TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

Geometrical Deep Learning on 3D Models: Classification for Additive Manufacturing

Authors	Bouziane, Nouhayla | Ebid, Ahmed | Srinivas, Aditya Sai | Bok, Felix | Kiechle, Johannes
Mentor(s)	Volkswagen Data:Lab Munich Dr. Kleshchonok, Andrii | Cesur, Guelce | Danielz, Marcus
Co-Mentor	Dr. Kerstin Lux (Department of Mathematics)
Project Lead	Dr. Ricardo Acevedo Cabra (Department of Mathematics)
Supervisor	Prof. Dr. Massimo Fornasier (Department of Mathematics)

Jul 2021

Abstract

Deciding on the manufacturability of 3D models by a 3D printer is still a task that is very time-consuming and requires expert knowledge. The main goal of the project is to automate this process using convolutional neural networks. We present a data pipeline that selects suitable input data and processes it through a series of transformations consisting of cleaning, scaling, alignment and voxelization. For the generation of non-manufacturable models we developed an algorithm that inserts various defects to 3D models. With this algorithm we were able to generate a dataset for our classification task, which we refer to as AMC dataset. We trained deep learning models, based on advanced convolutional neural network architectures, such as ResNet and InceptionNet on the AMC dataset. Using the InceptionNet architecture we were able to classify the manufacturability with an accuracy of 98% and an F1-score of 0.98. We further evaluated the performance of the deep learning models and the quality of the generated models in more detail.

Acknowledgements

First of all, we would like to express our deepest gratitude to our Volkswagen Data:Lab mentors Guelce Cesur, Dr. Andrii Kleshchonok and Marcus Danielz, for providing such a current and fascinating project, support whenever it was needed and all beneficial feedback during the whole project period. A special thanks also goes to our university supervisor Dr. Kerstin Lux for giving valuable feedback throughout the entire project phase. Kerstin was always very interested and curious about our progress and therefore demonstrated real interest in the topic. Finally, we are also very grateful to Prof. Dr. Massimo Fornasier and Dr. Ricardo Acevedo Cabra for enabling the TUM Data Innovation Lab and to the Leibnitz-Rechenzentrum (LRZ) for providing access to their compute infrastructure.

Contents

1 Introduction

1.1 Motivation

Additive Manufacturing (AM), commonly referred to as 3D printing, is the most general term which describes the process of adding material layer by layer, where each layer is a slice of a digital 3D model [1]. There are a plenty of different methods for AM, as described for example in [2]. What most of these methods have in common is that they offer a variety of high-impact benefits compared to classical manufacturing processes. More complex objects can be produced faster, more sustainable and on demand [3, 4, 5]. It is therefore no coincidence that the AM market faced an average growth rate of 27% over the last decade and is currently estimated at $12.8 billion [6]. Some industry experts estimate that the market will reach $100-250 billion by 2025 [3]. Additionally, companies in the automotive sector [7, 8, 9], aerospace sector [10, 11, 12] and many more are investing heavily in AM technology and its application to high volume manufacturing.

1.2 Problem Definition

AM still has many limitations at the moment, mainly lack of design knowledge, imperfections during the printing phase, and high costs in mass production. One significant reason and source of these limitations is that AM currently requires a lot of human expertise and supervision [3, 4].
In particular, the process of identifying whether a 3D model is manufacturable by a given 3D printer is a very time-consuming and complex task, as 3D models could have many geometric elements and the requirements of the 3D printer are very specific. In this project we want to automate this process using advanced convolutional neural networks (CNNs).

1.3 Project Goals

The overall goal of this project is to develop an inference algorithm, based on advanced CNN architectures, that is able to detect defects in 3D models, consequently classifying them as manufacturable or non-manufacturable. This is done by achieving two main technical goals:

1. Develop a data generation module that, given a 3D mesh model, returns the voxelized 3D model with a defect augmented into it. The defect is a hole that goes through the 3D voxelized model at a random location within the object.

2. Develop a deep learning module that offers a choice of several supported architectures, trains the deep learning models, and carries out performance analysis on the classified 3D models.

1.4 Project & Report Outline

The remainder of the report is organized as follows: We introduce similar literature work in section 2. This is followed by section 3 which gives an insight into the data source which has been exploited and corresponding data preprocessing steps (i.e. data selection, cleaning, normalization, alignment and voxelization). Subsequently, an approach for the synthetic data generation is outlined in section 4, which is called DefectorTopDownView. As the name already suggests, this algorithm inserts defects into existing 3D models. Section 5 showcases an in depth investigation of the ability of different neural network architectures to classify the generated data as printable or non-printable 3D models. This is followed by section 6 which presents the final results. The report is concluded by a discussion which is carried out in section 7.

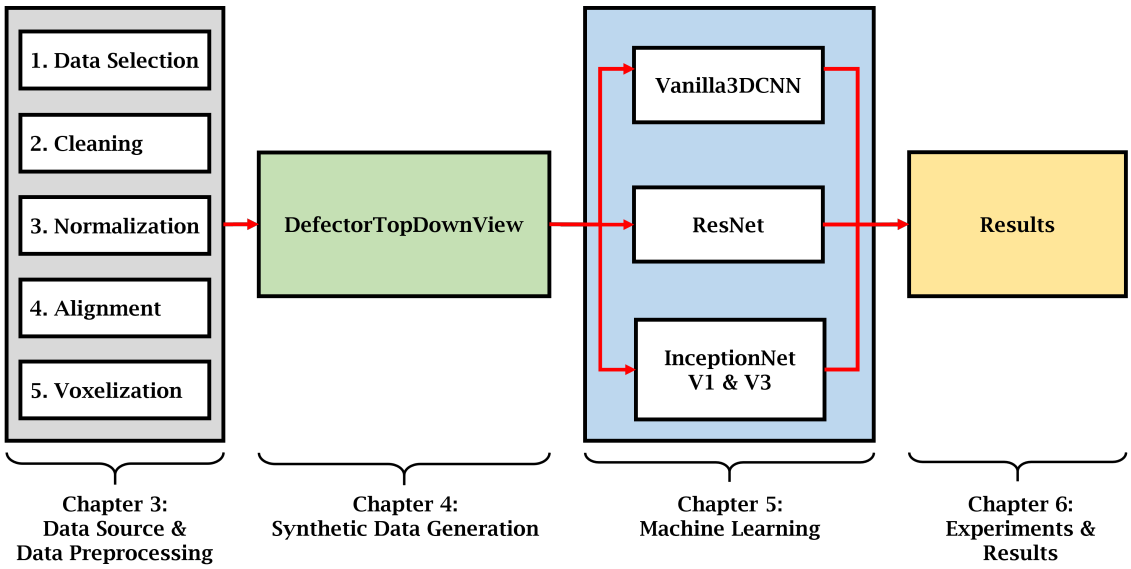


Figure 1: Overview Project and Report Structure.

2 Related Work

This sections aims for giving an overview about any similar work we are aware of, with respect to additive manufacturing and/or deep learning.

Generally, 3D CNNs have been used for object recognition based tasks using the voxelized shape of corresponding models. Balu et al., demonstrated the feasibility of using 3D CNNs to identify local features of interest using a voxel-based approach. The 3D CNN used, was able to learn local geometric features directly from the voxelized model, without any additional shape information. The deep learning based design for manufacturability (DLDFM) tool developed in the paper has successfully learned the complex Design for Manufacturability (DFM) rules for drilling which include not only depth-to-diameter ratio of the holes but also their position and type. As a consequence, the DLDFM framework out-performs traditional rule-based DFM tools which are currently available in computer-aided design (CAD) systems. The 3D CNN was able to identify the local geometric features irrespective of the external object shape, even in the non-representative test data. In contrast, within our project, we approached the problem by developing data driven model to classify whether a given 3D model is printable or not.

Fast and reliable industrial inspection is a main challenge in manufacturing scenarios. However, the defect detection performance is heavily dependent on manually defined features for defect representation. Daniel Weimer presented an approach for visual defect detection using deep machine learning, namely deep CNN. The performance of the proposed approach is measured on a data set representing 12 different classification categories with visual defects occurring on a heavily textured background. As opposed to hand-crafted features on pixel level, the CNN architectures are engineered by investigating different hyperparameters involved in the process. In this way, systems for optical quality control (OQC) can be developed with minimum prior knowledge within the problem domain. In our project, we proposed various deep learning architectures to classify a given 3D model with a very high accuracy.

Banadaki et al. proposed an automated quality grading system that uses a CNN for the additive manufacturing process. The CNN model was trained online using images of the internal and surface defects in the layer-by-layer deposition of materials. Furthermore, it was tested online by studying the performance of detecting and classifying the failure in the AM process at different extruder speeds and temperatures. The model achieved an accuracy of 94% and specificity of 96% for classifying the quality of the printing process to five classes in real-time. The online model offers an automated non-contact quality control inspector that eliminates the need for manual inspection of parts after they are completely built.

3 Data Source & Data Preprocessing

This chapter aims for giving an insight into the data source that was utilized and all consequent preprocessing steps applied. The holistic workflow is depicted in the block diagram below.

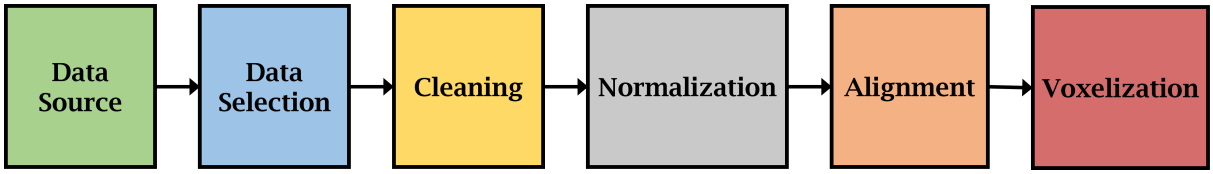


Figure 2: Holistic representation of the data source & data preprocessing workflow.

3.1 Data Source

As a baseline dataset the ABC dataset was used, which consists of computer-aided design (CAD) models for research of geometric deep learning methods and applications [16]. Generally, the dataset consists of one million mechanical 3D models, which are scattered over one hundred data chunks of equal size. The file format is given as standard triangle language (stl) which is one of the most common representations of 3D models [17]. The complexity level of the different 3D models is distributed from rather easy (i.e. simple 3D cube) to very complex examples (i.e. car engine block).

3.2 Data Selection

As a consequence of the non-uniform complexity distribution of the 3D models across the dataset, we tackled the problem by filtering the data right before any preprocessing is applied. Furthermore, in order to be able to make the assumption of the each selected model being printable, the complexity has to be restricted. From exploratory data analysis, it can be concluded that if the general complexity of the model increases, more triangular meshes are required to represent the exact shape of the 3D model. Consequently, this is an indicator, that more complex models result in stl files which may be much bigger than less complex 3D models (i.e. there is a positive correlation between non-complexity and printability). To this end, the final data, which is used for any further preprocessing steps, is selected according to a file size threshold, that was identified empirically.

In fact, the total number of files that are considered depends first of all on the number of initially selected data chunks. Building up on that, the file size parameter basically works like a cutoff criterion. First, the actual size of every 3D model is identified, followed by an ascending ordering of those. Finally, the selected file size parameter determined which files were regarded for further preprocessing steps and which were disregarded. In our current approach, three data chunks were taken and all models having a file size smaller or equal to 25 kilo bytes (kb) were selected, resulting in a total of 3457 initial models, that were considered for further preprocessing steps and in addition to be printable.

3.3 Data Preprocessing

Once a decision was made with respect to the 3D models that were considered for any further preprocessing steps, all models had to undergo cleaning, normalization, alignment and voxelization step. In the following, those steps are explained in more detail.

3.3.1 Cleaning

For a 3D mesh to be 3D printable, certain conditions have to be satisfied. The first condition is water tightness and the second condition is having manifold geometry. Water tightness is satisfied if the 3D mesh has no holes and if all normal vectors of the 3D mesh are facing outwards [18]. Manifold geometry is satisfied if no edges of the 3D mesh are shared by more than two faces [19]. The data cleaning applied involves cleaning of a mesh's vertices, edges, and triangles. The cleaning of mesh vertices is performed by applying the following:

1. Removing vertices that have identical coordinates
2. Removing vertices that are not referenced in any triangle

The cleaning of mesh edges is performed by removing non-manifold edges.

The cleaning of mesh triangles is performed by applying the following:

1. Removing triangles that reference the same three vertices (f.e. triangle1: [v1, v2, v3], triangle2: [v2, v1, v3])
2. Removing triangles that reference a single vertex multiple times in a single triangle (f.e triangle1: [v1, v2, v2])

All these cleaning functionalities were implemented using Open3D [20] functions.

In addition to that, the following filters defined by PyMeshlab, a Python library that interfaces to MeshLab, a well-known open source application for editing and processing 3D triangle meshes, were used:

- remove_duplicate_face: Two faces are considered equal if they are composed by the same set of vertices, regardless of the order of the vertices.
- remove_duplicate_vertices: If there are two vertices with same coordinates they are merged into a single one.
- repair_non_manifold_edges_by_removing_faces: For each non Manifold edge it iteratively deletes the smallest area face until it becomes 2-Manifold.

3.3.2 Normalization

Normalization is necessary in order to ensure having stable gradients while training neural networks [21]. Since different models come with different sizes and their vertices lie in different ranges, it is important to apply a normalization step to ensure that the vertices components lie in the same range. The main obstacle with 3D vertices is that all components have to be scaled with the same factor to not destroy the appearance/scale of the model. In traditional normalization, where each feature is scaled according to the range that the feature lies in independent of the other features will not work with the components of 3D vertices. To that end, normalization in two steps was applied. First, the model were centered on the origin by finding the center of mass of the model and then translating it to the origin.

$$\begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{1}$$

where (x_t, y_t, z_t) are the components of a three-dimensional point after translation to the origin, (x, y, z) is the three-dimensional point before translation, and (c_x, c_y, c_z) is the center of mass of the 3D points of a mesh. The center of mass of the 3D points is computed by finding the mean of the components.

