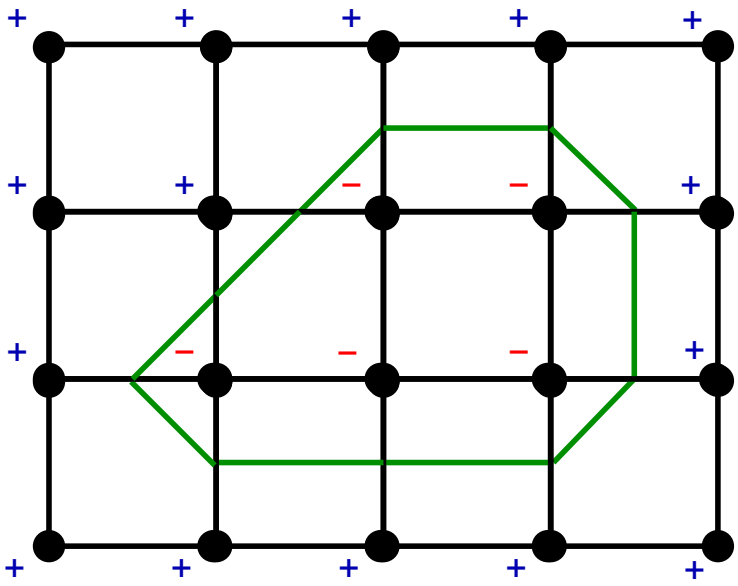


Fig. 3.3.: Simple example for the Marching Squares algorithm

In Figure 3.3 a very simple example is given which shows how the contour of a 2D object is estimated by the Marching Squares procedure. As in Figure 3.2 the signs provide the sign values of the vertices and the green line is the estimated contour.

The Block Marching algorithm uses the same type of procedure but in 3D, the bounding box as well as the objects in the lookup table will now be 3D. With this method a 3D mesh can be reconstructed from a given SDF.

3.4 Octree Sampling

Octree sampling solves problems that are inherent to the Marching Squares and Block Marching algorithms. The main issue which comes along when applying these algorithms is that there cannot be a very detailed approximation of the surface of a given SDF if one uses the same procedure as described in Section 3.3. This is due to the fact that the weight of every grid value is equal. Octree Sampling, first proposed in [41], provides a remedy for this issue. For example, consider an $M \times M \times M$ grid of cubes for a not too high number M. If it notices that there is

a sign change of the SDF among the vertices of a cube the procedure subdivides that block and recursively applies the algorithm to the subdivision. The recursion terminates once an a priory set level of precision is achieved. As a subdivision operator Octree uses the subdivision of a cube into 8 cubes arising by bisecting the cube parallel to the 3 coordinate planes. Its recursive structure and the fact that ”octa” is the (ancient) greek word for eight also explains the name of the algorithm. The following illustration Figure 3.4 is a visualization of the described algorithm. Consider the top cube as one of the cubes from the initial coarse grid. Assume that the SDF is not of constant sign on the 8 vertices of the cube – a certificate that the surface intersects the cube in its interior. The algorithm then subdivides the cube into 8 cubes depicted in the green figure below the initial cube. Now assume that there are sign changes of the SDF only on the vertices of the cube on the front side in the upper left corner and on the vertices of the cube on the backside in the lower right corner. Then these two cubes are subdivided each into 8 brown cubes depicted in the lowest cube. If now the precision threshold is reached the algorithm terminates. Note that the sign change in the first unsubdivided cube just suggested that the surface passes through the interior of the cube, the last image indeed suggests that the surface passes through the cube in two connected components one for each of the 8 brown cubes subdividing a green cube. On the right side of the image the tree structure is shown. The root node corresponds to the initial cube, the 8 child nodes are in correspondence with the 8 green cubes. The two green nodes corresponding to the green cubes with sign change then again have 8 children corresponding to the 8 brown cubes they are subdivided into.

3.5 Ray Marching

Sphere Tracing is a more advanced version of ray tracing, it uses spheres to find the intersection point of a given ray with an object S. We assume that S is described by an SDF f. In ray tracing and in Sphere Tracing we are given a ray from a point p pointing in a certain direction and we would like to determine the intersection point of the ray with the object S. In simple ray tracing the ray is probed (i.e., the SDF is evaluated) at constant intervals. This implies that for any given ray every point at a given step width is tested. For this procedure only the sign of the SDF at the probed points is important, the actual value is discarded. Moving along the ray in this fashion can take a really long time and can add a lot to the computational

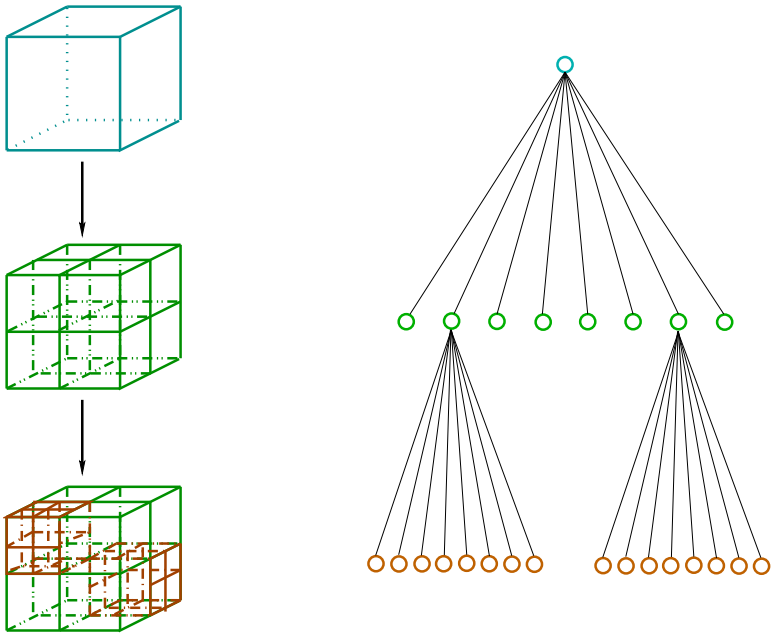


Fig. 3.4.: Octree Sampling

complexity of the used algorithm. On the other hand the value of the SDF at a point of the ray provides you with information about the maximal "safe" step you can take. To create a faster way to move along a ray and to see when it hits the given object S "Sphere Tracing" also known as "Ray Marching" was introduced in [14] taking advantage of the values of the SDF f.

Assume $p_0 = p \notin S$ is the initial point of the ray. The value $f(p_0)$ by definition provides you with the largest distance you can travel from p_0 in any direction without hitting the object. Now travel distance $f(p_0)$ on the ray to obtain a point p_1. Inductively, assume we have constructed the point p_i. We stop if $f(p_i) = 0$ – this value is reached unless S is unfavorably curved at the point of intersection in which case one has to work with threshold values. If $f(p_i) \neq 0$ then again travel distance $f(p_i)$ on the ray towards the surface to obtain point p_{i+1}.

In Figure 3.5 we provide an example of the procedure. Let $p_0 = p$. The interior of the circle C_0 of radius $f(p_0)$ is the safe region with no point from S. We proceed to the point p_1 (not labeled) which defines the circle C_1 whose interior is the safe region when starting in p_1. We continue inductively and the circle C_4 touches S at its intersection point with S and hence $p_5 = q$ is the desired intersection point of the ray with S.

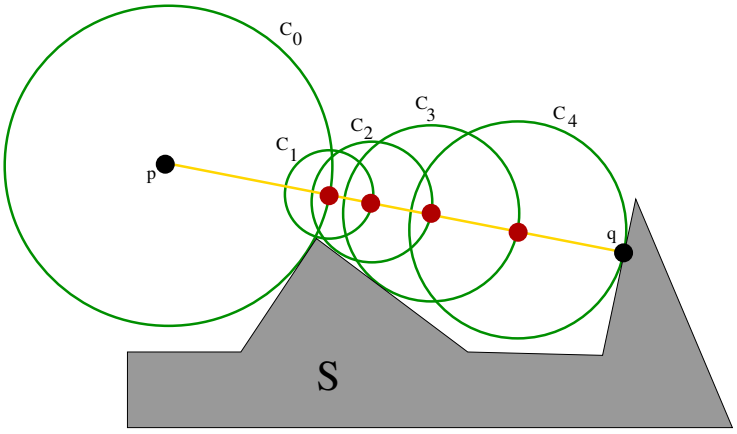
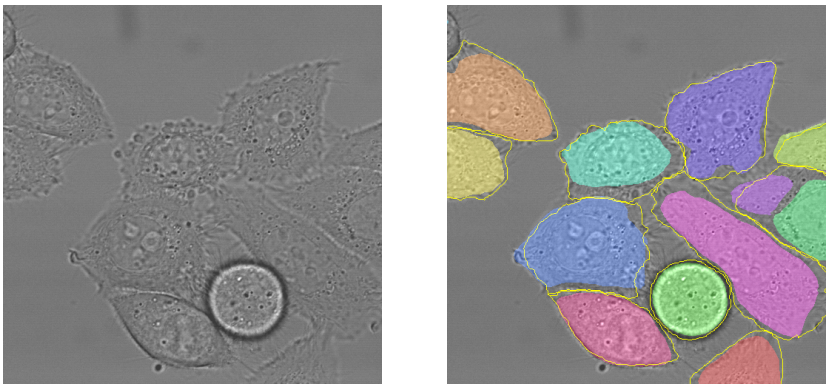


Fig. 3.5.: Sphere Tracing form a point p

3.6 Convolutional Neuronal Network – CNN

A ”Convolutinal Neural Network”, CNN for short, is a type of artifical neural net most commonly used in the field of visual data extraction and analysis. In most instances a CNN takes an image tensor I as its input. The architecture of a CNN is most commonly described as a multi-layered feed-forward network. This means that compared to an RNN [36], which uses a kind of loop in its data processing, a CNN uses a sequential design for data processing. The hidden layers in this kind of network are usually made up of mathematical convolutions operations. For that, for example flatten the image tensor to a vector of length N. Then consider this vector as a function t from $\{1, \ldots, N\}$ to $\mathbb{R}$. Now a convolution $t * t$ is an operation which produces a new function on the same or closely related domain by calculating a new value through ”neigboring values” for example multiplied and summed up. Physically a convolution can also be described as the application of a filter (see Section 3.2) to the image. The CNN learns many of these convolutions to produce higher dimensional vectors, so called feature vectors, for every pixel in the image. The process is iterated and the networks goal produce an output which allows the simple extraction of the desired information.

