



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. In this report we first present a data pipeline that identifies and selects the right input data and processes it such that it can be used for further processing, then we present an algorithm that adds different defined defects to 3D models and therefore generating a dataset for classification and last but not least we present deep learning models, based on advanced Convolutional Neural Network Architectures, that are able to correctly classify manufacturability with high accuracy and therefore build an inference algorithm to automate the previous described task.

Additive manufacturing technologies are widely used in the industrial world. They are used for rapid prototyping and visualization but also for fabrication of end-use products. However the process that identifies whether a 3D model is manufacturable is still done manually through CAD softwares and field application engineers.

Thus, the goal of our project is to automate this process using Convolutional Neural Networks. Our work focused on building solid Data Processing and Deep learning pipelines. We used Mesh models from the ABC dataset that we process through a series of transformations including cleaning, scaling, alignment and voxelization. An assumption of manufacturability of all these models was made. Therefore we include defects in our Data using synthetic data generation. 3 approaches of defects generation were developed. Afterwards we compared the performance of 4 convolutional neural network based architectures: Vanilla3DCNN, ResNet, InceptionNet V1 and InceptionNet V3.

Our results show that the InceptionNet architecture is the most suitable for our problem.(here I need to add more about the results)

## 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

<b>Abstract</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Definition . . . . .	1
1.3 Project Goals . . . . .	1
1.4 Project & Report Outline . . . . .	2
<b>2 Related Work</b>	<b>3</b>
<b>3 Data Source &amp; Data Preprocessing</b>	<b>4</b>
3.1 Data Source . . . . .	4
3.2 Data Selection . . . . .	4
3.3 Data Preprocessing . . . . .	5
3.3.1 Cleaning . . . . .	5
3.3.2 Normalization . . . . .	6
3.3.3 Alignment . . . . .	6
3.3.4 Voxelization . . . . .	8
<b>4 Synthetic Data Generation</b>	<b>10</b>
<b>5 Machine Learning</b>	<b>15</b>
5.1 Architectures (Adi) . . . . .	15
5.2 Implementation (Adi) . . . . .	18
5.3 Infrastructure & computational Resources . . . . .	18
<b>6 Experiments &amp; Results</b>	<b>21</b>
<b>7 Discussion</b>	<b>26</b>
<b>References</b>	<b>28</b>
<b>Appendix</b>	<b>33</b>

# 1 Introduction

## 1.1 Motivation

Additive Manufacturing (AM), or commonly referred to as 3D printing, in 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 [20]. There are a plenty of different methods for AM, as described for example in [22]. 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 [36], [11], [25]. It is therefore no coincidence that the AM market has an average growth rate of 27% over the last decade and is currently estimated at \$12.8 billion [39]. Some industry experts even estimate that the market will reach \$100-250 billion by 2025 [11]. Additionally, companies in the automotive [2], [10], [3], in aerospace [9], [6], [12] and many more are investing heavily in AM technology and its application to high volume manufacturing.

## 1.2 Problem Definition

AM has as of right now still a lot of limitations, especially a lack of design knowledge, imperfections during the printing phase and high cost in mass production. One significant reason and source of these limitations is that AM currently requires a lot of human expertise and oversight [25], [11].

In this project we want to automate one part of the manual steps needed for AM, namely identifying whether a given 3D model will be able to be manufacturable by a given 3D printer. Since a 3D model could have many geometric elements and the requirements for the 3D object can be quite complex and detailed, experts need to spend a lot of valuable time doing this task.

## 1.3 Project Goals

This project aims to achieve two main 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 3D mesh model given as input can be an *.obj* or *.stl* file. 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, given a 3D voxelized model, returns whether the voxelized 3D model is printable or not. The deep learning module offers a choice of several supported architectures, performs the classification, and performs performance analysis on the classified models.

## 1.4 Project & Report Outline

The remainder of the report is organized as follows: First of all, similar literature work is presented in section 2. This is followed by section 3 which gives an insight into the data source which has been exploit 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 method mainly focuses on the part how to add defects into existing 3D models. Section 5 showcases an in depth investigation of the generated data whether those can be used in order to differentiate between printable and non-printable 3D models using different neural network architectures. This is followed by section 6 which presents the final results. The report is concluded by a discussion which is presented in section 7.

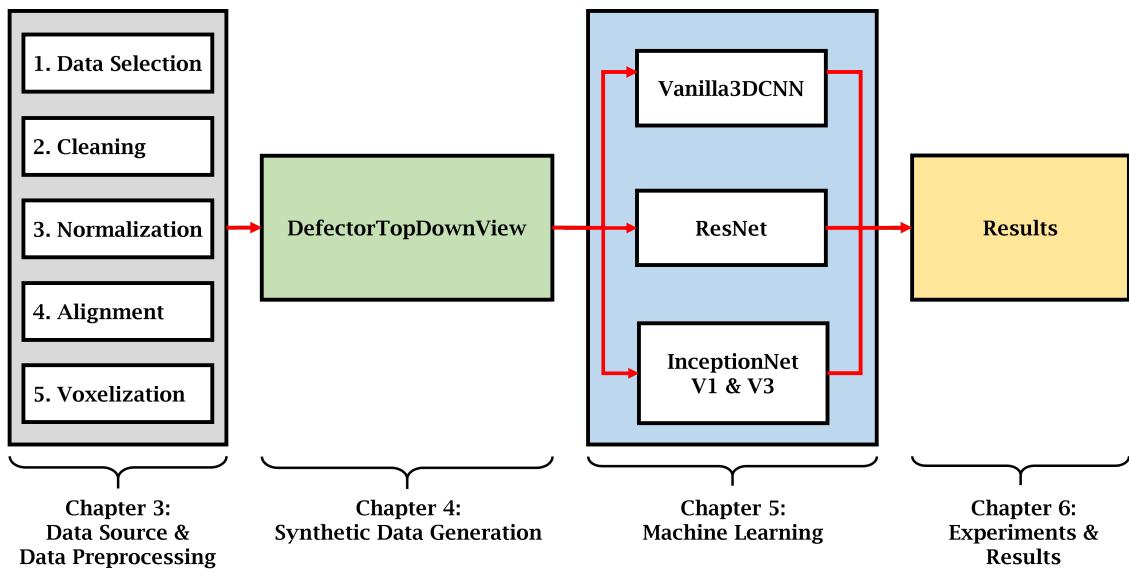


Figure 1: Overview Project and Report Structure.