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \frac{1}{n} \sum_{i=i}^{n} \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{2}$$

The points were then scaled such that all components (X, Y, Z) of the 3D vertices lie in $[-1, 1]$ range. This was done by finding the range of each component and then scaling the points by the component range with the maximum value.

$$\begin{bmatrix} x_s \\ y_s \\ z_s \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \div \max_{axis=x,y,z} \{range(axis)\} \tag{3}$$

3.3.3 Alignment

3D objects are generally represented in an arbitrary orientation [22]. The aim of this part of preprocessing is to have a common orientation of the models. An idea that is generic enough to be applied to all models is to find the axis of the minimum moment of inertia (MOI). The axis of the minimum moment of inertia is the axis around which most of the mass of the model is wrapped [23]. Thus, the resistance to rotation around that axis is minimal. The alignment proposed is to find the axis with the minimum moment of inertia and align it with one of the coordinate axes (X, Y, Z). To find the axis of minimum moment of inertia, principal component analysis (PCA) is applied on the inertia tensor. The eigenvector with the minimum eigenvalue is the axis of minimum moment of inertia for a given model [24]. Listed bellow are the elements of the inertia tensor. (x, y, z) denote the position vectors.

$$I_{xx} = \sum_{n=1}^{N} y^2 + z^2 \quad (4) \qquad I_{zz} = \sum_{n=1}^{N} y^2 + x^2 \quad (6) \qquad I_{xz} = \sum_{n=1}^{N} xz \quad (8)$$

$$I_{yy} = \sum_{n=1}^{N} x^2 + z^2 \quad (5) \qquad I_{xy} = \sum_{n=1}^{N} xy \quad (7) \qquad I_{yz} = \sum_{n=1}^{N} yz \quad (9)$$

The inertia matrix is a constant real symmetric matrix. The inertia matrix is formulated as follows:

$$I = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix} \quad (10)$$

The second step is the alignment between the axis of minimum moment of inertia and one of the coordinate axes. This is done by finding a rotation matrix that can be applied on the vertices of a mesh such that the two unit vectors (axis with minimum moment of inertia and a coordinate axis) are aligned [25]. Finding a rotation matrix R that rotates a unit vector a onto another unit vector b is done as outlined in the following.

Let $v = a \times b$ (cross product), $s = \|v\|$ (Frobenius norm of a vector) and $c = a \cdot b$ (dot product a), then the rotation matrix R is given by Equation 11:

$$R = I + [v]_\times + [v]_\times^2 \frac{1}{(1+c)} \quad (11)$$

where $[v]_\times$ is the skew-symmetric cross-product matrix of v, and can be expressed as

$$[v]_\times \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} \quad (12)$$

After finding the rotation R, matrix multiplication is applied with the $N \times 3$ points matrix,

$$P = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ \vdots & \vdots & \vdots \\ v_{N1} & v_{N2} & v_{N3} \end{bmatrix} \cdot R \quad (13)$$

where P is a $N \times 3$ matrix that represents the 3D points after applying the rotation. Figure 3 shows the results of the alignment algorithm using the equations mentioned above for an exemplary 3D model.

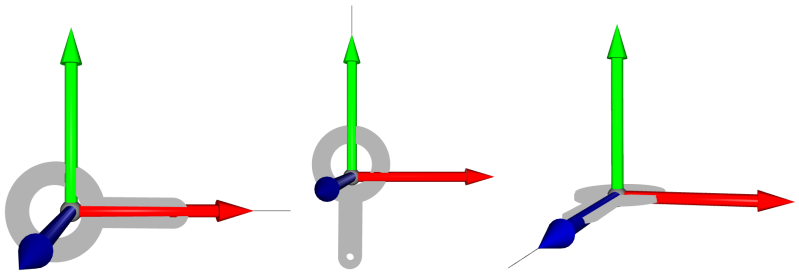


Figure 3: (a) Alignment of min MOI axis and x-axis (b) Alignment of min MOI axis and y-axis (c) Alignment of min MOI axis and z-axis.

3.3.4 Voxelization

All previously handled preprocessing steps were applied in the mesh status level of the 3D models. The final preprocessing step is devoted to the transformation of those 3D mesh models (see Figure 4) into a 3D voxel representation (see Figure 5). Generally, a voxel can be regarded as a pixel in a three-dimensional space. The reason why a voxelized representation of the 3D models was chosen is due to the following aspects. First, a voxel can be simply considered as an upscaling of 2D pixels into 3D. Thus, the assumption was made that state-of-the-art 2D vision models will show a rather good performance on the voxelized data by transforming them into 3D neural network architectures (i.e. 3D convolutions). Second, the voxel representation also allows an easy introduction of defects into existing models, as the 3D model is given as an easy interpretable 3D matrix and defects can be added by simply removing voxels (i.e. setting matrix elements to zero).

Generally, there are several techniques how to represent 3D models using voxels [26]. In this work, the occupancy grid was used [27]. It is the most straight forward method used in order to represent voxels. Occupancy grids represent a binary array/tensor with a value of 1, if the voxels are intersecting with the surface and 0 everywhere else.

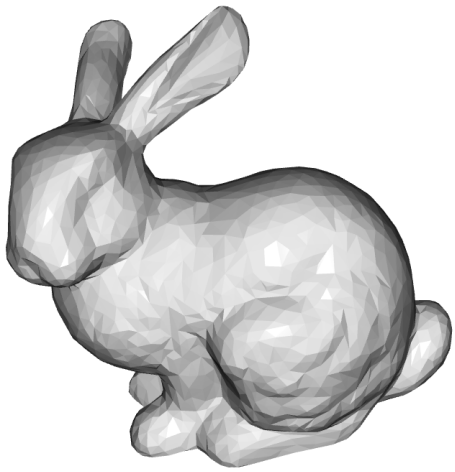


Figure 4: Stanford bunny in mesh format.

As there were a total of 3457 models initially selected, the process of transforming all models into a corresponding voxel representation is very compute intensive. Predominately, the computational demand depends on the final decision with respect to the occupancy grid resolution. As this was chosen to be a grid of $128 \times 128 \times 128$ voxels, a GPU accelerated voxelization method was leveraged in order to obtain the transformed models within a reasonable amount of time [28]. The GPU voxelization algorithm is built on the "Möller–Trumbore" intersection algorithm [29], which describes the ray-triangle intersection method used, and CUDA [30].

The voxelization step concludes the data preprocessing cycle. After all steps, an initial 3D mesh model now is transformed into a $128 \times 128 \times 128$ voxel representation model that is in addition cleaned, normalized and axis aligned. As the assumption was made, that all initial selected models are considered to be printable, a technique has to be found in order to create non-printable models in order to finalize the dataset for any further deep learning steps. As a consequence, the next chapter will give an in-depth insight into the synthetic data generation which solely focuses on the task of adding defects to all existing models and label them accordingly.

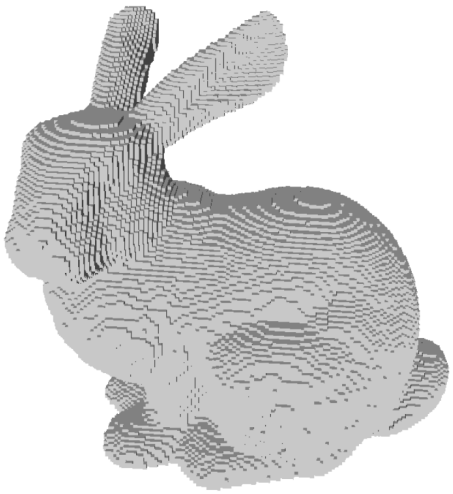


Figure 5: Stanford bunny in voxel format: Occupancy grid representation.

4 Synthetic Data Generation

After selecting the right models out of the ABC dataset and preprocessing the data, the next step is to extend the current dataset by models that are non-printable, in order to get the dataset that can later be used for classification, which we call AMC dataset (compare to Figure 6). This chapter aims to describe how printable and non-printable models are defined and how defects with different complexity are added to the 3D models.

Figure 6: Holistic representation of the synthetic data generation workflow.

Defects are defined here as holes that are added to the 3D model and can have an influence on the manufacturability of the resulting model. It was assumed that the selected models from the ABC dataset are printable and that the parameters deciding about manufacturability can be described by the radius of the hole and the distance between the border of the defect and the border of the model, that will be referred to as border. That results in four hyperparameters:

- r_p: radius printable
- r_{np}: radius non-printable
- b_p: border printable
- b_{np}: border non-printable.

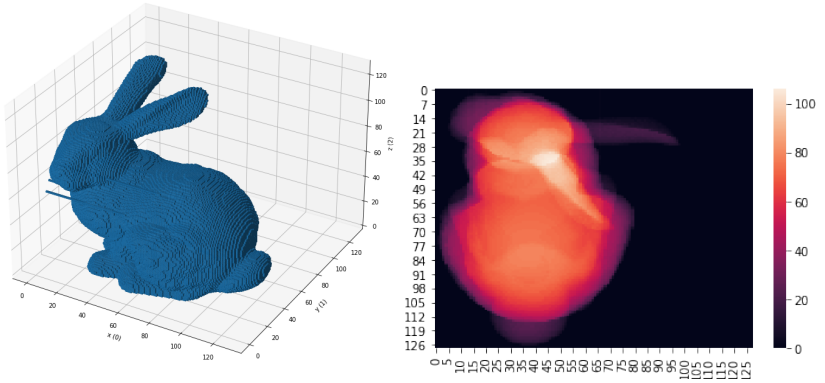
For a model dimension of 128x128x128 it was identified that a radius printable of 5, a radius non-printable of 10, a border printable of 5 and a border non-printable of 3 works best, which consequently were used to generate the AMC dataset. By combining a radius and a border out of the given hyperparameters the following models can be generated:

- **model with a non-printable defect in the middle** (m^m_{np}): radius of r_{np} and border larger or equal than b_{np} (Figure 8 (d))
- **model with a printable defect in the middle** (m^m_p): radius of r_p and border larger or equal than b_p (Figure 8 (f))
- **model with a non-printable defect at the border** (m^b_{np}): radius of r_p and border of smaller or equal than b_{np} (Figure 8 (h)).

Given the hyperparameters and the definition of the defect types, the problem of adding defects is reduced to finding an offset such that the requirements for each defect type is fulfilled. For this, two algorithms were developed and evaluated. The first algorithm, called DefectorExhaustive, uses an exhaustive approach which finds a suitable radius,

axis, and location of a cylinder to create a defect. The second algorithm, called DefectorTopDownView, projects the 3D data of the model onto a 2D grid and uses the resulting information to select a suitable offset. Since the second algorithm can add complex defects in a more robust way, it was decided to use this method for the generation of the AMC dataset (for more details about the first algorithm refer to the appendix).

The DefectorTopDownView algorithm is described in algorithm 1. The main idea of this algorithm was to complement a trial and error approach for adding defects, by transforming the 3D model data in a way that it can be used to add defects in a more efficient and controlled manner. The inspiration for this was given by the heatmap visualization. A heatmap is a 2D visualization of a 3D matrix [31]. The 3D model data is projected onto the (x, y) grid by summing the 3D matrix of the model over the z-axis. Since the voxelized models are represented as a 3D grid, by summing over the z-axis the color represents the number of voxels having the same (x, y) indices. The resulting 2D matrix was named **top_down_view** (*tdv*) (an example of a *tdv* is given in Figure 7 (b)).



(a) (b)

Figure 7: (a) Voxelized Stanford bunny. (b) *tdv* of the Stanford bunny represented as a heatmap. Scale on the right indicates the number of voxels at each (x, y) point.

By removing all points that have a previously set distance to the border of *tdv* in x and y direction a subsample of indices is generated, that defines suitable offsets for a (radius, border) combination, as seen in Figure 8 (a), (d) and (g) and described in step 5 and 10 in algorithm 1. Then, a random offset is selected and the *tdv* will again be used to check if the hole is fully in the model (compare to step 8 of algorithm 1). This additional check is needed to check the given criteria in the non-axis aligned directions. If all checks are met, a hole, that is aligned with the z-axis at the determined offset will be added. The output of the DefectorTopDownView algorithm is zero, two (input model without an defect and m^m_{np}) or four 3D models (previous two models, m^m_p and m^b_{np}), depending on whether it could find an offset satisfying the given conditions (example of the results are visualized in Figure 8 (b), (c), (e), (f), (h) and (i)). Note here that due to this fact, the defect types might not be equally distributed.

Applying the DefectorTopDownView the given dataset was extended to 7430 3D models. A few resulting models are visualized in Figure 20 in the appendix.

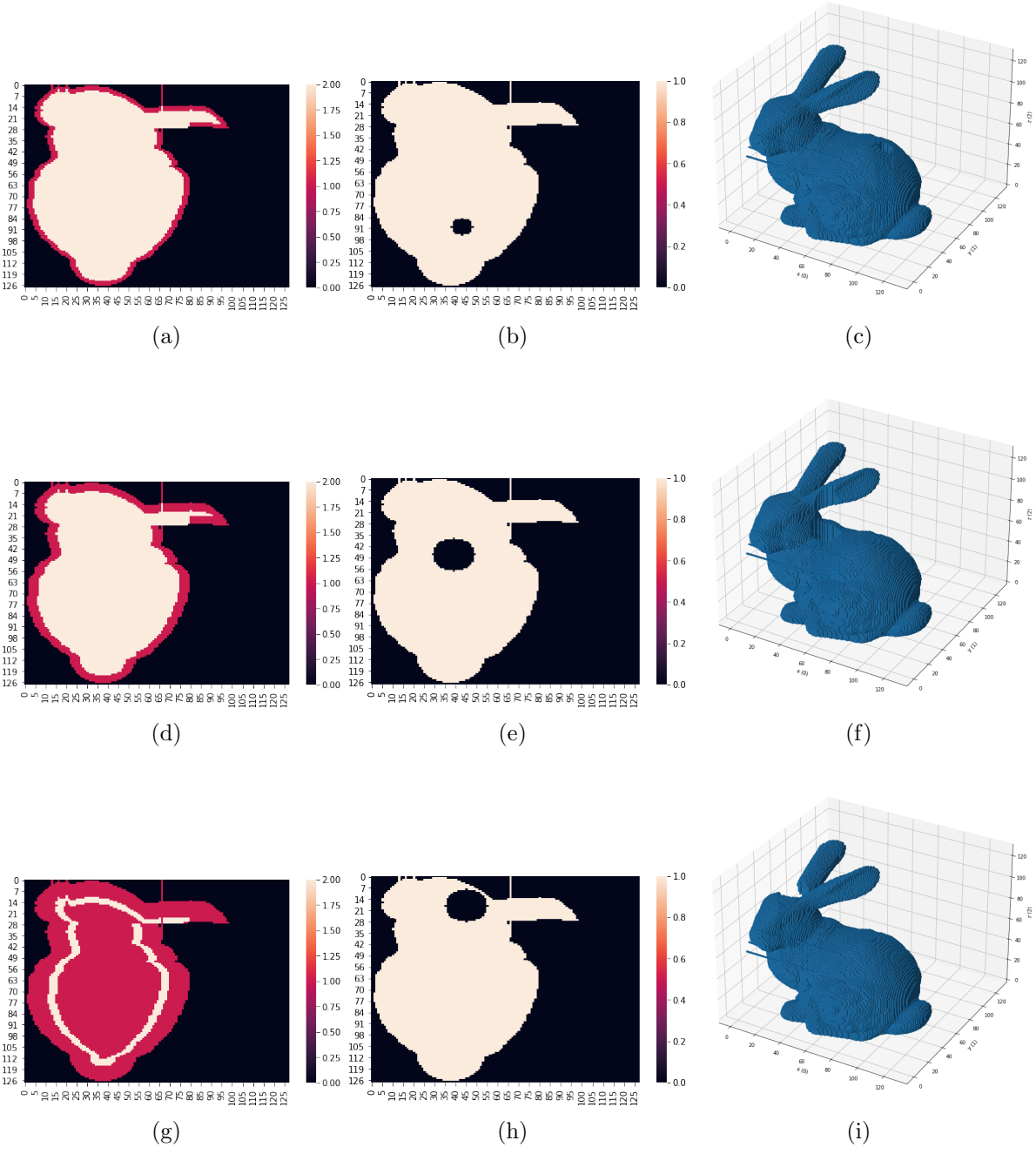


Figure 8: (a), (d) and (g) binary *tdv* of the Stanford bunny. The red area visualizes the indices that are removed by step 5 for (a) and (d), and step 15 for (g) of algorithm 1 by using a border of 5 for (a), a border of 10 for (d) and a border of 14 for (g). The light area indicates possible offsets. (b), (e), (h) binary *tdv* of the Stanford bunny with an added hole of radius of 5, 10 and 10 and consequently the *tdv* of m^m_{np}, m^m_p and m^b_{np}, respectively. For finding the offset the light area of indices out of Figure 8 (a), (d) or (g) has been used. (c), (f) and (i) resulting 3D models with added defects, i.e. m^m_{np}, m^m_p and m^b_{np} of the Stanford bunny.