A good example of this procedure and CNN’s in general is LeNet [1]. The LeNet network is able to match an image of hand drawn numbers with the corresponding natural number. It does this by using only a few convolution layers followed by a few fully connected layers and a soft-max activation to classify the image.



(a) Original (b) Segmented

Fig. 3.6.: Original and segmented image side by side

3.7 U-Net

An ”U-Net” is an example of a Convolutional Neuronal Network, which was first introduced in [33] by researcher from the University of Freiburg. Originally in [33] it was used for biomedical image segmentation. There the U-Net network is tasked to spatially separate cells in an image of many cells. For this the researchers first hand labeled a large portion of their data set. After that they also applied distortions to the hand labeled images. They did so in order to be able to expand the training data set without being forced to again embark in the tedious hand labeling procedure. Figure 3.6 provides an illustration of an example of the segmentation results of the U-Net described in [33]. The picture is a slightly simplified reproduction from the paper.

The architecture of U-Net can be subdivided into two parts, an encoder and an decoder. The encoder again can be divided into multiple double convolution blocks connected by a max pooling layer (see Figure 3.9). The encoder part uses these double convolutions and max pooling to down sample and learn the extraction of the important information contained in the image. On the other hand the decoder uses convolution as well together with an up sampling procedure (again some convolution) to reconstruct the image and provide a pixel aligned evaluation of where a cell ends and starts. The encoder process uses so called max pooling operations for the down sampling / pooling of the input values. A pooling operation in the field of image processing is a way to compress a given values grid into a smaller form factor without using to much information about it. The max pooling operation for example takes $k \times k$ blocks out of a layer of an input tensor. This block

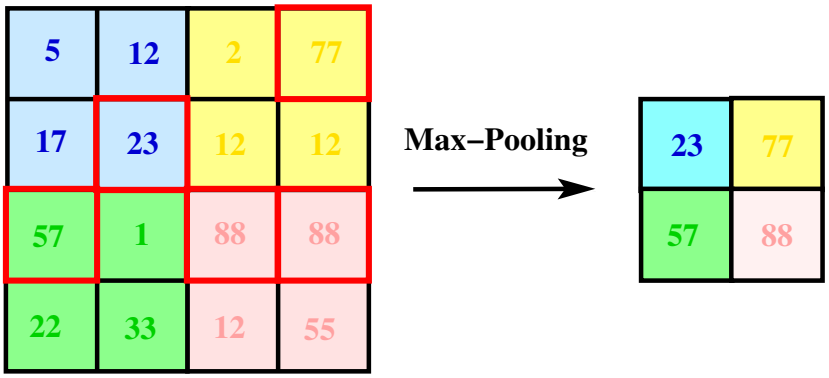


Fig. 3.7.: Down Sampling Process

then is processed, so that the largest (max) value of them is chosen and put into the compressed new image. The process will be outlined in Figure 3.7 for $k = 2$.

There are other pooling operations that could be used. For example there are mean pooling, min pooling etc. which function similarly except that the selection of the value for the compressed image is calculated differently. Indeed the respective naming of the pooling operation indicates the computation. But for this case the max pooling operation was chosen.

The up-sampling process that is used in the decoder, is a way to revert the down sampling. It uses the interpolation of the values, which the down sampling created, to create new values which are supposed to estimate the values lost by the down sampling procedure. In case of [33] the up-sampling process is not detailed in the paper. It can be guessed that the process used was probably that a every new value took the value of the pixel it came from. In Figure 3.8 we depict two possible such up-sampling procedures. The process of up-sampling allows a CNN, to be more compact and as a consequence requires less time to train it.

The process of max pooling and up-sampling is widely used in the design of pixel segmentation processes using CNNs as well as other kind of neuronal nets. The result of the process of up-sampling is outlined in Figure 3.8. This will also solidify the fact that pooling and are pseudo inverse operations. Indeed, the real grid values will be preserved.

To ensure the flow of data into the deeper layers of the network sip connections are used. In [21] this procedure has been shown to produce a better loss landscape. This means that there is a lower risk of running into local minima / maxima during the optimization process of the network. The skip connections of the network in Figure 3.9 are drawn as the horizontal light red arrows. The operation behind them usually is the addition of the output of the layer from which the skip connection

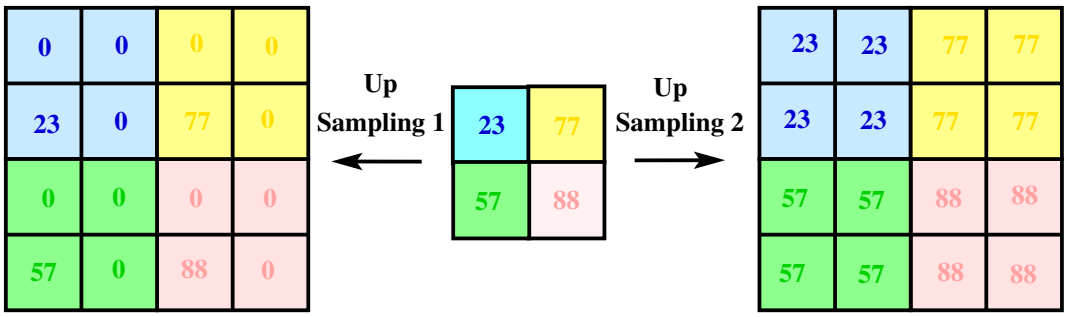


Fig. 3.8.: Examples of pp-sampling procedures

originates to the input of the layer it points to. This then in turn most commonly provides the desired flow of data into the deeper layers.

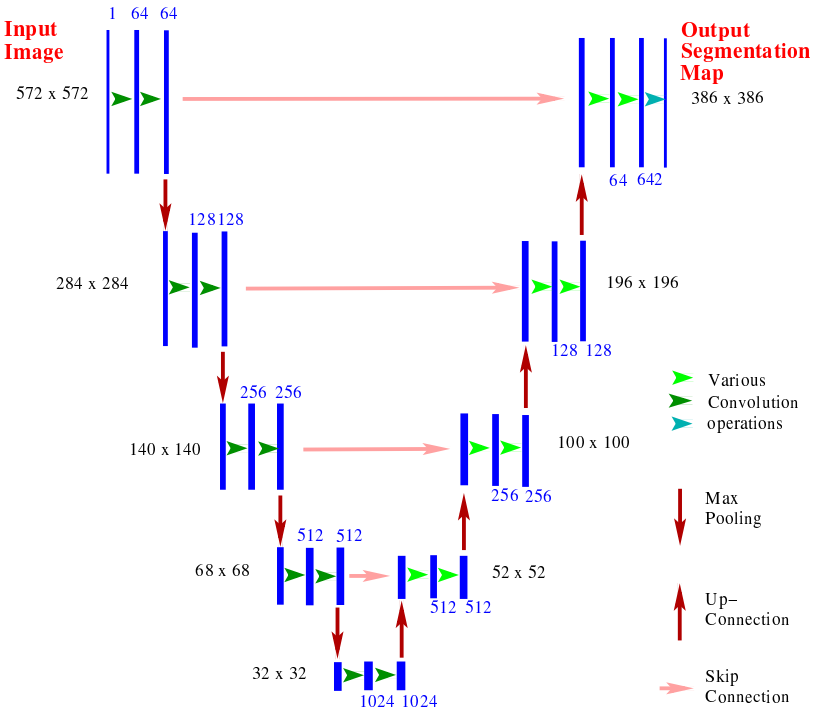


Fig. 3.9.: Example of an U-Net

The ”U” in U-Net is motivated by the fact that the architecture of down and up sampling can be pictured as an U-shape. In this work we adapt this kind of network to our setting on order to produce pixel aligned feature vectors for every given pixel of an input image (see 5).

4 Data Set and Image Rendering

The data-set used in this thesis is provided by ”Name of Dude” and includes OBJ data (see Section 4.2) of heads as well as an image that is used to the texture of the head. My approach to rendering the $512 \times 512px$ images used for training was to apply the ”Blender API” (see Section 4.3) to render out images. In this chapter we describe both and provide additional information on how they were used in our projects.

4.1 Model Creation

The data set was created by an online dealership (see https://www.3dscanstore.com/) which focuses on 3D scanning of humans and animals. The services also include textures for all their 3D models as well. For the scanning process they use a 3D scanning rig that utilizes the process of laser scanning. Their process yields a very high resolution of the objects – humans and animals – which they scan. This in turn makes their 3D models a good base for deep learning approaches with the goal to estimate 3D models.The data provided to us included 5 data sets each of human faces of males and females. Each data set of a face is comprised of 37 different facial expression for the face. Since the process of 3D scanning of hair is a really hard task, every human who did not have short hair had to wear a hairnet while scanning. Otherwise the 3D modelling process could get confused by the hair. This limitation of 3D scanning is widely known, because the extraction of structures as fine as hairs is only possible with a calibrated camera which has almost infinite precision in its parameters.