## 2 Related Work

### **Learning Localized Geometric Features Using 3D-CNN [7]:**

In the paper, the feasibility of using 3D-CNNs to identify local features of interest using voxel-based approach has been demonstrated. The 3D-CNN 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 learnt the complex 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 currently available in 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.

### **Deep CNNs for automated feature extraction in Industrial Inspection [14]:**

In this work an approach in visual defect detection using deep machine learning, namely deep CNN, is presented. The performance of the proposed approach is measured on a data set representing 12 different classification categories with visual defects occurring on heavily textured background. As opposed to hand-crafting features on pixel level, with CNN architectures are engineered by investigating different hyper-parameters involved in the process. In this way, systems for Optical Quality Control (OQC) can be developed with minimum prior knowledge on the problem domain.

### **Toward Enabling a Reliable Quality Monitoring System for Additive Manufacturing Process using Deep Convolutional Neural Networks [8]:**

In this paper, an automated quality grading system that uses a deep CNN model for the additive manufacturing process was proposed. The CNN model was trained online using images of the internal and surface defects in the layer-by-layer deposition of materials and tested online by studying the performance of detecting and classifying the failure in 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 which has been exploited and all consequent applied preprocessing steps whilst also give a reasoning for all individual steps. The holistic workflow is depicted in the block diagram below.

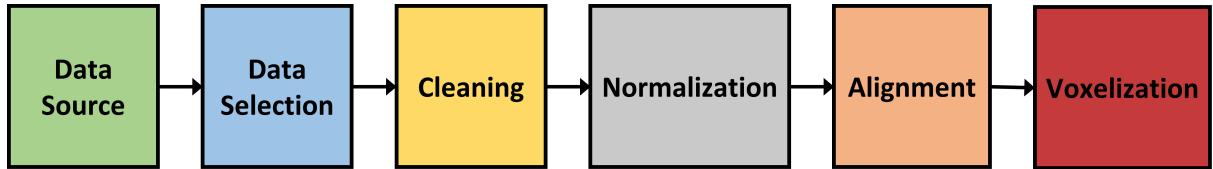


Figure 2: Holistic representation of the data & data preprocessing workflow.

#### 3.1 Data Source

As a baseline dataset we used the so called ABC Dataset, which is a collection of Computer-Aided Design (CAD) models for research of geometric deep learning methods and applications [24]. 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. 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 dissimilar complexity distribution of the 3D models across the dataset and even within each data chunk, we are aiming for tackle this 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. Out of exploratory data analysis it can be concluded that the more complex the model gets, the more triangular meshes are required in order to represent the exact shape of the 3D model. Consequently, this also 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, which was identified empirically.

Apparently, the total number of files which are initially considered depends first of all on the number of 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 determines which files will be regarded for further preprocessing steps and which are disregarded. In our current approach, three data chunks are taken and all models having a file size smaller or equal to 25 kilo bytes (kb) are selected, resulting in a total of 3457 initial models, which are considered for further preprocessing steps and in addition to be printable.

### 3.3 Data Preprocessing

Once a decision has been made with respect to the 3D models which are considered for any further preprocessing steps, all models have to undergo a 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 normals of the 3D mesh are facing outwards. Manifold geometry is satisfied if no edges of the 3D mesh are shared by more than two faces [33]. 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, i.e., (triangle1: [v1, v2, v3], triangle2: [v2, v1, v3])
2. Removing triangles that reference a single vertex multiple times in a single triangle, i.e., (triangle1: [v1, v2, v2])

All these cleaning functionalities were implemented using Open3D [1] functions.

In addition to that some 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_faces`: 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 to ensure having stable gradients when training neural networks. Since different models come at 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 same factor to not destroy the appearance/scale factor of the model. Traditional normalization where each feature is scaled according to the range that the feature lies in independent of the other features won't work with the components of 3D vertices. To that end, we apply normalization in 2 steps. First, we center the model on the origin by finding the center of mass of the model and translating it to the origin.

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

where  $(x_t, y_t, z_t)$  are the components of a 3D point after translation to the origin,  $(x, y, z)$  is the 3D 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.

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

Then, we scale the points such that all components (X, Y, Z) of the 3D vertices lie in  $[-1, 1]$  range. We do this by finding the range of each component and then scaling the points by the component range with the maximum value.

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

### 3.3.3 Alignment

3D objects are generally represented in an arbitrary orientation [26]. The aim of this part of pre-processing 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. The axis of the minimum moment of inertia is the axis around which most of the mass of the model is wrapped. Resistance to rotation around that axis is minimal. The alignment proposed is to find the axis with the minimum moment of interia and align it with one of the coordinate axes (X, Y, Z). To find the axis of minimum moment of inertia, we apply Principal Component Analysis on the inertia tensor. The eigen vector with the minimum eigen value is the axis of minimum moment of inertia for a given model [37]. The inertia tensor is formulated as follows:

Listed below are the elements of the inertia tensor.  $(x, y, z)$  are the position vectors.

$$I_{xx} = \sum_{n=1}^N y^2 + z^2 \quad (4)$$

$$I_{zz} = \sum_{n=1}^N y^2 + x^2 \quad (6)$$

$$I_{xz} = \sum_{n=1}^N xz \quad (8)$$

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

$$I_{xy} = \sum_{n=1}^N xy \quad (7)$$

$$I_{yz} = \sum_{n=1}^N yz \quad (9)$$

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 [4].

