ROSKILDE UNIVERSITY

MASTER THESIS

MASTER OF SCIENCE IN COMPUTER SCIENCE AND INFORMATICS

3D Mesh Super-Resolution using Deep Learning

AUTHOR

FREDERIK ZEILBERGER THULSTRUP

Email: frzeth@ruc.dk
Student number: 58223

June 1, 2020

SUPERVISION BY

MADS ROSENDAHL

Abstract

Super-resolution is the task of creating a high-resolution output from a low-resolution input. Deep learning has proved to be able to achieve super-resolution on images, and now also on some 3D structures. It is, however, not as trivial to perform deep learning on 3D structures compared to images. It has been done on 3D structures such as point clouds and voxels, but not on meshes. Therefore the aim of this thesis is to uncover why deep learning on meshes, in general, is difficult and then to investigate how deep learning can be used to achieve super-resolution on meshes. Because the techniques that are used to achieve super-resolution cannot be directly applied to meshes, other methods were investigated. Firstly it was tested whether a network could learn from the mesh structure alone using a feedforward network. The network turned out to overfit and not to be able to achieve super-resolution. It was thought that it was due to the lack of convolutional layers. Then particular types of convolutional layers working on meshes were tested, but still, the method showed no definite signs of being able to perform super-resolution. It could overfit the data, but generalizing was much more challenging. However, it is believed that with the right network architecture and enough training data, super-resolution on meshes using deep learning is possible.

Contents

Chapter 1

Introduction

Artificial neural networks are machine learning algorithms inspired by how the human brain works, although in a more simplified and more rigidly structured way. Deep learning is a field within machine learning involving particular types of artificial neural networks [38]. It will be covered later why it is called deep learning and how it works. Deep learning has achieved a state of the art performance on several tasks involving images such as image classification with over 99% accuracy [23] and semantic segmentation, which can correctly classify the class of every single pixel with over 80% accuracy [64]. Also, deep learning has made it possible to synthesize new images that look real to the human eye [21], and the resolution of images increased so that details that were never present in the image become visible [30][26]. This technique of improving the resolution is called super-resolution.

There are also examples of deep learning with 3D data performing classification, segmentation, and more [1]. There are many different types of 3D structures, and some of them have a structure where traditional deep learning techniques can be applied, but other structures such as meshes are so different in their structure that other techniques are necessary in order to understand the patterns in the structure [1].

A mesh is a 3D structure constructed from points in 3D space called vertices, edges that connect the vertices and surfaces connecting the vertices and edges called faces [28]. Below in Figure 1.1 is a simple sphere mesh and a monkey head. The triangles are the faces, the corners of the triangles are the vertices, and the lines between the vertices are the edges. It shows the many possibilities meshes can offer as anything should be possible to model.

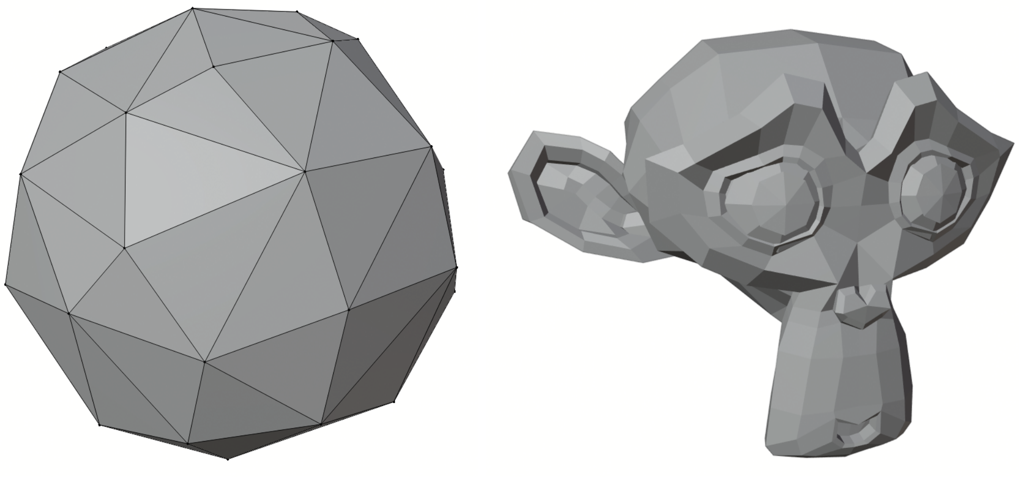

Figure 1.1: A simple sphere mesh and a monkey head mesh (from Blender [11])

The idea of this thesis is to combine the research of deep learning on 3D meshes with the research of super-resolution on different data types. Super-resolution has been achieved on some types of 3D structures [63][49], but to my knowledge, not on meshes. Therefore this thesis will investigate whether it is possible to achieve super-resolution on 3D meshes using deep learning.

Below in Figure 1.2 are four meshes: A low-resolution mesh with 100 faces, a high-resolution mesh of 400 faces, and two meshes that are smooth versions of the low-resolution mesh. MeshLab [3] is a software that can smooth meshes using different algorithms. This smoothing is actually increasing the number of faces of the meshes and this is called subdivision. The algorithms Butterfly Subdivision and L3S loop Subdivision created the smooth versions of the low-resolution mesh and in the process increased the faces to 400. There are several other types of algorithms, like the two. It is, however, clear that there is a big difference between the original high-resolution mesh and the modified low-resolution meshes even though they all have 400 faces. Note that the two modified meshes look more smooth than the original meshes, and this is because the edges were smoothed out by MeshLab and it was not known how to disable this smoothing. The two new meshes contains rounded faces instead of flat faces which makes them appear smoother. Although when looking at the outline of the meshes, it is clear to see the difference. Especially in the top part of the meshes.

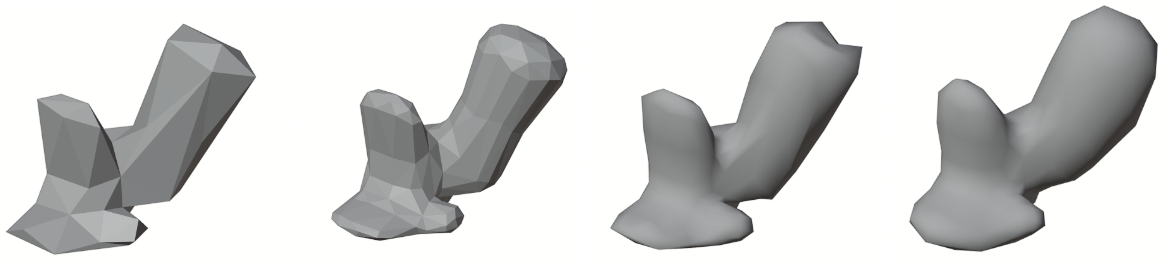
Figure 1.2: A model presented as a low-resolution mesh (first from left), a high-resolution mesh (second from left), the low-resolution mesh subdivided with Butterfly Subdivision (second from right) and the low-resolution mesh subdivided with L3S Subdivision (first from right)

It is not only the number of triangles that define the detail level of a mesh. It is also the placement of the vertices. Smoothing a mesh using known methods, as shown above, is not the aim of this thesis. What this thesis aims for is to make artificial neural networks learn to add more detail to a low-resolution model to achieve a high-resolution model. It is a type of displacement mapping which is subdividing a mesh's faces and moving its vertices around to achieve a more detailed look [59]. Displacement mapping is used to achieve higher resolution graphics in DirectX 11 [8]. It is this displacement mapping from the low-resolution mesh to the high-resolution mesh that is the goal. The specific problem statement this paper will investigate is:

"How can deep learning be used to achieve super-resolution on 3D meshes?"

By answering this research question, this thesis could contribute to the current research in multiple ways: By providing an overview of the current research of how deep learning algorithms work with meshes, by providing an overview of current ways to achieve super-resolution on multiple types of 3D structures, and by investigating different ways, super-resolution can be applied to meshes using deep learning and compare them with each other.

The structure of this thesis unfolds the following way: In Chapter 2, the theory about machine learning, deep learning, and 3D structures is covered. In Chapter 3 the current research about super-resolution on images, and multiple 3D structures will be covered as well as an overview of deep learning on meshes. Then Chapter 4 will discuss and decide the methodology and outline the experiments. Chapter 5 will cover the experiments and the results. A discussion of the results is in Chapter 6. And at last, the thesis will present its conclusion in Chapter 7.

Chapter 2

Theory

This thesis concerns two subfields of computer science; machine learning [58] and computer graphics [2]. More specifically, deep learning within the field of machine learning and 3D structures within the field of computer graphics. The goal is to combine the fields to investigate options of how deep learning could improve the quality of meshes. The following sections will describe the two fields. More specifically, the sections will cover machine learning, deep learning, and a selection of relevant 3D structures. This chapter also serves as the foundation to understand the related research, which will be presented in Chapter 3.

2.1 Machine learning

Deep learning is a sub-field of machine learning [12], and therefore it is viable to know the basics of machine learning before diving into details about deep learning.

2.1.1 Machine learning algorithms

Machine learning is when machines can learn from data. When a machine can improve its performance on a task by looking at data, it means it can learn from it. Machine learning has a wide range of applications such as classification of images, regression analysis, transcription of voice to text, translation between languages, and many more. Depending on the task the machine learning algorithm has, it needs a performance measure, and this can vary from task to task. When developing a machine learning algorithm it is essential to define the measures that make it successful [12].

There are different types of Machine learning algorithms: Supervised learning where the data has labels that help the machine understand what the data is. A label can be many things such as a binary value, a categorical value, an image, or a string of text. This label helps the algorithm know what the correct answer is. Then there is unsupervised learning, which only gets data without label and can, therefore, only learn patterns and cluster the data. If some of the data is labeled and some not, it is called semi-supervised learning. Furthermore, there is reinforcement learning, which learns by doing an action and getting feedback on that action [12]. This thesis focuses on supervised learning since the data will be labeled. A low-resolution mesh will have a high-resolution mesh as its label.

2.1.1.1 Training and testing

A trained supervised machine learning algorithm should be able to process unseen data just as well as seen data. Being able to work on unseen data is called generalization. The algorithm should have access to training data and test data. The algorithm will only be able to learn from the training data but will use the test data to test its ability to generalize. The first goal of training a machine learning algorithm is to make it perform well on the training set. The second goal is then to make the difference between the performance on the training set and test set as small as possible. When a machine learning algorithm is capable of generalizing, it has proved that it learns rather than just memorizing. The performance of a machine learning algorithm is defined by something called a loss, which is a value that in supervised learning is calculated using the algorithms output and the labels [12].

Overfitting and underfitting
There are two terms to describe a trained machine learning algorithm that does not perform optimally. Underfitting is when the algorithm cannot learn to minimize training loss. This means that the algorithm cannot even memorize the training data, and changes to the algorithm are necessary. Then there is overfitting, which is when the test loss is higher than the training loss. Overfitting means that the algorithm has memorized some or maybe all of the training data. It is now essential to figure out what changes to the algorithm that could reduce overfitting. The ultimate goal is not to underfit or overfit the algorithm to achieve generalization and produce useful results. A trained machine learning algorithm is essentially a model that takes an input and produces an output [12].

Model capacity
The number of learnable parameters in a model defines its capacity to learn. Each parameter adds a new dimension to the model hence creating more possibilities to represent the data. If there are too many parameters, the model is likely to overfit since it will memorize the training data, and if there are too few parameters, the model cannot learn from complex data [12].

2.1.1.2 Hyperparameters

The machine learning algorithm has parameters it can learn with, but it also has some hyperparameters that can be adjusted by the developers. The hyperparameters are some settings that affect the way the algorithm learns and can make it learn better and faster if adjusted correctly. The number of parameters in the model is a hyperparameter in itself, and so is the learning rate, which is a scalar of how quickly the model should learn. There are many different hyperparameters, and they will be covered more in-depth in the following section. When a learning algorithm is altered in any way to achieve better generalization performance, it is called regularization, and there are many different ways of doing this [12].

2.2 Deep learning

Deep learning algorithms are types of machine learning algorithms that are powerful at recognizing patterns that are usually hard for other algorithms to find, such as patterns in images [12]. In order to understand deep learning, it is necessary to know what artificial neural networks are first.

2.2.1 Artificial neural networks

Artificial neural networks are inspired by how the biological brain works. The brain is a network of neurons that are cells able to process information and pass it on to other connected neurons. This network of neurons is the inspiration for artificial neural networks, which is often called neural networks within computer science. In order to understand how a neural network works, it is necessary to know how these artificial neurons work [48].

2.2.1.1 Neurons

Neurons are small processors that process several numerical inputs and give one numerical output. Between each connected neuron, there is a connection. This connection is weighted. The information that flows through gets multiplied by a weight. When the neuron receives its weighted inputs, they are added together and sent through an activation function, which gives a final output. There are many different activation functions, and they each have their strengths and weaknesses [48].

Activation functions
A commonly used activation function is a rectified linear unit, which is also called ReLU. This activation function outputs the input directly for positive values and outputs zero for all negative values. It seems to work well in many cases, and it is recommended as one of the default activation function for most tasks [12]. However, sometimes it is necessary to get negative values from an activation function, and then a standard linear function can be used. Another common activation function is sigmoid, which returns a number between zero and one. Sigmoid used in the output layer can serve as a probability measure. There is also a variant of ReLU called Leaky ReLU (LReLU). Instead of returning zero for all values under zero, it returns the input multiplied with a small number [38]. Then there is another variant of ReLU called Parametric ReLU (PReLU) which returns linear values when the output is positive, but the negative values are multiplied with a variable that is learned through training [33].

A network of neurons
Because the brain's neural network structure is complex and hard to model in a computer, the idea has been simplified into a layered structure. This structure consists of an input layer, at least one hidden layer and an output layer. The input layer takes in the information that the network has to process, sends it through the hidden layers, and finally, the output layer produces a result, which is a vector with a size depending on the number of neurons in the output layer [48].

2.2.1.2 Deep neural networks

Deep learning is a machine learning approach utilizing deep neural networks. The "deep" in deep learning refers to the depth of the neural network, and must have multiple hidden layers before it is a deep neural network. With each layer of neurons in the network, the network becomes deeper and more advanced. The layered structure described until now where information flows only one direction is called deep feedforward neural networks. There are other types of networks where the flow is different, but they will not be covered in this thesis. Each layer has a bias, which is a constant that is often one, multiplied with a weight. The bias multiplied with its weight is added to the sum of the weighted neurons, and then sent through the activation function. This bias can shift the possible outputs of the activation function. The weights for all the neurons should be initialized to small random values, and the bias weights to small positive random values or just zero. The weights of a deep neural network are what makes it able to learn since these weights are all variables and can be adjusted to give the best output. Adjusting the weights is done by training the network [12].

2.2.1.3 Training a neural network

In the previous sections, it was covered how a neural network works. It receives input and gives output, but before the network is trained, the output will not make sense. Training a network is when the weights are adjusted in such a way that the output comes closer to an expected result.

The weights get adjusted only a little every time, and it takes many iterations before the output makes sense [12].

Loss functions
The loss determines the performance of a neural network, and a loss function calculates this loss. The loss is an indication of how close the network output is to an expected output. The loss function returns a single positive number, and the closer to zero it is, the better. There are many different types of loss functions that are good for each their task. A common loss function is mean squared error, which gives us the average of the squared differences. Training essentially revolves around optimizing the network's weights so that it gives a lower loss. It as an error measure between the network's output and the expected output. [38]. Relevant loss functions concerning meshes will be covered later.

Optimization
The way the optimization of the weights works is basically to take the derivative of the loss function to find the slope of it. Then the parameters (weights) to that function can be adjusted so that the loss is lower in the next iteration. The weights are adjusted with an algorithm called backpropagation, which will be covered later. Optimization algorithms have a variable called the learning rate, which controls the magnitude of the changes made to the weights. There are different methods for how the optimization should take place. One of the most known methods is gradient descent, which has to calculate all training samples' losses before adjusting the network. Gradient descent ensures that it makes a change to the network in relation to all training samples, but there is also a method called stochastic gradient descent, which only looks at small batches of the training data or even a single training sample before adjusting the network. It has in many cases been proven to work more efficiently than gradient descent. Improvements have further been made to stochastic gradient descent by adding more variables to the algorithm, such as an adaptive learning rate utilizing acceleration and momentum [12]. An example of such an optimization algorithm is Adam which will be used in this thesis since it has proved to work well on many different deep learning tasks.

Backpropagation
Backpropagation is the math behind how the weights are adjusted through the network. They are adjusted from the output layer to the input layer. Taking the derivative of the loss function is, in fact, a composite function because data from the input layer goes through multiple layers, and each neuron in a layer is a function itself. Taking the derivative of a composite function requires the use of the chain rule from calculus. The derivative gives the slope of the activation functions, and they indicate if the weights should be adjusted up or down. The slope will also provide information about the magnitude of the change to make.

Minima and saddle points
When the network is being adjusted, it is to minimize the loss of the loss function. This loss function can have local minima, a global minimum, and saddle points. Local minima are when the network has been adjusted in such a way that any optimization will lead to a worse higher loss. It looks as if the model is perfect, but it is just locally the best, not globally. It is like being stuck in a hole on top of a hill. The global minimum is the absolute best state the model can be in, and therefore any further optimization will make the model worse. Saddle points are when some adjustments make the loss go up, and some make it go down, which is very common when there are many variables. The scenario to be feared most is the local minima as the optimization might get stuck here unless it can break out of these minima, which depends on the learning rate to be large enough. A too big learning rate can make it jump over a minimum, because it takes

too big steps. Therefore a learning rate should not be too big or too small [12].