4.2 OBJ Data

OBJ is an open 3D geometry file format that was developed by Wavefront Technologies for its flagship product, the Advanced Visualizer, in the 1980s and 1990s. It is

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

Tab. 4.1.: OBJ description of cube

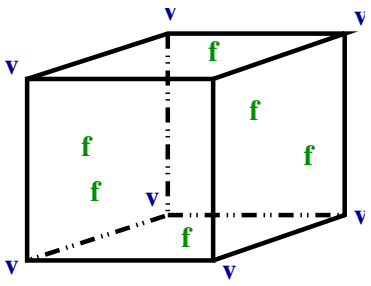


Fig. 4.1.: Cube describe by OBJ data from Table 4.1

used to describe the geometry of 3D objects and their related properties. If stored in ASCII format the OBJ files have extension ".obj". In the file format itself you can then define vertices v and their relations. The code example in Table 4.1 provides an example of a cube with vertices v and faces f. Every single vertex is referenced by the line of code it was defined in. In Figure 4.1 a schematic picture of the cube described in Table 4.1 is given.

The OBJ datatype also has the capability to include references to material, as well as many features not pertinent to the topic of this thesis. The surface normals of an OBJ object can be easily computed, this fact comes in handy as we get to defining losses later in this work (see Section 5.6).

4.3 Blender API

Blender [7] is a free (GPL licence) open-source 3D computer graphics software tool. It is used for animation, visual effects, 3D modeling and many other applications in the field of computer graphics. Its first version came out 1994, the current

version is 3.2 and it is still under development. The ”Blender API” and a compatible Python version is already included in Blender itself. For rendering images from OBJ data as well as the texture, we used pre-defined functions which are provided by the Blender API. It has the capability to load the OBJ data itself and layer the given material image file on top of it. After this the object center of mass is positioned at the world origin. This allows us to use the same camera and camera movement to produce the images themselves. The authors of [4] also used a HDR background to render out the training image. This principle is also applied to our setting and a result is shown in Figure 4.2. Blender provide means to use an HDR image as a background. The code for using the Blender API is included in our code base on GitHub [40] as well as a docker image that the code is able to run on. A full documentation of the Blender API can be found at https://docs.blender.org/api/current/index.html. From it you can deduced that the range of functions it provides is far richer than the ones we used when implementing the data reading and the rendering process. The code base for the python library that is the Blender API can be found at https://github.com/TylerGubala/bpy-build. If one wants to build a Blender python application, an in depth installation guide can be found at https://pypi.org/project/bpy/.

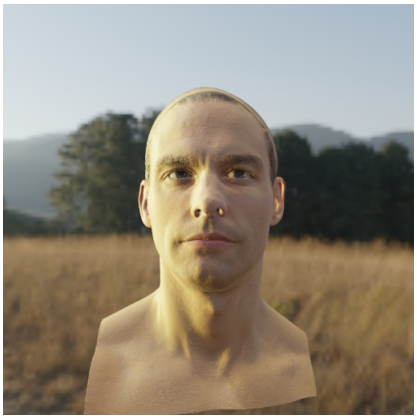


Fig. 4.2.: Rendering of human head with HDR background

5 Architecture

The reconstruction of 3D geometry given just a single input image I has been and continues to be a central research topic in the field of computer vision (see for example [34, 35, 3, 10, 16, 19, 22, 25, 27, 37, 47, 5, 32, 45, 4]). In this chapter we describe the architecture of the approach taken to this problem in this thesis. It is heavily influenced by the work in [4], where the authors explored an attempt to estimate pictures of humans wearing clothing from single images. As mention in Section 1.2 the goal for us is different and poses different challenges. We want to reconstruct a human head given a single image I. Note that even though this is a part of a human body the task is not a special case of [4] and additional and different problems arise.

5.1 Setting

In our project we assume that the image I is a 512×512 pixel image with RGB-values. From the image we estimate a set S (in our case S is the head displayed in the picture), which then determines the the surface $S(I)$ as well as the color of the 3D model. For the definition of $S(I)$ an extension of an SDF f, as defined in Section 3.1, is employed. This extension in addition to the signed distance d from the surface also returns a color a. For the definition of the surface we use sphere tracing (see Section 3.5) based on the fact that the SDF's distance component will return 0 if a point is on the surface of the object. For later use we assume that $a = [R, G, B]$ so that we can interpret it as an RGB color.

$$S(I) := \{x \in \mathbb{R}^3 : f(g(I, x), \gamma(x)) = (0, a) \text{ for some } a\} \tag{5.1}$$

In Section 3.1 the argument of the SDF f was a point in $\mathbb{R}^3$. For the extended SDF the arguments are $g(I, x)$ which is a pixel aligned feature vector $z_x \in \mathbb{R}^{256}$ corresponding to the point $x \in \mathbb{R}^3$ and a positional encoding $\gamma(x) \in \mathbb{R}^{256}$ of x. We must now define and detail the functions f , g as well as γ. This will be done in

Sections 5.2, 5.4 and 5.3 respectively. In reality all functions, that define $S(I)$, will be realized as artificial neuronal networks with learnable parameters.

5.2 SDF with color component

We have introduced the concept of a signed distance function in 3.1. We now extend this definition to a signed distance function f that has a color component as well. The input of this function will not only take the positional encoded $3D$ coordinates $\gamma(x)$ (see Section 5.3), it will also take the pixel aligned feature vector z_x (see Section 5.4) of the corresponding point x. This further input will be used so that the function f will not only return the signed distance value d but a color value a as well. Indeed the f evaluates to the pairs (d, a).

By querying the function with, for example, octree (see Section 3.4) or taking the gradient of the f sampling, we can estimate the whole surface geometry in great detail without losing the related color values for a given polygon. The function f is called as follows:

$$f(z_x, \gamma(x)) := (d, a) \tag{5.2}$$

5.3 Positional Encoding

As shown in other papers (see e.g., [30]), deep neuronal networks are more error prone when fed raw (low dimensional) data. As a remedy the authors of [4] propose a form of positional encoding with a sinusoidal function. Thus an encoding function γ is proposed, that for example in [26] it has already been show to greatly improve the results of novel view syntheses.

$$\gamma(p) = \Big\langle (sin(2^t \pi p), cos(2^t \pi p)) \Big\rangle_{t=0}^{L} \tag{5.3}$$

$$\gamma : \mathbb{R} \longmapsto \mathbb{R}^{2L+2}$$

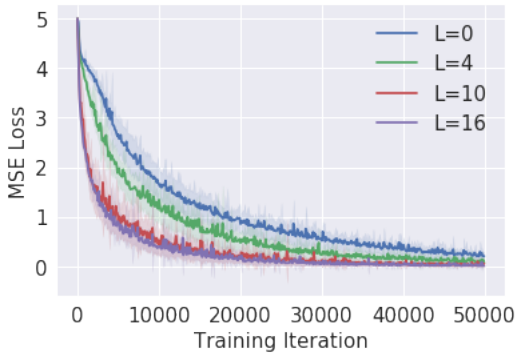


Fig. 5.1.: Loss of MLP fit to function λ with increasing values of L for input data [30]

In general, [30] has shown that the training of artificial neuronal nets yields better and faster results on higher frequency / dimensional data compared it results on lower dimensional / frequency data. A exemplary proof for this statement based on experimental data is provided in the paper [30]. Here the authors of fit MLPs with increasing values of L and by that increasing the dimension / frequency of the input data. We are able to see from their Figure 5.1 [30] that the loss of the network with a low value for L has a far smaller drop per iteration than with a higher value for L. Form this we can deduct that the network learns quicker. As one can also see the loss seems to converge to a smaller value. This means that the MLP also will yield better results on higher frequency data.

In [4] the factor L is chosen so that the dimension of the new vector will be 256. This is not explicitly mentioned in [4] but can also be derived from the fact that the input of f is supposed to be 512 dimensional. Thus $\gamma(p)$ together with its corresponding feature vector can be passed on to f.

5.4 Feature Extraction

The feature vector extraction function g takes as arguments the image I (in our project a $512 \times 512px$ picture) and a point $x \in \mathbb{R}^3$.

$$g(I, x) = z_x$$

The value of g is a feature vector z_x with is a numerical representation of so called features. One of the most simple forms of a feature vector is an RGB color value in

an image. Here the color of every pixel is denoted by a 3-dimensional vector with red, green and blue values that are in between 0 and 255. In general, there is no limit to the dimension of a feature vector. But the purpose of such vectors in almost every case is to encode information of the input in the format of a vector. In our case a feature vector $z_x \in \mathbb{R}^{256}$ will be produced for every pixel of the input image I. We then call these vectors the ”pixel aligned feature vectors”.

In our case, the feature vector extraction function is a composition

$$g(I, x) = b(G(I), \pi(x)) = z_x. \tag{5.4}$$

Here G is a feature extractor function/network G. It uses a modified version of the U-Net architecture (see Section 3.7) to produce these vectors. The function b is now defined as a function which can extract the corresponding feature vector of every pixel given $G(I)$ and $\pi(x)$, where $\pi(x)$ defines the pixel coordinate of the given point $x \in \mathbb{R}^3$. The process behind b is a so called bi-linear interpolation over the image plane and the feature tensor.

5.5 Shading Network

The shading will be done in yet another network s which will be taught to estimate the surface shading by per-point shading. For this we decouple the shading model from the surface color completely. Indeed, we first calculate the surface normal for a given point x and denote it as n_x. We calculate the surface normal as the gradient Δ_x of the surface distance d_x in the point x described by the (enhanced) SDF f.

$$\Delta_x d_x = n_x$$

Another input to this shading function will be the illumination model l which will be derived from feature extractor network G. It will be derived from the bottleneck of G, also called the half point of G. This procedure can be used, because of the architecture that G will be implemented in (see 3.7). The dimensionality of l will also be further reduced. The function s will then produce a shading vector s_x for the point given its surface normal n_x and its illumination model l.

$$s(n_x, l) = s_x$$

The final shading color can then be produced by the elementwise multiplication of a_x and s_x, where a_x is the result of $f(z_x, x) = (d, a_x)$.