Finding a rotation matrix  $R$  that rotates a unit vector  $a$  onto another unit vector  $b$  is done as follows:

Let  $v = a \times b$  (cross product)

Let  $s = \|v\|$  (Frobenius norm of a vector)

Let  $c = a \cdot b$  (dot product a)

Then the rotation matrix  $R$  is given by eq. (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  $v$ ,

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

After finding the rotation matrix  $R$ , 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 5 shows the results of the alignment algorithm using the equations mentioned above.

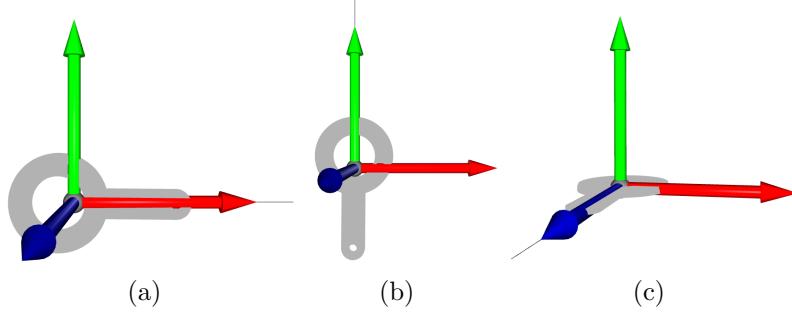


Figure 3: (a) Alignment of min MOI axis and X-axis (b) Alignment of min MOI axis and Y-axis (c) Alignment of min MOI axis and Z-axis

Additionally, we also tested the results of a PyMeshLab alignment algorithm that follows the same concept. Figure 4 shows the results of the alignment algorithm using PyMeshLab.

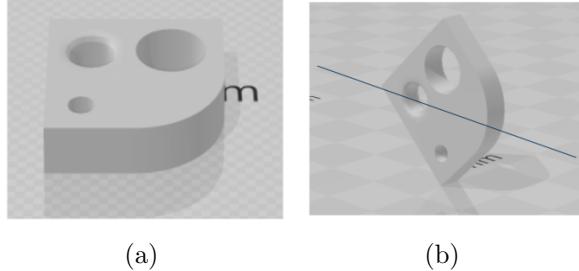


Figure 4: (a) 3D model before alignment, (b) 3D model with PCA aligned with x axis

### 3.3.4 Voxelization

All previously handled preprocessing steps have been applied in the mesh status level of the 3D models. However, last but not least, the final preprocessing step which is missing is devoted to the transformation of those 3D mesh models into a voxel representation. Generally spoken, a voxel can be regarded as a pixel in a three-dimensional space. The reason why a voxelized representation of the 3D models has been chosen is due to several aspects. First, as a voxel can simply considered as an upscaling of a 2D pixel into 3D, the assumption was made that also state-of-the-art 2D vision models will show a rather good performance on the data by transforming them into neural network architectures which act on 3D data (i.e. 3D convolutions). Second, the voxel representation also allows an easy introduction of defects into existing models, as !!!GIVE REASONING HERE!!!.

Generally, there are several techniques how to represent 3D models using voxels !!REFERENCE!. In this work, the so-called occupancy grid has been used [15]. This method is based on the 1990 paper published in Proceedings of the sixth conference on Uncertainty

in Artificial Intelligence (UAI). Occupancy grid is the most straight forward method used to represent voxels. Occupancy grids represent a binary field with a value of 1, if the voxels are intersecting with the surface and 0 everywhere else. However, due to cubically growing compute and memory requirements, only low resolution occupancy grids (max  $128^3$ ) can be handled. Given SDF value representation and binary threshold, all the SDF values beyond the binary threshold are set to 0 and the SDF values within the binary threshold are set to a value of 1. The cycle representing transformation of the given mesh file into a  $64^3$  occupancy grid is shown in Figure Figure 5.

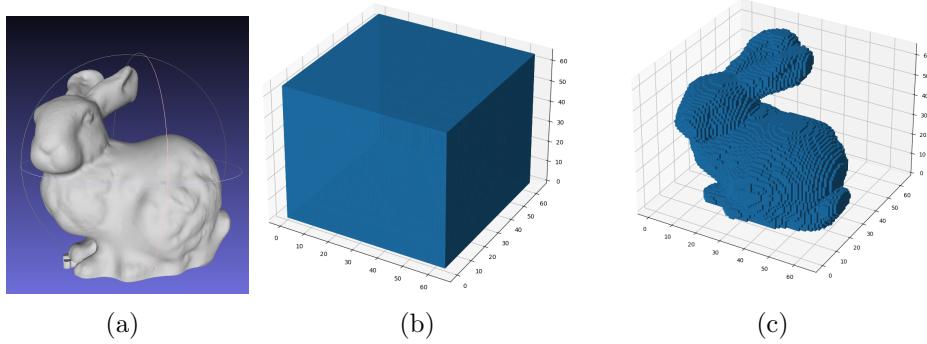


Figure 5: (a) Stanford bunny .stl file (b)Signed Distance Function (SDF) of the stanford bunny (c)Occupancy grid representation of the bunny rabbit

Finally, as there were initially a total of 3457 models selected, the process of transforming all models into a corresponding voxel representation is very compute intensive. Predominately, the computational demand depends on the final 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 an reasonable amount of time [21]. The GPU voxelization algorithm is build on the "Möller–Trumbore" intersection algorithm [27], which describes the ray-triangle intersection method used, and CUDA [13].

The voxelization step concludes the data preprocessing cycle. After all steps, an initial 3D mesh model (i.e. stl file) now got transformed into a  $128 \times 128 \times 128$  voxel representation model which 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 focus on the task of adding defects to all existing models and label them accordingly.

## 4 Synthetic Data Generation



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

After selecting the right models out of the ABC dataset and preprocesing 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. We called the resulting dataset AMC Dataset (compare to Figure Figure 6). In the following, we will describe how printable and non-printable models are defined and how we added defects with different complexity to the 3D models in a controlled way.