Algorithm 1: DefectorTopDownView

input : 3D model data in occupancy grid format (m), hole radius printable (r_p) and non-printable (r_{np}), border printable (b_p) and non-printable (b_{np}), number of trials (t)

output: 0/2/4 models containing different defects that are printable or non-printable, depending on the input model

```
1  begin
2      tdv ← sum m over z axis
3      possible_offsets ← determine non-zero indices of tdv
4      for r, b in [(r_np, b_np), (r_p, b_p)] do
5          remove all indices of possible_offsets were the corresponding points in tdv
             are b away from the border of tdv in x and y direction
6          offset ← randomly select offset from possible_offsets
7          for 0 up to t do
8              if for every point given by a hole with radius = r + b at offset the value
                 of tdv at this point is not zero then
9                  new_model ← m with added hole aligned with z-axis at offset with
                     radius r
10                 add new_model to pre_output
11                 break
12             end
13         end
14     end
15     possible_offsets ← determine points of tdv that are b_np + r_p + 1 away from the
         border of tdv in x and y direction
16     repeat code block 6-13 with radius=r_p and border=1
17     if model_with_non_printable_defect_middle in pre_output then
18         add m and model_with_non_printable_defect_middle to output
19     end
20     if model_with_non_printable_defect_border and
         model_with_printable_defect_middle in pre_output then
21         add model_with_non_printable_defect_border and
             model_with_printable_defect_middle to output
22     end
23     if output is not empty then
24         return output
25     else
26         return empty list
27     end
28 end
```

5 Machine Learning

Machine Learning is the science (and art) of programming computers so that they can learn from data [32]. Generally, there are a lot of aspirations which give a definition of machine learning. The following showcases two attempts, first a rather generic definition and second an engineering-oriented notation:

> *"Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed"*
>
> – Arthur Samuel, 1959

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*
>
> – Tom Mitchel, 1997

5.1 Architectures

In this project, a 3D convolution neural network based method is proposed in order to learn distinct local geometric features of interest within a three-dimensional object. The task at hand is a binary classification task to recognize whether a part is printable or not. Thus, the feature of interest to be learned is, whether there are holes inside the 3D models which make them non-printable. Since, it's a simple binary classification task, it is essential to obtain an appropriate feature extractor architecture which is subsequently followed by dense layers and a final sigmoid activation. The general structure of the classification task used within the project is shown in Figure 9.

Overall, four different 3D convolution neural network architectures were examined in order to perform the feature learning. Vanilla3DCNN, ResNet and both InceptionNet architectures are designed and built up on state-of-the-art 2D vision models. The following will introduce the reader into these architectures and will give a detailed outline of their specific layout.

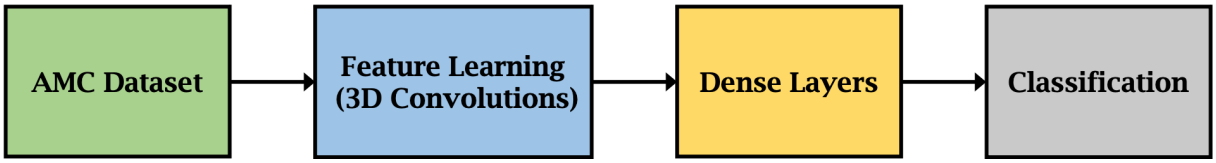


Figure 9: Holistic representation of the deep learning workflow using 3D convolutions.

Vanilla3DCNN is the simplest Neural Network used. The architecture based on the VGGNet [33] has 1.8M trainable parameters with 4 convolution layers using kernel sizes of 9, 7, 5 and 3. The detailed description of the Vanilla3DCNN architecture can be found in Figure 26 in the appendix. Maxpooling layers are added in between to reduce the spatial size of the input data. The batch normalization layers were initially absent in the

architecture resulting in failure of the model to train. These were then added in between all the CNN layers to stabilize the training process and standardize the inputs to all the layers for each mini-batch [34]. As Vanilla3DCNN is a very simple network, it requires hyperparameter tuning if stochastic gradient descent (SGD) optimizer is used. A gradient descent method based on momentum is hence recommended for this approach to ensure faster convergence.

Much of the success of deep neural networks is attributed to additional layers. The function of these layers is to progressively learn more complex features of the input data. Despite the popular meme shared in AI communities from the Inception movie stating that "We need to go Deeper", He et al. empirically showed that there is a maximum threshold for depth with the traditional CNN model. The increase in the training error with increasing depth of the neural network is due to vanishing and exploding gradients. The problem of training deep neural networks was alleviated with the introduction of the ResNet architecture[35]. For the project, a customized ResNet architecture was built. The detailed description of the architecture can be found in Figure 28 in the appendix. This architecture has 9.1M trainable parameters. The basic residual block shown in Figure 27 in the appendix contains 2 convolution layers with batch normalization and dropout layers in between and a skip connection layer that adds the output from the previous layers to the output of stacked layers. Each residual block is repeated twice. From the description Figure 28, it can be found that the number of residual blocks that are repeated are reduced as the data set size is relatively small and using a larger ResNet model would lead to overfitting of the model to the data. It is a well known fact that CNNs are very sensitive to sudden dimensional changes. In order to avoid rapid dimensionality changes, two pooling layers were added before converting the CNN layers into feature vectors followed by 1 x 1 x 1 convolutions to increase the number of channels without exponentially increasing the number of operations.

The Inception network [36] was an important milestone in the development of CNN classifiers. Prior to this, most popular CNNs just stacked convolution layers deeper and deeper, hoping to get better performance. For the project, 3D InceptionNet V1 and InceptionNet V3 were built. The detailed description of InceptionNet V1 and InceptionNet V3 can be found in Figure 30 and Figure 32 in the appendix, respectively. Salient parts in the 3D model can have large variation of size. Due to this variation, choosing right kernel size for the convolution operation is difficult. A larger kernel is preferred for information that is distributed more globally, and a smaller kernel is preferred for information that is distributed more locally. InceptionNet tries to solve the problem of choosing the right kernel size to capture salient features in the data by using multiple kernel sizes operating at the same level [37]. InceptionNet V1 has 9.6M trainable parameters. From the InceptionNet V1 module as shown in Figure 29 in the appendix, it can be observed that kernel sizes 1, 3 and 5 at the same level were used, hence providing the ability the network the ability to learn salient features of the input data of all dimensions. This InceptionNet V1 block is repeated four times to reduce the input 3D model into a feature vector. InceptionNet V3 contains 17.9M trainable parameters and is an improvement over the InceptionNet V1 architecture. Large reduction in input dimensions due to the filter size of 5 x 5 x 5 leads to decrease in the accuracy of prediction hence making the neural network prone to loss

of information. 5 x 5 x 5 kernel is hence replaced with 3 x 3 x 3 kernel and the depth of each inception block is increased. From Figure 31 (see appendix), it can be observed that the the depth of a branch containing 3 x 3 x 3 kernel is increased. This particular branch containing 5 x 5 x 5 kernel is replaced with 3 x 3 x 3 kernel to avoid sudden reduction in the input dimensions. The detailed description of the InceptionNet V3 architecture is shown in Figure 31 (see appendix).

5.2 Implementation Details

Since the task is a binary classification task, sigmoid activation function is used in the final dense layer while the layers in between use ReLU activation functions [38]. Use of ReLU activation function in all the other layers prevents saturation of parameters, hence preventing vanishing gradients allowing the model to learn faster [38].

Binary cross-entropy (BCE) [39] loss also called as sigmoid cross-entropy loss is a sigmoid activation plus a cross-entropy loss. Unlike softmax loss [39], it is independent for each vector component (class), meaning that the loss computed for every CNN output vector component is not affected by other component values. That's why it is used for multi-label classification, where the insight of an element belonging to a certain class should not influence the decision for another class. It is called binary cross-entropy loss because it sets up a binary classification problem between C=2 classes for every class in C. So when using this loss, the following formulation of cross-entropy loss for binary problems is often used:

$$BCE = -\sum_{i=1}^{C=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

where, t_1 and s_1 are the predicted label and ground truth label of class C_1 respectively and t_2 = 1 - t_1 and s_2 = 1 - s_1 are the predicted label and ground truth label of class C_2 respectively.

Gradient descent [39] is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters [39]. The learning rate η determines the size of the steps the algorithm takes to reach a (local) minimum. In the project, Adam optimizer [40] was used to optimize the parameters with a learning rate of 0.00001.

Regarding the dataset it has to be mentioned, that all of the 7430 data samples of the AMC dataset were used in order to train and validate the respective experiments. Within that, a proper dataset split of 80 % belonging to the training data and the remaining 20 % belonging to the validation data were used. The neural networks were trained and validated in 100 epochs.

5.3 Infrastructure & computational Resources

Artificial Intelligence in general has been around since the middle of the last century. Since that time, challenges solved by machine learning techniques, especially deep learning and corresponding neural networks, faced an ever-expanding boom in terms of computational demand. Above all, this is mainly due to rather easy accessibility to big datasets, open-source frameworks and the continuous development of more and more sophisticated neural network architectures, which are available online and therefore represent a key driver in order to come up with extensive algorithms. Training a large neural network on a single machine with a single central processing unit (CPU) can take days or even weeks. As a consequence, powerful computational resources are indispensable in order to tackle the problem of continuously increasing computational requirements. This subsection introduces the infrastructure that was exploited, distributed learning across multiple graphic processing units (GPUs) using horovord [41] and monitoring neural network learning lifecycle using MLflow [42].

Conducting deep learning experiment on different neural network architectures, tuning hyperparameters and using massive dataset can clearly be considered inefficient without having access to powerful compute nodes. Especially, for the task at hand, where a single 3D model already consists of more than two million voxels (i.e. $128 \times 128 \times 128$) the input dimensions ruthlessly comes to light and thus make the availability of GPUs indispensable [38]. Thanks to Leibniz Rechenzentrum (LRZ), we were equipped with powerful resources all along the way and were able to distribute experiments across multiple devices simultaneously in order to excessively speed up the training and hyperparameter tuning procedure. To this end, we have luckily been able to make use of up to eight Nvidia Tesla V100 GPUs in parallel. This kind of GPU is exceptionally powerful as it offers 640 tensor cores per device in order to accelerate the training process. The exact course of action of how the parallelization is done is outlined in the subsequent paragraph.

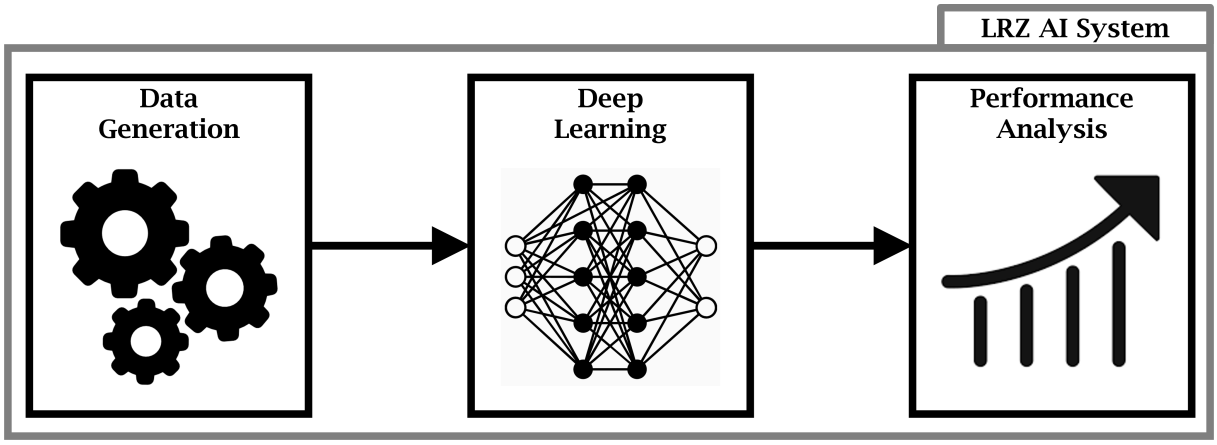


Figure 10: Big picture representing the project workflow using the LRZ infrastructure.

Apart from computations with respect to neural networks (i.e. gradient calculation using backpropagation or weight updating using various optimization techniques) GPUs can also be leveraged to perform general purpose computation tasks. As a consequence, the provided resources were additionally used within the data generation step. Therein, the heavy computation of transforming the 3D mesh models to 3D voxel models is completed by unloading the task to a CUDA capable device [43, 30].

The data parallelism strategy exploited in this project relies on the distributed deep learning framework called horovod, that was developed by researchers at UBER [41]. As deep learning in general is commonly compute intensive and thus resource demanding, GPU utilization or even multi-GPU usage is essential in order to counteract ever lasting training processes. Conceptually, a distributed deep neural network task across multiple GPU devices can be outlined as follows: Every device which is intended to be involved in the training process gets a copy of the to be executed training script. Every GPU individually reads a chunk of the provided data, performs the forward pass and computes the gradients with respect to the deviation of the predicted and the actual label using an appropriate loss function (i.e. in this case sigmoid). The gradients across those multiple devices are averaged using Baidu's algorithm [44], which in turn is based on a paper that describes bandwidth optimal all-reduce algorithms for clusters of workstations [45]. Once the averaged gradients are computed, the model gets updated and the next iteration again starts by reading data chunks for each device which is involved. The schematic workflow of the distributed training process is depicted in Figure 11 below.