5.6 Losses

Following [4] the training of the networks will be, done by rendering scans with sphere tracing 3.5 of faces and drawing samples form the mesh itself. For estimating the quality of our predictions we define loss functions and an optimizer to tweak the internal parameters accordingly. For the optimization process we have used the optimizer ADAM [20]. We first describe the optimizer and then the different loss functions.

ADAM Adam is a further development of a classical gradient decent that with great success has been widely used in the field of deep learning in recent years (see [12, 9, 26]. The authors of [20] refine gradient descent by two further algorithms to create ADAM. These two extensions are the ”Adaptive Gradient Algorithm” (AdaGrad) and the ”Root Mean Square Propagation” (RMSprob). By doing this the ADAM is able to use the benefits of both of these extensions. This results in a slowly degrading learning curve at fist but a faster and faster approach to the local extremum in the longer run. It does this by adapting the learning rate during the process as well as using the concept of momentum and weighted gradient movement. The use of these algorithms also been shown to speed up the optimization process and by that allows machine learning researchers to, more quickly and efficiently train their models. The simple procedure behind ADAM also allows it to run on a small amount of memory compared to other optimizers and in addition it is computationally efficient.

Geometry and Color Losses To be able to calculate a geometry loss we first define the ground truth mesh for a given image I as $\mathbf{M}$ describing $S(I)$. The surface/geometry loss $\mathbf{L}_g$ will be calculated by drawing a set of samples $\mathbf{O}$ of the surface, and enforcing the distance d_x of them to return zero and the distance of the gradient (surface normal) and the corresponding normal $\hat{n}_x$ of the ground truth mesh to return zero as well. By n_x we denote the estimated normal. We also define weights

λ_{g_1} which are linearly increased from 0 to 15 over the iterations and $\lambda_{g_2} = 1.0$ to weigh the difference.

$$\mathbf{L}_g = \frac{1}{|\mathbf{O}|} \sum_{x_i \in \mathbf{O}} \lambda_{g_1} |d_{x_i}| + \lambda_{g_2} \|n_{x_i} - \hat{n}_{x_i}\| \tag{5.5}$$

The second loss $\mathbf{L}_l$ will be defined over another sample set $\mathbf{F}$, that consists of samples drawn from around the surface. Here we want to supervise the sign of the signed distance function by calculating the mean $\mathbf{BCE}$ (Binary Cross Entropy) loss $\mathbf{L}_l$ over the drawn samples. The ground truth inside our label will be denoted as l_x and compared with the label belonging to d_x. This will be calculated by taking the sigmoid ϕ (see (5.6)) function value of the d_x multiplied with a learnable parameter k, that determines the sharpness of the decision boundary.

$$\phi(x) = \frac{1}{1 + \exp(-x)} \tag{5.6}$$

The formula for the loss can now be formulated as :

$$\mathbf{L}_l = \frac{1}{|\mathbf{F}|} \sum_{x_i \in \mathbf{F}} \mathbf{BCE}(l_{x_i}, \phi(k d_{x_i})) \tag{5.7}$$

To ensure the normalization of the surface normals we define a second loss over the set $\mathbf{F}$.

$$\mathbf{L}_e = \frac{1}{|\mathbf{F}|} \sum_{x_i \in \mathbf{F}} (\|n_{x_i}\| - 1)^2 \tag{5.8}$$

The loss over the estimated albedo color a_x will be a combination of values assigned to the sample $\mathbf{F}$ and to the sample $\mathbf{O}$. The process used can be described as a weighted mean error function with weights λ_{a_1}, λ_{a_2} over the provided data point sets. Following previous conventions a_x is the estimated value and $\hat{a}_x$ the ground truth.

$$\mathbf{L}_a = \lambda_{a_1} \frac{1}{|\mathbf{O}|} \sum_{x_i \in \mathbf{O}} |a_{x_i} - \hat{a}_{x_i}| + \lambda_{a_2} \frac{1}{|\mathbf{F}|} \sum_{x_i \in \mathbf{F}} |a_{x_i} - \hat{a}_{x_i}| \tag{5.9}$$

Rendering Losses We also use rendering losses to further improve the output of the network. A rendering loss will be defined over the process of ray / sphere tracing given the estimated signed distance function f. We will render out 32×32px image patches during the training process. For this we first define a camera position π and its corresponding set of image rays $\mathfrak{R}$. At first we must locate the surface of the object S defined by f (see 5.2) from camera position π. We do this by applying ray tracing to the rays r in $\mathfrak{R}$ to locate the surface of S. We then determine a bounding box for S such that the surface of the box has a minimum distance value $\hat{\sigma} = 0.5$ to S. To find the box we proceed as follows. A value σ_r is calculated for every $r \in \mathfrak{R}$. This value will return the minimum distance to the object along the ray and can be described the following way.

$$\sigma_r = \phi(k \min_{t \geq 0} d_{\pi + tr}) \tag{5.10}$$

We then determine the bounding box by considering the subset $R \subset \mathfrak{R}$ that describes the set of rays where $\sigma_r \leq \hat{\sigma}$. These rays are then the set of rays that are hitting or are close to hitting the object itself. For the points along the ray we then compute the inside and outside labels already mentioned in the geometry loss paragraph (see page 29). The subset $R \subset \mathfrak{R}$ will then be used to exactly determine the surface using sphere tracing (see Section 3.5). To determine both the front and the back surface a form of sphere tracing has to be used that not only draws samples on the way from the camera to the object but from infinity to the object as well. By doing this we are able to calculate the back surface as well as the color of it. We now denote the calculated front and back points of our calculated 3D model as x_f and x_b respectively, where f stands for front and b stands for back. We then define our rendering losses the following way.

$$\mathbf{L}_c = \frac{1}{|R|} \sum_{i \in |R|} |a_{f_i} - \hat{a}_{f_i}| + |a_{x_{b_i}} - \hat{a}_{b_i}| \tag{5.11}$$

This loss is designed in a way that it should enforce the albedo colors of the surface. The shading loss will be defined later. The original albedo colors are calculated by rendering out images of the front and the back of the original 3D model. The image of the backside of the original model then can be calculated by then inverting the Z-buffer.

The authors of [4] also include a VGG loss L_{VGG} that is further explained in [15]. We refer the reader to [15] for more details.

To further enforce the right shading for the we supervise the shading of the network with :

$$\mathbf{L}_c = \frac{1}{|R|} \sum_{i \in |R|} |\hat{a}_{x_{f_i}} \odot s_{x_i} - p_i| \tag{5.12}$$

With p_i being the corssespnding pixel color in our input image I corresponding to ray i.

At the end the authors of [4] found out that it is useful supervise the shading of the image I itself using the ground truth normals $\hat{n}$ as well as the ground truth albedo colors $\hat{a}$.

$$\mathbf{L}_s = \frac{1}{|R|} \sum_{i \in |R|} |\hat{a}_{x_{f_i}} \odot s(\hat{n}, l) - p_i| \tag{5.13}$$

Combined Loss Because the network is meant to be trained front to back in one go, all losses will be combined into one loss $\mathbf{L}_*$. The loss $\mathbf{L}_*$ will be a linear combination of all losses mentioned above and be defined the following way:

$$\mathbf{L}_* = \mathbf{L}_g + \lambda_e \mathbf{L}_e + \lambda_l \mathbf{L}_l + \lambda_a \mathbf{L}_a + \lambda_r \mathbf{L}_r + \lambda_c \mathbf{L}_c + \lambda_s \mathbf{L} + \lambda_{VGG} \mathbf{L}_{VGG} \tag{5.14}$$

Where the given scalar values $\lambda_?$ will be defined as following way:

$$\begin{aligned} \lambda_e &= 0.1 & \lambda_l &= 0.2 & \lambda_r &= 1.0 & \lambda_s &= 50 \\ \lambda_{VGG} &= 1.0 & \lambda_{g_2} &= 1.0 & \lambda_{a_1} &= 0.5 & \lambda_{a_2} &= 0.3 \end{aligned}$$

The only variable will be λ_{g_1} it will be described as a linear function that adapts the value form 1.0 at the beginning to 15.0 over the duration of training iterations. In our approach, we are combining this whole process with the ADAM optimizer. We

want to again mention that all losses where pre defined in [4] and we did not come up with them ourselves.

6 Implementation

6.1 Framework

The framework that we used for implementing the neuronal nets, was ”PyTorch”.

PyTorch PyTorch [29] is an open source deep learning framework. It is most commonly used for computer vision and natural language processing tasks. It was developed by ”Meta AI” a sister company of ”Meta” the company ”Facebook Inc.” belongs to. We chose PyTorch since it is the framework we had the most experience. For this reason we did not chose to implement in its biggest competitor framework ”TensorFlow” [24] (developed by Google / Alphabet). Another reason for choosing "PyTorch" as a deep learning framework was it, that other especially computer vision python libraries were used that work hand in hand together with ”PyTorch”.

6.2 Network

Following [4] the implementation of the network itself at the same time was simple and challenging. Even though the global structure of their network is clearly described in [4] many details are lacking. We begin our description of implementation details by describing the implementation of the U-Net G (see Section 3.7).

Feature Extractor ”G” In [4, Section 3.3] the authors describe their feature extractor G as an U-Net with 13 encoder/decoder layers and skip connections (see Section 3.7 for the rough structure of an U-Net). To us the definitions of layer and skip-connections in [4] appear to be ambiguous. We will comment on possible interpretation in the next paragraphs. Their first layer contains 64 filters and the filter size is doubled in the encoder in each layer up to 512 at the maximum. The decoder halves the filter size at the 11^{th} layer, which effectively means that G produces features in $\mathbb{R}^2 56$. We use Leaky ReLU activation functions and blur-pooling [46] for the encoder and bilinear resizing for the decoder, respectively.