2.2.1.4 Regularization

Regularization has been mentioned briefly earlier. It is different changes in a machine learning algorithm that increases its generalization ability. This means reducing the loss of unseen data. A machine learning algorithm can have an easy time overfitting, since it basically memorizes the training data, and then struggle with unseen data. Regularisation may decrease the training loss to increase the test loss, but the test loss is all that matters [12]. Below are some regularization techniques specific for deep learning algorithms that are relevant for this thesis.

Batch normalization
Batch normalization reduces training time, along with being a regularization technique. It works by normalizing a batch of data so that the mean is zero, and the variance is one. Instead of having very different values as input in the layers, they now all take in a number between minus one and one, and it makes it able to learn faster [19].

L2 parameter regularization
The L2 parameter regularization is also called weight decay. It is a modification to the optimization of the loss function. It alters the weight adjustments in such a way that the weights with little covariance with the output get penalized [12].

Dataset augmentation
Dataset augmentation is a way to make more training data from a dataset. It is a way to modify the current data to seem different for the algorithm. It is often done with images, and examples of image augmentations are: translating, scaling, rotating, and tinting of the images. Generally, noise can be applied to any kind of training data to generate more training data [12].

Early stopping
It can be a good idea to stop before the training loss has reached its minimum since this will make the network less likely to overfit. When the network stops improving over several epochs, training is stopped. It can be useful to look at the test loss and stop training once then test loss stops improving [12].

Dropout
Dropout is a way to regularize where each training epoch will exclude a random set of nodes in the network. This means there is less information being passed through the network. The excluded nodes will only be excluded from one weight adjustment. This can ensure that the model does not overfit since it will not be able to memorize the training data directly [12].

2.2.2 Network types

So far only feedforward neural networks has been accounted for. There are many different types of networks that work best on specific tasks. Below are some of the most well-known types of neural networks.

Convolutional neural networks
Convolutional neural networks (CNN) are feedforward networks with one or more convolution layers. The input data to a convolutional layer has to have a relation to each other spatially as a grid has. It has to be the case that an entity in a grid has a relation to its neighbor entities. An image is a good example of this since only the pixels pieced together form meaning. Also,

time-series data, which is a list of data points, have some relation to each other in the form of past and future.

Convolutional layers make the network able to see the bigger picture. The way this works is by detecting simple features in the first layer, and the following layers will then detect more and more advanced features. An example with images could be that the first layer detects edges of various directions, the following layer combines these edges to form shapes, and the next layer combines these shapes, which then can be used to classify an image.

The convolutional operation takes in the input with one or more channels. An RGB image has an input size of width and height along with three channels for the colors. The channels are the depth of the matrix. The result of a convolutional operation is a matrix with the same width and height, but with any number of channels. Each of these channels is found by using a unique filter. A filter is a window that extracts features of a small selection of pixels. This window is slid on the whole input space, and this is done for all the filters. Each filter has the ability to detect different patterns. In the first layers, they can be simple edges and curves, and in the later layers, more advanced structures.

Another layer that often follows a convolutional layer is a pooling layer. It works by defining an area that should be pooled together to form a smaller matrix. It could be a 2x2 grid becoming a 1x1 grid which reduces the size from four pixels to one pixel. There are different pooling options. The average can be taken, the maximum or something else, but is it often these two that are used [12].

Additionally there exist operations that does the opposite of the convolution and pooling operations. They make it possible to upsample data. Convolutional layers would take the values through a filter and compute one number, then put in a cell in the result matrix and finally slide the window. A deconvolution would only look with a small filter and produce more values and put them in the result matrix. The unpooling adds padding around the data in a way that there are zero-valued entries around each input entry [56].

Graph convolutional neural network
A Graph convolutional neural network (Graph-CNN) is a type of CNN that works on graph data. Graph data is data where any data point can have a relation to each other. They are not positioned in a grid like a regular CNN, but instead the relation between the points is defined [66].

Residual neural networks
Residual neural networks (ResNet) have proven to work well on very deep neural networks. They work by having skip connections between the layers so that that data can move directly past a layer [15].

Autoencoders
Autoencoders are neural networks that aim to learn how to reconstruct data from a few features. It is used to extract the most significant features of the data. The network's structure is that its input layer starts with a number of neurons, followed by one or more hidden layers, but with fewer neurons than the input layer and at last, the output layer has more neurons than the hidden layers. It has a structure like an hourglass. After training such a network, it can learn to reconstruct the input data even with a significant bottleneck in the hidden layer [12].

Generative adversarial networks
General adversarial networks (GAN) are networks consisting of two separate networks: a gen-

erator that generates data and a discriminator that judges data. The generator can learn to synthesize new data that looks real. The way it learns this is by trying to fool the discriminator network. The discriminator network's task is to figure out whether the data is fake or real. The discriminator will, therefore, be presented with real data and fake data. It is essentially a classifier that decides between the two classes, real or fake [12].

2.2.3 The methodological approach to deep learning

This section describes how deep learning architectures can be done efficiently. Designing deep learning architectures are challenging because there are so many parameters to tweak, and the effect of each of them cannot fully be known without trying. Therefore it quickly becomes time-consuming and hence it is important to tweak the hyperparameters in a smart way. Improvements to the network could be achieved by getting more or better data, increasing or decreasing the size of the network, finding the most appropriate loss functions, finding the right regularizers and finding the right optimizer with the right learning rate. It can also be to debug the code if there are bugs in it. These points are just some of the ways of how models can be improved when conducting deep learning experiments [12].

The following process [12] of conducting experiments can be employed. It is a number of recommendations that can be used to navigate experiments in the right direction:

- Figure out the goal for the experiment, what makes a model successful, and what the metrics should be.
- Get a model up and running and start experimenting. Start simple, and then expand. Try different optimizers and regularizers.
- Figure out if more data is needed. If results are good on the training data, but not on the test data, then more data can help. If it performs badly on training data, then the model should be improved first.
- Improve the model by tweaking the hyperparameters correctly. Some will improve training time, and some will improve the quality of the output. The adjustments of the hyperparameters can be done either manually or automatically.
- Debug the code. Look at the output of the network and determine whether or not it makes sense. It can be a good idea to see if it is even possible to overfit a tiny dataset first.

These points will be used in the Chapter 5 to help conducting the experiments.

2.3 3D structures

3D models have a wide range of applications, such as being used when making 3D games in the game engine Unity [52], making 3D animations in Blender [11], and Ikea has an app with 3D models of their furniture [16]. These are just examples of the many applications of 3D models. To represent these models, there are multiple different 3D data representations with each their strengths and weaknesses. The types of 3D data can be split into two groups. Euclidean and non-euclidean [1].

When data is euclidean, there is a spatial relationship between the data points. It means that the data point itself along with its placement in relation to other data points both matter. Examples of euclidean data are RGB images, RGB-D images, which are RGB images with a depth map, Multi-View images, which is a series of images of an object from different angles and volumetric data, which is a 3D grid of occupied and unoccupied cells which will be explained in the next sections. Since euclidean data follow a grid structure, 2D Deep Learning practices such as regular convolutional neural networks can be more or less directly applied to this type of data [1].

When data is non-euclidean, there is no fixed structure in how the data points relate to each other. Examples of non-euclidean data are meshes, which consist of vertices, edges and faces, and point clouds, which consist of points. They will both be explained in the following sections. Since the data is not structured in a grid format, the deep learning practices such as regular convolutional neural networks cannot be directly applied [1]. There is some research on how to make a type of convolutional neural network that works on meshes [14] and point clouds [43], which will be explained in-depth in the next chapter.

Meshes are the 3D structure that will be focused upon in this thesis with the goal of achieving super-resolution using deep learning. Since it is possible to convert between meshes, point clouds, and voxels [20], all three structures are included. One option to achieve super-resolution on meshes could be to perform super-resolution on a point cloud or voxel and then converting the result back into a mesh. However, this option is excluded from this thesis. Figure 2.1 illustrates a comparison between the three selected structures.

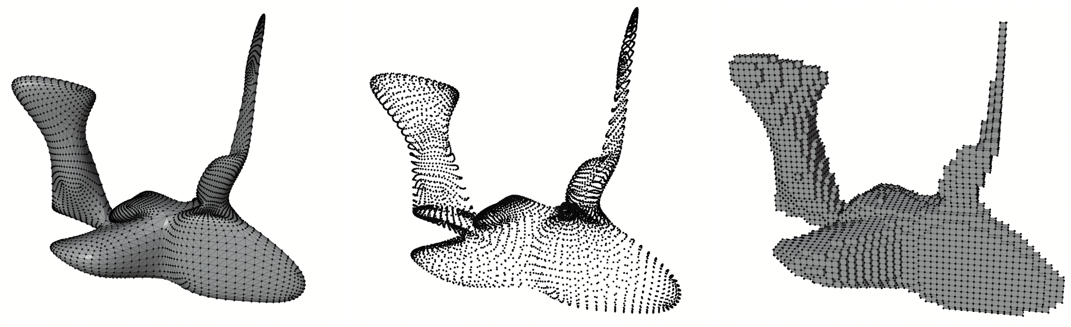

Figure 2.1: A mesh (left), a point cloud (middle) and a voxel (right)

The following sections will cover each of the three structures explaining how they are structured along with their strengths and weaknesses.

2.3.1 Meshes

Meshes are a widely used 3D structure and are, for example, used as a standard 3D structure in Unity [52], and Blender [17]. Meshes are constructed of vertices, edges, and faces [28], and is a non-euclidean data structure [1]. A vertex is a point in a 3D space. An edge is a connection between two points and forms as a straight line. A face is a surface where three or more edges form a connection. A face with any number of edges can be "triangulated" into triangular faces without losing detail [55]. Faces also have a property called normals, which indicates which direction the face is facing [34] and can be used to ensure that light is cast correctly onto the model when rendering [31]. A mesh can be seen in Figure 2.1 above.

2.3.2 Point clouds

A point cloud is a number of points that together form a shape. There is no defined structure between the points by itself like meshes have faces and edges, but holistically it forms a 3D shape when viewed. It is non-euclidean data [1] for this reason. Each point consists of an x, y, and z coordinate and can be extended with a color property. Some ways to obtain point clouds can be through 3D scanning [46] or by sampling points on the surface of a mesh [44]. A point cloud can be seen in Figure 2.1 above.

2.3.3 Voxels

Voxels are 3D structures that are based on a grid structure in 3 dimensions. It is, therefore, euclidean data [1] like images. The grid represents which space is occupied and unoccupied and therefore uses data storing what is not in the model. This is an inefficient way of storing 3D data and can end up taking more memory to process than alternative 3D data structures. Another flaw is that they need a very high resolution to represent smooth surfaces. The good thing about voxels is that the same deep learning techniques that work on images can be directly applied to voxels since it follows the same structure just with an extra dimension. Another drawback of voxels is that they require huge computational resources when they get big enough [1]. Voxel graphics are well known from the game Minecraft [29]. A 3D voxel model can be seen in Figure 2.1 above.

2.4 Summary

This chapter has covered the basics of machine learning, which is essential to understand deep learning. Relevant topics within deep learning have been covered to be able to understand the state of the art research that is presented in the next chapter. Also, a methodological approach when doing deep learning has been outlined. Furthermore, the most fundamental 3D structures, meshes, point clouds, and voxels have been explained and visualized as they are important to understand in the following chapter that will dive into current research related to the research question.

Chapter 3

Related research

With the knowledge from the previous chapter about machine learning, deep learning, and 3D structures, it can now be used to understand the relevant state of the art research about super-resolution in both 2D and 3D and to give an overview of existing research about 3D deep learning on meshes. The findings from this chapter will be used to select the experiments that will be conducted. This will be discussed in Chapter 4.

3.1 2D super-resolution

The inspiration for this thesis initially came from the current research on super-resolution on images. The current methods can achieve very good results that are almost as good as an original high-resolution image [26][30]. Openly available research shows how super-resolution can be achieved using a combination of a GAN, ResNet, and CNN's. This research covers the deep learning architecture that can learn to generate a high-resolution image from a low-resolution image input. The results are promising and almost as good as the original high-resolution image when shown to a group of people. The loss function used was a combination of the loss between the original high-resolution image and the super-resolution image along with the discriminator's result. The networks took the use of PReLU in the generator and LReLU in the discriminator [26].

3.2 3D super-resolution

3D super-resolution has in fact been done multiple times as it will be described in this section. It has been done on point clouds and voxels.

3.2.1 Point cloud super-resolution

The currently best performing point cloud super-resolution method [36] takes use of a combination of a GAN, a Graph-CNN, and a ResNet. A point cloud can be represented as a graph and can then be used in a Graph-CNN. The Graph-CNN allows for unpooling and pooling, which is used in the generator and discriminator networks. The generator upsamples the low-resolution input progressively over multiple unpooling layers. The discriminator pools on the input features until an output of two classes, real or fake, has been determined. The point cloud is converted into a graph format by finding 'k' neighbors of each of the points. The value k was set to eight, meaning there were eight neighbors of every point in the point cloud. In many of the tests made, this research exceeded previous research, which is especially evident in the generated super-resolution point clouds that clearly look better than other research results [63].

3.2.2 Voxel super-resolution

On voxels, there has been an approach on how to achieve super-resolution by extracting six depth-maps of a low-resolution voxel from different angles and then performing image super-resolution on these depth-maps. Then these super-resolution depth maps can be used to carve the low-resolution voxel to achieve a high-resolution voxel. Their depth map super-resolution relies heavily on research on image super-resolution. The smart aspect about this solution is that it avoids 3D voxels from entering a neural network, which can be resource-heavy. The method is called Multi-view Decomposition. The results show that this method is able to achieve super-resolution on voxels. The super-resolution voxels are vastly more detailed than the input voxels [49].

3.3 Deep learning on meshes

There is plenty of research on deep learning on meshes. Below, the research is split into two subsections. One section about mesh classification where a mesh is given as input and the output is a class prediction. And another section about mesh synthesis and deformation where the input can be in all sorts of formats, but a mesh or deformations to a mesh is given as output. Mesh deformations are where the vertices of a mesh is shifted, but the faces stay connected the same way.

3.3.1 Mesh classification

There are multiple methods of doing convolutions on meshes, and it is currently used to classify meshes.

3.3.1.1 MeshNet

MeshNet is a neural network capable of classifying meshes. The method consists of multiple parts: A spatial descriptor which takes in the vertex coordinates, and a structural descriptor which takes in the three vectors that are from the center of a face to each corner. The structural descriptor also gets the normals of the faces. The information is processed in each of these descriptors, and it is passed to a mesh convolution. The mesh convolution can then use these spatial and structural features of the mesh and do convolutional operations as well as pooling operations. The results show a classification accuracy of over 90% [10].

3.3.1.2 MeshCNN

MeshCNN is another method that is able to do convolutions on meshes and can classify as well as segment meshes. Instead of using regular CNN's where a pixel is the center of the convolution, an edge is the center of the convolution in MeshCNN. A mesh will be encoded in a way so that it becomes a 5 x n dimensional vector where n is the number of edges in the mesh. The five features of the encoded mesh are the angle between the two faces that are connected by the edge, the two outer angles of each face connected to the edge (the corners that are not connected to the edge), and the shortest distances from these corners to the edge. This 5 dimensional vector is ordered in such a way that it is known what the neighbor edges are of each edge. The results show a classification accuracy over 90% [14].

3.3.2 Mesh synthesis and deformation

Below is a selection of recent research on different ways of synthesizing and deforming meshes using neural networks.

3.3.2.1 Mesh autoencoder

There is a method for generating 3D meshes with the use of an autoencoder network. The encoder part of the network learns to parameterize the input meshes into arrays of 100 parameters, and the decoder is then able to generate the mesh again. Changing these 100 parameters will deform the output. The way the generation of the output mesh works is by calculating depth maps of the input mesh and then fitting a simple mesh within this depth map. This simple mesh is then subdivided until it has reached an optimal resolution and fits perfectly within the depth map. The method was only used on car models and is therefore limited to this class, but it proves that it is possible to use an autoencoder for meshes [57].

3.3.2.2 Scan2Mesh

Scan2Mesh aims to construct a mesh from range scans, which is depth images. The meshes are made by using a neural network to predict vertices and edges, and then they use another neural network to predict the faces, which results in a complete mesh. The method outperforms other surface reconstruction methods, which is the task of finding faces for a point cloud. The method employs a graph neural network in order to do convolutions on the range scans [5].

3.3.2.3 Pixel2Mesh

Pixel2Mesh is a method employing graph neural networks that can construct a mesh from a single image input. It is done by iteratively deforming a simple mesh, which at first is an ellipsoid. By deforming a mesh, it ensures that all the edges and faces are connected correctly and therefore form a valid mesh with no gaps or overlapping faces. It is only the placement of the vertices that has to be changed. Along with the deformations, an unpooling operation is used to create more faces. It is done by inserting vertices in between the already made vertices and connecting the faces between them. It still assures a valid mesh. Four mesh specific loss functions were employed to judge the output mesh' correctness. They were chamfer distance, edge length, normal consistency, and laplacian losses. These will all be covered in Chapter 4. The results show that the network is able to generate 3D meshes representing the input image, and it seems to work quite well [61]

3.3.2.4 Image2Mesh

Image2Mesh is another network that can learn to generate a mesh from a single image. In this case, it is achieved by using the neural network to find a 3D model of a set of models that looks most like the image and then deform this mesh. It performs really well, but it is also dependent on having a good model catalog that makes it easy for the deformation to take place [39].