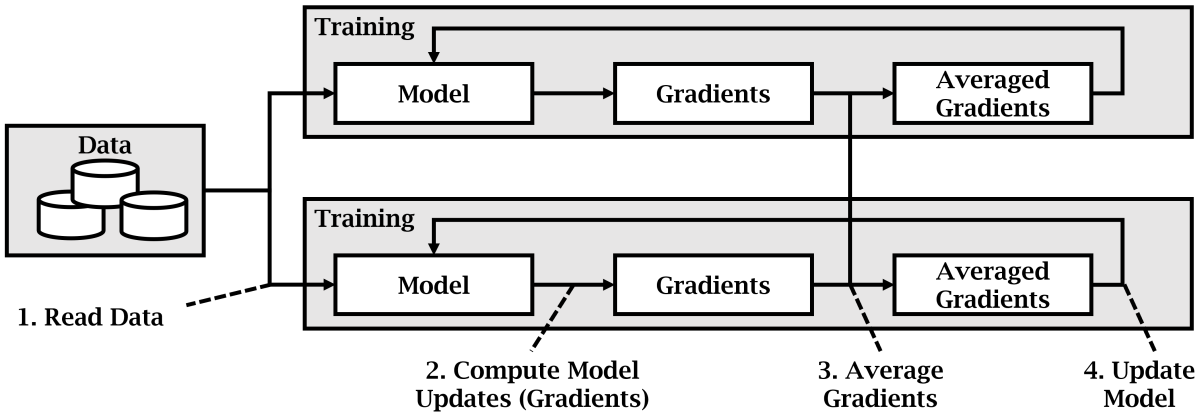


Figure 11: Schematic workflow of distributed training using two GPUs, inspired by [41].

During the life-cycle of any deep learning model, different hyperparameters are tuned, in order to perform best possibly at the given task. Furthermore, different state-of-the-art architectures are commonly investigated, which often constitutes the baseline for further hyperparameter optimization. In order to not loose track of any architecture experiments which have been performed and corresponding hyperparamter tuning, MLflow is used to monitor the life-cycle of all our attempts. Basically, MLflow is an open source platform to streamline machine learning development, including tracking experiments, packaging code into reproducible runs, and sharing and deploying models [42].

6 Experiments & Results

"All models are wrong, but some are useful."

– George Box, 1976

Generally, every machine learning model represents merely an approximation of the hidden data pattern to be learned. Thus, as George Box put it in 1976, every model will never represent the exact behaviour of the task at hand. Even though, the respective data pattern can not be learned exactly, machine learning models can be useful by only using approximations of the data features. This section aims for giving an insight into the experiments that were carried out alongside with a corresponding performance analysis and their respective results.

In order to perform an in-depth performance analysis of our trained neural network models, different metrics were employed. A very common way in order to evaluate the output quality of a classifier is to make use of the so-called receiver operation characteristic (ROC) and the corresponding area under the curve (AUC). Typically, ROC curves are exploit in binary classification problems in order to examine the performance of the classifier. In general, those curves are comprised of a true positive rate (tpr) indicated on the y-axis, and a false positive rate (fpr) indicated on the x-axis. Thus, the top left corner of the plot represents the most desirable point, that exhibits a false positive rate of zero, and a true positive rate of one. Furthermore, a larger AUC score and the steepness of the ROC curve are valid indications, in order to conclude a better performing classifier, as the tpr is maximized while the fpr is minimized [32].

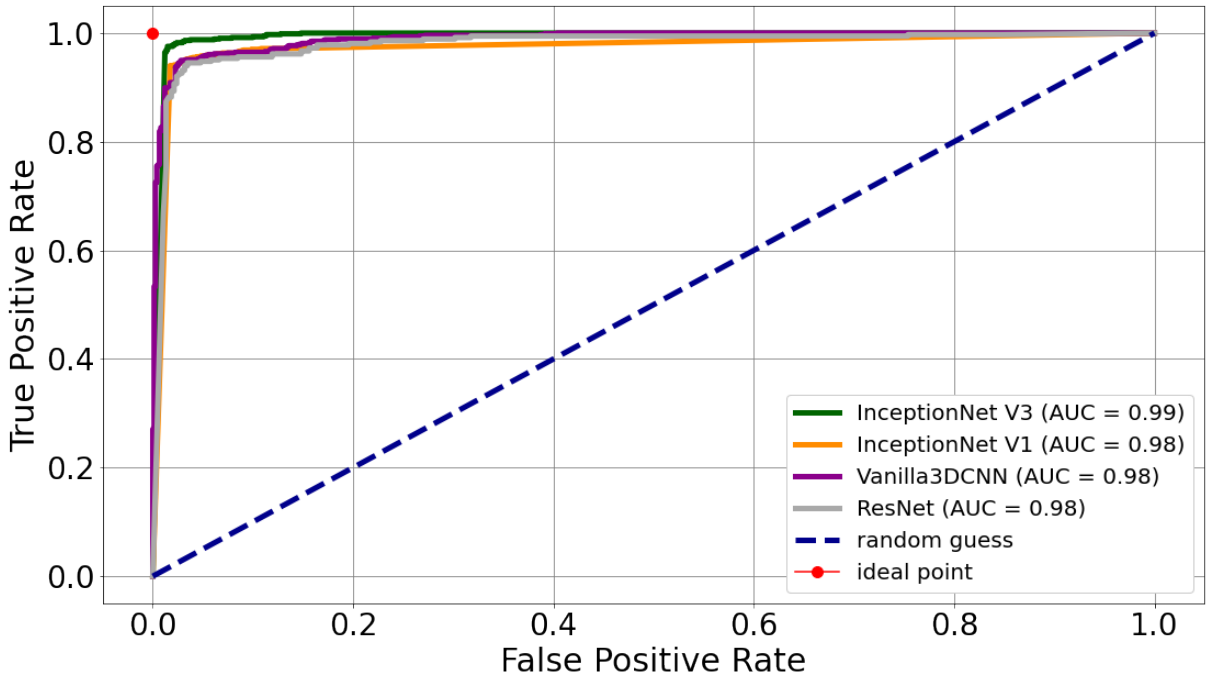


Figure 12: ROC curve with AUC, indicating the performance of different architectures.

From Figure 12 above, it is clearly visible that both InceptionNet, as well as the ResNet and Vanilla3DCNN architectures are capable of handling the binary classification task in a rather convincing manner. According to the ROC curves and their corresponding AUC score depicted above, it can be concluded that the best performing model is InceptionNet V3, followed by InceptionNet V1, ResNet and Vanilla3DCNN. Nevertheless, as both InceptionNet models are based on the same intuition (i.e. inception block) with different parameter settings (e.g filter size), it was decided to only consider the slightly better performing InceptionNet V3 model alongside the ResNet and Vanilla3DCNN architectures in the remainder of this performance evaluation.

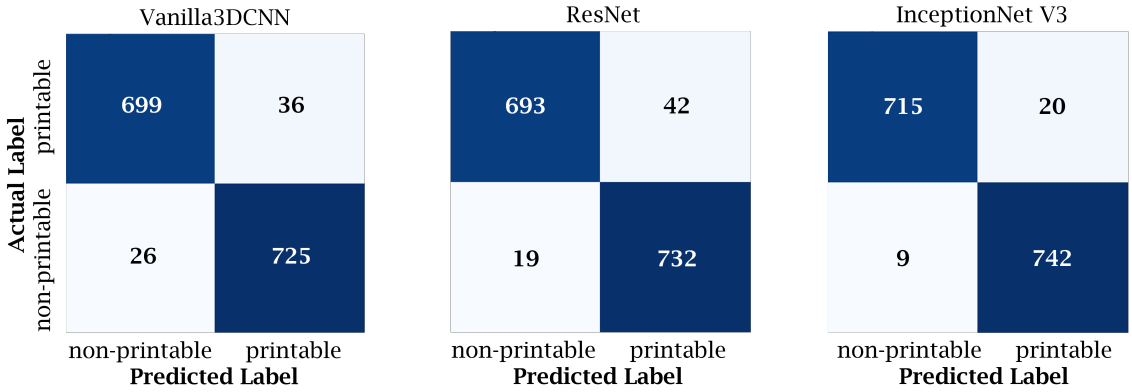
Besides a rather graphical performance evaluation using the ROC curve, another common way to assess the output quality of a classifier is to rely on numeric measurements. To this end, the F1 score metric, which represents the harmonic mean of precision and recall, was used [32]. Thus, the F1 score mirrors a trade-off between precision and recall and weights both at the same importance. Generally, the highest possible value score is one, indicating perfect precision and recall, and the lowest possible value is zero. The latter happens, if either the precision or the recall is zero.

$$F1 = \frac{2}{recall^{-1} + precision^{-1}} \qquad recall = \frac{tp}{tp + fn} \qquad precision = \frac{tp}{tp + fp} \tag{14}$$

The overall goal of any classifier is to be as good as possible in correctly predicting the positives and negatives. When comparing the 3D models classified by the deep neural networks and the ground truth, there are following possibilities (the notation is given with respect to the context of additive manufacturing):

- **true positive (tp):** A printable 3D model is classified as printable.
- **true negative (tn):** A non-printable 3D model is classified as non-printable.
- **false positive (fp):** A printable 3D model is classified as non-printable.
- **false negative (fn):** A non-printable 3D model is classified as printable.

Since the AMC dataset, that was used for training the neural networks, is balanced in terms of printable and non-printable 3D models, the accuracy is already giving rather good unbiased indication of the actual model performance. Besides an overwhelming accuracy of 98.1 % for InceptionNet V3, 95.9 % for ResNet and 95.8 % for Vanilla3DCNN on the validation dataset, all architectures were also able to achieve outstanding F1 score values of 0.980, 0.959 and 0.957 respectively. In contrast to straight forward numerical measurements like the F1 score or the accuracy metric, another common approach in order to evaluate the performance is to leverage confusion matrices [32]. They are especially beneficial as they provide a more detailed analysis than just a numeric value. Within a confusion matrix, diagonal elements represent the number of samples for which the predicted label is equal to the true label, whereas off-diagonal elements are those that are mislabeled by the classifier. The higher the number of diagonal elements within the confusion matrix, the better, consequently indicating many correct predictions.



(a) (b) (c)

Figure 13: Confusion Matrix: (a) Vanilla3DCNN (b) ResNet (c) InceptionNet V3

Figure 13 shows the resulting confusion matrix for the InceptionNet V3, ResNet and Vanilla3DCNN architecture. The main goal of having the majority of the elements on the diagonal of the matrix is fulfilled in all cases, whereas the total number of off-diagonal elements is minimal. Consequently, all classifiers predicted most of the samples correctly that resulted in a rather high accuracy score and F1 value as aforementioned. The few off-diagonal elements on which the architectures failed to predict correctly (i.e. false positives and false negatives), will be elaborated at the end of this section by an explicit failure analysis on 3D model level.

Finally, one of the most apparent and common ways to get an insight into the training behaviour of any model and to get an impression about its performance, is to look at the development of the loss and accuracy for both training and validation data over the epochs. Corresponding plots for all of the three in-depth investigated architectures is provided in Figure 14. Within the plot, the x-axis represents the training process in terms of epochs, while the improvement is indicated as accuracy on the left and as loss on the right y-axis. Overall, it can be summarized that all architectures can handle the binary classification problem with a very high final accuracy and low loss value. Moreover, the development of the curves (see Figure 14) show that there is a continuous improvement without any sudden or unexpected drop within the given number of epochs. Thus it can be concluded, that all architectures exhibit a very good fit in order to handle the classification task at hand.

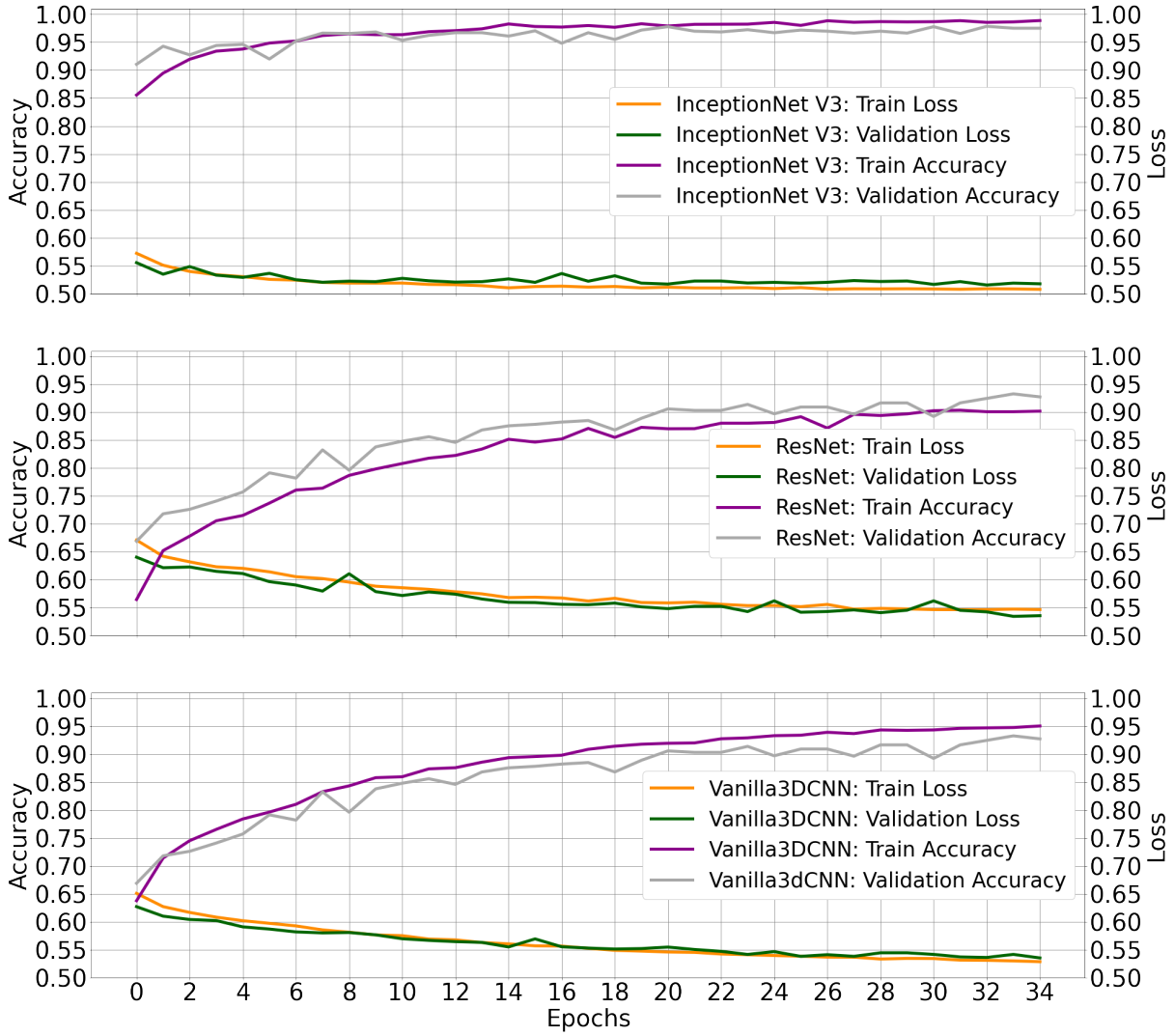


Figure 14: Development of loss & accuracy on the train & validation data depicted for the first 35 epochs. Top: InceptionNet V3, middle: ResNet, bottom: Vanilla3DCNN.