As mentioned in the preceding paragraph we struggled with the definition of a ”layer” in a U-Net [4] which appears to be ambiguous and not well defined in the text. The two interpretations we considered as reasonable were the following.

- a layer is meant to be a double convolution in an U-Net.

  If that were the case the U-Net would be in a way to deep to be trained in a acceptable amount of time. Since the authors of [4, Section 3.3] mention that the pipeline of the network itself is relatively small, we could rule out this interpretation.

- a layer in the U-net is meant to be a single convolution inside the network.

  This interpretation could later be shown to be consistent and appears to coincide with the one the authors of [4] had in mind.

The second hurdle we faced when following our approach was the definition of a skip-connection. In other contexts [] a skip connection is essentially a directed edge between two layers. It takes the result of the layer where the skip connection originates and the input of the layer where the connection points to and performs some kind of mathematical operation to them. The following possibilities for the operation we considered were the following.

- A skip connection is the addition of the output tensor of the layer where the skip connection originates to the input tensor of the layer where it points to.

  This definition of a skip connection seems to contradict the way the layers are defined. One example for this is the skip connection from the first double convolution $\mathcal{L}_f$ layer to the last / output layer L_o. The dimensions output of $\mathcal{L}_f$ by the definition given by the authors of [4] is $256 \times 256 \times 64$ whereas the input of $\mathcal{L}_o$ is defined to be $256 \times 256 \times 256$. This addition in a mathematical sense is not possible and for that reason this option can be ruled out.

- A skip is the concatenation of the output tensor of the layer where the skip connection originates and the input tensor of the layer where it points to.

  We found out that this must be the case by a private correspondence [2] with the authors of [4]. The interaction with the authors, will be provided in Appendix A.1. In this interaction it was made clear to us that the concatenation must be used to implement the U-Net architecture.

Another important implementation detail is the use of the blur pooling operation. The blur pooling process was first introduced in [46] and combines the process

of applying a Gaussian filter (see Section 3.2) and the ”usual” max pooling operation. This pooling operation was shown in [46] to improve the consistency of convolutional neuronal nets. The authors of [46] included an implementation of the blur pooling algorithm which also uses ”PyTorch”. The bi-linear interpolation for up-sampling was already included in ”Torch-Vision” and thus did not have to be implemented.

SDF with color component ”f” For the implementation of the SDF network a ”simple” MLP was used. It has eight 512 dimensional fully connected layers with a SWISH activation function. The SWISH activation function was first introduced in [31] in work exploring new kinds of activation functions and their impact on neuronal nets. Our SWISH function is defined in (6.1) and was already implemented in ”PyTorch” as well. In [31] SWISH also has been shown to improve the accuracy of neuronal nets, mostly in computer vision tasks.

$$SWISH(x) = x\phi(\beta x) \tag{6.1}$$

In the SWISH activation function the parameter beta is a trainable parameter. We can imagine the activation function as some kind of an enhanced sigmoid (see (5.6)) function already mentioned in Section 5.6. In the output layer of f the ”normal” sigmoid function is also used as an activation function, but only applied to the color component a of the output. The implementation also includes a skip connection to again ensure the flow of data into the deeper layers and improve the loss landscape. See Section 3.7) where this was already listed as a benefit of implementing skip connections in artificial neuronal nets. In our implementation of this skip connection, unlike the skip connection in the implementation of f (see Section 5.2) an addition and not a concatenation of the values was used. This implementation choice again is deduced from the dimensionality of the inputs and outputs of every layer. If we had used a concatenation instead the input of the layer of the skip connection (the middle layer in this case) would have been $\mathbb{R}^{512\times1}$ whereas the actual input is $\mathbb{R}^{512\times2}$.

Shading Network ”S” The shading network ”S” is made up out of a simple MLP and returns a 16 dimentional illumination code. It has three 256 dimensional fully connected layers with SWISH (see 6.1 [31])activation functions as well as an ReLU activation function for the output layer.

Positional Encoding In [4] it is not specified which positional encoding algorithm is used. The one we chose is the same as the one used in NeRF 5.3. Because of the simplicity of its structure its implementation was quite straight forward.

6.3 Sphere Tracing

A problem we had to solve when calculating the losses of the network was that we had to have some kind of method for rendering out images. More precisely, we needed to render out exact image patches (32×32px) given an SDF with a color component. The process we used was sphere tracing (see 3.5). We did not implement the whole process on our own, we rather used an existing implementation of sphere tracing that had already been shown to be efficient rendering out image patches. For chosing an existing implementation, there were a few options available, we went for an implementation called Torch Sphere Tracer . It uses features of ”PyTorch” , ”TorchVision” as well as ”pytorch3D”, a python deep learning library designed for 3D geometry data. All of the above given libraries support ”CUDA” [28] and by that can outperform CPU only libraries by huge margin.

Our chosen implementation was designed to render out images given an SDF. It only supports mono colors for a given SDF, here mono color means that only one single color was supported for the surface of the SDF. We had to adapt it so that an SDF with a color component could be used. We proceeded by first creating a mock-up SDF with a color component that we used as an input SDF. In our case this was the SDF for a sphere (see Section 3.1). As a mock color component we used the position x that also is the input for the SDF. This color component then must be saved for every convergent ray during the sphere tracing process so that we can define a value for every point that can be viewed in the camera plane. For this we had to change the implementation so that for a given SDF with a color component the color values as well as the distance d is saved separately. Another good feature which this implementation provides is the calculation of surface normals given an SDF. In particular, given their definitions this will be handy when calculating the losses. From them both the ground truth normals are needed as well as the one produced by our und SDF f (see Section 5.2). Another process that we introduced earlier (octree sampling Section 3.4) could have been used for this process as well. The problem of the octree sampling algorithm here would be that all surface normals must be calculated so that the exact sample normal’s that we need can be calculated. Using octree this process then in turn would have a way worse run time compared to the sphere tracing algorithm. The authors of [4] also used the sphere tracing

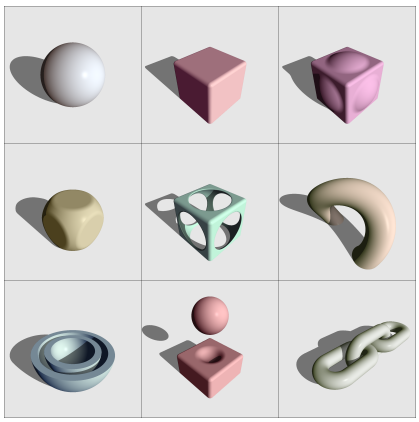


Fig. 6.1.: Mono color image

Fig. 6.2.: Sphere Tracing with color component

gradient approach to calculate the surface normals. We can speculate that this was due to this exact reason.

The following two images in Figure 6.1 and 6.2 are examples of rendered out images. Figure 6.1 uses mono color and Figure 6.2 an SDF with a color component.

6.4 Losses

For the implementation of the losses from Section 5.6 defined a new ”pytorch.nn.module” and then adapted it to always take in the right input. In the following code snippet we provide a blueprint example of a custom loss function.

```
class MyLoss(nn.Module):
    def __init__(self, weight=None, size_average=True):
        super(MyLoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):

        inputs = F.sigmoid(inputs)

        inputs = inputs.view(-1)
        targets = targets.view(-1)

        intersection = (inputs * targets).sum()
        loss = (2.*intersection + smooth)/(inputs.sum()

        return 1 - loss
```

Listing 6.1: Custom Loss in PyTorch

This custom loss fuction was then adapted so that all the losses provided in Section 5.6 were implemented. The implementation sometimes themselves include additional loss function calculations (e.g., BCEloss). We learned [2] (see Appendix A.1) from one of the authors of [4] that for the functioning of the whole network the implementation of the loss functions is the key part. The network itself will be trained end to end combining all before mentioned networks as well as the error calculation, and combing it with the ADAM [20] optimizer with a learning rate of 10^{-4} decaying with a factor of $.9$ over 50.000 training steps.

An important thing to denote here is that the implementation of the sphere tracing algorithm is only a part of process of the rendering loss implementation described in Section 5.6.

7 Conclusion

In the final and last chapter of this thesis we talk about the achieved results and possible improvements of the implementation and of the concept itself.

7.1 Implementation

General As already mentioned we chose to implement the networks themselves in ”PyTorch.” The implementation from [4] was realized in ”Tensorflow.” This fact can be deduced from the e-mail exchange in Appendix A.1). On the other hand the choice is not surprising, since the researchers that developed the method of [4] work at ”Google”, which develops and advertises ”TensorFlow”. This detail of the implementation should not make any difference since ”PyTorch” and ”TensorFlow” provide equivalent functionality. For example there are already frameworks that can translate ”PyTorch” code into ”TensorFlow” code.

In Section 4 we mentioned Blender as our choice of rendering tool. Blender proved to be an excellent choice, both for the implementation of the rendering and for the data loading process with the Blender API. One of the benefits which contributed to the usefulness of Blender is, that Blender includes an interface for coding in its application. The coding interface in turn includes a kind of translation tool/logging interface that allows to see the an equivalent python operation for every button that is pressed inside of Blender itself. This tool made it quite easy to implement an automated version of the rendering process. The log can be seen in Figure 7.1

Losses A crucial point of the implementation is defining the losses in PyTorch as well as extracting the data needed from the network, this fact was particularly mentioned by one of the authors of [4] in (see A.1). The authors of [4] did not provide any code together with their paper, so everything had to be implemented by hand or existing implementation had to be adapted in a way so that it somewhat resembled the approach taken by the authors. The results of rendering out images from the 3D Models provided to us can be seen in Figure 7.2.