3.4 Summary

This chapter has covered relevant research starting with 2D super-resolution and then diving into 3D super-resolution on point clouds and voxels. Then current research on deep learning on meshes has been covered. The research in this section, along with the theory, lays the foundation for the research question of this thesis to be answered.

Chapter 4

Method

This chapter's purpose is to explain the overall method of the experiments, choose the dataset and explain the tools and the code. Then the most interesting findings from the previous sections will be extracted, and the foundations will be set for the experiments that can help to reach the goal. The goal is to explore how deep learning can be used to achieve super-resolution on meshes which is a complex 3D data structure, which therefore requires special methods.

4.1 Overall pipeline

The pipeline of the experiments requires a set of high-resolution models at first. These models will then be decimated to create a low-resolution twin model. Decimation reduces the face count of a mesh in a way that it loses the least amount of detail possible and it is done using Blender [7]. A neural network is then fed with the low-resolution model, and the output of the network is used to create a super-resolution model. The super-resolution model is a prediction of how the low-resolution model would look like as a high-resolution model. This super-resolution model is then compared to the actual high-resolution model, which then gives a loss. This loss is then used to adjust the weights in the neural network. Below in Figure 4.1 is an image visualizing the pipeline.

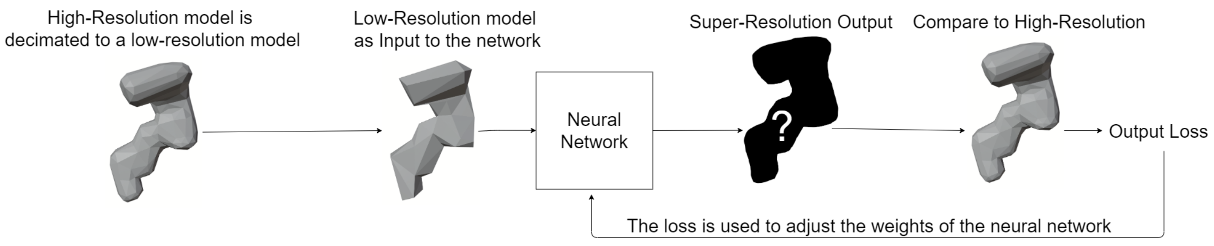
High-Resolution model is decimated to a low-resolution model
Low-Resolution model as Input to the network
Neural Network
Super-Resolution Output
?
Compare to High-Resolution
Output Loss
The loss is used to adjust the weights of the neural network

Figure 4.1: Overall pipeline

In the following section, it will be discussed which dataset should be used in the experiments.

4.2 Dataset

As a start, it would be satisfactory if super-resolution could be applied to simple 3D models, and then later to be tested on more complex models. Therefore the goal is firstly to find a dataset with simple shapes of great variety. It does not matter if the models are representing something meaningful. If the super-resolution could work on randomly looking models, there is a high possibility that it could work on other models.

4.2.1 Available 3D datasets

When searching for a dataset, there was one by Google [47], which initially seemed perfect. It was a 1000 procedurally generated models, and hence it seemed perfect for the task. However, many of the models had very few faces due to its sharp edges. Since the number of faces was too few, another dataset was needed.

There are some very popular 3D datasets such as ModelNet [65] and ShapeNet [51]. However, these were avoided because many of the models were of the same class. ModelNet has 40 different classes, and ShapeNet has 55 classes. The optimum would be to have a lot of different classes and only a few samples of each class. None of the found datasets could deliver this kind of variety, and it was considered to create a custom dataset.

4.2.2 Creating a custom dataset

At some point, a Blender plugin "Shape Generator" [22] was found that made it possible to generate random shapes. It was decided to use this plugin to generate as many models as needed. The shape generator plugin could generate all kinds of shapes. The settings were limited so that the models did not get too simple (with very sharp edges) and not too complex (with too many unnecessary faces). With the settings, found to be best, the models ended up having animal-like shapes, and they were still varying a lot from each other. Below are ten of the models that were created.

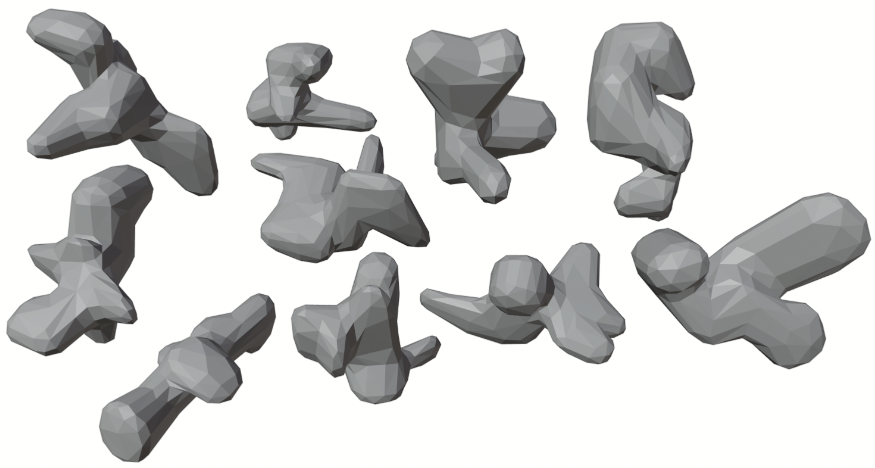

Figure 4.2: A sample of ten models from the dataset

5000 models were generated because it was around the maximum amount of models Blender could handle on the computer that was used without crashing. If more models were needed, more could simply be generated. They are available from Github:
github.com/frederikzt/Master-Thesis-3D-mesh-super-resolution

4.2.2.1 Creating the dataset

With the 5000 models created, the only thing left to do with the dataset was to create a high-resolution and low-resolution version of each model. Since all models had at least 400 faces, it was chosen that the high-resolution models should have 400 faces. It was then chosen that 100 faces should be the number of faces for the low-resolution meshes. By choosing 100 faces, it is possible to subdivide the faces into 400 because one triangle can be split into four triangles. Then the low-resolution meshes would only require mesh deformation in order to achieve the high-resolution mesh. The models were decimated in order to achieve the desired amount of faces.

Below in Figure 4.3, is a illustration of the differences between the two resolutions.

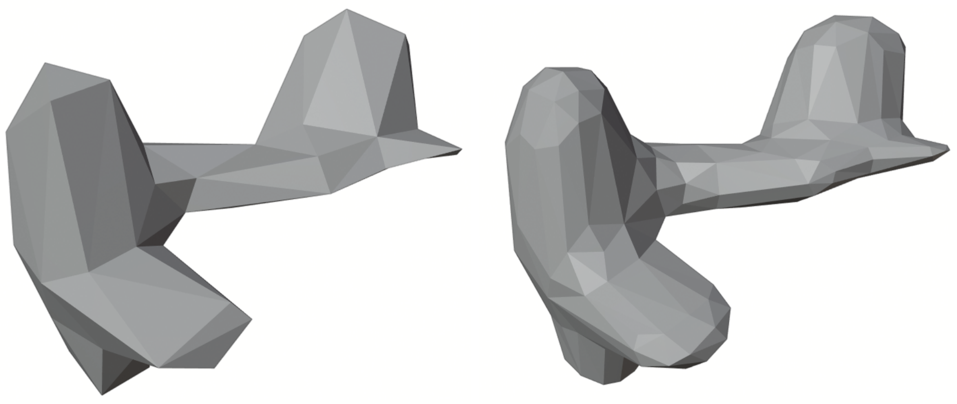

Figure 4.3: The low-resolution mesh (left) and the high-resolution mesh (right)

It is clear to see that some details have disappeared in the low-resolution model. It is mainly at the round edges of the model that detail has been lost. It can be argued that what is aimed at achieving is "de-decimation". It is what the network learns, but learning de-decimation would make it possible to add detail to unseen data, which is the goal.

4.2.2.2 Train and test set

It was chosen to split the 5000 models into a training set of 4500 models and a test set of 500 models. More training models gives better training, but a good test can not be done with too few test models.

4.2.3 Dataset size

The dataset is only 5000 models, which are not many samples compared to what is normally used. As an example, the MNIST dataset [25] has 70000 images. It might be limited to what that can be achieved with only 5000 models, but there is an option to generate more data if needed.

4.3 Tools, libraries and code description

This section will be about what tools are used as well as describing the structure of the code.

4.3.1 Tools

The code is written in Python [41] using Jupyter Notebooks [40]. These tools make it possible structure an experiment and to be able to run code blocks easily. When changes are made to some hyperparameters, it will be easy to re-run the necessary blocks to train the network.

To run the notebooks, Google Colab [13] is used because the only GPU that was access to was an NVIDIA MX150, which is not a strong GPU. Also, it was made sure that the training of the networks ran on the GPU rather than the CPU. It was found to be around 60 times faster when running on the GPU rather than the CPU.

Google Colab also has the strength of being the only way to run some libraries, such as TensorFlow Graphics[54]. However, there is a weakness of Google Colab, which is that it is required to reinstall external libraries every time connection to their servers is lost, and there is a limit of how long time users can be connected. Installing these libraries can take over 20 minutes because some of them have to be built from source.

4.3.2 Libraries

As explained above Python will be used as the programming language. Below is a list of the python libraries that will be used.

4.3.2.1 Deep learning specific libraries

It was chosen PyTorch [37] as the deep learning framework because there were found multiple 3D deep learning libraries that use PyTorch as their primary driver. PyTorch3D [44] is a framework extending PyTorch that has ways to encode meshes into tensors that PyTorch can use, and it also has the mesh loss functions that are needed. Kaolin [20] is an alternative to PyTorch3D, but it was hard to install correctly, and therefore, was PyTorch3D chosen. The four loss functions from PyTorch3D that will be used are chamfer distance, mesh edge length, Laplacian smoothing, and mesh normal consistency. These are the same loss functions used in Pixel2Mesh [61].

Chamfer distance
Chamfer distance is the distance between two point clouds. Imagine two point clouds; point cloud A and point cloud B. The distance is then calculated by finding the distance of each point from one point cloud to the closest point of another point cloud. These distances are then averaged, and this value is the chamfer distance. In order to use chamfer distance as a loss function for meshes, point clouds need to be sampled from the meshes. The loss function also has the ability to calculate the chamfer distance between the normals of two point clouds. This means we can make sure that the angles of the normals are facing the correct directions [62].

Mesh edge length
The mesh edge length is a function that indicates how close the average length of the edges of a mesh is to a target length. By selecting the target length to zero, it will encourage edges to have the minimum distance possible [62].

Laplacian smoothing
Laplacian smoothing is a method that can be used to smooth meshes [24]. This loss function calculates the current level of laplacian smoothness in a mesh [62]. Laplacian smoothing is done by adjusting the vertices position in a mesh so that they are more evenly distributed [60].

Mesh normal consistency
Makes sure that normals are consistent with its neighbor normals. [62]

4.3.2.2 Mesh Convolution

MeshCNN [14] will be used to do mesh convolutions. It was found to be easier to use than MeshNet, and that is why MeshCNN was chosen over MeshNet.

4.3.2.3 Miscellaneous libraries

Trimesh [6] is a Python library made to load meshes into Python, and it can modify the mesh in various ways. It will be used to extract the faces and vertices. This is used to visualize the models using Tensorflow-Graphics [54] that can visualize 3D models.

Through the notebook, the library NumPy [32] is used to store arrays of data as well as saving and loading this data.

Matplotlib [27] is a python library making it possible to plot data. It will be used to plot results.

The python library tqdm [4] makes it possible to display a progress bar that can be used in loops to inform the user of the current progress. It is beneficial when training for many hours and to know if it has gotten stuck.

4.3.3 Code description

Each of the Jupyter notebooks for the experiments are available on GitHub:
github.com/frederikzt/Master-Thesis-3D-mesh-super-resolution
They are all a complete package, meaning that they have all functionalities inside to work. They all follow the same overall structure, which is:

- Import libraries
- Load data
- Test that data was loaded correctly
- Training
- Testing
- Plotting results

4.4 Choosing the experiments

In this section, it will be discussed what the options for the experiments are based on the findings from the two previous chapters. At last, the outline for the experiments will be presented.

4.4.1 Network architecture options

From the papers on the state of the art super-resolution of images, point clouds, and voxels, it seems like there is a pattern of how the network structure should be. They all use CNN's, and some use a GAN and ResNet on top of that. The ideal for these experiments would be to use a combination of the three for the network architecture.

4.4.2 Input options

Even though none of the above papers has done it, one option would be to give the vertices and faces directly to the network in the form of a flattened 1D array, but by doing this we cannot make use of the 3D convolutional methods, like MeshNet, MeshCNN or Graph-CNN's offer us. If any of these convolutional options should be used, the meshes need to be converted into a format that is supported by each of the methods. Another option could be to employ Multi-View Decomposition and extract depth maps of the meshes and give these images as the input.

4.4.3 Output options

As discussed in the input options, an output option would be to output vertices and faces directly. Another option would be to employ Scan2Mesh's method of first predicting vertices and edges and then use this to predict the faces. However, other options such as mesh deformations are probably more likely to work, since there are high similarity between the low-resolution mesh and high-resolution mesh. Therefore it could be smart to exploit these similarities and just deform the input mesh.

4.4.4 Choosing amongst the options

A rough list of what experiments that will be done will be outlined. They start simple and develop to be more advanced. Every experiment will be based on a previous experiment where one hyperparameter is changed every experiment. This is to understand what effect every single change has. Below is the plan for the experiments.

1. To start simple, an experiment will be performed where the vertices and faces are fed into a normal feedforward network and and the vertices and faces are received as output. The goal is to see if the network is unable to learn without convolutional layers. Then it will be tested if vertex deformations work as a viable output option.
2. The next experiments will include convolutional layers. As mentioned earlier, MeshCNN will be used. The goal is to expand on one of the previous experiments by adding the convolutional layers to see the effect.

4.5 Summary

This chapter has explained the pipeline of how the data is fed to the network and how the output is used to adjust the weights using the calculated loss. Then the dataset was chosen and it was a custom generated one. The tools and code was described. Lastly the outline for the experiments was presented.

Chapter 5

Experiments

The last section briefly discussed the possibilities for the experiments, and in this chapter, they will be conducted. There are two overall experiments. One being with only feedforward networks and a second being with mesh convolutional layers. The hypothesis is that the mesh convolutional layers will outperform the standard feedforward network, because it is able to do convolutional operations on the meshes.

Each overall experiment starts with an initial network structure, and it is then changed one hyper-parameter at a time to try to achieve better performance. To make the comparison between the results of the experiments, there will be images of the output meshes. There will be a mesh from the training set and the test set that will be visualized as well as some information about the losses. The two selected meshes are shown below in Figure 5.1 and 5.2. Note that a side by side comparison of the meshes from the experiments with the original meshes are available in the appendix.

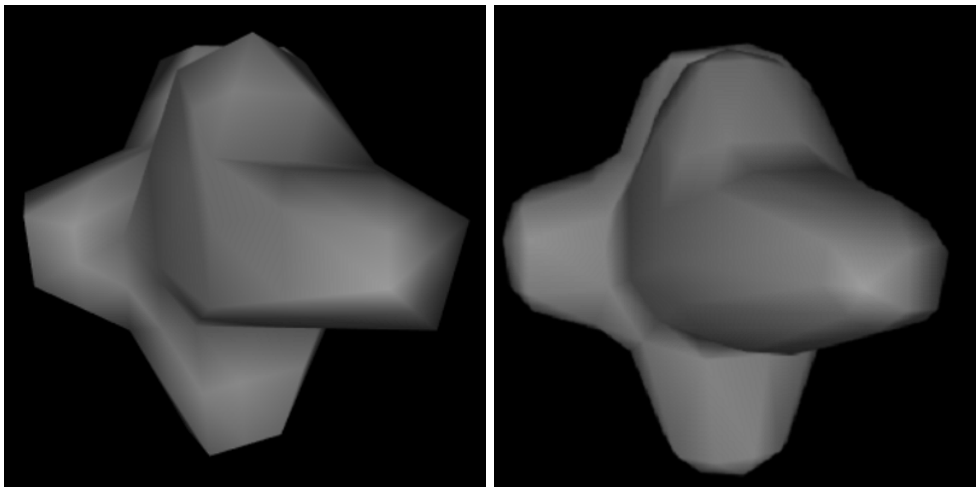
Figure 5.1: Training model presented in low-resolution (left) and high-resolution (right)

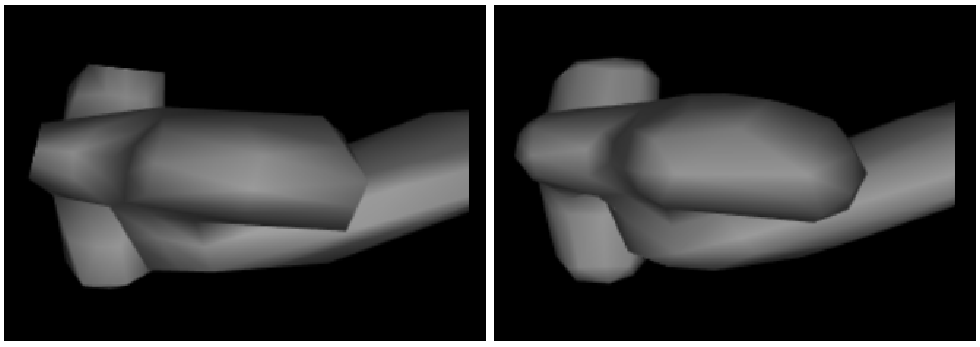
Figure 5.2: Test model presented in low-resolution (left) and high-resolution (right). Note that this test models were cut off in the right side due to their length.