In order to obtain a more accurate assessment of the AMC dataset quality and the performance of the deep learning architectures, an in detail failure analysis was carried out. In this part, the focus is on the validation dataset, therein specifically on incorrectly predicted data samples by InceptionNet V3. The wrongly predicted models were split according to the different defect types and were visualized. Please note, that due to the implementation of the DefectorTopDownView algorithm, the defects are not equally distributed in the AMC dataset. As Table 1 shows, the most incorrectly classified models have a non-printable defect in the middle of the model.
Examples of these models are visualized in Figure 15. Since this missclassification mostly happens for large models, a potential reason for this could be that, it is hard to detect a hole in large models, especially with a hole that has a small radius. Additionally, if there is already a hole in the model, then the InceptionNet V3 can fail to detect the non-printable hole (compare to Figure 15 (b)).

Defect type	**Count**
Non-printable defect in the middle (m_{np}^{m})	12
Printable defect in the middle (m_{p}^{m})	5
Non-printable defect at the border (m_{np}^{b})	6
No defect	9

Table 1: Incorrectly predicted models by InceptionNet_V3, splitted into the different defect types. The validation dataset, consisting of 1486 randomly selected models, was used.

It is also interesting to note that nine models that had no defect were incorrectly classified as non-printable. The first identified source of this is that the used models from the ABC dataset either already have holes or are just too complex (compare to Figure 15 (c) and (d)). Therefore, the assumption that all selected models are printable does not hold and as a consequence the label for these few models are wrongly set. The second potential source could be that the 3D models contain rounded edges, that could be interpreted as a part of a hole (compare to Figure 15 (e)).

The missclassified models having a non-printable defect at the border, all have in common that the border contains barely three voxels. Mostly, there is only one removed voxel having a distance of three to the border of the model, which of course is really hard to detect (compare to Figure 15 (f)).

An identified reason for the missclassification of models having a printable defect in the middle is that a few models contain a difficult 3D structure that the DefectorTopDownView can as of right now not detect, since it uses only a 2D representation of the model to check the defect conditions and label the model accordingly (compare to Figure 15 (g), where the model is labeled as printable but due to the 3D structure the added hole at the second layer should be non-printable).

What also has to be noted, is that the InceptionNet V3 fails to classify a few models that do not show any obvious problems, regarding added defects, the labels and the models itself (compare to Figure 15 (h) and (i)).

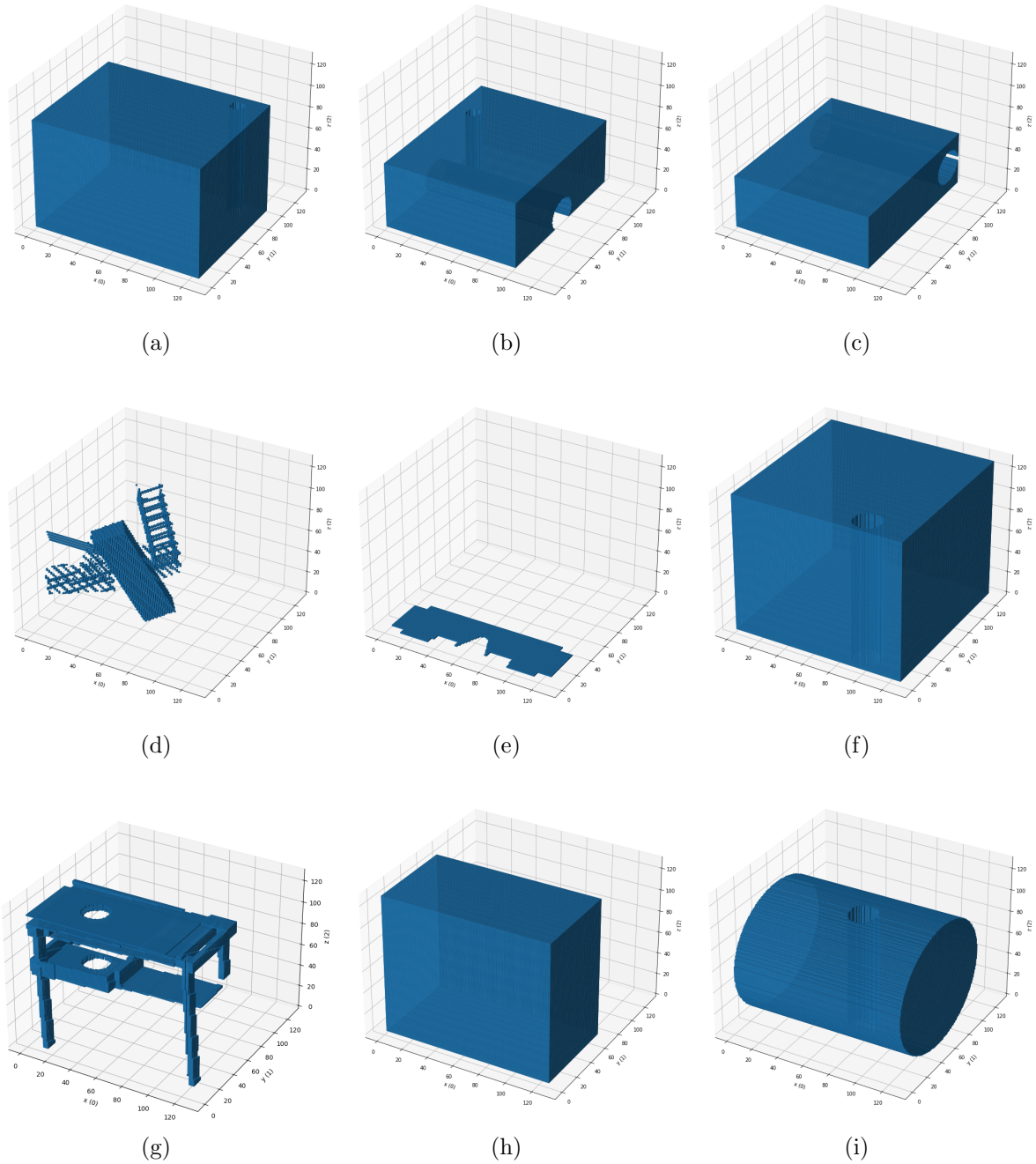


Figure 15: Visualizations of missclassified models out of the validation set by the trained InceptionNet_V3 model. (a) Example of a model having a non-printable defect in the middle. (b) Example of a model having a non-printable defect in the middle and additionally a printable hole that was already in the model coming from the ABC dataset. (c) Example of model having a hole that was already in the model coming from the ABC dataset. (d) Example of a too complex selected model, labeled as printable by the DefectorTopDownView. (e) Example of model having a rounded edge that can be interpreted as part of a removed hole. (f) Example of a model having a non-printable defect at the border, with a distance of barely three voxels between the border of the model and the removed voxels. (g) Example of a model having a printable defect according to the assumptions made by the DefectorTopDownView, but due to the shifted layers should be non-printable. (h) and (i) Examples of a model having no defect and that give no clues about why the InceptionNet V3 failes to classify it correctly.

7 Discussion

Starting from mesh models in the format of stl files, we designed a data processing pipeline with a series of transformations: normalization, alignment, cleaning and voxelization. The resulting models are considered as 3D printable. To construct the non-printable 3D models, we developed the DefectorTopDownView algorithm to insert specific defects in the original models. This algorithm allows us to insert defects in the shape of holes with different radius and border values. Thus, the resulting dataset, referred to as AMC dataset, consists of the processed original models, the models with a printable defect and the models with a non-printable defect since the added hole has a too small radius or the hole is too close to the border of the model. The DefectorTopDownView is implemented such that the number of printable and non-printable models are equally distributed within the AMC dataset.

In the deep learning pipeline we implemented and trained four different CNN architectures. The Vanilla3DCNN was our baseline model achieving an accuracy of 95.8% and a F1-Score of 0.957. The ResNet architecture had a similar performance with an accuracy of 95.9% and a F1-Score of 0.959. Furthermore, we were able to obtain a performance increase by using the InceptionNet V3 model with an accuracy of 98.1% and a F1-Score of 0.980. We think that the reason behind such a good performance is the ability of InceptionNet to learn the salient features of the 3D input data having large variation in dimension by using multiple kernel sizes at the same level.

As an extra evaluation method of the deep learning models, it would be highly interesting to generate an additional test set from another chunk of the ABC dataset or from the Thingi10K dataset [46]. Also explainable AI techniques can be applied to have a better understanding of the performances of the deep learning models.
The failure analysis led us to think of some considerations that could further enhance the quality of the AMC dataset as well as the performances of the deep learning models. For example by removing 3D models already having a hole and too complex models, f.e. by using other criteria for the selection such as a compactness measure (see appendix). Moreover, one could think about extensions of the DefectorTopDownView algorithm, such as adding defects from another axis-aligned directions and non-axis aligned directions to the 3D models, using different defect shapes and defect parameters and using more of the 3D structure of the 3D model to determine the position of the defect f.e. by using similarity checks for the area where the whole should be added (see appendix).

Since the defector is a deterministic algorithm with limited capabilities as outlined in the failure analysis, and the results of the deep learning module are highly influenced by this algorithm, we propose to try the following approaches that do not require a defector but make use of labeled data by experts. A possible idea is to use self supervision with an autoencoder as a pretrained model. Also few-shot learning methods can be tested. Another possible approach is to train generative adversarial networks (GANs) on a dataset of few complex models that are expert-labeled, and using the GAN to generate new models with complex defects.

References

[1] Brent Stucker Mahyar Khorasani Ian Gibson, David Rosen. *Additive Manufacturing Technologies*. Springer International Publishing, 3 edition, 2021. ISBN 978-3-030-56126-0. URL https://link-springer-com.eaccess.ub.tum.de/book/10.1007%2F978-3-030-56127-7#about. pg. 2.

[2] Ian GibsonDavid RosenBrent StuckerMahyar Khorasani. *Additive Manufacturing Technologiesy*. Springer International Publishing, 3 edition, 2021. ISBN 978-3-030-56126-0. URL https://link-springer-com.eaccess.ub.tum.de/book/10.1007%2F978-3-030-56127-7#about. pg. 77ff.

[3] Jörg Bromberger and Richard Kelly. Additive manufacturing: A long-term game changer for manufacturers, 2017. URL https://www.mckinsey.com/business-functions/operations/our-insights/additive-manufacturing-a-long-term-game-changer-for-manufacturers.

[4] Mehrshad Mehrpouya, Amir Dehghanghadikolaei, Behzad Fotovvati, Alireza Vosooghnia, Sattar Emamian, and Annamaria Gisario. The potential of additive manufacturing in the smart factory industrial 4.0: A review. *Applied Sciences*, 9: 3865, 09 2019. doi: 10.3390/app9183865.

[5] Mary Kathryn Thompson, Giovanni Moroni, Tom Vaneker, Georges Fadel, R. Ian Campbell, Ian Gibson, Alain Bernard, Joachim Schulz, Patricia Graf, Bhrigu Ahuja, and Filomeno Martina. Design for additive manufacturing: Trends, opportunities, considerations, and constraints. *CIRP Annals*, 65(2):737–760, 2016. ISSN 0007-8506. doi: https://doi.org/10.1016/j.cirp.2016.05.004. URL https://www.sciencedirect.com/science/article/pii/S0007850616301913.

[6] T.T. Wohlers, I. Campbell, O. Diegel, R. Huff, J. Kowen, and Wohlers Associates (Firm). *Wohlers Report 2021: 3D Printing and Additive Manufacturing : Global State of the Industry*. Wohlers Associates, Incorporated, 2021. ISBN 9780991333271. URL https://books.google.de/books?id=vGJszgEACAAJ.

[7] Einsatz von 3d-druck bei volkswagen. URL https://www.volkswagenag.com/de/news/stories/2018/12/brake-calipers-and-wheels-now-from-a-3d-printer.html#.

[8] Der additive manufacturing campus: Fahrzeugteile aus dem drucker., 2020. URL https://www.bmwgroup.com/de/news/2020/additive-manufacturing.html.

[9] Vivienne Brando. Wie der metallische 3d-druck die fertigung verändern könnte, 2020. URL https://www.daimler.com/magazin/technologie-innovation/metallischer-3d-druck-nextgenam.html.

[10] Richard Aston. 3d printing done right. URL https://www.boeing.com/features/innovation-quarterly/nov2017/feature-thought-leadership-3d-printing.page.

[11] Vivienne Brando. 3d printing - decarbonisation meets digitalisation. URL https://www.airbus.com/public-affairs/brussels/our-topics/innovation/3d-printing.html.

[12] Molly Porter Clare Skelly. Nasa looks to advance 3d printing construction systems for the moon and mars, 2020. URL https://www.nasa.gov/centers/marshall/news/releases/2020/nasa-looks-to-advance-3d-printing-construction-systems-for-the-moon.html.

[13] Aditya Balu, Sambit Ghadai, Kin Gwn Lore, Gavin Young, Adarsh Krishnamurthy, and Soumik Sarkar. Learning localized geometric features using 3d-cnn: An application to manufacturability analysis of drilled holes. *arXiv preprint arXiv:1612.02141*, 2016.

[14] Moshe Shpitalni Daniel Weimer, Bernd Scholz-Reiter. Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection. *CIRP Annals,Volume 65, Issue 1, 2016, Pages 417-420*, 2016.

[15] Yaser Banadaki, Nariman Razaviarab, Hadi Fekrmandi, and Safura Sharifi. Toward enabling a reliable quality monitoring system for additive manufacturing process using deep convolutional neural networks, 2020.

[16] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[17] T. Grimm. *User's Guide to Rapid Prototyping*. Raymond F. Boyer Library Collection. Society of Manufacturing Engineers, 2004. ISBN 9780872636972. URL https://books.google.de/books?id=o2B7OmABPNUC.