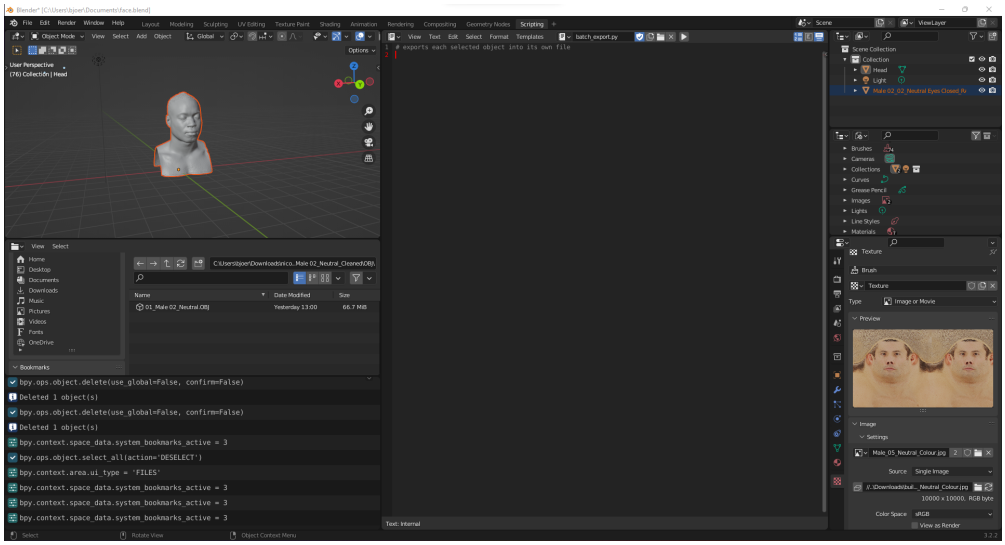


Fig. 7.1.: Blender Implementation tool

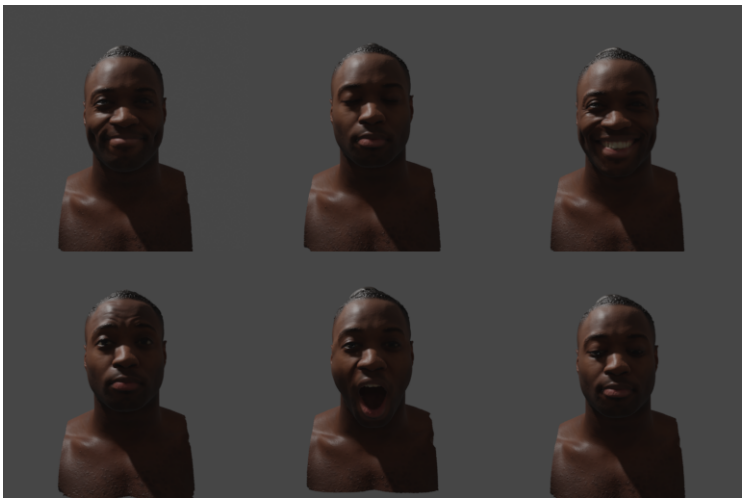


Fig. 7.2.: Facial Expressions in the data set

The implementation of the losses was most difficult to develop and understand. This is due to the fact that not all methods and processes documented in [4] were clear to us from the beginning. A lot of time went into understanding the methods described and understanding the choices mode by the authors. One of the best examples of this is the way they describe the method of ray and sphere tracing they use to calculate the rendering loss (see Section 5.6). The way the data extraction for the loss is described is completely different to the commonly used image rendering process. As already mentioned, the authors did not provide any detailed implementation description or code of this process. As a consequence its implementation stalled the application of the results from [4]. To implement it a completely new sphere tracer that follows the approach taken by [4] in a fashion which somewhat resembles the efficiency of Torch Sphere Tracer would have taken another fair amount of time. With this achieved the way is free to the training of the network. Another hurdle that we encountered was the incompatibility issues of some python libraries and their the versions of CUDA [28]. The sphere tracing implementation we used was recommended to be used with a version of CUDA [28] which was incompatible with the GPU we used. This was later fixed by adapting the installation process of the ”PyTorch3D” library. This is also why we included a Docker container for the Blender code due to similar compatibility issues. This results from the fact that there is no way to use the graphical interface to Blender over an SSH [43] connection and by that a ”stand alone” python environment had to be built.

This experience shows that the implementation of a version of the software from [4] adapted to the estimation of 3D models of human heads is a feasible but non-trivial task.

Although we introduced the octree sampling method for 3D model extraction , we later realised the an implementation of it is only needed for extracting the whole 3D structure. This is also why we chose not to implement the algorithm on our own, but still set out to find an appropriate implementation for our network. The search itself in this case did not yield any satisfying results for an implementation.

Data Set The data set (see Chapter 4) provided to us did not include 3D models of heads of people that was as diverse and rich as the models used for training in [4]. So even if we had been able to train our network, this fact could have led to a less successful generalization process. This shortcoming is due to the fact that the data set focuses on facial expressions of a small set (5 male and 5 female) of people and does not provide a wide range of human heads.

For the process of rendering out the images we used the Blender API (see Chapter 4). In the process of implementation we later realized that, the same results could have been archived with ”pytorch3D” as well. The use of the later could have expedited the implementation of the loss related value extraction a lot. By general software design principles, the implementation under a single framework in ”Python” would have been beneficial. In addition, choosing to render the images with ”pytorch3d” would have reduced the learning part on our side since familiarity with a single library would have been sufficient. This fact too could have expedited the implementation process. Our reasons for choosing to render out images with Blender API were that Blender API is a widely used software, which is highly efficient and optimized. In addition, when first working on the rendering process it was the only suitable software familiar to us which included a Python implementation of its processes.

The experience with our data set shows that it is a suitable data set which should be more diverse when used for serious training. It can be handled by various existing software, none of which is the clear cut first choice.

7.2 Future Work

In general, it can be said that apart from [4] there is no prior implementation of the methods that were described in the paper. Moreover, we had to unravel some of the detail of [4] not made explicit in the text. This hold in particular for defining and extracting the important information from the rendering loss. Clearly these facts slowed process of implementation. Nevertheless, the thesis has paved the path for an application of [4] to the estimation of 3D models of heads from single images.

There is a good chance that the method described in [4] to work on human heads as well. Our implementation can definitely be further improved in terms of efficiency and speed of learning, so that we are able to really train the designed network. Another point of improvement should be the extension of the data set to include a wider range of people and their facial expressions so that for a given 2D image of a face a estimation of a 3D reconstruction can be created. In the future there can probably also be improvement to the architecture of the model itself. An improvement we can see, is for example changing the network f 3.1, currently a signed distance function, to be more a NeRF [26]. In recent months the visualisation of NeRFs has be shown to be greatly speed up the network and deduce results much faster. Examples for this can for example be found in [44]. Here volumetric representations of NeRFs are produced within seconds of starting the training

process. The downside of this process is, that in turn the exceedingly fast process of sphere tracing can no longer be used in for rendering out images any more.

As a conclusion, much remains to be done in this exciting field of research.

Bibliography

[1] Rashad Al-Jawfi. “Handwriting Arabic character recognition LeNet using neural network”. In: *Int. Arab J. Inf. Technol.* 6.3 (2009), pp. 304–309 (cit. on p. 15).

[2] Thiemo Alldieck. *private communication* (cit. on pp. 4, 36, 40).

[3] Thiemo Alldieck, Marcus Magnor, Weipeng Xu, Christian Theobalt, and Gerard Pons-Moll. “Video based reconstruction of 3D people models”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* (Cit. on pp. 2, 5, 25).

[4] Thiemo Alldieck, Mihai Zanfir, and Cristian Sminchisescu. *Photorealistic monocular 3D reconstruction of humans wearing clothing*. 2022. arXiv: 2204.08906 (cit. on pp. v, vii, 2–6, 23, 25–27, 29, 32, 33, 35, 36, 38, 40, 41, 43, 44, 57).

[5] S. Asteriadis, K. Karpouzis, and S. Kollias. “Visual focus of attention in non-calibrated environments using gaze estimation”. In: *International Journal of Computer Vision* 107 (2014), pp. 293–316 (cit. on pp. 2, 5, 25).

[6] Bharat Lal Bhatnagar, Garvita Tiwari, Christian Theobalt, and Gerard Pons-Moll. “Multi-garment net: Learning to dress 3d people from images”. In: *Int. Conf. Comput. Vis. IEEE*. 2019 (cit. on p. 5).

[7] Blender Foundation. *Blender*. Version 3.2 (cit. on p. 22).

[8] Federica Bogo, Michael J. Black, Matthew Loper, and Javier Romero. “Detailed full-body reconstructions of moving peo- ple from monocular RGB-D sequences”. In: *Int. Conf. Com- put. Vis.* 2015, pp. 2300–2308 (cit. on p. 5).

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. *Language Models are Few-Shot Learners*. 2020 (cit. on p. 29).

[10] Zhiqin Chen and Hao Zhang. “Learning implicit fields for generative shape modeling”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2019, pp. 5939–5948 (cit. on pp. 2, 5, 25).

[11] E. M. Chutorian and M. M. Trivedi. “Head pose estimation and augmented reality tracking: an integrated system and evaluation for monitoring driver awareness”. In: *IEEE Transactions on Intelligent Transportation Systems* 11 (2010), pp. 300–311 (cit. on p. 5).

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018 (cit. on p. 29).

[13] Valentin Gabeur, Jean-Sebastien Franco, Xavier Martin, Cordelia Schmid, and Gregory Rogez. “Moulding humans: Non-parametric 3d human shape estimation from single images”. In: *Int. Conf. Comput. Vis.* 2019, pp. 2232–2241 (cit. on p. 5).