Defects are defined here as holes that has been added to the 3D model and can have an influence on the manufacturability of the resulting model. We assumed that the selected models from the ABC dataset are printable and that the defining parameters that decide 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, which we will call just border for now. We therefore get 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 we 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 we can generate the following defects:

- **model with a non printable defect in the middle ( $m_{np}^m$ )**: radius of  $r_{np}$  and border larger or equal than  $b_{np}$  (Fig. Figure 7 (d))
- **model with a printable defect in the middle ( $m_p^m$ )**: radius of  $r_p$  and border larger or equal than  $b_p$  (Fig. Figure 7 (f))
- **model with a non printable defect at the border ( $m_{np}^b$ )**: radius of  $r_p$  and border of smaller or equal than  $b_{np}$  (Fig. Figure 7 (h)).

Given the hyperparameters and the definition of the defect types the problem of adding defects reduces to find a offset such that the requirements for each defect type is fulfilled. For this we developed and evaluated two algorithms. The first algorithm we developed, called DefectorExhaustive, uses an exhaustive approach with various checks and with a variable radius and the second algorithm, called DefectorTopDownView, projects the 3D data of the model onto a 2D grid and uses the resulting information's to select a suitable offset. Since the second algorithm enabled us to add more complex defects in a more robust way, we decided to use this method for the generation of the AMC Dataset (for more details about the first algorithm see section 7).

The DefectorTopDownView algorithm is described in Algorithm 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 such 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, that consists of a x and y label that together generate a grid. Each tile is then colored based on the value in the given matrix [38]. Therefore, 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 x and y axis keep their meaning and 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 is given in Figure Figure 7 (b)). With the help of the *tdv* a subsample of indices can be generated that defines suitable offsets for a (radius, border) combination, as seen in Figure Figure 7 (c), (e) and (g) and described in step 5 and 10 in Algorithm 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 algorithm 1. This additional check is needed to check the given criterion's in the non-axis aligned directions. The output of the DefectorTopDownView algorithm are then either zero, two (input model without an defect and  $m_{np}^m$ ) or four 3D models (previous two models,  $m_p^m$  and  $m_{np}^b$ ) depending on whether it could find a offset satisfying the given conditions.

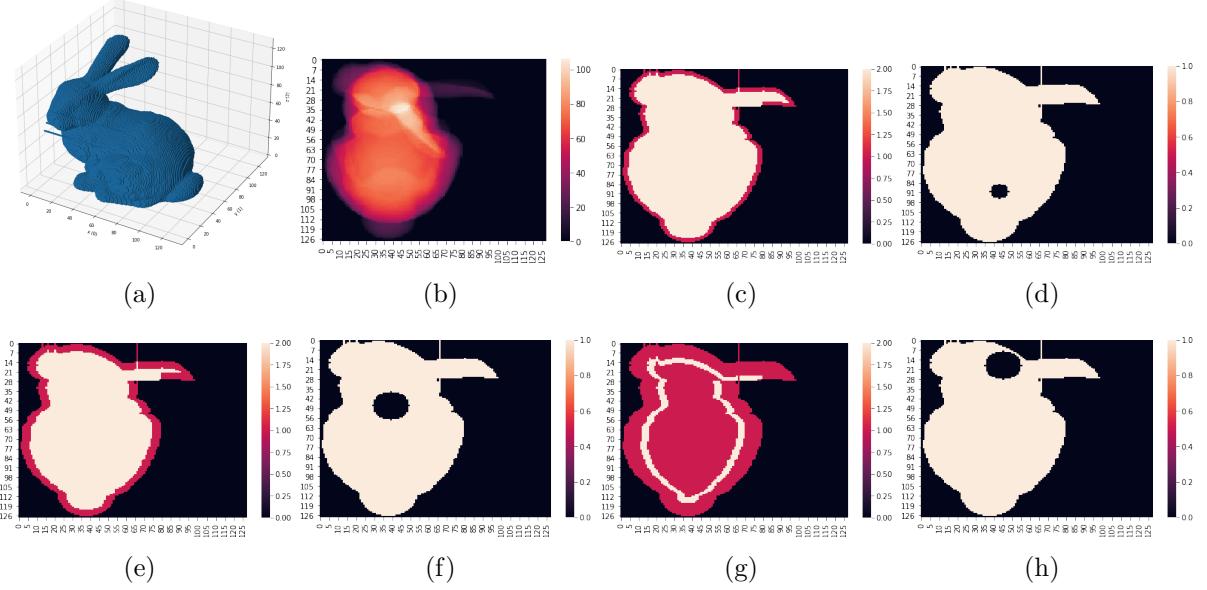


Figure 7: (a) Stanford bunny represented by an occupancy grid. (b)  $tdv$  of the Stanford bunny represented as a heatmap. Scale on the right indicates the number of voxels at each  $(x, y)$  point (c), (e) and (g)  $tdv$  of the Stanford bunny. The red area visualizes the indices that are removed by step 5 for (c) and (e) and step 15 for (g) of Algorithm algorithm 1 by using a border size of 5 for (c) and (g) and a border size of 10 for (e). The light area indicates possible offsets. (d), (f), (h)  $tdv$  of the Stanford bunny with an added hole of radius of 5, 10 and 10 and consequently the  $tdv$  of  $m_{np}^m$ ,  $m_p^m$  and  $m_{np}^b$ , respectively. For finding the offset the light area of indices out of Figure Figure 7 (c), (e) or (g) has been used.

The advantage of this method is that the models are transformed in such a way that the resulting informations can be used to precisely add the defects needed. Additionally, the algorithm can be extended such that defects with all kinds of shapes and properties can be added and it is implemented such that the resulting dataset has balanced labels.