To compare the average losses that are found in these experiments, two tests has been made to find the lowest loss possible. An average loss was calculated of the low-resolution models against the high-resolution models, and a loss was calculated of the high-resolution models compared to themselves. It turns out that even when taking the same exact same high-resolution model against itself does not give a loss of zero. The average for all the models in the training set was as follows:

- Low-resolution versus high-resolution: 0.445
- High-resolution versus high-resolution: 0.348

Therefore a loss less than 0.348 means that the network has produced a mesh even better than the target mesh, and a higher loss than 0.445 has produced a mesh that is worse than the low-resolution mesh. Note that these results are based on the loss functions, and they might not be perfect, but they still seem to indicate which mesh is closer to the target correctly.

Because there are so many hyperparameters that can be adjusted in the network it was chosen to have some of them being static. As far as the loss functions, the four loss functions that works on meshes [61] were used and they were weighted based on an example from PyTorch3D [42]. An additional loss function chamfer distance normals was added to ensure that the normals were not turned upside down which could make the mesh be rendered incorrectly. Specifically for the chamfer distance losses, a point cloud sampling of the meshes were needed and 5000 points were chosen as the number of samples. It seemed to be enough points to achieve a detailed point cloud of the mesh, and just doubling the points would increase the training time drastically. The weights chosen for the loss functions were:

- Chamfer distance = 1.0
- Chamfer distance normals = 1.0
- Edge length = 1.0
- Normal consistency = 0.01
- Laplacian smoothing = 0.1

Since they seemed to work well on some initial experiments with a small set of meshes, these settings were chosen. Adam was chosen as the optimizer because of its ability to adapt dynamically. The learning rate was a hyperparameter that was going to be changed between the experiments. The training time is not included, but the experiments took between 2 and 8 hours each. Note that the figures with the loss during training has on its x-axis "Epochs" written. But it is in fact the loss from each single mesh going through the network. It should also be mentioned that a full comparison between the test meshes from all experiments can be seen together in the discussion and they can be seen next to the low-resolution and high-resolution models in the appendix.

5.1 Experiments using feedforward networks

For these experiments only feedforward neural networks will be used, and the plan is to test different input and output types. The results can hopefully tell which input and output types to continue with further on. The different types of experiments are:

- Vertices as input and vertices as output (E1_V_V)

- Vertices and faces as input and vertices and faces as output (E1_VF_VF)
- Vertices as input and vertex deformations as output (E1_V_VD)
- Vertices and faces as input and vertex deformations as output (E1_VF_VD)

The advantage of the vertex deformation methods is that it ensures a valid mesh without gaps and overlapping faces should be less likely to happen.

The metric that is used to compare the results between the methods are the visual results of the output meshes and the average losses of the test set and the training set.

In this first experiment will all the four networks are trained on the full dataset of 4500 models and for 30 epochs. The learning rate is set to 0.00001 because it seemed to work best during initial experiments. The network is a fully connected feedforward network with 5000 neurons in each hidden layer. The input and output layers' size is determined by the different input and output shapes. Note that this network structure might be too big or too small, but it is the starting point of the experiments and will therefore be adjusted further on.

5.1.1 E1_V_V

The input for this network is only the vertex coordinates and the output is the vertex coordinates again. The faces are reused from the input mesh. It is a very simple input and output and it has its strengths and weaknesses. It should easier for the network to learn how to place the vertices at good positions, but the faces will be hard for the network to learn how to connect properly. The results after training can be seen below.

Average training set loss: **1.42**
Average test set loss: **1.72**

The loss during training can be seen below in Figure 5.3. It quickly converged from having a huge loss to a point where it started to improve very slowly.

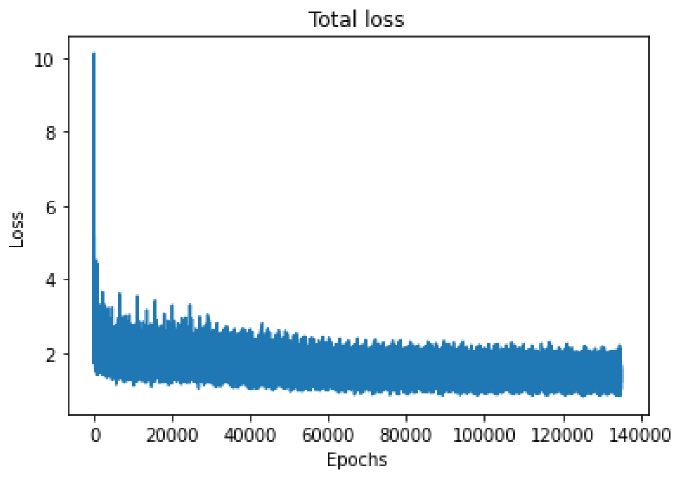

Figure 5.3: Loss during training

The output meshes can be seen below in Figure 5.4. They were very far from the desired meshes. It looks like it is very hard for the network to learn how to construct a valid mesh.

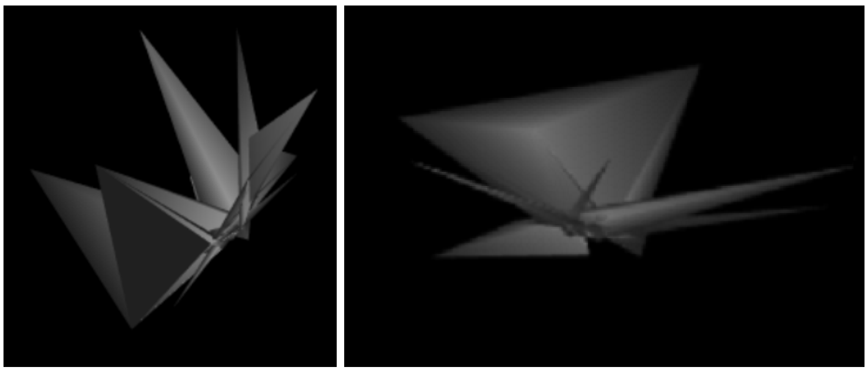

Figure 5.4: Training mesh (left) and test mesh (right)

The results of this experiment did not show any signs of super-resolution, but as suspected it is probably difficult to connect the faces correctly with the limited information about them.

5.1.2 E1_VF_VF

The idea behind this experiment was to give the face information to the network and receive it back, but it turned out not to work. It is suspected (although not for sure) to be due to a bug [9] in PyTorch3D, that made it unable to sample points (for the point cloud used for Chamfer distance) from an invalid mesh. In the beginning of training the output mesh is not valid from the direct output of the network since the first outputs of the network is likely to be zero for all values. This means that the faces' corners all share the same coordinate. There was probably a workaround for this, but instead the time was spent on other experiments.

5.1.3 E1_V_VD

Giving the vertices as input and getting vertex deformations as output would hopefully solve the problems from E1_V_V and E1_VF_VF. Since the network only provides mesh deformations it should be able to produce valid meshes that looks close to the input mesh at least. The results are definitely better than the previous results, even though it looks like it has been overfitted. The results show an decrease in the average loss as it can be seen below:

Average training set loss: **0.392**
Average test set loss: **0.499**

Below in Figure 5.5 are the training losses during training. The loss starts much lower than before and improves slowly. Perhaps the learning rate is too big since it jumps so much, or maybe it is simply due the the differences between the meshes. Later on it would definitely be interesting to see what a lower learning rate would do.

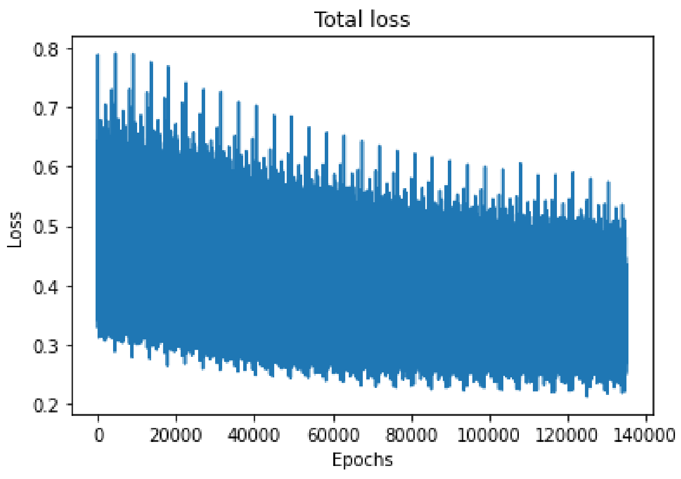

Figure 5.5: Loss during training

Below in Figure 5.6 are the produced meshes. The training set mesh looks really good, and it is a result like this for the test set that is desirable. The test set mesh looks to be deformed in random ways which is not desirable, but still the network has made changes which is a step in the right direction.

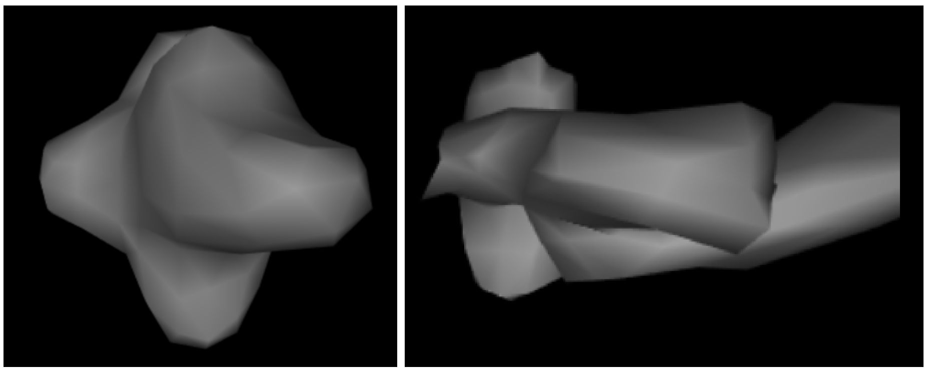

Figure 5.6: Training mesh (left) and test mesh (right)

It looks like it has been overfitted, but it is definitely the best results so far. It will be interesting to see if additional information about the faces will improve the results further.

5.1.4 E1_VF_VD (1)

This experiment has more information available as input than the previous experiment. It is the indexes of the faces and it continues to have the vertex deformations as output which seemed to work well in the previous experiment. The results for this experiment seem to overfit less, but to have a higher train loss as shown below:

Average training set loss: **0.473**
Average test set loss: **0.49**

Looking at figure 5.7 it seems to improve very slowly. It would be interesting to see if another learning rate would help it converge down better.

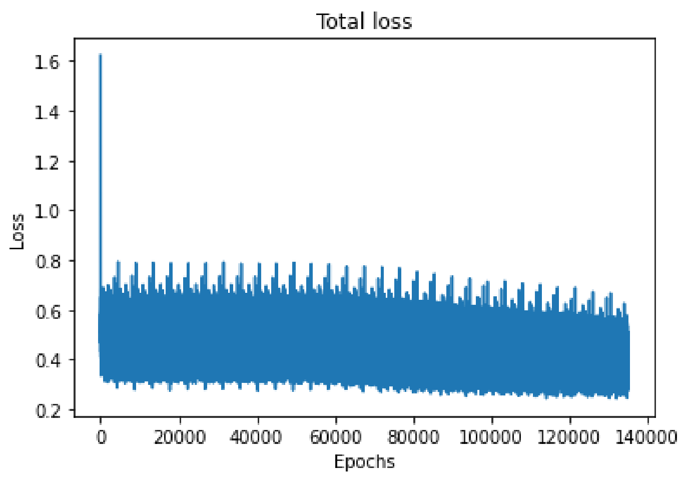

Figure 5.7: Training loss

Below in Figure 5.8 are the produced meshes. The training set mesh looks similar to the one from E1_V_VD, but the test mesh looks a little better, although it still seems to be a bunch of random vertex deformations that are applied.

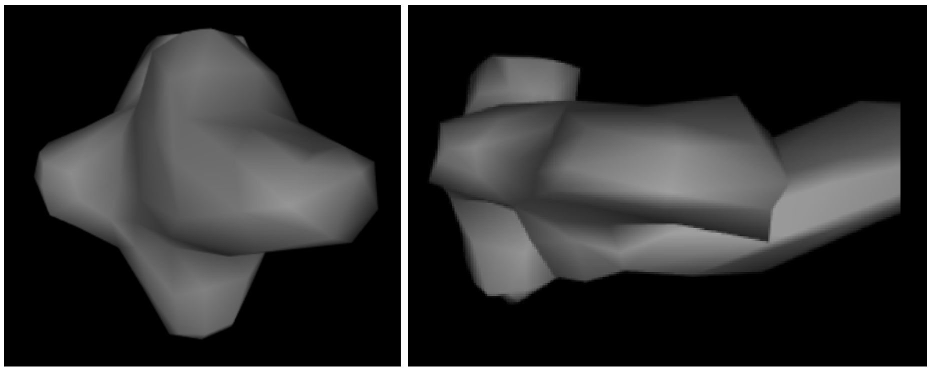

Figure 5.8: Training mesh (left) and test mesh (right)

This experiment probably has more potential. With a different learning rate and perhaps more layers it could potentially show much better test results.

5.1.5 Moving on with E1_VF_VD

The train mesh output of E1_V_VD looked good and it had the lowest training loss, but it had been overfit severely. E1_VF_VD had a lower test loss, and it was therefore thought that it had more potential for improvement and therefore it was chosen to move on with this method.

5.1.5.1 E1_VF_VD (2) - Adding two extra layers

It was thought that the network was too shallow to learn properly. Therefore the network was expanded with two layers to try be able to learn better. The results from this experiment showed a lower test loss than before which can be seen below.

Average training set loss: **0.447**
Average test set loss: **0.449**

Looking at the loss during training below in Figure 5.9 it shows us that the learning rate might be too high, since it still jumps around.

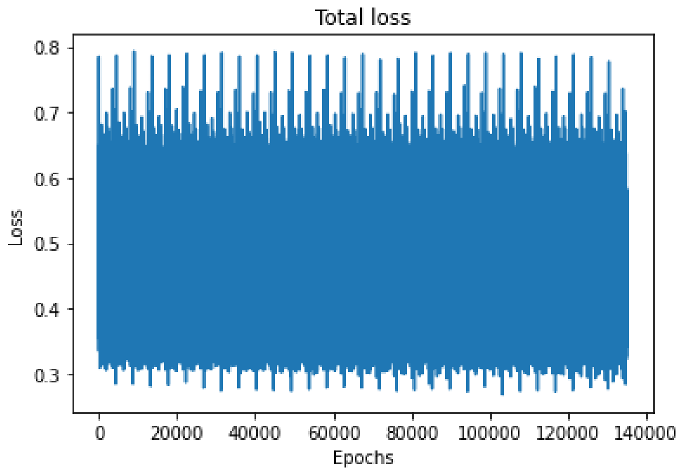

Figure 5.9: Loss during training

The mesh results were less promising than previously. It seems to not have learned anything. The meshes look a lot like the input meshes. They can be seen below in Figure 5.10

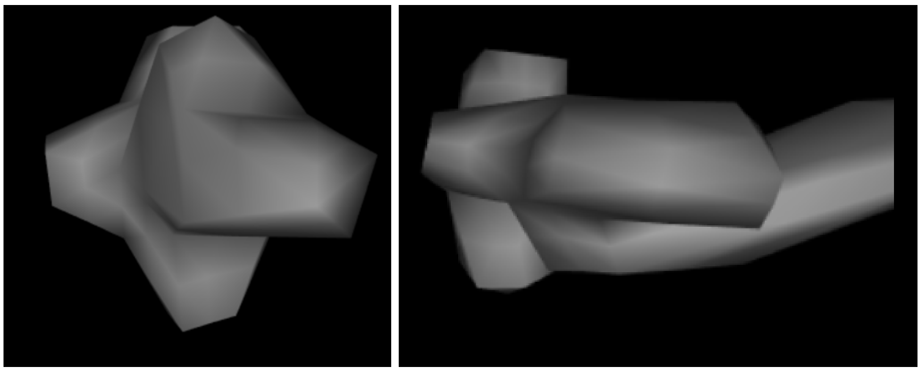

Figure 5.10: Training mesh (left) and test mesh (right)

It seems to have a harder time learning with the new hidden layers. It basically stops learning completely. It is interesting that a lower loss does not mean it will produce better meshes visually. For the next experiment it would be interesting to see what a lower the learning rate and training for more epochs would do.

5.1.5.2 E1_VF_VD (3) - Lowering the learning rate

In this experiment the learning rate is halved and the network is trained for a 100 epochs. Hopefully this can make the new larger network able to learn more. The results show an increased test loss which comes as a surprise, since it should have had more capabilities than before. However there is a chance that it had too much capacity which can make it worse at learning patterns. The results can be seen below.

Average training set loss: **0.491**
Average test set loss: **0.528**

The loss during training can be seen below in Figure 5.11 seems like it could improve, but this would probably just result in overfitting more, because it already is quite overfit.