[14] J. C. Hart. “Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces”. In: *The Visual Computer* 12 (1996), pp. 527–545 (cit. on p. 14).

[15] Tong He, John Collomosse, Hailin Jin, and Stefano Soatto. *Geo-PIFu: Geometry and Pixel Aligned Implicit Functions for Single-view Human Reconstruction*. 2020 (cit. on p. 32).

[16] Tong He, John Collomosse, Hailin Jin, and Stefano Soatto. “Geo-PIFu: Geometry and pixel aligned implicit functions for single-view human reconstruction”. In: *Adv. Neural Inform. Process. Syst.* 2020 (cit. on pp. 2, 5, 25).

[17] Tong He, Yuanlu Xu, Shunsuke Saito, Stefano Soatto, and Tony Tung. “Arch++: Animation-ready clothed human reconstruction revisited”. In: *Int. Conf. Comput. Vis.* 2021, pp. 11046–11056 (cit. on p. 5).

[18] Zeng Huang, Yuanlu Xu, Christoph Lassner, Hao Li, and Tony Tung. “Arch: Animatable reconstruction of clothed humans”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2020, pp. 3093–3102 (cit. on p. 5).

[19] Aaron S Jackson, Chris Manafas, and Georgios Tzimiropoulos. “3d human body reconstruction from a single image via volumetric regression”. In: *ECCV Workshops*. 2018 (cit. on pp. 2, 5, 25).

[20] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014 (cit. on pp. 29, 40).

[21] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. *Visualizing the loss landscape of neural nets*. 2017. arXiv: 1712.09913 (cit. on p. 17).

[22] Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. “SMPL: A skinned multi-person linear model”. In: *ACM Trans. Graph.* 34 (2015), pp. 1–248 (cit. on pp. 2, 5, 25).

[23] William E. Lorensen and Harvey E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Computer Graphics* 21.4 (1987), pp. 163–169 (cit. on p. 8).

[24] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 35).

[25] Lars M. Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. “Occupancy net-works: Learning 3d reconstruction in function space”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2019, pp. 4460–4470 (cit. on pp. 2, 5, 25).

[26] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, et al. “NeRF: Representing scenes as neural radiance fields for view synthesis”. In: *ECCV*. 2020 (cit. on pp. 26, 29, 44).

[27] Ryota Natsume, Shunsuke Saito, Zeng Huang, et al. “Siclope: Silhouette-based clothed people”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2019, pp. 4480–4490 (cit. on pp. 2, 5, 25).

[28] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007 (cit. on pp. 38, 43).

[29] Adam Paszke, Sam Gross, Francisco Massa, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on pp. 2, 35).

[30] Nasim Rahaman, Aristide Baratin, Devansh Arpit, et al. “On the Spectral Bias of Neural Networks”. In: (2018) (cit. on pp. 26, 27).

[31] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions*. 2017 (cit. on p. 37).

[32] G. Riegler, D. Ferstl, M. Rüther, and H. Bischof. “Hough networks for head pose estimation and facial feature localization”. In: *Proc. British Machine Vision Conference*. 2014 (cit. on pp. 2, 5, 25).

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597 (cit. on pp. 16, 17).

[34] Shunsuke Saito, Zeng Huang, Ryota Natsume, et al. “PIFu: Pixel-aligned implicit function for high-resolution clothed human digitiza- tion”. In: *Int. Conf. Comput. Vis.* 2019 (cit. on pp. 2, 5, 25).

[35] Shunsuke Saito, Tomas Simon, Jason Saragih, and Hanbyul Joo. “PIFuHD: Multi-level pixel-aligned implicit function for high-resolution 3d human digitization”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2020 (cit. on pp. 2, 5, 25).

[36] Alex Sherstinsky. “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network”. In: *CoRR* abs/1808.03314 (2018). arXiv: 1808.03314 (cit. on p. 15).

[37] David Smith, Matthew Loper, Xiaochen Hu, Paris Mavroidis, and Javier Romero. “Facsimile: Fast and accurate scans from an image in less than a second”. In: *Int. Conf. Comput. Vis.* 2019, pp. 5330–5339 (cit. on pp. 2, 5, 25).

[38] David Smith, Matthew Loper, Xiaochen Hu, Paris Mavroidis, and Javier Romero. “Facsimile: Fast and accurate scans from an image in less than a second”. In: *In Int. Conf. Comput. Vis.* 2019, pp. 5330–5339 (cit. on p. 5).

[39] D. J. Tan, F. Tombari, and N. Navab. “Real-Time accurate 3d head tracking and pose estimation with consumer RGB-D cameras”. In: *International Journal of Computer Vision* (2017), pp. 1–26 (cit. on p. 5).

[40] Björn Bastian Welker. *Code for Bachelor Thesis: Monocular 3D Reconstruction of Human Heads*. https://github.com/br0wnbeer/3Dhumanheads. 2022 (cit. on pp. 2, 23).

[41] Jane Wilhelms and Allen van Gelder. “Octrees for faster isosurface generation”. In: *Transactions on Graphics* 11.3 (1992), pp. 201–227 (cit. on p. 12).

[42] Hongyi Xu, Eduard Gabriel Bazavan, Andrei Zanfir, et al. “GHUM & GHUML: Generative 3d human shape and articulated pose models”. In: *IEEE Conf. Comput. Vis. Pattern Recog.* 2020, pp. 6184–6193 (cit. on p. 5).

[43] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254 (Proposed Standard). Internet Engineering Task Force, 2006 (cit. on p. 43).

[44] Alex Yu, Ruilong Li, Matthew Tancik, et al. “PlenOctrees for Real-time Rendering of Neural Radiance Fields”. In: *ICCV*. 2021 (cit. on p. 44).

[45] Hui Yuan, Mengyu Lib, Junhui Houc, and Jimin Xiaod. “Single image-based head pose estimation with spherical parametrization and 3D morphing”. In: *Pattern Recognition* 103 (2020), p. 107316 (cit. on pp. 2, 5, 25).

[46] Richard Zhang. “Making Convolutional Networks Shift-Invariant Again”. In: *ICML*. 2019 (cit. on pp. 35–37).

[47] Zerong Zheng, Tao Yu, Yixuan Wei, Qionghai Dai, and Yebin Liu. “DeepHuman: 3d human reconstruction from a single image”. In: *Int. Conf. Comput. Vis.* 2019 (cit. on pp. 2, 5, 25).