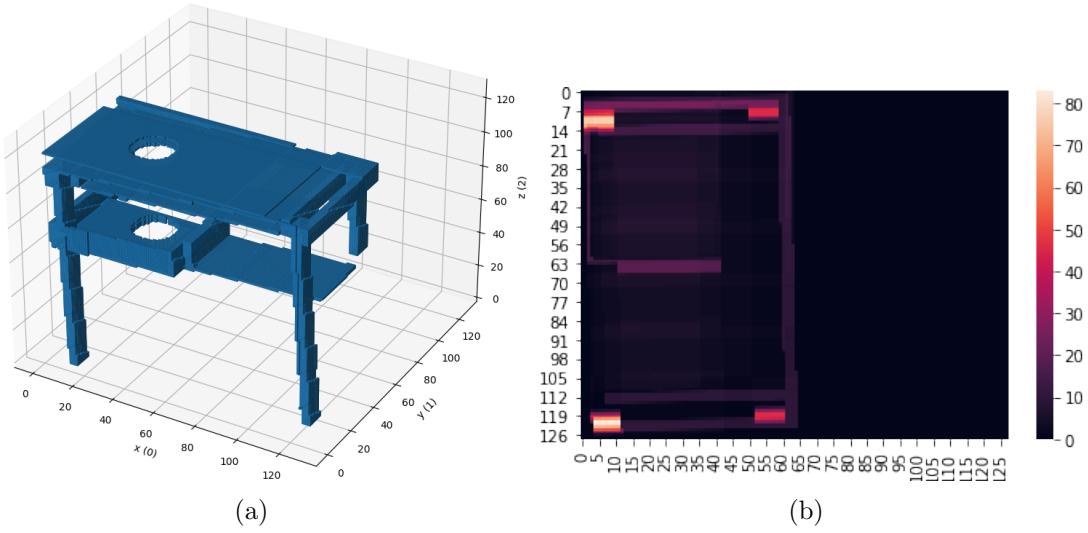


Figure 8: (a) Voxel model labeled by Algorithm algorithm 1 as manufacturable, however on the second layer the hole is too close to the boarder making this model non-manufacturable (b)  $tdv$  of the model visualized in (a)

However, a identified drawback of Algorithm 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 been observed and is visualized in Figure Figure 8 (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 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 8 (a) is only around ten voxels (Fig. Figure 8 (b)). Therefore, for the generation of the AMC dataset, these checks were not included, tolerating the few models that are potentially wrongly labeled.

Applying the Algorithm 1 we were able to extend the given dataset by a factor of 2.15, generating 7430 3D models. A few additional resulting models are visualized in Figure Figure 23 in the Appendix.

```

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

begin
     $tdv \leftarrow$  sum  $m$  over z axis
    possible_offsets  $\leftarrow$  determine non-zero indices of  $tdv$ 
    for  $r, b$  in  $[(r_{np}, b_{np}), (r_p, b_p)]$  do
        remove all points of  $tdv$  that are  $b$  away from the border of  $tdv$  in x and y
        direction
        offset  $\leftarrow$  randomly select offset from previously determined points of  $tdv$ 
        for  $0$  up to  $t$  do
            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
                new_model  $\leftarrow m$  with hole aligned with z-axis at offset with radius
                 $r$ 
                add new_model to pre_output
                break
            end
        end
    end
    possible_offsets  $\leftarrow$  determine points of  $tdv$  that are  $b_{np} + r_p + 1$  away from the
    border of  $tdv$  in x and y direction
    repeat code block 7-13 with radius= $r_p$  and border=1
    if model_with_non_printable_defect_middle in pre_output then
        | add  $m$  and model_with_non_printable_defect_middle to output
    end
    if model_with_non_printable_defect_border and
        model_with_printable_defect_middle in pre_output then
        | add model_with_non_printable_defect_border and
        | model_with_printable_defect_middle to output
    end
    if output is not empty then
        | return output
    else
        | return empty list
    end
end

```

**Algorithm 1:** DefectorTopDownView

## 5 Machine Learning

Machine Learning is the science (and art) of programming computers so they can learn from data [16]. Generally, there are a lot of aspirations which are giving a definition about 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 (Adi)

3D convolution neural networks have been used for object recognition based on voxelized shape of the object. In this project, a 3D-CNN based method is proposed to learn distinct local geometric features of interest within an object. The task at hand is a binary classification task to recognise if a part is printable or not. Since, it's a simple binary classification task, it is essential to obtain appropriate feature extractor architecture which are then followed by dense layers. The general structure of the neural network used in the project is shown in Figure 9.

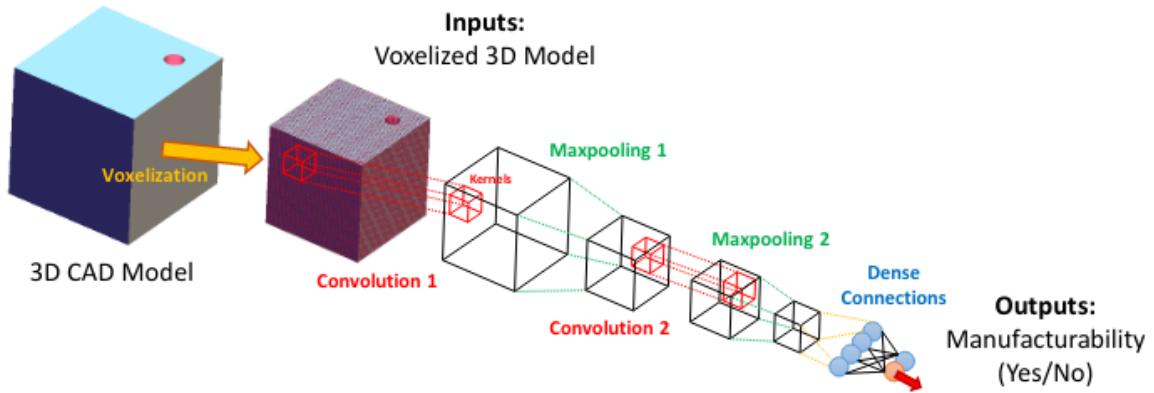


Figure 9: General structure of the architecture used in the project[7]

The following architectures have been used as feature extraction networks:

**Vanilla3DCNN:** Vanilla3DCNN is the simplest Neural Network used. It uses kernel sizes of 9, 7, 5 and 3 with batchnorm layers and maxpool layers in between to reduce the spatial size of the input data.