[18] David Stutz. A formal definition of watertight meshes. URL https://davidstutz.de/a-formal-definition-of-watertight-meshes/.

[19] shapeways. Tips for successful modeling.

[20] Open3d: A modern library for 3d data processing. URL http://www.open3d.org/docs/release/index.html.

[21] Sebastian Raschka. About feature scaling and normalization. URL https://sebastianraschka.com/Articles/2014_about_feature_scaling.html.

[22] Anne Verroust-Blondet Mohamed Chaouch. Alignment of 3d models. *HAL Id:hal-00804653*, 2015.

[23] James Dann and James J. Dann. *The People's Physics Book*. third edition, 2006.

[24] user197851. How to find the axis with minimum moment of inertia?, 2018. URL https://physics.stackexchange.com/questions/426273/how-to-find-the-axis-with-minimum-moment-of-inertia.

[25] Aaron. Find rotation matrix to align two vectors. URL `https://stackoverflow.com/questions/67017134/find-rotation-matrix-to-align-two-vectors`.

[26] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation, 2019.

[27] A. Elfes. Occupancy grids: A stochastic spatial representation for active robot perception, 2013.

[28] kctess5. Cuda c mesh voxelizer. URL `https://github.com/kctess5/voxelizer`.

[29] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28, 1997.

[30] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780124159334.

[31] Leland Wilkinson and Michael Friendly. The history of the cluster heat map. *The American Statistician*, 63(2):179–184, 2009. doi: 10.1198/tas.2009.0033. URL `https://doi.org/10.1198/tas.2009.0033`.

[32] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[34] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[37] Bharat Raj. A simple guide to the versions of the inception network, 2018. URL `https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202`.

[38] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[39] Kevin P Murphy. *Machine Learning-A Probabilistic Perspective*. MIT press Cambridge.

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[42] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41 (4):39–45, 2018.

[43] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020. URL `https://developer.nvidia.com/cuda-toolkit`.

[44] Andrew Gibiansky. Bringing hpc techniques to deep learning. URL `https://web.archive.org/web/20180128132031/http://research.baidu.com/bringing-hpc-techniques-deep-learning/`.

[45] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[46] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models, 2016.

[47] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[48] Neeraj Sujan Martin Herrmann, Simon Trendel. Tsdf volume reconstruction, 2015. URL `https://vision.in.tum.de/_media/teaching/ss2015/gpucourse_ss2015/kinectfusion.pdf`.

[49] Eman Ahmed, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Bjorn Ottersten. A survey on deep learning advances on different 3d data representations. *arXiv preprint arXiv:1808.01462*, 2018.

Model Selection using Compactness Parameter

The compactness parameter describes another facility how to control the 3D model complexity and thus to subselect data. It can be used in combination with the filesize or individually. The calculation of the compactness of every 3D model is applied in the mesh-representation status, where the condition of the mesh being watertight has to be fulfilled [18]. The mathematical formulation of the compactness calculation can be outlined as follows:

$$compactness = \frac{volume}{boundingVolume} \tag{15}$$

In our current approach, we are not making use of any compactness parameter, as it empirically turned out that an appropriate file size parameter is sufficient. Nevertheless, the compactness may have the ability to leverage the data selection in cases of big data scenarios.

Data Representations

This section aims for introducing additional 3D data representations which are commonly used in literature to apply deep learning. The following listing does not claim completeness, it is rather giving an overview of the most usual representations:

Depth images represent the actual physical structure of an object or a scene being kept by the camera lense. As the pixel values are representing distances instead of color values, any intensities are not incorporated. Thus, depth images are not affected by the level of ambient light (e.g sunlight) what may lead to improvements with respect to the robustness in applying deep learning.

Point clouds are clusters which represent collections of data points in a three-dimensional space. Therein, each data point is determined by a particular position which is given by x, y and z values and can be further attributed with RGB colour values. As the points are not linked among each other in the point cloud representation, high-quality geometric information of the scene or object can be kept, however also leading to a large degree of freedom and high- dimensionality.

Voxel grids are a derivative of point clouds. Generally, a voxel can be considered as an ordinary pixel which is well known from a 2D representation, however in this case in a three-dimensional space. Moreover, a voxel grid can be regarded as a quantized point cloud being of fixed size. Therein, voxels usually take values of either zero or one (i.e. occupancy grid), where the zero means that the voxel does not belong to the object and one vice-versa accordingly. Nevertheless, there are also other techniques instead of just being zero or one, used in the domain of voxels.

- **Signed Distance Function (SDF) [47]:** A signed distance function is a continuous function that, for a given spatial point, outputs the distance of the point to the closest surface, whose sign encodes whether the point is inside (negative) or outside (positive) of the watertight surface:

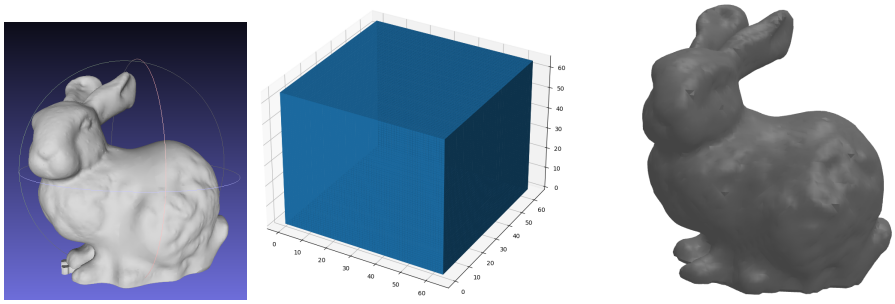
$$SDF(x) = s : x \in \mathbb{R}^3,\ s \in \mathbb{R} \tag{16}$$

  If f(x) represents the signed distance function that maps a value in the 3 dimensional vector space to a scalar value that represents the distance of the point considered in the 3D space, we have,

$$x \in \mathbb{R}^3 = \begin{cases} outside & \text{if } f(x) > 0 \\ surface & \text{if } f(x) = 0 \\ inside & \text{if } f(x) < 0 \end{cases} \tag{17}$$

  Since the signed distance function embeds more information regarding than any other voxel representations, it is easy to obtain the direction of the surface of the 3D model by looking into the gradients of the normals of the signed distance function. In the project, mesh_to_sdf function is used to create an array of N x N x N

array of SDF values. Marching cubes algorithm is used to reconstruct the mesh from the SDF values and finally it's rendered using Pyrender.The cycle of converting a mesh file into SDF voxel representation is showed in Figure 3.



(a) (b) (c)

Figure 16: (a) Mesh: Stanford bunny (b) Signed Distance Function (SDF) (c) Reconstructed bunny using marching cubes algorithm.

- **Truncated Signed Distance Function (TSDF) [47]:** Truncated signed distance function is similar to the signed distance function discussed above. the only difference lies in the fact that, this kind of voxel representation is used if the values near the surface of the object are important. It defines the limited SDF near the surface and truncates the unsigned distance above a specified threshold.The values of TSDF lies in the range of -1 to 1 as shown in Figure 4.To achieve this, the SDF values obtained in the previous section from mesh_to_sdf module are truncated to be in the range [-1,1].The mesh is then reconstructed using marching cubes algorithm and rendered using Pyrender.

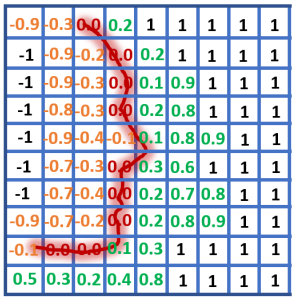


Figure 17: TSDF Representation[48]

Polygon meshes are entities that are comprised of edges, vertices and faces which together define the volume and thus can approximate the shape of a geometric object. Likewise the voxel grid representation, also the polygon mesh representation can be regarded with respect to the point cloud representation. In this case, a mesh can be considered as a three-dimensional point set, which was sampled from a set of continuous surfaces.

Furthermore, mesh faces can not only be quadrilateral as depicted below, they can also be triangular or a convex polygon.

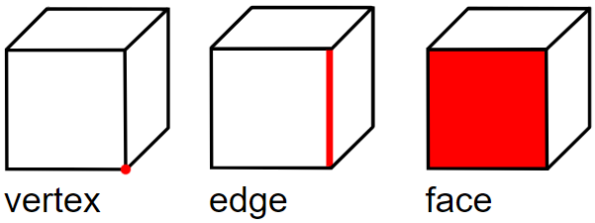


Figure 18: Example: Vertex, edge and face of a cube.

Multi-view representations are as the name already suggests, a collection of two-dimensional images of an object or scene from multiple perspectives. It is the simplest way to apply deep learning, as the 3D context is represented in 2D but still allows (restricted) justifications about the geometric structure.

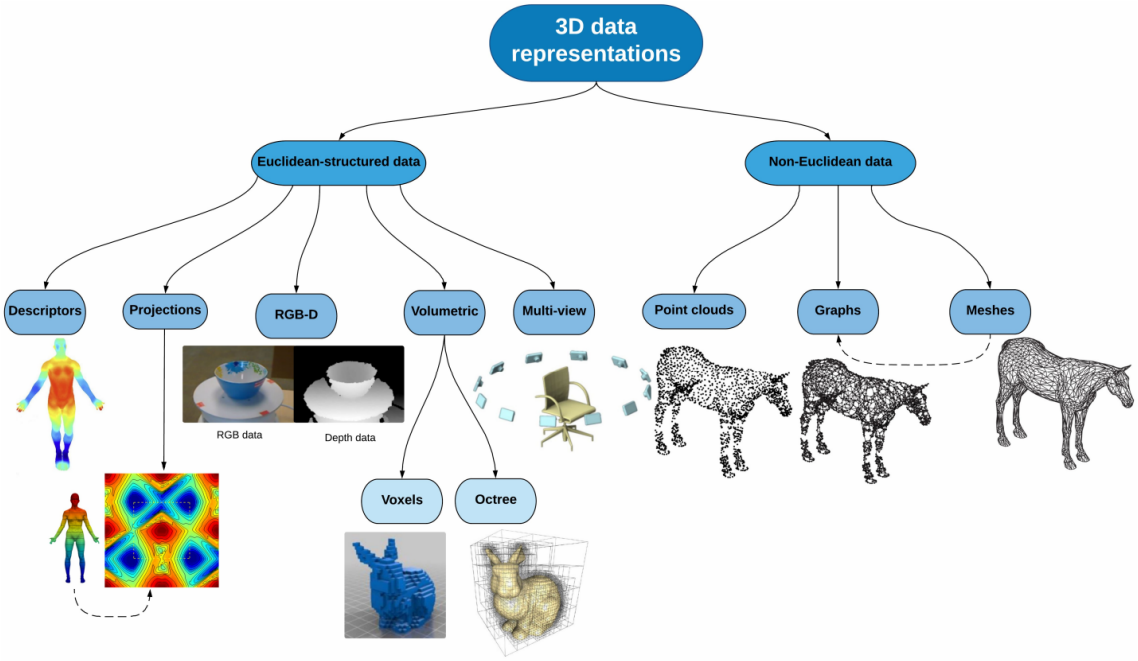
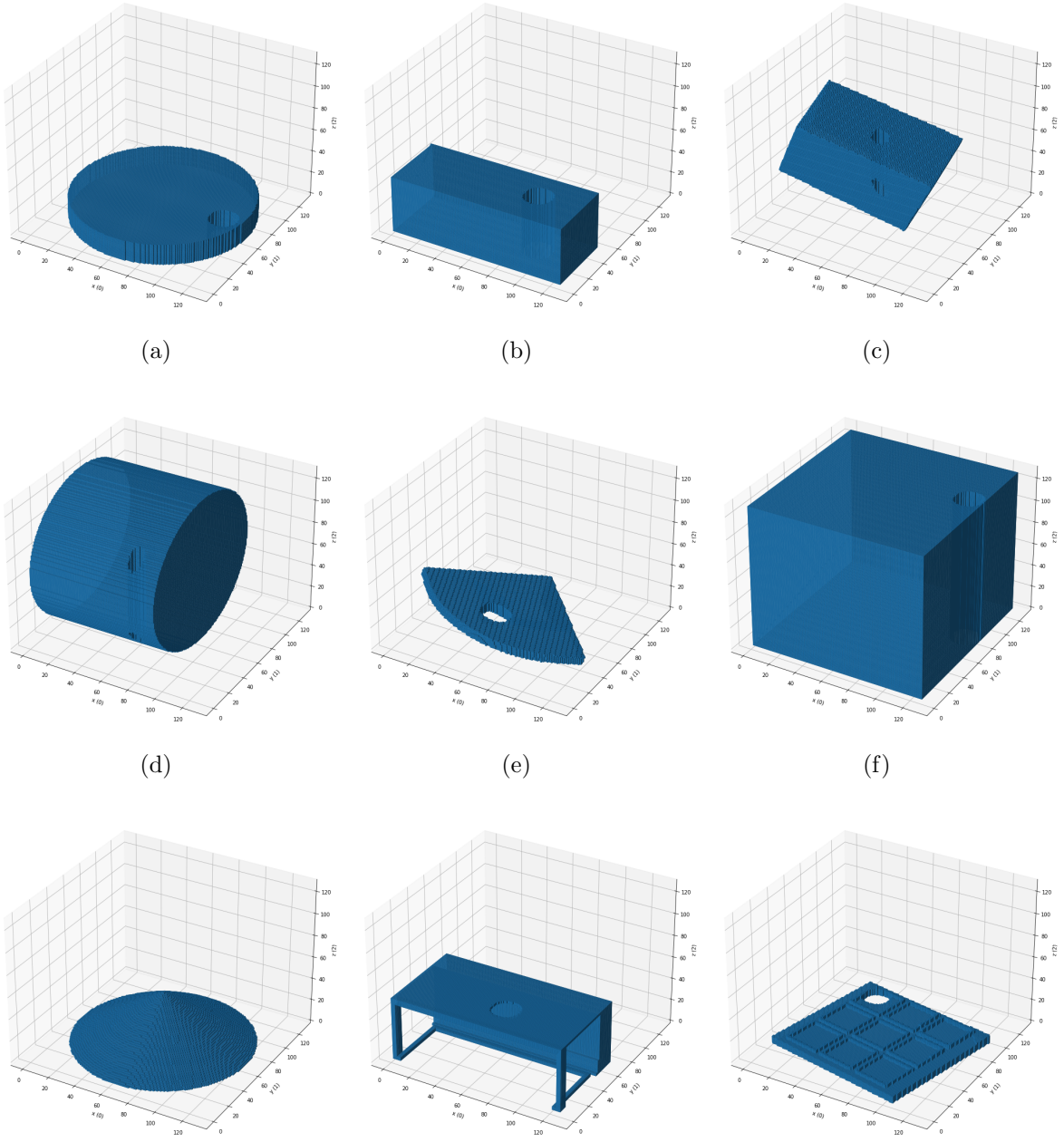


Figure 19: Overview: Different 3D data representations. Source: [49]

AMC Dataset Examples



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 20: Examples of the AMC dataset. (a), (f) and (i) have a non-printable defect since it is too close to the border of the 3D model (m^b_{np}). (b), (e) and (h) have a printable defect (m^m_p). (c) and (d) have a non-printable defect since its radius is too small (m^m_{np}) and (g) is a model with no defect. Note that all visualized models are also in the AMC dataset without the defect.