List of Figures

List of Tables

List of Listings

Appendix

A

A.1 Correspondence with authors of [4]

```
Hi Bjoern,

happy to help. In general it does not matter too much how
the architecture exactly looks as long you get the
losses right.
Not sure why you think the skip connections cannot work the
way we describe the network.
Maybe what you got wrong
is that we describe the decoder
dimensions before concatenating
the skip connection. Maybe this summary log
of the models help, unfortunately the
layers are not logged in order of execution:

+-------------------------------------------------------------+-------------------
| Name                                                        | Shape
| Size      | Mean      | Std      |
+-------------------------------------------------------------+-------------------
| while/unet_encoder_decoder/decoder/decoder_0/conv_0/bias:0   | (512,)
| 512       | -0.000579 | 0.00131  |
| while/unet_encoder_decoder/decoder/decoder_0/conv_0/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.0001   | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_0/conv_1/bias:0   | (512,)
| 512       | -0.000529 | 0.00116  |
| while/unet_encoder_decoder/decoder/decoder_0/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | 8.06e-05  | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_1/conv_0/bias:0   | (512,)
| 512       | -0.000686 | 0.00109  |
| while/unet_encoder_decoder/decoder/decoder_1/conv_0/kernel:0 | (3, 3, 1024, 512)
|
| while/unet_encoder_decoder/decoder/decoder_1/conv_1/bias:0   | (512,)
| 512       | -0.000854 | 0.000882 |
| while/unet_encoder_decoder/decoder/decoder_1/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | 8.59e-05  | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_2/conv_0/bias:0   | (512,)
| 512       | -0.00069  | 0.000849 |
```

```
| while/unet_encoder_decoder/decoder/decoder_2/conv_0/kernel:0 | (3, 3, 1024, 512) | 4,7
|
| while/unet_encoder_decoder/decoder/decoder_2/conv_1/bias:0   | (512,)
| 512       | -0.000812 | 0.00103  |
| while/unet_encoder_decoder/decoder/decoder_2/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -9.62e-06 | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_3/conv_0/bias:0   | (512,)
| 512       | 7.81e-05  | 0.00185  |
| while/unet_encoder_decoder/decoder/decoder_3/conv_0/kernel:0 | (3, 3, 1024, 512) | 4,7
|
| while/unet_encoder_decoder/decoder/decoder_3/conv_1/bias:0   | (512,)
| 512       | -0.000124 | 0.00283  |
| while/unet_encoder_decoder/decoder/decoder_3/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | 6.17e-07  | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_4/conv_0/bias:0   | (256,)
| 256       | 0.000594  | 0.00325  |
| while/unet_encoder_decoder/decoder/decoder_4/conv_0/kernel:0 | (3, 3, 768, 256)
| 1,769,472 | -7.18e-07 | 0.0148   |
| while/unet_encoder_decoder/decoder/decoder_4/conv_1/bias:0   | (256,)
| 256       | 0.000257  | 0.00385  |
| while/unet_encoder_decoder/decoder/decoder_4/conv_1/kernel:0 | (3, 3, 256, 256)
| 589,824   | 6.52e-05  | 0.0209   |
| while/unet_encoder_decoder/decoder/decoder_5/conv_0/bias:0   | (256,)
| 256       | 0.000518  | 0.0039   |
| while/unet_encoder_decoder/decoder/decoder_5/conv_0/kernel:0 | (3, 3, 384, 256)
| 884,736   | 3.45e-05  | 0.0188   |
| while/unet_encoder_decoder/decoder/decoder_5/conv_1/bias:0   | (256,)
| 256       | 0.00096   | 0.00398  |
| while/unet_encoder_decoder/decoder/decoder_5/conv_1/kernel:0 | (3, 3, 256, 256)
| 589,824   | -1.82e-05 | 0.0209   |
| while/unet_encoder_decoder/decoder/decoder_6/conv_0/bias:0   | (256,)
| 256       | 0.000583  | 0.00352  |
| while/unet_encoder_decoder/decoder/decoder_6/conv_0/kernel:0 | (3, 3, 320, 256)
| 737,280   | 6.81e-05  | 0.0197   |
| while/unet_encoder_decoder/decoder/decoder_6/conv_1/bias:0   | (256,)
| 256       | 0.000485  | 0.00365  |
| while/unet_encoder_decoder/decoder/decoder_6/conv_1/kernel:0 | (3, 3, 256, 256)
| 589,824   | -3.91e-05 | 0.0208   |
| while/unet_encoder_decoder/encoder/encoder_0/conv_0/bias:0   | (64,)
| 64        | 0.00105   | 0.00389  |
| while/unet_encoder_decoder/encoder/encoder_0/conv_0/kernel:0 | (3, 3, 3, 64)
| 1,728     | 0.00038   | 0.0568   |
| while/unet_encoder_decoder/encoder/encoder_0/conv_1/bias:0   | (64,)
| 64        | 0.000692  | 0.00379  |
| while/unet_encoder_decoder/encoder/encoder_0/conv_1/kernel:0 | (3, 3, 64, 64)
| 36,864    | -2.14e-05 | 0.042    |
| while/unet_encoder_decoder/encoder/encoder_1/conv_0/bias:0   | (128,)
| 128       | 8.8e-05   | 0.00358  |
```

```
| while/unet_encoder_decoder/encoder/encoder_1/conv_0/kernel:0 | (3, 3, 64, 128)
| 73,728    | 0.000393  | 0.0344   |
| while/unet_encoder_decoder/encoder/encoder_1/conv_1/bias:0   | (128,)
| 128       | 0.00019   | 0.00348  |
| while/unet_encoder_decoder/encoder/encoder_1/conv_1/kernel:0 | (3, 3, 128, 128)
| 147,456   | -0.000135 | 0.0297   |
| while/unet_encoder_decoder/encoder/encoder_2/conv_0/bias:0   | (256,)
| 256       | -0.000327 | 0.00302  |
| while/unet_encoder_decoder/encoder/encoder_2/conv_0/kernel:0 | (3, 3, 128, 256)
| 294,912   | 1.41e-05  | 0.0242   |
| while/unet_encoder_decoder/encoder/encoder_2/conv_1/bias:0   | (256,)
| 256       | 0.000269  | 0.00288  |
| while/unet_encoder_decoder/encoder/encoder_2/conv_1/kernel:0 | (3, 3, 256, 256)
| 589,824   | -0.000193 | 0.021    |
| while/unet_encoder_decoder/encoder/encoder_3/conv_0/bias:0   | (512,)
| 512       | -0.000645 | 0.00216  |
| while/unet_encoder_decoder/encoder/encoder_3/conv_0/kernel:0 | (3, 3, 256, 512)
| 1,179,648 | -0.000342 | 0.0172   |
| while/unet_encoder_decoder/encoder/encoder_3/conv_1/bias:0   | (512,)
| 512       | -0.000341 | 0.00214  |
| while/unet_encoder_decoder/encoder/encoder_3/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.000268 | 0.0148   |
| while/unet_encoder_decoder/encoder/encoder_4/conv_0/bias:0   | (512,)
| 512       | -0.00132  | 0.00175  |
| while/unet_encoder_decoder/encoder/encoder_4/conv_0/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.000308 | 0.0148   |
| while/unet_encoder_decoder/encoder/encoder_4/conv_1/bias:0   | (512,)
| 512       | -0.00164  | 0.00167  |
| while/unet_encoder_decoder/encoder/encoder_4/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.000307 | 0.0148   |
| while/unet_encoder_decoder/encoder/encoder_5/conv_0/bias:0   | (512,)
| 512       | -0.00136  | 0.00201  |
| while/unet_encoder_decoder/encoder/encoder_5/conv_0/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.000205 | 0.0148   |
| while/unet_encoder_decoder/encoder/encoder_5/conv_1/bias:0   | (512,)
| 512       | -0.00193  | 0.00158  |
| while/unet_encoder_decoder/encoder/encoder_5/conv_1/kernel:0 | (3, 3, 512, 512)
| 2,359,296 | -0.000323 | 0.0148   |
| while/unet_encoder_decoder/output/bias:0                     | (256,)
| 256       | -0.000254 | 0.00351  |
| while/unet_encoder_decoder/output/kernel:0                   | (3, 3, 256, 256)
| 589,824   | -4.72e-05 | 0.0209   |
+-------------------------------------------------------------+-------------------

+-----------------------------------+------------+---------+-----------+---------+
| Name                              | Shape      | Size    | Mean
| Std     |
+-----------------------------------+------------+---------+-----------+---------+
```

```
| mlp_geometry_net/dense/bias:0      | (512,)      | 512      | 0.000345
| 0.00372 |
| mlp_geometry_net/dense/kernel:0    | (262, 512) | 134,144 | 2.67e-05
| 0.0509  |
| mlp_geometry_net/dense_1/bias:0    | (512,)      | 512      | 0.000373
| 0.00344 |
| mlp_geometry_net/dense_1/kernel:0 | (512, 512) | 262,144 | 0.000103
| 0.0443  |
| mlp_geometry_net/dense_2/bias:0    | (512,)      | 512      | 5.22e-05
| 0.00346 |
| mlp_geometry_net/dense_2/kernel:0 | (512, 512) | 262,144 | 4.28e-07
| 0.0444  |
| mlp_geometry_net/dense_3/bias:0    | (512,)      | 512      | 9.41e-06
| 0.00359 |
| mlp_geometry_net/dense_3/kernel:0 | (512, 512) | 262,144 | 3.57e-05
| 0.0443  |
| mlp_geometry_net/dense_4/bias:0    | (512,)      | 512      | 0.000113
| 0.0036  |
| mlp_geometry_net/dense_4/kernel:0 | (774, 512) | 396,288 | -0.000135 | 0.0396
|
| mlp_geometry_net/dense_5/bias:0    | (512,)      | 512      | -0.000234 | 0.00353 |
| mlp_geometry_net/dense_5/kernel:0 | (512, 512) | 262,144 | 0.000147
| 0.0443  |
| mlp_geometry_net/dense_6/bias:0    | (512,)      | 512      | -0.000228 | 0.0038
|
| mlp_geometry_net/dense_6/kernel:0 | (512, 512) | 262,144 | -3.85e-05 | 0.0443
|
| mlp_geometry_net/dense_7/bias:0    | (512,)      | 512      | 0.000139
| 0.00401 |
| mlp_geometry_net/dense_7/kernel:0 | (512, 512) | 262,144 | -5.89e-05 | 0.0444
|
| mlp_geometry_net/dense_8/bias:0    | (4,)        | 4        | -0.00656
| 0.00568 |
| mlp_geometry_net/dense_8/kernel:0 | (512, 4)    | 2,048    | 0.00111
| 0.0679  |
+-----------------------------------+------------+---------+-----------+---------+


+-----------------------------------------------------------+------------------+---------+
| Name                                                      | Shape
| Size    | Mean       | Std       |
+-----------------------------------------------------------+------------------+---------+
| while/normals_shading_net/dense_10/bias:0                 | (256,)
| 256     | 0.0022     | 0.00513   |
| while/normals_shading_net/dense_10/kernel:0               | (256, 256)
| 65,536  | 0.0012     | 0.0627    |
| while/normals_shading_net/dense_11/bias:0                 | (256,)
| 256     | 0.00171    | 0.00518   |
```

```
| while/normals_shading_net/dense_11/kernel:0             | (256, 256)
| 65,536   | 0.000938   | 0.0628    |
| while/normals_shading_net/dense_12/bias:0               | (3,)
| 3        | 0.0065     | 0.000654 |
| while/normals_shading_net/dense_12/kernel:0             | (256, 3)
| 768      | 0.0112     | 0.09      |
| while/normals_shading_net/dense_9/bias:0                | (256,)
| 256      | 0.00208    | 0.00533   |
| while/normals_shading_net/dense_9/kernel:0              | (19, 256)
| 4,864    | -0.000479 | 0.0865    |
| while/normals_shading_net/encoder_block/conv_0/bias:0   | (128,)
| 128      | 0.000546   | 0.0035    |
| while/normals_shading_net/encoder_block/conv_0/bias:0   | (64,)
| 64       | 0.00114    | 0.00414   |
| while/normals_shading_net/encoder_block/conv_0/bias:0   | (32,)
| 32       | 0.000658   | 0.00537   |
| while/normals_shading_net/encoder_block/conv_0/bias:0   | (16,)
| 16       | 0.00135    | 0.00401   |
| while/normals_shading_net/encoder_block/conv_0/kernel:0 | (3, 3, 512, 128) | 589,8
|
| while/normals_shading_net/encoder_block/conv_0/kernel:0 | (3, 3, 128, 64)
| 73,728   | 8.47e-06   | 0.0339    |
| while/normals_shading_net/encoder_block/conv_0/kernel:0 | (3, 3, 64, 32)
| 18,432   | 5.61e-05   | 0.048     |
| while/normals_shading_net/encoder_block/conv_0/kernel:0 | (3, 3, 32, 16)
| 4,608    | 0.000312   | 0.0681    |
+---------------------------------------------------------+------------------+-----


The Swish activation we use is simply tf.nn.swish with default values.

Ja, ich spreche deutsch :-)

Viele Gruesse
Thiemo
```