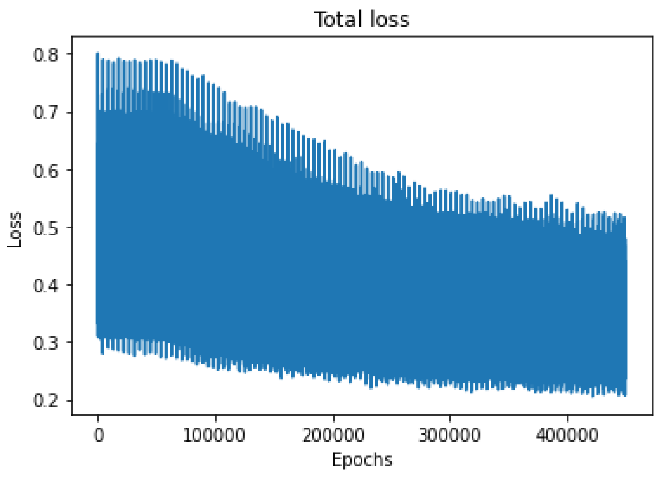

Figure 5.11: Loss during training

Looking at the meshes below in Figure 5.12 it is clear that the training mesh is smooth while the test mesh stays less smooth.

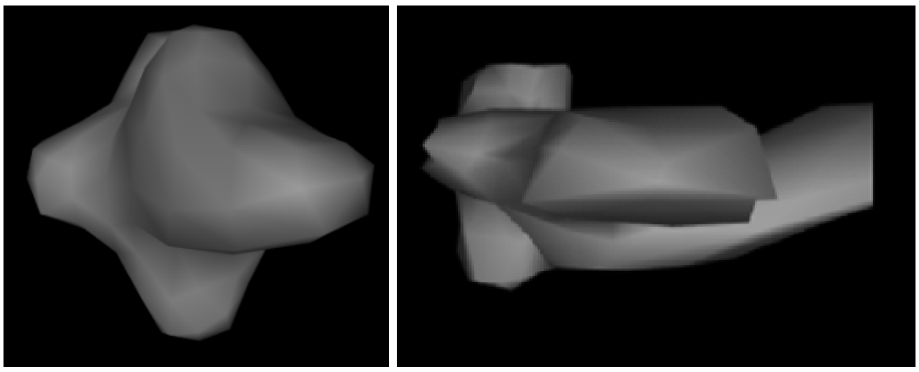

Figure 5.12: Training mesh (left) and test mesh (right)

Training more would probably not result in a better test loss, but just the opposite. Regularization could possibly fix this. Now dropout layers will be added in the hope of decreasing the test loss.

5.1.5.3 E1_VF_VD (4) - Adding dropout

A dropout layer with a dropout rate of 50% is added between each hidden layer in the hope of the test results to be better. The network is trained for a 100 epochs like the previous experiment. The results show an decrease of the test loss which is good. The results can be seen below.

Average training set loss: **0.446**
Average test set loss: **0.448**

The loss during training quickly dropped, but did not improve for the rest of the training. It can be seen below in Figure 5.13

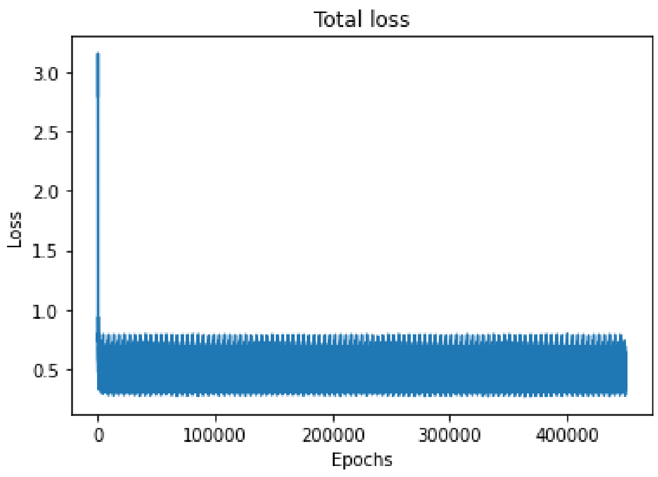

Figure 5.13: Loss during training

The meshes seem to be very close to the input meshes which is not the desired result. It can be seen below in Figure 5.14.

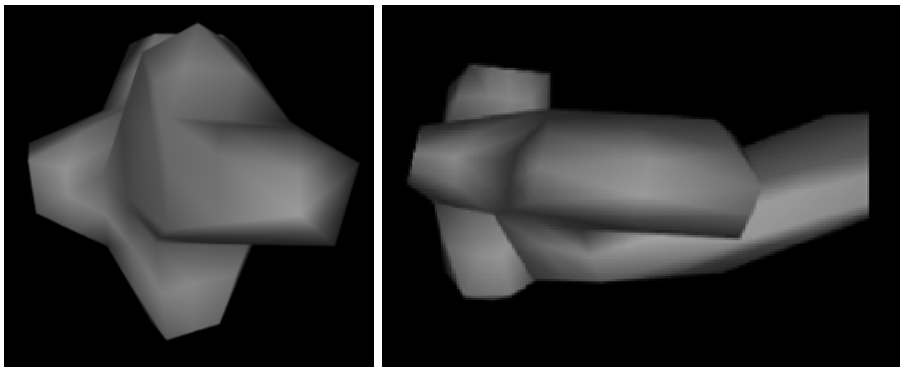

Figure 5.14: Training mesh (left) and test mesh (right)

Perhaps the learning rate is too small and it is stuck in a local minimum. There could be ways to fix this, but instead convolutional layers will now be employed in the hope of better results.

5.2 Experiments using convolutional neural networks

For these experiments MeshCNN's convolutional layers will be used. Since vertex deformations has proved to be the best output type yet it is chosen as the output. The input is going to be the MeshCNN specific input. Therefore each mesh will be converted into a 5 dimensional array with the input features for the network. As a starting point, the network structure is built with 5 mesh convolutional layers. The first has 5 input channels and 64 output channels. The following layer has 64 input channels and 128 output channels. The remaining hidden layers has 128 input channels and 128 output channels. The last layer is a fully connected layer which takes in the last convolutional layer's output in a flattened format. The idea is to increase the number channels slowly, but it is not known whether it has an effect in these experiments.

The learning rate is set to 0.00001 like it was in some of the previous experiments and the network is trained on the full dataset. It is trained for 10 epochs because it took a very long time compared to the last experiments. The idea is that the number of epochs could be increased if it showed room for improvement. Everything else stayed the same as in the previous experiments.

5.2.1 E2_VD (1) - Initial experiment with mesh convolutional layers

The results from this experiment shows it has been overfit. It can be seen below.

Average training set loss: **0.399**
Average test set loss: **0.5**

Below in Figure 5.15 are some figures showing the loss during training as well as the average loss of the different loss functions each epoch during training. It seems to learn well and would be able to learn more, but again it would probably end up overfitting more.

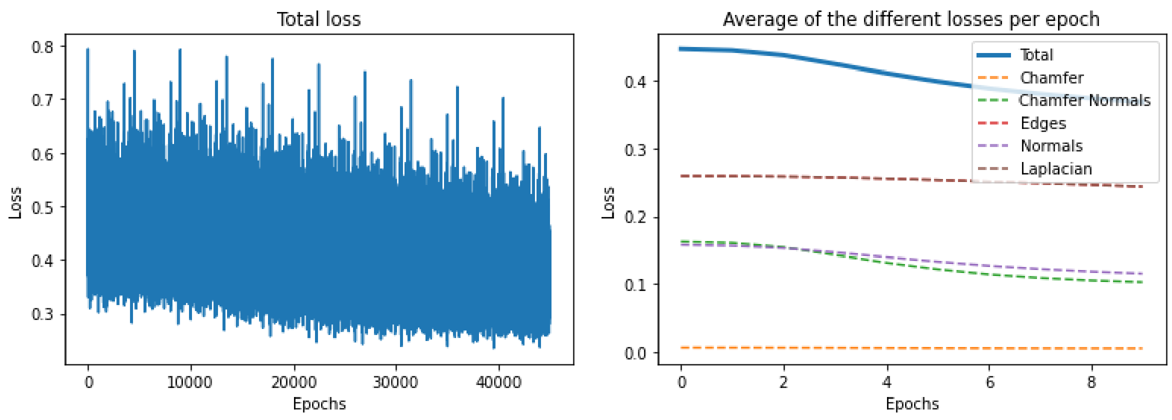

Figure 5.15: Losses during training

Looking at the output meshes in Figure 5.16 it seems that the training mesh is a little smoother than the test mesh, but the test mesh actually looks like it has been smoothed a little bit.

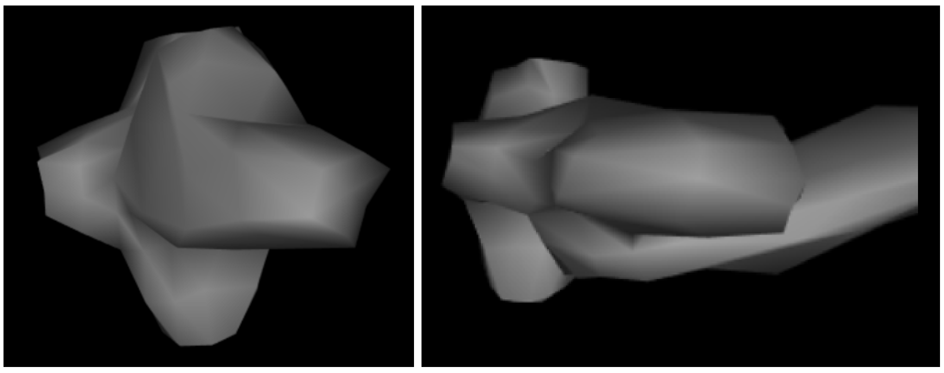

Figure 5.16: Training mesh (left) and test mesh (right)

Since it had overfit it would be interesting to try and add some dropout and see how it affected the network.

5.2.2 E2_VD (2) - Adding dropout

A dropout rate of 50% was added between each of the hidden layers. The results show an improved test loss and it looks as follows.

Average training set loss: **0.446**
Average test set loss: **0.448**

Looking at the losses during training in Figure 5.17 it seems to completely stop learning very early.

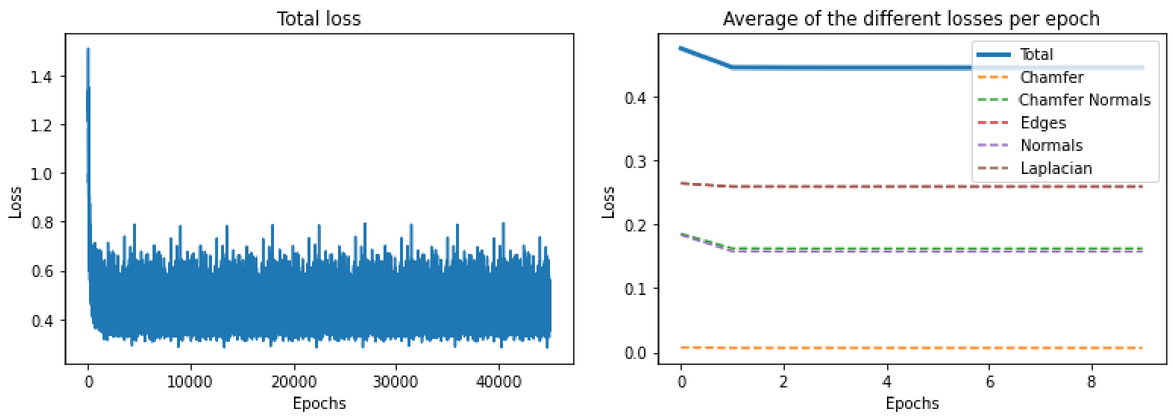

Figure 5.17: Losses during training

Looking at the output meshes in Figure 5.18 it seems like they are just like the original input meshes.

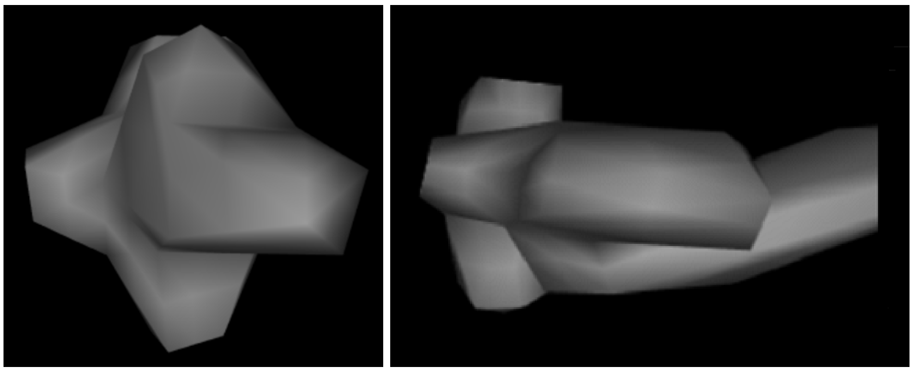

Figure 5.18: Training mesh (left) and test mesh (right)

It has unfortunately not learned much, but now the learning rate will be lowered to the half in the hope of improvement. It is interesting that the lower loss does not necessarily give a better output mesh.

5.2.3 E2_VD (3) - Halving the learning rate

The learning rate is halved to 0.000005 in the hope of it being able to achieve a lower loss. However the results show no improvement in the average losses.

Average training set loss: **0.446**
Average test set loss: **0.448**

It also turned out not to do anything for the losses during training as seen below in Figure 5.19.

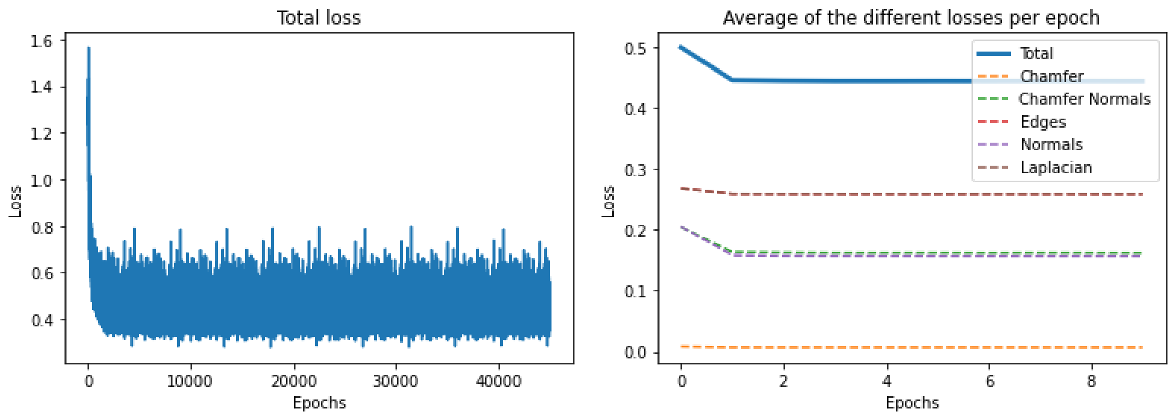

Figure 5.19: Losses during training

Also the meshes looks the same as before as seen in Figure 5.20.

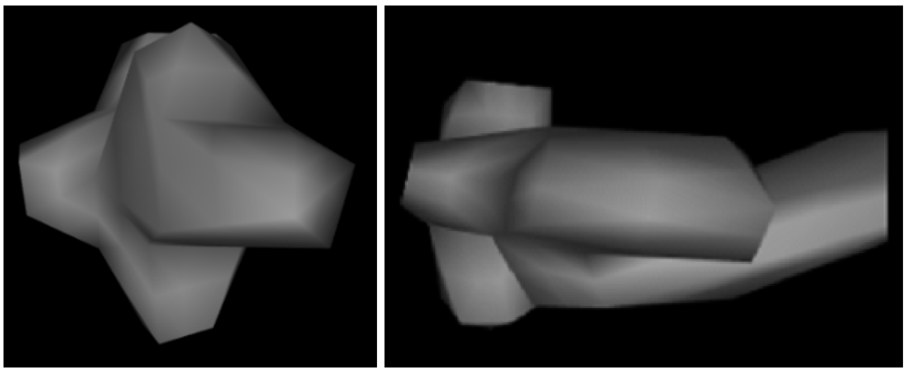

Figure 5.20: Training mesh (left) and test mesh (right)

Perhaps it is stuck in a local minimum and therefore would the lower learning have no effect. Instead it should be higher. But first it would be interesting to see if it is the size of the network that is the problem. Therefore three more hidden layers are added.

5.2.4 E2_VD (4) - Adding three hidden layers

The three layers added are similar to the last layers meaning they have 128 input channels and 128 output channels. The results from this change was again nothing new as seen below.

Average training set loss: **0.446**
Average test set loss: **0.448**

As seen in Figure 5.21 there were again no improvement.

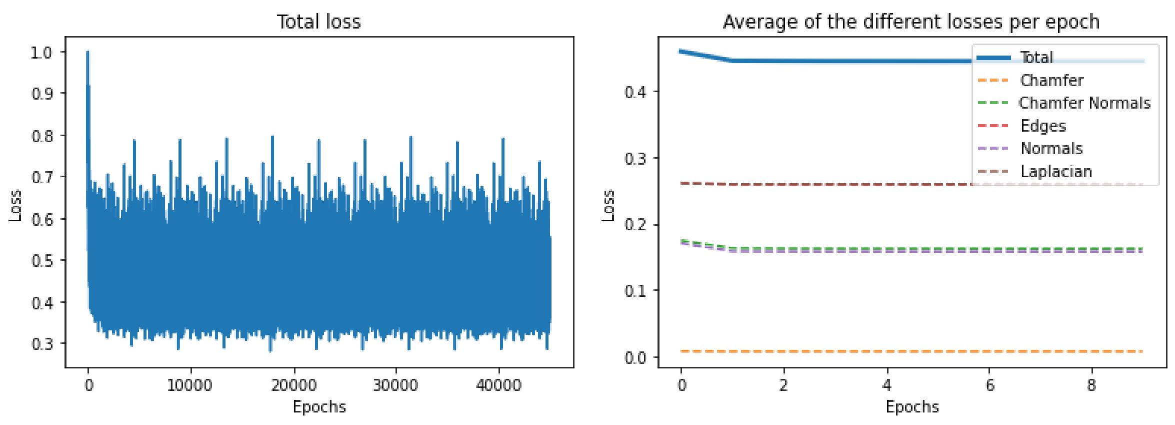

Figure 5.21: Losses during training

The meshes look the same as before which is seen below in Figure 5.22.