DefectorExhaustive

The motivation behind this defector is to design an algorithm that given a voxelized 3D model, decides on a suitable hole to be augmented into the voxelized model. The hole is augmented by placing a cylinder within the model and removing the voxels within the cylinder. The cylinder is defined by three parameters, namely:

- *radius*: size of the cylinder radius
- *axis*: axis through which the cylinder is defined (X, Y, Z)
- *location*: location of the cylinder center

To find the three parameters that define the cylinder, the algorithm uses four hyperparameters:

- d_{max}: maximum possible size of the cylinder diameter
- *trials*: number of trials of finding a cylinder center
- $voxels_{remain}$: number of voxels that should remain in a 1D side after removing the voxels

Through experiments on models with dimensionalty of 128x128x128, we set the hyperparameters to the following values. $d_{max} = 10$, $trials = 20$, and $voxels_{remain} = 30$.

The algorithm finds the three cylinder parameters in the same order as stated. The first parameter is the radius. To determine an optimal radius, each one of the coordinate axes (X, Y, Z) are tested. Given an axis, to find a suitable cylinder radius the following procedure is applied:

1. Find the perpendicular 2D plane to the axis being tested
2. Get the length of each 1D side out of the 2D plane
3. Choose the smaller side
4. Starting with a defined maximum cylinder diameter $x = d_{max}$:
   (a) subtract x from the smaller side
   (b) if more than $voxels_{remain}$ voxels remain, choose this diameter
   (c) else, set $x = x - 1$ and repeat (a)

If the smallest possible diameter (which is 2) can not be used, it is not possible to create a hole in the given model.

The second parameter is the axis. To determine an optimal axis, each one of the coordinate axes (X, Y, Z) are tested. For each axis, the following procedure is applied:

1. Find the radius of the cylinder through that axis using the previously explained procedure
2. Choose the axis that provides the largest radius

The third parameter is the center location. Given a cylinder radius and axis, to find the location of the cylinder center the following procedure is applied:

1. Get the voxels out of the occupancy grid
2. For a number of trails defined by the hyperparameter: *trials*, randomly choose voxels:
   (a) skip a voxel that is too close to the plane boundaries
   (b) find the area of the circle defined by the voxel as a center and the radius as computed previously
3. Choose the voxel that has the maximum area
4. Make sure that the chosen voxel has an area greater than or equal to a full circle

If the chosen voxel has a surrounding area less than a full circle, it is not possible to create a hole in the given model. Figure 21 shows some samples of the defects added by the ExhaustiveDefector.

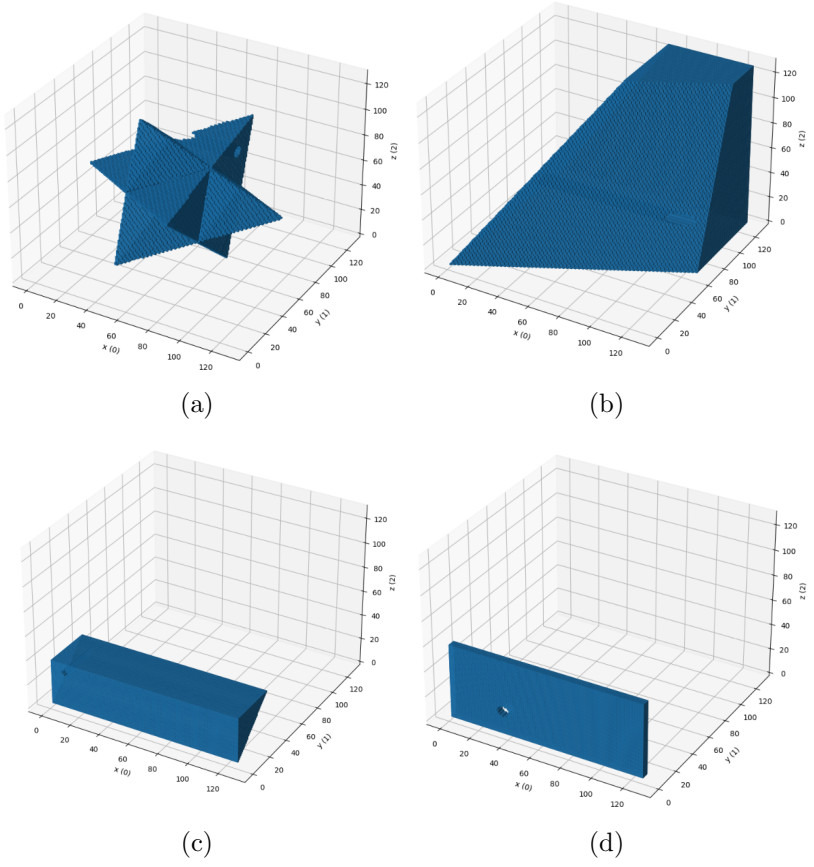
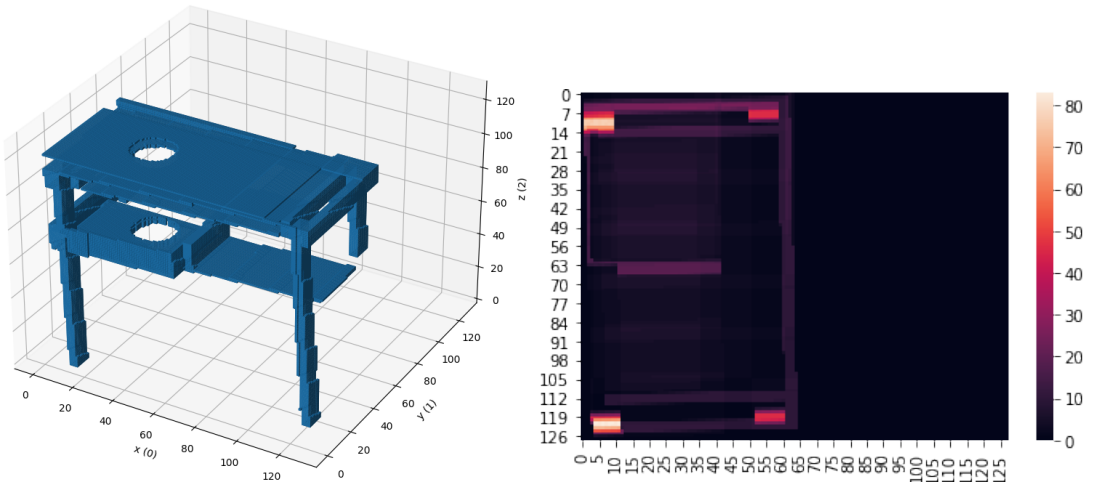


Figure 21: Four data example outputs of the ExhaustiveDefector approach.

DefectorTopDownView Similarity Check Add-on



(a) (b)

Figure 22: (a) Voxel model labeled by algorithm 1 as manufacturable, however on the second layer the hole is too close to the boarder making this model non-manufacturable (b) *tdv* of the model visualized in (a).

A identified drawback of algorithm 1 is that the information given for the *tdv* could be misleading. For example, if two layers of voxels are stacked onto each other in the z-dimension with a slight gap in between but are moved in the (x, y) plane such that a hole added through the first layer would be printable but a hole through the second layer would be non-printable, since it lies too close to the border of the second layer. This case has for example been observed and is visualized in Figure 22 (a).
We therefore developed a global and a local uniformity check for the area that will be removed by the defect. Both can be added to the if-clause in step 8 of algorithm 1. The global uniformity check calculates the difference between the maximum and the minimum of the *tdv* at the indices that will be removed by the defect at the determined offset and compares it to a given threshold. The local uniformity check works similar but with the difference that the difference will be calculated for each given index using its direct neighbors. The benefit for the local approach would be that small step wise changes in the z-dimension could still be accepted. Both checks, given the right threshold value, can prevent the problem described above. However, the cost of using this would be that a lot of models will be filtered out, if they are not perfectly uniform, and especially since the difference of the *tdv* in the critical area for adding the defect to the model visualized in Figure 22 (a) is only around ten voxels (Figure 22 (b)). Therefore, for the generation of the AMC dataset, these checks were not included, tolerating the few models that are potentially wrongly labeled.

DefectorTopDownView Rotation Add-On

The purpose of this part is to insert rotated holes with random angles as a generalization of the second method described above. Three approaches were tested:

Rotated holes insertion using Scipy

- The first step is to rotate with random angles ϕ_x, ϕ_y, ϕ_z the object around respectively x,y and z axes. In order to do that we use the scipy library and more precisely the method scipy.ndimage.interpolation.rotate.

- After rotating the model we use DefectorTopDownView algorithm to find a suitable offset defined by the coordinates (x_{offset}, y_{offset}) of the voxels.

- Once the $offset = (x_{offset}, y_{offset})$ defined we will select the indices of the cylinder centred around this $offset$.

Since there is some information lost after the rotation of the model, we prefer here to only rotate back the cylinder and remove its indices from the original model.

Unfortunately, this approach couldn't be used since there was an issue with the rotation function defined by the scipy library. In fact this function always adds an unknown value to the indices making the rotation back of the cylinder indices different from the original ones. Therefore the insertion of the hole was not coherent.

Rotated holes insertion using Rotation Matrices

To overcome the problem from the first approach we define a rotation function based on rotation matrices. The idea is to multiply the coordinates which here are the indices of the voxels by the following rotation matrices:

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\phi_x) & -sin(\phi_x) \\ 0 & sin(\phi_x) & cos(\phi_x) \end{pmatrix}, R_y = \begin{pmatrix} cos(\phi_y) & 0 & sin(\phi_y) \\ 0 & 1 & 0 \\ -sin(\phi_y) & 0 & -cos(\phi_y) \end{pmatrix}$$

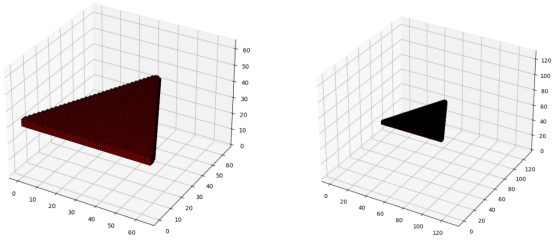
$$R_z = \begin{pmatrix} cos(\phi_z) & -sin(\phi_z) & 0 \\ sin(\phi_z) & cos(\phi_z) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

After applying the rotation, the new coordinates are transformed to an occupancy grid. This occupancy grid is the input of the algorithm DefectorTopDownView that will define the offset coordinates $offset = (x_{offset}, y_{offset})$. Then we select the coordinates of the cylinder centered around this $offset$.
In the last step and similarly to the first approach, we only rotate back the cylinder coordinates using the rotation matrices. And we remove these coordinates from the original object.

Unfortunately the negative values of indices after rotation are hard hard to work with and the transformation to occupancy doesn't give the desired results.

Basic holes insertion in rotated model

The idea of this approach is to add a padding to the 3D models before applying the rotations. For example we add a padding $p = 32$ from each side for models with a 64 resolution. The following figure shows a model before and after padding:



(a) (b)

Figure 23: (a) 3D model with 64 resolution, (b) 3D model with 128 resolution.

Then the model is rotated randomly around x,y and z axis using the function scipy.ndimage.interpolation.rotate from scipy library. Afterward, the algorithm DefectorTopDownView is used to find a suitable offset. A hole is inserted by removing the coordinates of a cylinder centered around the defined offset through the z axis. The following hyperparameters were used in the algorithm DefectorTopDownView:

- $r_p = 6$: radius printable
- $r_{np} = 3$: radius non-printable
- $b_p = 2$: border printable
- $b_{np} = 5$: border non-printable.

The final step is to rotate back the hole model.

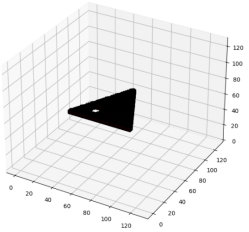


Figure 24: 3D model with a randomly rotated hole.

Detailed Description of the Architectures Used

Vanilla3DCNN

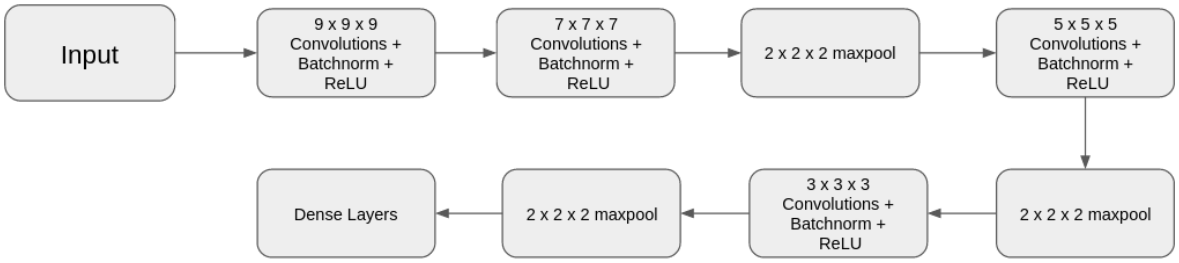


Figure 25: Vanilla3DCNN architecture.

Vanilla3DCNN Architecture		
Layer Name	**Output Size**	**CNN - Layers**
conv_1	120 x 120 x 120 x 32	9 x 9 x 9/1,32
conv_2	114 x 114 x 114 x 64	7 x 7 x 7/1, 64
Maxpool layer	57 x 57 x 57 x 64	2 x 2 x 2 maxpool,stride 2
conv_3	53 x 53 x 53 x 96	5 x 5 x 5/1, 96
Maxpool layer	26 x 26 x 26 x 96	2 x 2 x 2 maxpool,stride 2
conv_4	24 x 24 x 24 x 128	3 x 3 x 3/1, 128
Maxpool layer	12 x 12 x 12 x 128	2 x 2 x 2 maxpool,stride 2
Average pool layer	11 x 11 x 11 x 128	2 x 2 x 2 maxpool,stride 1
Maxpool layer	1 x 1 x 1 x 128	11 x 11 x 11 maxpool,stride 1
FC_1		32-d fc,ReLU
FC_2		Scalar value, Sigmoid

Figure 26: Detailed description of Vanilla3DCNN architecture used in the project.