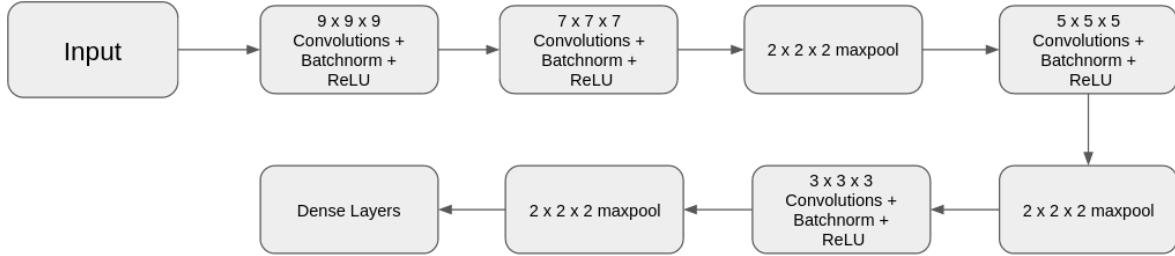


Figure 10: Vanilla3DCNN Architecture.

**ResNet [19]:** 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, the authors of the paper empirically show that there is a maximum threshold for depth with the traditional CNN model. The author conclusively showed that adding more layers leads to increase in the training error which is ideally not expected. This increase in the training error is due to vanishing and exploding gradients. The problem of training deep neural networks has been alleviated with the introduction of ResNet architecture. For the project, a customized ResNet architecture has been built with the basic residual block as shown in Figure 11. The detailed description of the architecture can be found in Figure 28. From the description, it can be found that the number of times residual blocks are repeated are changed. 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 have been added before converting the CNN layers into feature vectors followed by  $1 \times 1 \times 1$  convolutions to increase the number of channels without exponentially increasing the number of operations.

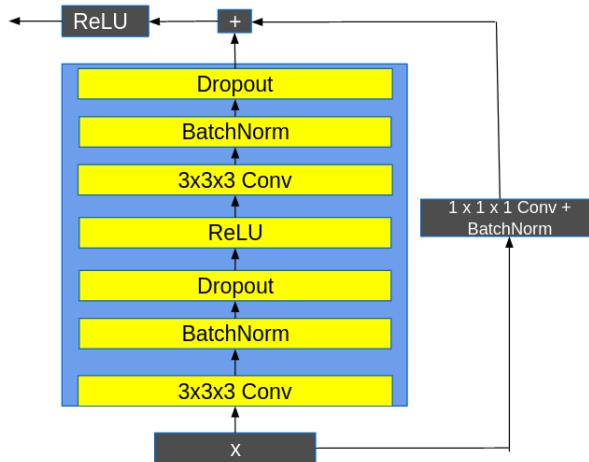


Figure 11: Basic Residual Block

The Skip Connections between layers add the outputs from previous layers to the outputs of stacked layers.

**InceptionNet [35]:** The Inception network 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 have been built. The detailed description of InceptionNet\_V1 and InceptionNet\_V3 can be found in Figure 29 and Figure 30 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.

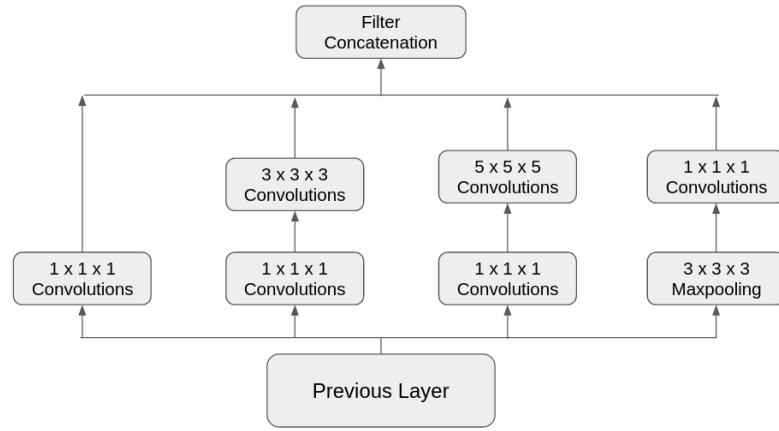


Figure 12: InceptionNet\_V1 block

InceptionNet\_V3 is an improvement over InceptionNet\_V1 architecture. The authors realized that the input large reduction in input dimensions due to the filter size of  $5 \times 5$  leading to decrease in the accuracy of prediction hence making the neural network prone to loss of information. The authors hence proposed to replace the  $5 \times 5$  kernel with  $3 \times 3$  kernel and increasing the depth of each inception block. The inception block used in InceptionNet\_V3 along with the detailed architecture is shown below in Figure 13.

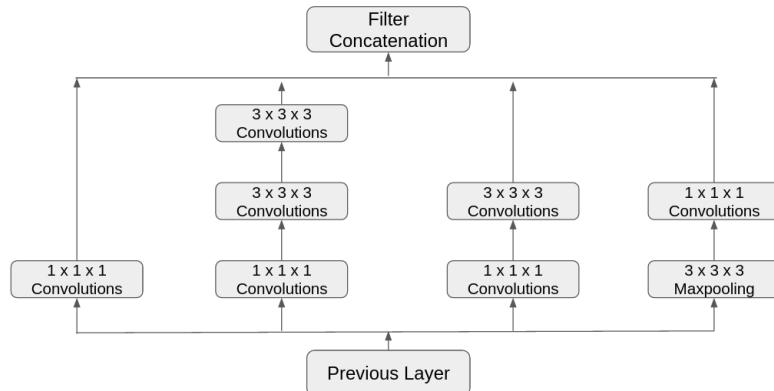


Figure 13: InceptionNet\_V3 block

## 5.2 Implementation (Adi)

**Activation Functions:** Since the task is a binary classification task, Sigmoid[18] activation function is used in the final dense layer while the layers in between use ReLU activation functions. Use of ReLU[18] activation function in all the other layers prevents saturation of parameters, hence preventing vanishing gradients allowing the model to learn faster.