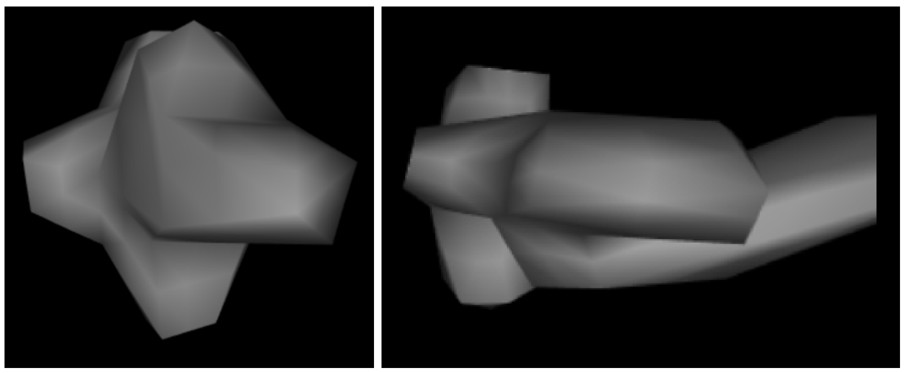

Figure 5.22: Training mesh (left) and test mesh (right)

It is again suspected to be due to a too low learning rate and it could be stuck in a local minimum. Therefore the learning rate will be increased to the double of the initial learning rate which is 0.00002.

5.2.5 E2_VD (5) - Increasing the learning rate to 0.00002

As the last experiment it was tested if it was the learning rate that was too low. It was set to 0.00002 which was the double learning rate of the first experiments. The results of the average loss did not change as seen below.

Average training set loss: **0.446**
Average test set loss: **0.448**

The losses during training can be seen in Figure 5.23 and it seems to learn faster in the start and then get stuck at the same level as before.

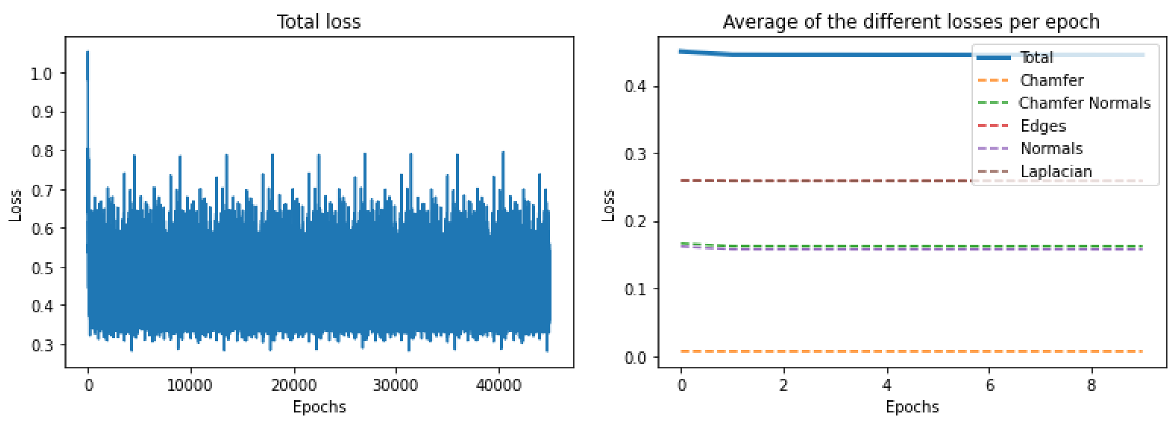

Figure 5.23: Losses during training

The output meshes as seen below in Figure 5.24 looks similar to the original input meshes as well as it did in the last experiments.

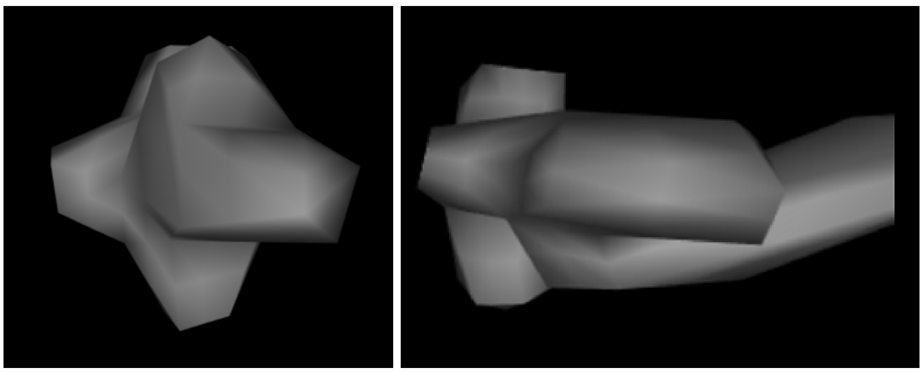

Figure 5.24: Training mesh (left) and test mesh (right)

As a last attempt it was tested if the bigger learning rate had a positive effect on the results. It turned out to give the same result as before, but with a faster learning in the beginning.

5.3 Summary

The results show signs over overfitting, and did not directly prove to be able to achieve proper super-resolution. Also, it was thought that the mesh convolutional networks would outperform the feedforward networks, but both methods did not perform well. It is however not the end, and super-resolution is definitely possible, but just not with the given architecture and the given data in these experiments. The results will be discussed further in the next chapter.

Chapter 6

Discussion

This discussion contains four sections about the results and possible improvements. The first section will cover an analysis of the results and explore the output meshes a further. The second section will discuss why the network was challenged when generalizing. The third section will suggest future improvements that can be further researched. And the final section will discuss alternative methods to achieving the goal of this thesis.

6.1 A closer look at the results

The results show that the network was unable to generalize, at least not to a level of the original high-resolution mesh. It could produce nice smooth meshes from the training set, but the results from the test set looked like the vertices had been shifted either randomly or not at all. There is a chance that there are other meshes from the test set that looks like its high-resolution model, but they are not investigated in this thesis.

The first experiment with the convolutional networks looks interesting, because it has rounded some of the edges. Below in Figure 6.1 are a collection of all the generated meshes as well as the original low-resolution and high-resolution mesh.

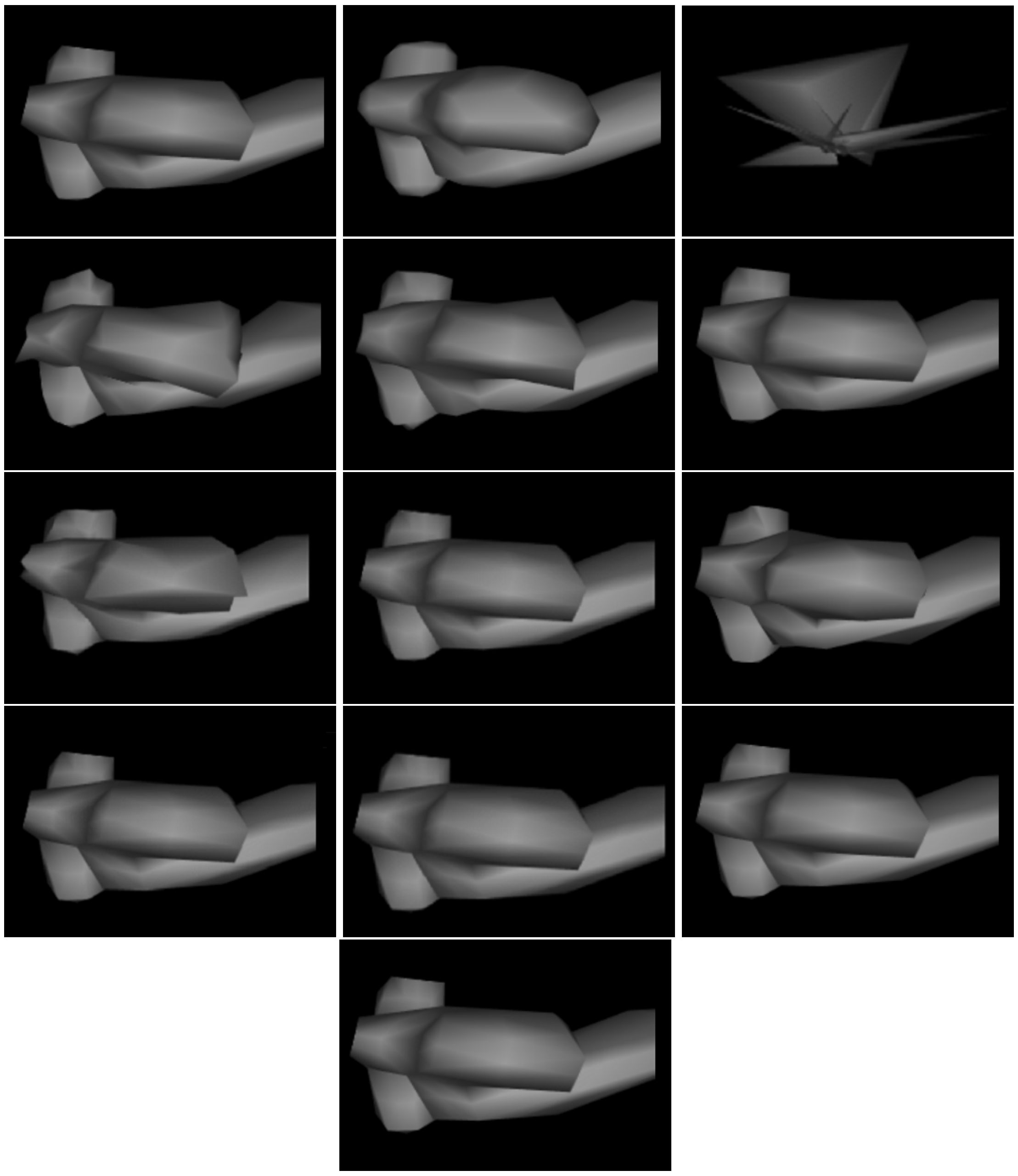

Figure 6.1: The meshes from the experiments presented in the order left to right then top to bottom. Original low-resolution mesh, original high-resolution mesh, E1_V_V, E1_V_VD, E1_VF_VD (1), E1_VF_VD (2), E1_VF_VD (3), E1_VF_VD (4), E2_VD (1), E2_VD (2), E2_VD (3), E2_VD (4), E2_VD (5)

6.2 What made it challenging for the network to generalize?

There were several factors that could have challenged the network to learn to generalize.

6.2.1 The loss functions

A big factor for the results were the used loss functions. Because they were kept unchanged it is not known what effect it would have to change them. They were tested for some initial experiments and the weights showed that the network was able to overfit on a small set of samples. This was understood to be a good sign since it then knows which is a well made mesh. The first concern would be if it were the right choice of loss functions. They were based on multiple sources [61][42] that used these loss functions and with weights similar to the chosen ones. Only chamfer distance normals was added to ensure the direction of the normals were accurate. It would be very interesting to test different weights for the different loss functions to find the optimal values. Or perhaps there are other loss functions that compare two meshes with each other in a better way. Because one of the most influential loss functions of them all are probably chamfer distance and it works by comparing two point clouds that are sampled from meshes. Therefore it is not a comparison of the meshes directly.

6.2.2 The inputs and outputs

The experiments with vertices and faces directly as input had a strength, which was that the input was in the same format as the output. Also, the ones where the input was vertices and faces and the output was vertex deformations, it can be argued that the output was of the same type as the input, namely a three dimensional vector. A big drawback of this method is that convolutional layers cannot be applied, and therefore it can be too hard for the network to learn the patterns of the meshes.

In the experiments using MeshCNN's convolutional layers the input and output type were completely different. The problem was that the network should learn to convert from a 5 dimensional vector with angles and lengths instead of vertices into vertex deformations. This conversion could be very hard for the network to do. That means it had two tasks. Firstly converting the 5 dimensional input into vertex deformations and second to make sure that the vertex deformations achieved super-resolution.

MeshCNN seems to have a way to convert the 5 dimensional vector back to a mesh which would probably solve the problem and make it able to do proper super-resolution on meshes. Unfortunately it was never discovered how this worked. But when discovered the super-resolution can probably be done using deconvolutional and unpooling layers that MeshCNN offer.

Another flaw with the vertex deformations is that it might not even be possible to move the vertices around properly. Imagine that one side of the mesh is more detailed, but since the faces are evenly distributed on the whole mesh, it would have to shift the faces towards the detailed area.

6.2.3 Suboptimal training

Because training took a long time it would take a really long time to try out all hyperparameter combinations. However it would have been very useful to know whether the experiments were on the right path or completely off. It could be that the learning rate was way too small, or maybe the optimizer was not right for this task. It could be because of the lack of data. 4500 models are not many and they are not even augmented. It should definitely have been augmented. The training should also have been done with mini-batches of more than just one mesh as it was done. Also L2 parameter regularization and batch normalization could have been used. When it comes to the amount of training, it would probably not make the networks perform better to train more since some of them stopped learning and others just overfitted more and more. It is also unknown whether the size of the network was appropriate. One possibility why it could not learn to generalize could also be due to a bug, but it is not suspected.

6.3 Future work

Apart from the things mentioned above there are some ideas for future work to be done. The most important thing would be to figure out a way to get a mesh back from the 5 dimensional vector. When this is done it could vastly improve the test loss and the looks of the output meshes. Further on it could be combined with a GAN and a ResNet, just as in other super-resolution solutions [63][35]. Before the issue with the conversion is solved i do not think that the GAN and ResNet would do much. When using the GAN network PReLU and LReLU could be used as well.

Another option would be to convert the mesh into a graph and employ a Graph-CNN. If convolutional layers could be applied to this graph it could be a new way of doing convolutions on meshes. The output of the network could even be a mesh directly which could make it easier for the network to learn.

6.4 Alternative methods

It is possible to achieve super-resolution with meshes, using deep learning with known methods. This was however not the goal of this thesis. The method available is to convert meshes into point clouds or voxels. Then do the super-resolution using deep learning on this data. Finally whenever the model is used to generate a super-resolution point cloud or voxel the result can then be converted back to a mesh. Kaolin [20] provides methods for converting between the three 3D structures. This method was even used before in point cloud super-resolution [63] to compare the method to the original meshes.

Chapter 7

Conclusion

What was learned throughout this thesis was that deep learning on meshes, in general, is difficult, and that there does not exist a way to perform super-resolution on meshes using deep learning directly. The goal of this thesis was to investigate this. It turned out that traditional convolutional layers only work on data that is structured in a grid-like format such as images are. These convolutional layers are crucial for deep learning architectures for these structures. Therefore it would make good sense if deep learning on meshes required convolutional layers. This thesis conducted experiments with two types of network architectures. One with only feedforward layers and another with special mesh convolutional layers. Both experiments did not prove to be able to perform super-resolution on meshes. There is, however, room for improvement both of the network architecture, and the input and output for the network. Especially the output of the network could be optimized. If there was a method to convert the 5 dimensional array from MeshCNN back to a mesh in an effective way, it would probably make it able to learn much better. Even though both the feedforward and the mesh convolutional experiments performed quite similarly, it is thought that the mesh convolutional method has the most potential. This is believed to be the case because convolutional layers have advantages with all sorts of structures, such as images, point clouds, and voxels. Convolutional layers should give the network a better understanding of the mesh structure. It is also very possible that it could not generalize because of the lack of data. Perhaps there should be much more training data for the network to learn to generalize.

After these issues have been solved and a network is able to perform super-resolution, it is speculated what further improvements could be made. If the deep learning architecture were combined with a GAN and a ResNet like other super-resolution methods do, it could potentially give even better results. Another option would be to employ a Graph-CNN, which could be another way of using convolutional layers on a mesh. As a final word, it is still believed that super-resolution on 3D meshes with deep learning is possible within the near future. There is active research on both deep learning on meshes as well as super-resolution on different structures, and it is therefore only a matter of time before someone discovers an optimal way of achieving super-resolution on meshes.

Chapter 8

Appendix

8.1 Meshes from the experiments

The purpose of this section is to set up the output meshes from the experiments up side by side to the original low-resolution and high-resolution meshes. There is no new models presented, but they are easier to compare against each other.