ResNet

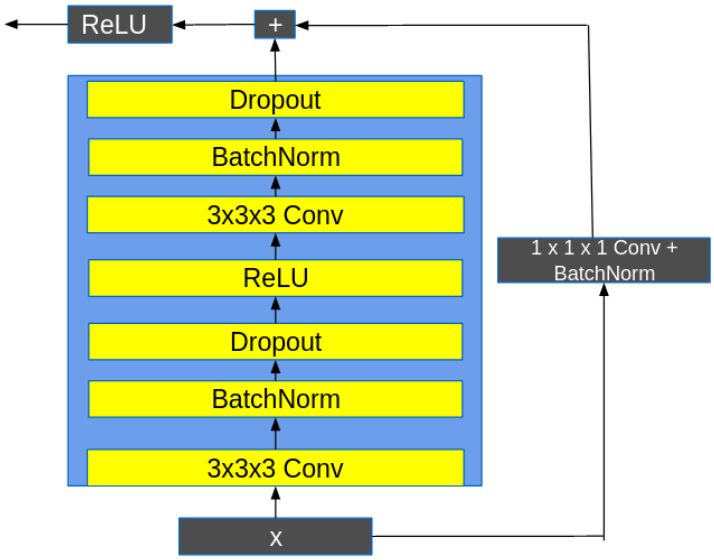


Figure 27: Basic layout of a residual block showing skip connections [35].

ResNet Architecture		
Layer Name	**Output Size**	**CNN - Layers**
conv_1	64 x 64 x 64 x 64	5 x 5 x 5/2,64 (Same)
	32 x 32 x 32 x 64	3 x 3 x 3 maxpool,stride 2
conv_2	32 x 32 x 32 x 64	(3 x 3 x 3/1,64) x 2
		(3 x 3 x 3/1,64) x 2
conv_3	16 x 16 x 16 x 128	(3 x 3 x 3/2,128) x 2
		(3 x 3 x 3/2,128) x 2
conv_4	8 x 8 x 8 x 256	(3 x 3 x 3/2,256) x 2
		(3 x 3 x 3/2,256) x 2
Maxpool layer	4 x 4 x 4 x 256	5 x 5 x 5 maxpool,stride 1
Average pool layer	1 x 1 x 1 x 256	4 x 4 x 4 average pool,stride 1
conv_5	1 x 1 x 1 x 1024	(1 x 1 x 1,256)
FC_1		512-d fc,ReLU
FC_2		128-d fc,ReLU
FC_3		64-d fc,ReLU
FC_4		Scalar value,Sigmoid

Figure 28: Detailed description of ResNet architecture used in the project.

InceptionNet V1

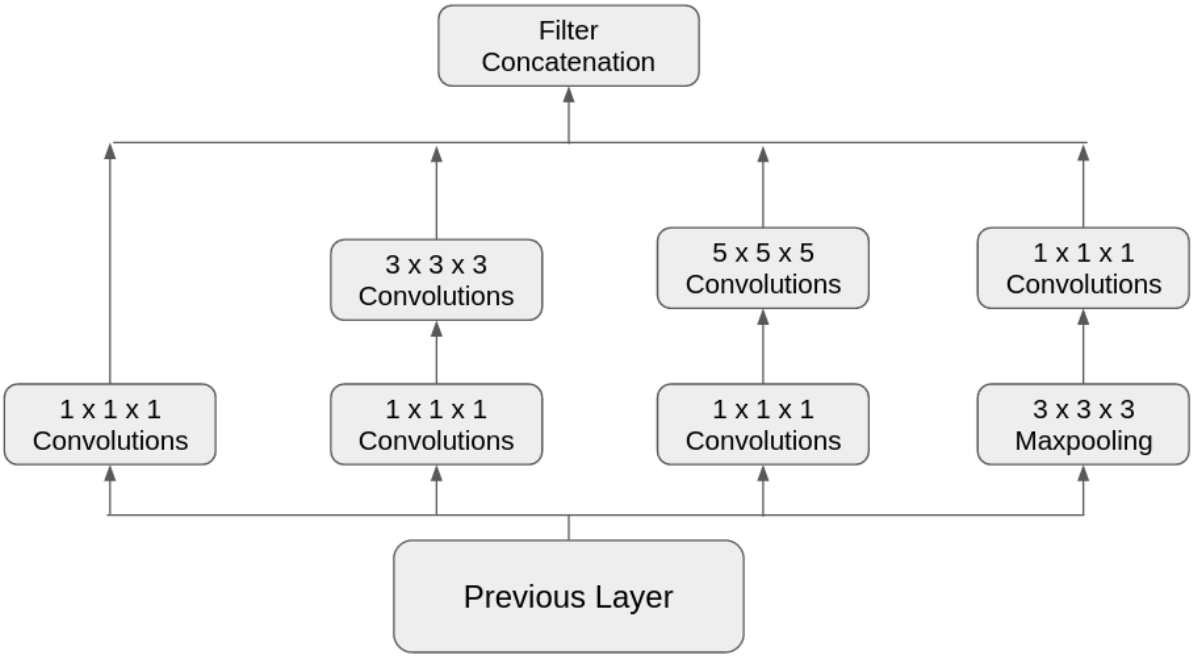


Figure 29: InceptionNet V1 block [36].

InceptionNet_v1 Architecture									
Type	**Patch_size/Stride**	**Output_size**	**Depth**	**#1 x 1 x 1**	**#3 x 3 x 3 reduce**	**#3 x 3 x 3**	**#5 x 5 x 5 reduce**	**#5 x 5 x 5**	**Pool Proj**
Convolution	5 x 5 x 5/1	64 x 64 x 64 x 64	1						
Max pool	2 x 2 x 2/2	32 x 32 x 32 x 64							
Convolution	3 x 3 x 3/1	32 x 32 x 32 x 192	1						
Max pool	2 x 2 x 2/2	16 x 16 x 16 x 192							
Inception(3a)		16 x 16 x 16 x 256	2	64	96	128	16	32	32
max pool	2 x 2 x 2/2	8 x 8 x 8 x 256	0						
Inception (4a)		8 x 8 x 8 x 512	2	192	96	208	16	48	64
Max pool	2 x 2 x 2/2	4 x 4 x 4 x 512	0						
Inception(5a)		4 x 4 x 4 x 1024	2	384	192	384	48	128	128
Maxpool	2 x 2 x 2/2	2 x 2 x 2 x 1024	0						
Inception(6a)		2 x 2 x 2 x 1024	2	384	192	384	48	128	128
Average pool	2 x 2 x 2/1	1 x 1 x 1 x 1024	0						
Dropout	p = 0.4								
FC1		512							
FC2		64							
FC3		1							

Figure 30: Detailed description of InceptionNet V1 architecture used in the project.

InceptionNet V3

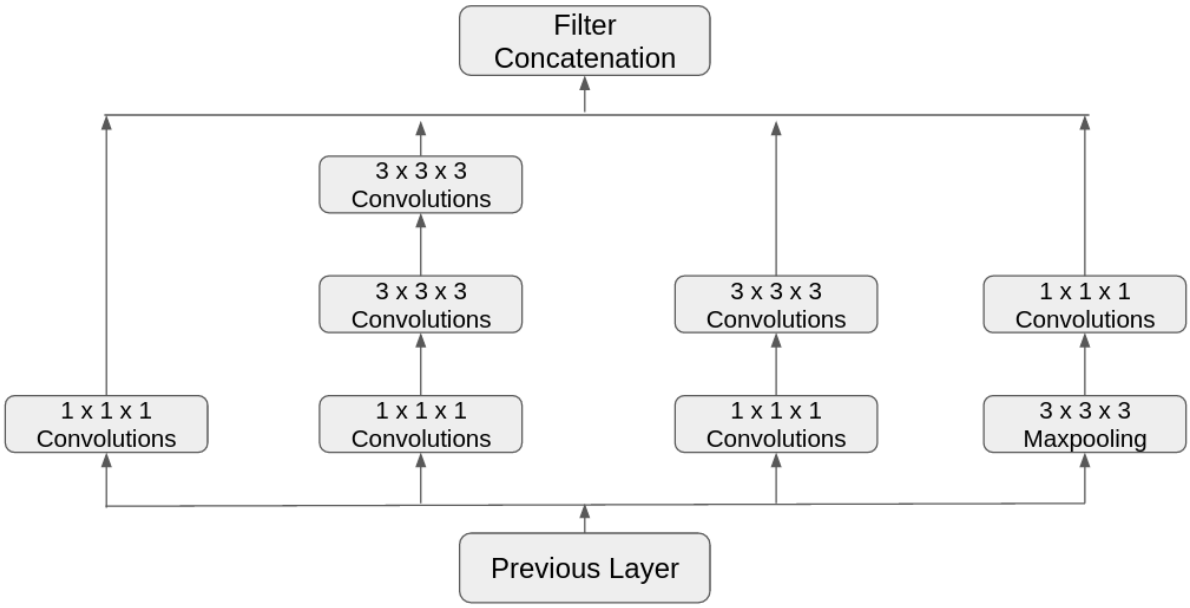


Figure 31: InceptionNet V3 block [36].

InceptionNet_v3 Architecture										
Type	Patch_size/Stride	Output_size	Depth	#1 x 1 x 1	#3 x 3 x 3 reduce_1	#3 x 3 x 3 reduce_2	#3 x 3 x 3	#3 x 3 x 3 reduce	#3 x 3 x 3	Pool Proj
Convolution	5 x 5 x 5/1	128 x 128 x 128 x 64	1							
Max pool	2 x 2 x 2/2	64 x 64 x 64 x 64								
Convolution	3 x 3 x 3/1	64 x 64 x 64 x 192	1							
Max pool	2 x 2 x 2/2	32 x 32 x 32 x 192								
Inception 3		32 x 32 x 32 x 256	3	32	64	64	128	64	64	32
max pool	2 x 2 x 2/2	16 x 16 x 16 x 256	0							
Inception 4		16 x 16 x 16 x 512	2	128	128	128	128	64	128	128
Max pool	2 x 2 x 2/2	8 x 8 x 8 x 512	0							
Inception 5		4 x 4 x 4 x 1024	2	128	256	256	384	256	384	128
Maxpool	2 x 2 x 2/2	2 x 2 x 2 x 1024	0							
Inception 6		2 x 2 x 2 x 1024	2	128	256	256	384	256	384	128
Average pool	2 x 2 x 2/1	1 x 1 x 1 x 1024	0							
Dropout	p = 0.4									
FC1		512								
FC2		64								
FC3		1								

Figure 32: Detailed description of InceptionNet V3 architecture used in the project.

Authors' contributions

Bok, Felix (FB):

- *FB was responsible for project management.*
- *Within the data generation part, FB provided ...*
  - *the idea, the implementation and the evaluation of the DefectorTopDownView.*
  - *the code structure.*
  - *key functionalities such as the BatchDataProcessor, Voxel- and MeshModel and the main script (together with JK).*
  - *the exploratory data analysis of the ABC dataset, in order to find suitable models and the selection threshold.*
  - *the download and the selection of the input 3D models from the ABC dataset.*
  - *the generation of the AMC dataset.*
- *Within the deep learning part, FB provided ...*
  - *the setup of the pipeline for training deep learning models, i.e. implementing the AMC dataset, the ClassifictationTask Class and the main script.*
  - *the in detail evaluation of the deep learning models and the AMC dataset, i.e. writing the Failure Analysis part.*
- *FB wrote the Abstract (together with NB) and the Motivation chapter.*

Bouziane, Nouhayla (NB):

- *NB was responsible for the presentation and report.*
- *Within the data preprocessing part, NB provided ...*
  - *the implementation of some approaches for Cleaning and Alignment.*
  - *the implementation and evaluation of the DefectorTopDownView Rotation Add-On.*
  - *the generation of the data using DefectorTopDownView Rotation Add-On.*
- *Within the deep learning part, NB provided ...*
  - *the training and evaluation of the InceptionNet V1 model on DefectorTopDown-View Rotation Add-On data.*
  - *the implementation of the hyper-parameters tuning part.*
- *NB wrote the Abstract (together with FB), Problem definition and Discussion chapters.*

Ebid, Ahmed (AE):

- *AE was responsible for the data.*
- *Within the data generation part, AE provided ...*
  - *the idea and implementation of data cleaning.*
  - *the idea and implementation of data normalization.*
  - *the idea and implementation of data alignment.*
  - *the idea, implementation and evaluation of the DefectorExhaustive.*
  - *functionalities for reading and visualizing meshes.*
  - *development using the Open3D library and interfacing between its data representation and ours.*
  - *the generation of the data using DefectorExhaustive.*
  - *the textual formulation of corresponding sections in the final report.*
- *Within the deep learning part, AE provided ...*
  - *the adapation of the deep learning container scripts to Horovod distributed training.*
  - *the adapation of MLflow logging to Horovod distributed training.*
  - *the training and evaluation of the InceptionNet V1 model on DefectorExhaustive data.*
- *AE wrote the Project Goals and the Related Work (together with AS) chapters.*

Srinivas, Aditya Sai (AS):

- *AS was responsible for the deep learning part within the project.*
- *Within the data preprocessing part, AS provied ...*
  - *implemention of the voxelizer to convert the mesh file into three kinds of voxel representations: Occupancy grid, SDF and TSDF.*
- *Within the deep learning part, AS provided/carried out ...*
  - *implementation of ResNet, InceptionNet V1 and InceptionNet V3.*
  - *implementation the deep learning pipeline excluding the failure analysis and the ROC curves.*
  - *various experiments on all the neural network architectures.*
  - *solution to the issue of the inability of Vanilla3DCNN and ResNet to train initially.*
- *AS wrote down the Related Work (together with AE), Deep Learning architectures and Implementation details of the final report.*

Kiechle, Johannes (JK):

- *JK was responsible for the Development Operations (DevOps) part within the project. This included the entire connection to the LRZ infrastructure for ...*
  - *the data generation pipeline*
  - *the deep learning pipeline*
- *Within the data preprocessing part, JK provided ...*
  - *an automated script in order to download our source data (ABC dataset).*
  - *the data selection method based on filesize and compactness.*
  - *the interface of python source code and the CUDA accelerated voxelization method C++ source code.*
  - *the textual formulation of corresponding sections in the final report.*
- *Within the deep learning part, JK provided / carried out ...*
  - *the implementation of the Vanilla3DCNN architecture.*
  - *the implementation of the deep learning source code pipeline including the failure analysis module.*
  - *dozen of training runs with different parameter settings for all four investigated architectures on the AMC dataset.*
  - *the implementation of distributed training using Horovod.*
  - *the implementation of MLflow in order to monitor the neural networks training life-cycle and result storage.*
  - *evaluation of the experiments / results and their corresponding interpretations and visualizations.*
  - *the textual formulation of corresponding sections in the final report.*

All authors gave final approval for publication and agree to be held accountable for the work performed therein.