**Loss Function:** Binary cross entropy (BCE)[28] loss also called Sigmoid Cross-Entropy loss is a Sigmoid activation plus a Cross-Entropy loss. Unlike Softmax loss[28], 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's 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 formulation of Cross Entropy Loss for binary problems is often used:

$$BCE = - \sum_{i=1}^{C=2} t_i \log(f(s_i)) = -t_1 \log(f(s_1)) - (1 - t_1) \log(1 - f(s_1)) \quad (14)$$

where,  $t_1$  and  $s_1$  are the predicted label and ground truth label of class C<sub>1</sub> respectively and  $t_2 = 1 - t_1$  and  $s_2 = 1 - s_1$  are the predicted label and ground truth label of class C<sub>2</sub> respectively.

**Optimizer:** Gradient descent[28] 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. The learning rate  $\eta$  determines the size of the steps we take to reach a (local) minimum. In the project, Adam optimizer[23] was used to optimize the parameters with a learning rate of 0.00001.

**Dataset:** Regarding the dataset it has to be mentioned, that all of the 7430 datasamples 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 entire process of training and validating the neural network architectures took place within 100 epochs for each of the experiments.

## 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 the rather easy accessibility of 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 continuous increasing computational requirements. Thus, this subsection aims for giving an insight into the infrastructure which has been exploit, how learning across multiple graphic processing units (GPUs) is distributed, using horovod [32] and how the neural network learning life-cycle of several architectures is monitored using MLflow [40].

**Infrastructure:** Conducting deep learning experiments on different neural network architectures whilst also tuning hyperparameters and making use of massive datasets 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 curse of dimensionality ruthlessly comes to light and thus make the availability of GPUs indispensable [18]. Thanks to Leibniz Rechenzentrum (LRZ), we have been equipped with powerful resources all along the way and were even 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 especially powerful as it offers 640 tensor cores per device in order to accelerate the training process. The exact course of action how the parallelization is done is outlined in the paragraph called "distributed deep learning using horovod".

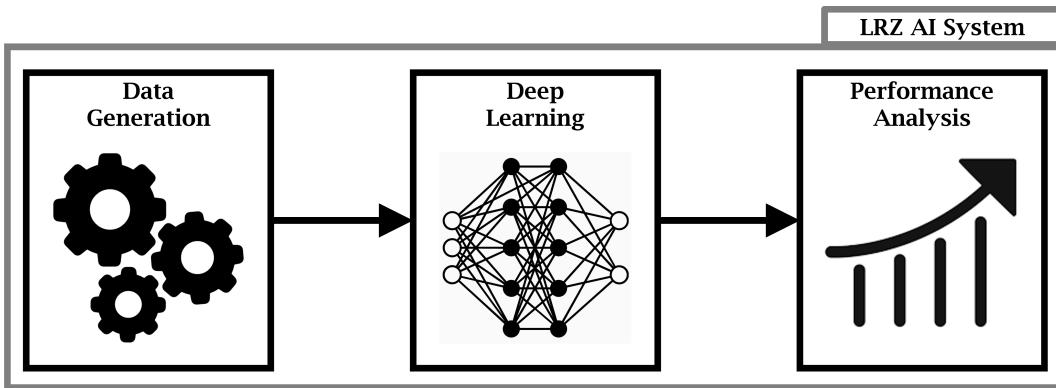


Figure 14: Big picture representing the project workflow using the LRZ infrastructure.

Apart of computations with respect to neural networks (i.e. gradient calculation using backpropagation or weights updating using various optimization techniques) GPUs can be also leveraged in order to perform general purpose computation tasks. As a consequence, the provided resources are 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 [29], [13].

**Distributed deep learning using horovod:** The data parallelism strategy which is exploit in this project relies on the distributed deep learning framework called horovod,

which initially was invented by researchers working at UBER [32]. 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. The gradients across those multiple devices are averaged using Baidu's algorithm [17], which in turn is based on a paper that describes bandwidth optimal all-reduce algorithms for clusters of workstations [31]. 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 below.

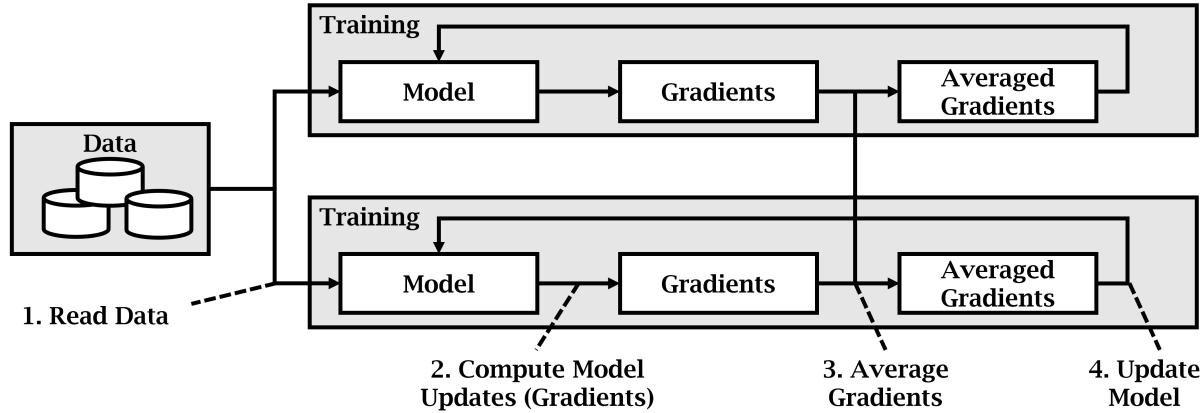


Figure 15: Schematic workflow example of distributed training using two GPUs. Source: [32]

**Deep learning life-cycle monitoring using MLflow:** 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 hyperparameter tuning, MLflow is used to monitor the life-cycle of all our attempts. Basically, MLflow is a open source platform to streamline machine learning development, including tracking experiments, packaging code into reproducible runs, and sharing and deploying models [40].