8.1.1 E1_V_V

Training set models

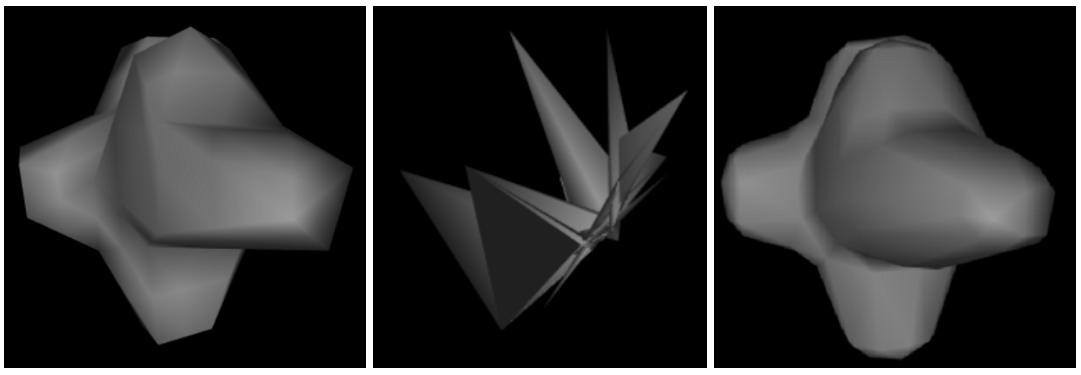

Figure 8.1: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

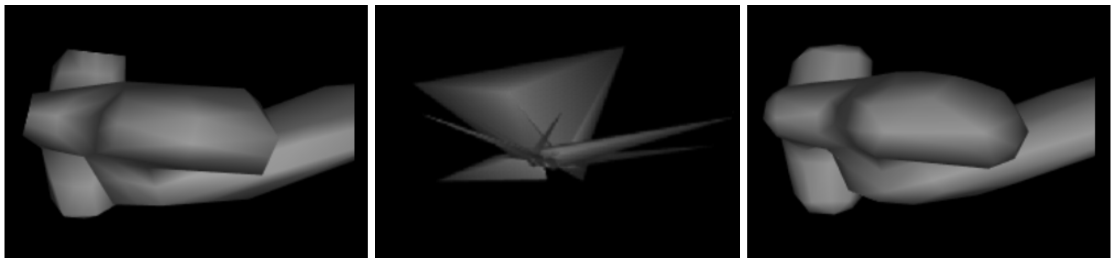

Figure 8.2: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.2 E1_V_VD

Training set models

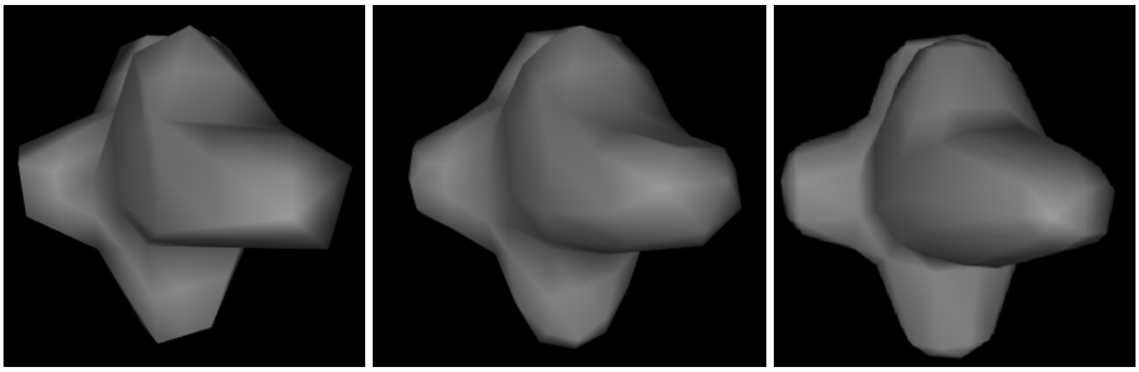

Figure 8.3: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

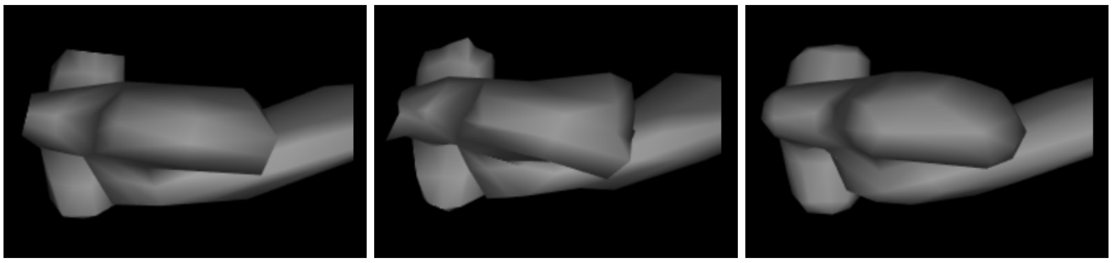

Figure 8.4: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.3 E1_VF_VD (1)

Training set models

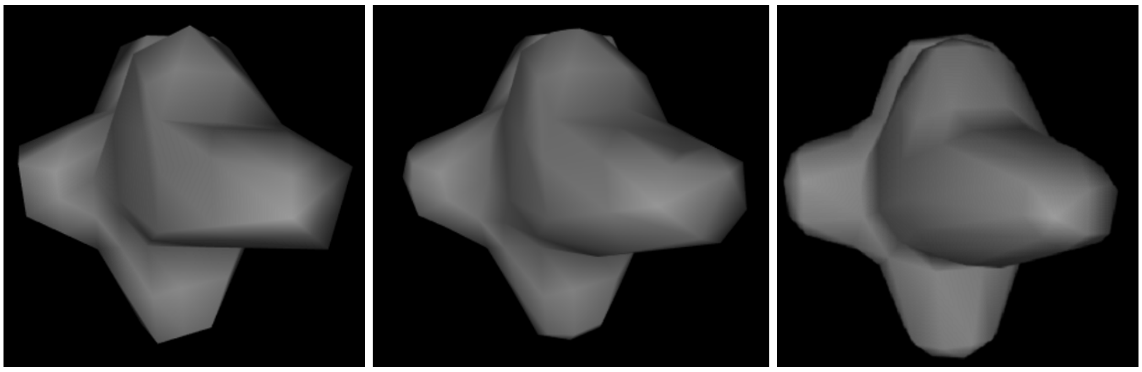

Figure 8.5: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

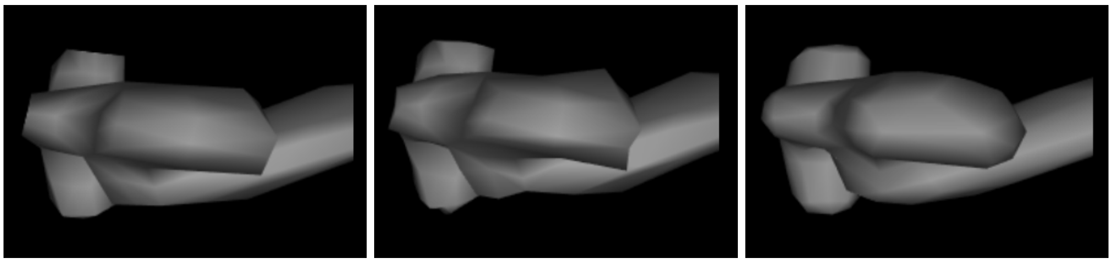

Figure 8.6: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.4 E1_VF_VD (2)

Training set models

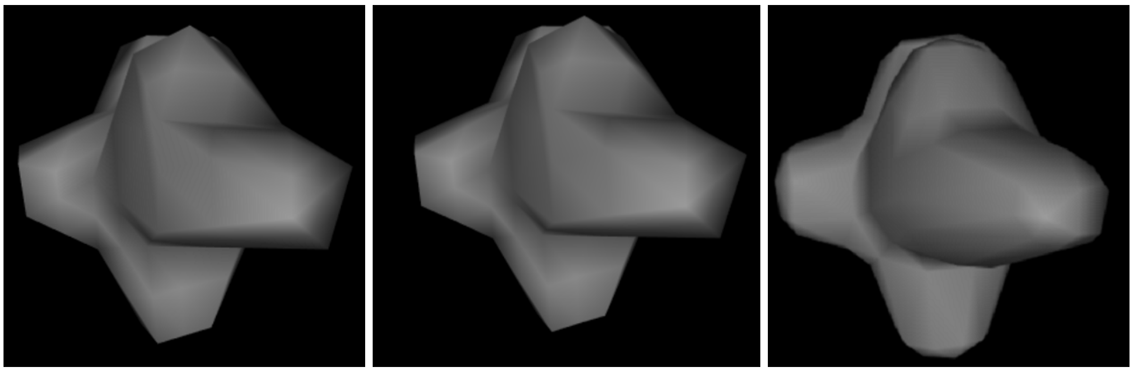

Figure 8.7: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

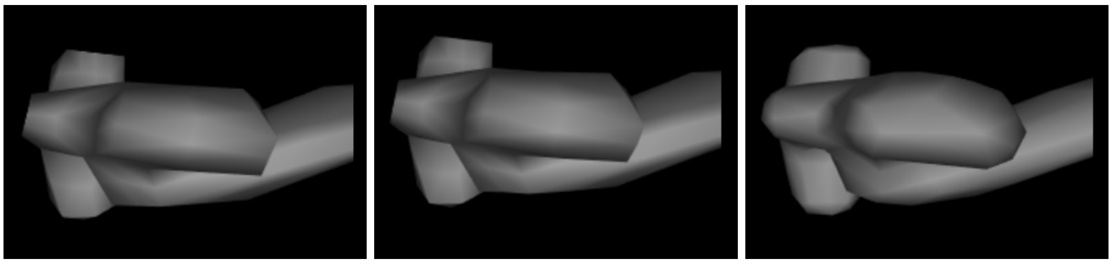

Figure 8.8: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.5 E1_VF_VD (3)

Training set models

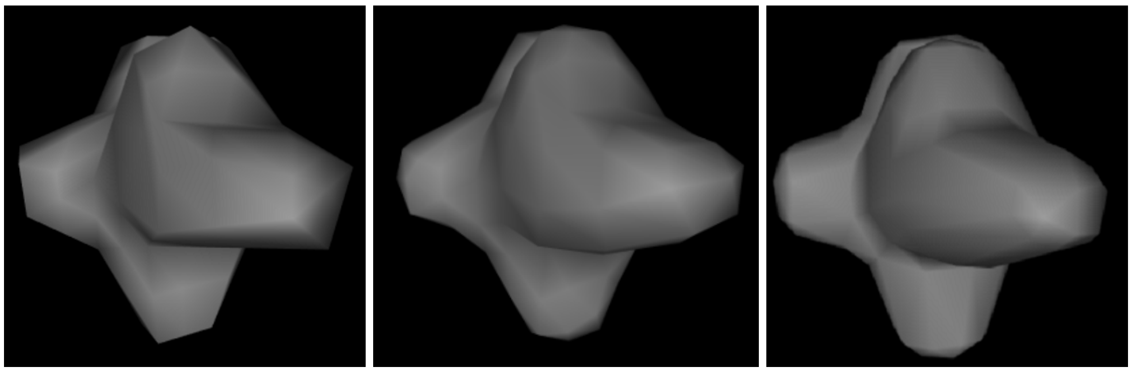

Figure 8.9: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

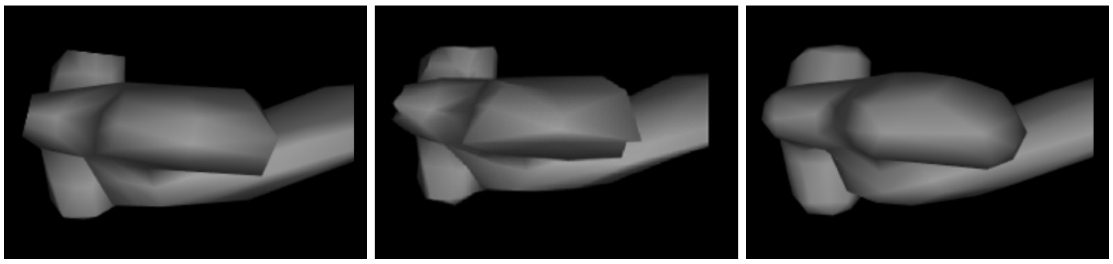

Figure 8.10: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.6 E1_VF_VD (4)

Training set models

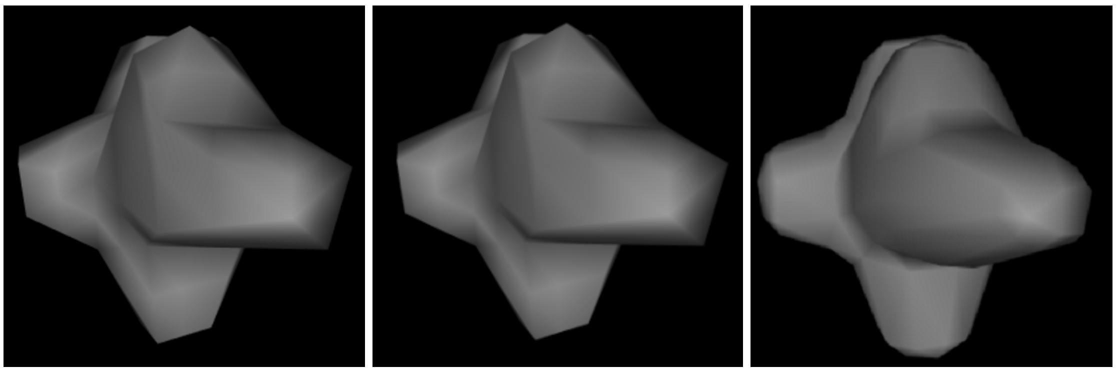

Figure 8.11: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

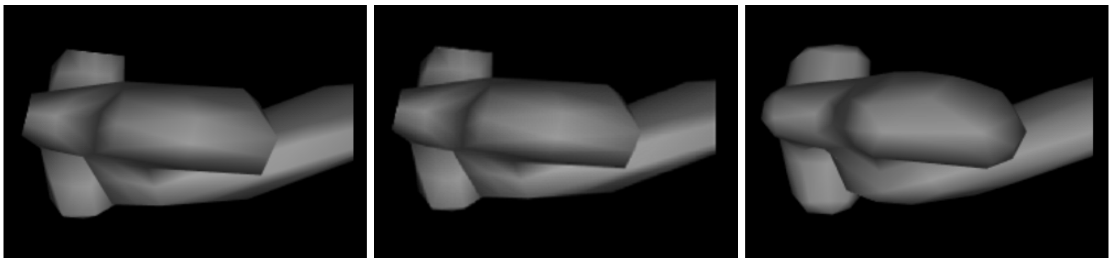

Figure 8.12: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.7 E2_VD (1)

Training set models

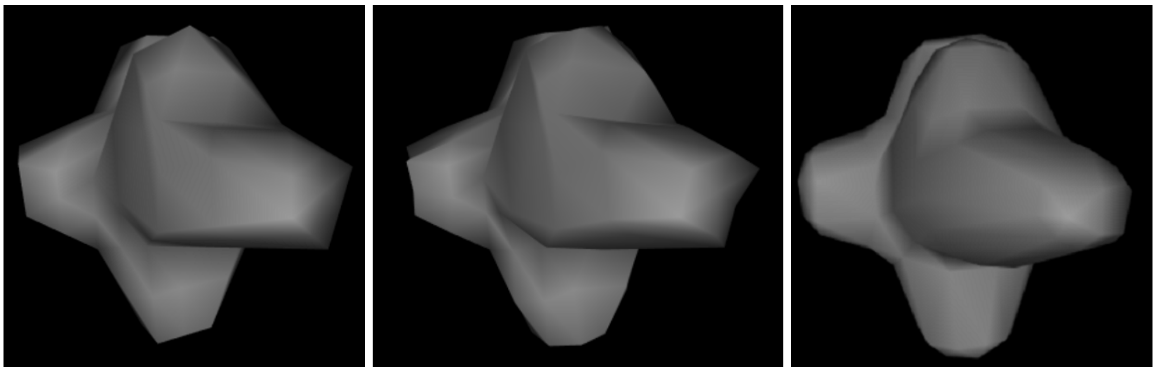

Figure 8.13: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

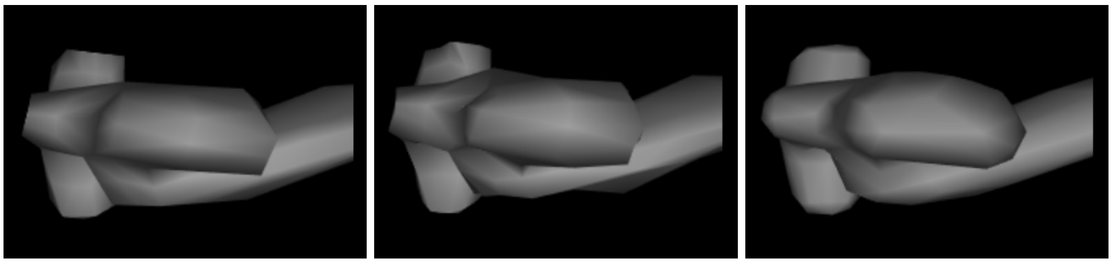

Figure 8.14: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.8 E2_VD (2)

Training set models

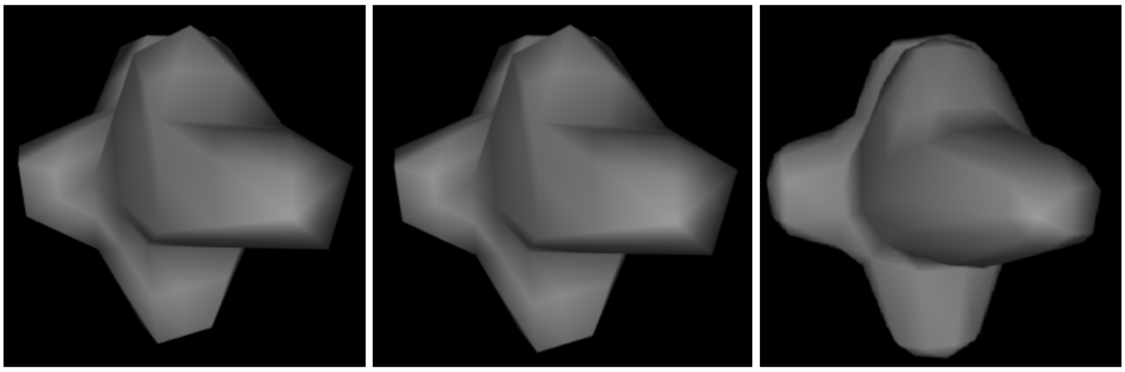

Figure 8.15: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

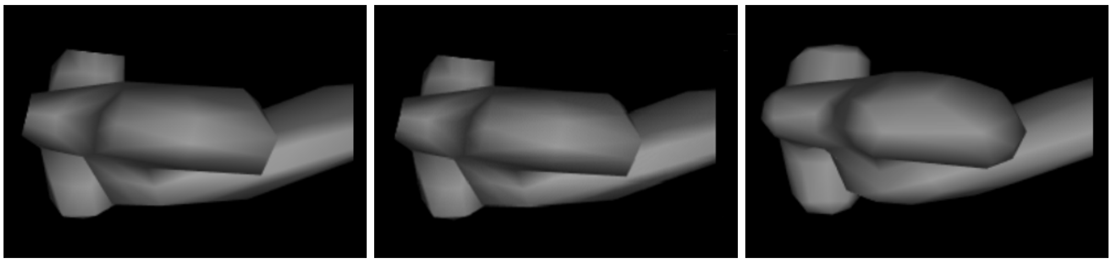

Figure 8.16: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.9 E2_VD (3)

Training set models

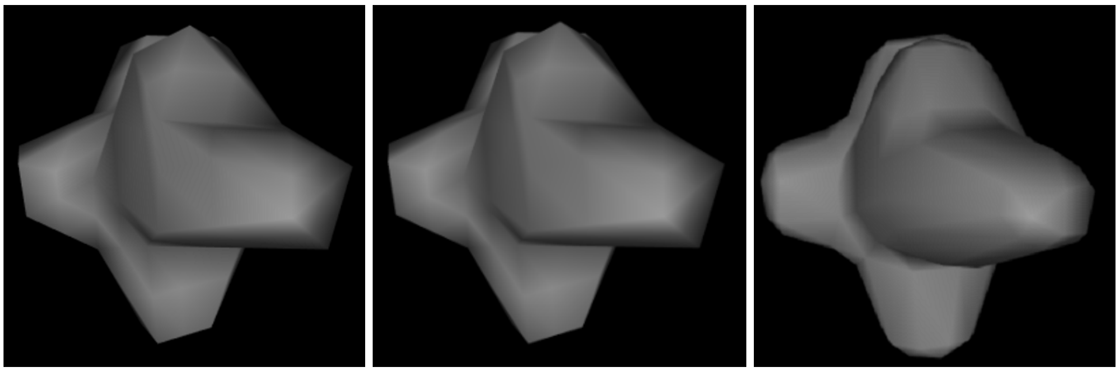

Figure 8.17: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

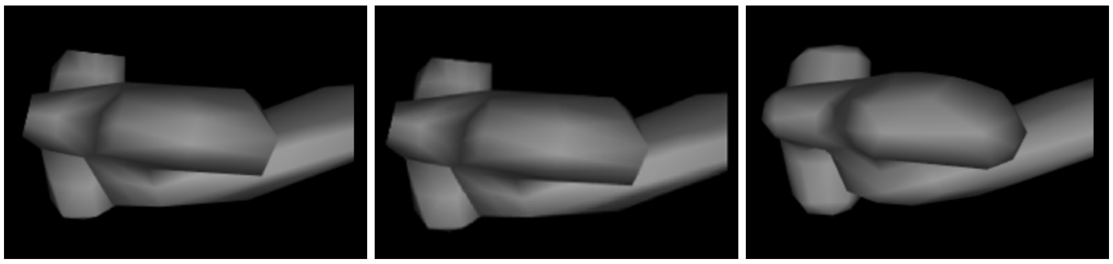

Figure 8.18: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.10 E2_VD (4)

Training set models

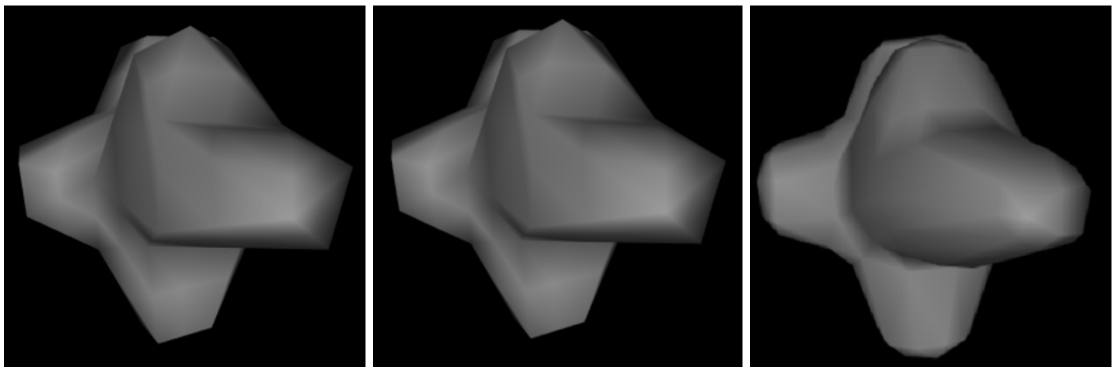

Figure 8.19: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

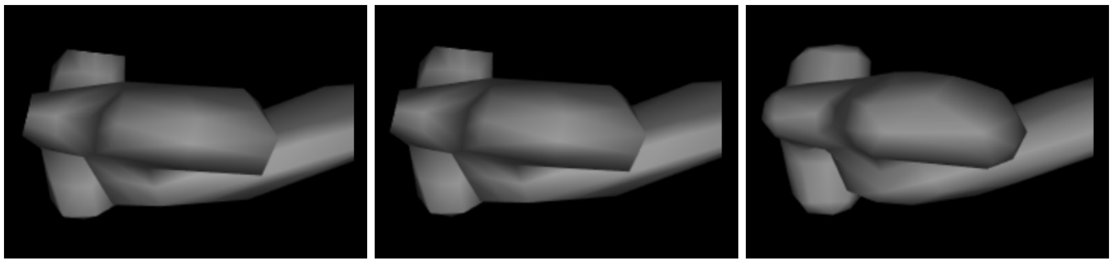

Figure 8.20: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

8.1.11 E2_VD (5)

Training set models

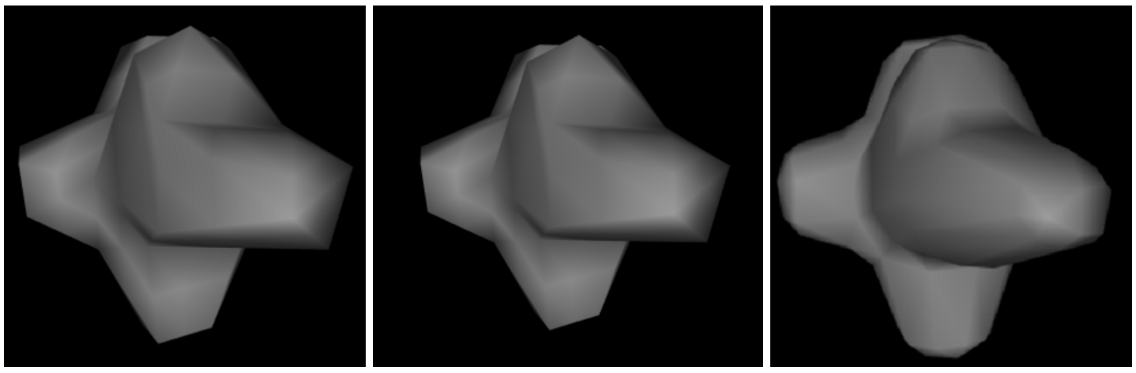

Figure 8.21: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Test set models

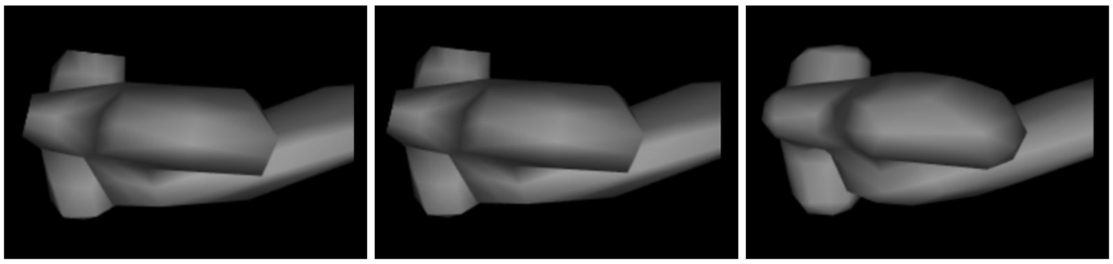

Figure 8.22: Training models. Low-resolution mesh (left), network output mesh (middle) and high-resolution mesh (right)

Bibliography

[1] Eman Ahmed et al. *A survey on Deep Learning Advances on Different 3D Data Representations*. 2018. eprint: `arXiv:1808.01462`.

[2] Glenn Brookshear. *Computer Science: An Overview, Global Edition*. Pearson Education Limited, 2014.

[3] Paolo Cignoni et al. "MeshLab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara, and Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: `10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136`. URL: `http://www.meshlab.net/`.

[4] Casper da Costa-Luis. "'tqdm': A Fast, Extensible Progress Meter for Python and CLI". In: *Journal of Open Source Software* 4.37 (2019), p. 1277. DOI: `10.21105/joss.01277`. URL: `https://doi.org/10.21105/joss.01277`.

[5] Angela Dai and Matthias Nießner. *Scan2Mesh: From Unstructured Range Scans to 3D Meshes*. 2018. eprint: `arXiv:1811.10464`.

[6] Dawson-Haggerty et al. *trimesh*. Version 3.2.0. URL: `https://trimsh.org/`.

[7] *Decimate Modifier*. URL: `https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/decimate.html?highlight=decimate`.

[8] *DirextX 11 Tesselation*. URL: `https://www.nvidia.com/object/tessellation`.

[9] Facebookresearch. *PyTorch3D Sample points from meshes Bug*. URL: `https://github.com/facebookresearch/pytorch3d/blob/master/pytorch3d/ops/sample_points_from_meshes.py`.

[10] Yutong Feng et al. *MeshNet: Mesh Neural Network for 3D Shape Representation*. 2018. eprint: `arXiv:1811.11424`.

[11] Blender Foundation. *Home of the Blender project - Free and Open 3D Creation Software*. URL: `https://www.blender.org/`.

[12] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. Cambridge, MA, USA: MIT Press, 2016.

[13] *Google Colab*. URL: `colab.research.google.com`.

[14] Rana Hanocka et al. "MeshCNN: A Network with an Edge". In: (2018). DOI: `10.1145/3306346.3322959`. eprint: `arXiv:1809.05910`.

[15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. eprint: `arXiv:1512.03385`.

[16] *Ikea App*. URL: `https://www.ikea.com/au/en/customer-service/mobile-apps/say-hej-to-ikea-place-pub1f8af050`.

[17] *Importing and Exporting Files in Blender*. URL: `https://docs.blender.org/manual/en/latest/files/import_export.html`.

[18] *Introduction to Meshes*. URL: `https://docs.blender.org/manual/en/latest/modeling/meshes/introduction.html`.

[19] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: `arXiv:1502.03167`.

[20] Krishna Murthy J. et al. "Kaolin: A PyTorch Library for Accelerating 3D Deep Learning Research". In: *arXiv:1911.05063* (2019).

[21] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2018. eprint: `arXiv:1812.04948`.

[22] Mark Kingsnorth. *Shape Generator*. URL: `https://blendermarket.com/products/shape-generator`.

[23] Alexander Kolesnikov et al. *Big Transfer (BiT): General Visual Representation Learning*. 2019. eprint: `arXiv:1912.11370`.

[24] *Laplacian Smooth Modifier*. URL: `https://docs.blender.org/manual/en/latest/modeling/modifiers/deform/laplacian_smooth.html`.

[25] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.

[26] Christian Ledig et al. *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*.

[27] *Matplotlib*. URL: `https://matplotlib.org/`.

[28] *Mesh Structure*. URL: `https://docs.blender.org/manual/en/latest/modeling/meshes/structure.html`.

[29] *Minecraft*. May 2020. URL: `https://en.wikipedia.org/wiki/Minecraft`.

[30] *Neural network image super-resolution and enhancement*. URL: `https://letsenhance.io/`.

[31] *Normal (geometry)*. Feb. 2020. URL: `https://en.wikipedia.org/wiki/Normal_(geometry)`.

[32] *NumPy*. URL: `https://numpy.org/`.

[33] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. eprint: `arXiv:1811.03378`.

[34] Tian Ooi. *Blender: Recalculate Normals – Simply Explained*. Mar. 2019. URL: `https://all3dp.com/2/blender-recalculate-normals-simply-explained/`.

[35] *Papers with Code - Image Super-Resolution*. URL: `https://paperswithcode.com/task/image-super-resolution`.

[36] *Papers with Code - Point Cloud Super Resolution*. URL: `https://paperswithcode.com/task/point-cloud-super-resolution`.

[37] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[38] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly, 2017. ISBN: 978-1-4919-1425-0.

[39] Jhony K. Pontes et al. "Image2Mesh: A Learning Framework for Single Image 3D Reconstruction". In: (2017). eprint: `arXiv:1711.10669`.

[40] *Project Jupyter*. URL: `https://jupyter.org/`.

[41] *Python.org*. URL: `https://www.python.org/`.

[42] *Pytorch 3D Fitmesh*. URL: `https://pytorch3d.org/tutorials/deform_source_mesh_to_target_mesh`.

[43] Charles R. Qi et al. *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space*. 2017. eprint: `arXiv:1706.02413`.

[44] Nikhila Ravi et al. *PyTorch3D*. `https://github.com/facebookresearch/pytorch3d`. 2020.

[45] *Remesh Modifier*. URL: `https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/remesh.html`.

[46] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

[47] Laura Downs Sergey Levine Chelsea Finn. *3D Models - Google Brain Robotics Data*. URL: `https://sites.google.com/site/brainrobotdata/home/models`.

[48] D. Shiffman, S. Fry, and Z. Marsh. *The Nature of Code*. D. Shiffman, 2012. ISBN: 9780985930806. URL: `https://books.google.dk/books?id=hoK6lgEACAAJ`.

[49] Edward Smith, Scott Fujimoto, and David Meger. *Multi-View Silhouette and Depth Decomposition for High Resolution 3D Object Representation*. 2018. eprint: `arXiv:1802.09987`.

[50] *Subdivide Modifier*. URL: `https://docs.blender.org/manual/en/latest/modeling/meshes/editing/subdividing/subdivide.html`.

[51] ShapeNet Research Team. *ShapeNet*. URL: `https://www.shapenet.org/`.

[52] Unity Technologies. *3D formats*. URL: `https://docs.unity3d.com/560/Documentation/Manual/3D-formats.html`.

[53] Unity Technologies. *Unity*. URL: `https://unity.com/`.

[54] *TensorFlow Graphics*. URL: `https://www.tensorflow.org/graphics`.

[55] *Triangulate Modifier*. URL: `https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/triangulate.html`.

[56] Sik-Ho Tsang. *Review: DeconvNet-Unpooling Layer (Semantic Segmentation)*. Mar. 2019. URL: `https://towardsdatascience.com/review-deconvnet-unpooling-layer-semantic-segmentation-55cf8a6e380e`.

[57] Nobuyuki Umetani. *Exploring Generative 3D Shapes Using Autoencoder Networks*. 2017. URL: `https://www.autodeskresearch.com/publications/exploring_generative_3d_shapes`.

[58] Jake VanderPlas. *Python data science handbook: essential tools for working with data*. OReilly, 2017.

[59] *Vector Displacement Node*. URL: `https://docs.blender.org/manual/en/latest/render/shader_nodes/vector/vector_displacement.html`.

[60] J. Vollmer, R. Mencl, and H. Müller. "Improved Laplacian Smoothing of Noisy Surface Meshes". In: *Computer Graphics Forum* 18.3 (1999), pp. 131–138. DOI: `10.1111/1467-8659.00334`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00334`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00334`.

[61] Nanyang Wang et al. "Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images". In: (2018). eprint: `arXiv:1804.01654`.

[62] *Welcome to PyTorch3D's documentation!* URL: `https://pytorch3d.readthedocs.io/en/latest/index.html`.

[63] Huikai Wu, Junge Zhang, and Kaiqi Huang. *Point Cloud Super Resolution with Adversarial Residual Graph Networks*. 2019. eprint: `arXiv:1908.02111`.

[64] Yuhui Yuan, Xilin Chen, and Jingdong Wang. *Object-Contextual Representations for Semantic Segmentation*. 2019. eprint: `arXiv:1909.11065`.

[65] et.al. Z. Wu. *3D ShapeNets: A Deep Representation for Volumetric Shapes*. URL: `https://modelnet.cs.princeton.edu/`.

[66] Yingxue Zhang and Michael Rabbat. *A Graph-CNN for 3D Point Cloud Classification*. 2018. eprint: `arXiv:1812.01711`.