## 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. Nevertheless, even though, the respective data pattern can not be learned exactly, machine learning models can leverage a dozen of tasks if they are to approximate the data structure as close as possible. As a consequence, this section aims for giving an insight into the experiments which have been 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 are 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 exploited 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, which 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 [16].

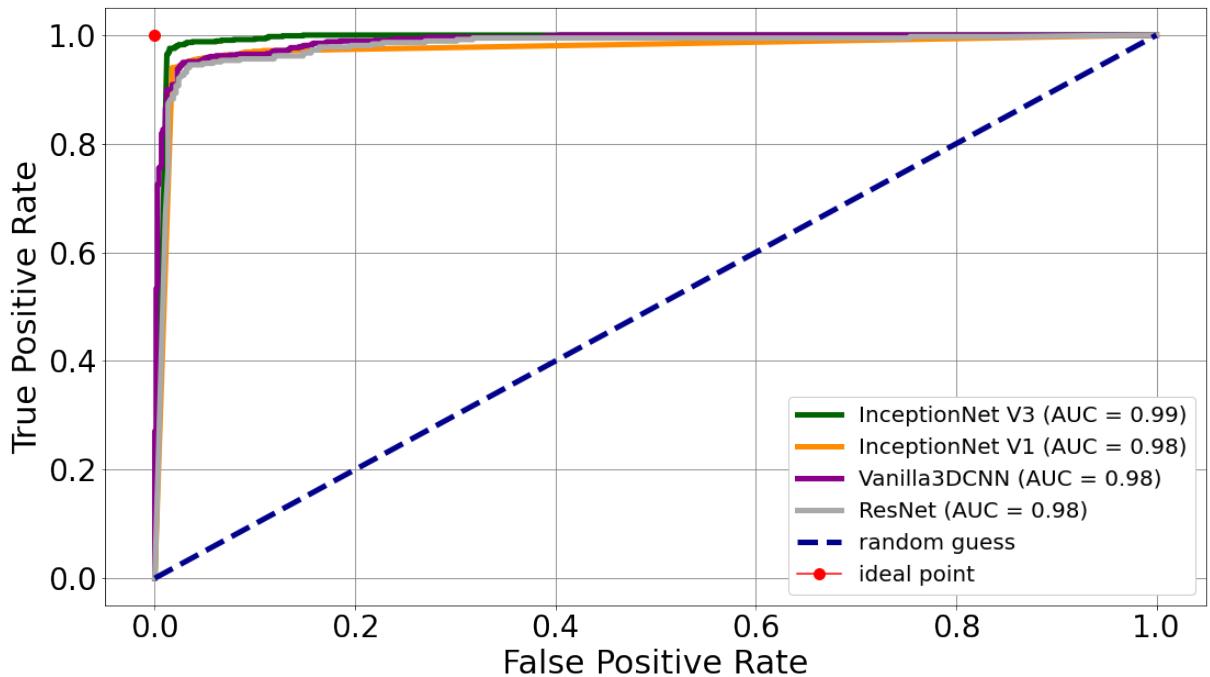


Figure 16: ROC curve with AUC, indicating the performance of different architectures.

From fig. 16 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 comprised of the same architecture basis (i.e. inception block) and thus mainly differ in their parameter setting (e.g. filter size), it is 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 is used, which represents the harmonic mean of precision and recall [16]. 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.

$$F1 = \frac{2}{recall^{-1} + precision^{-1}} \quad recall = \frac{tp}{tp + fn} \quad precision = \frac{tp}{tp + fp} \quad (15)$$

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, which 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 are 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 accuracy metric, another common approach in order to evaluate the performance is to leverage confusion matrices [16]. 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, thus indicating many correct predictions.

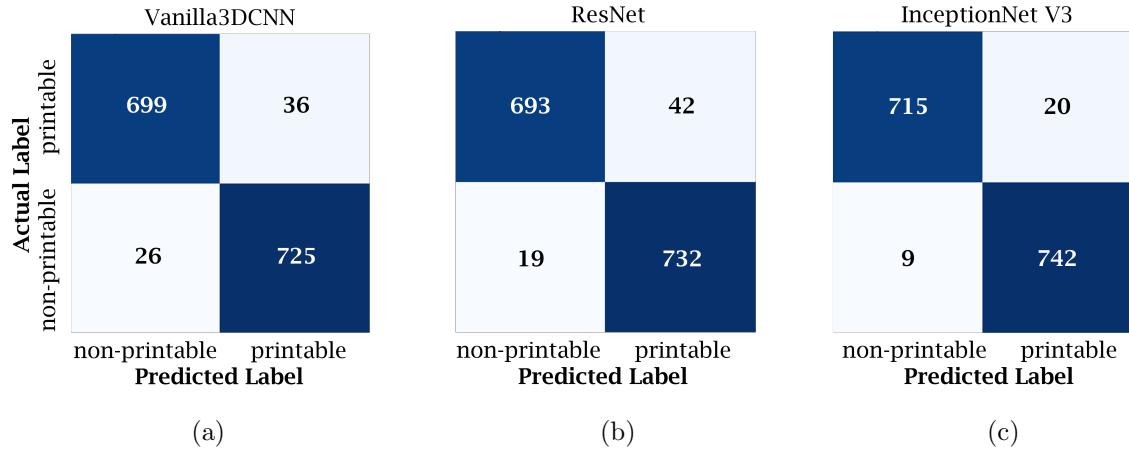


Figure 17: Confusion Matrix: (a) Vanilla3DCNN (b) ResNet (c) InceptionNet V3

The fig. 17 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 off-diagonal elements are belonging to a rare minority. Consequently, all classifiers are predicting most of the samples correct which results in a rather high accuracy score and F1 value as aforementioned. The few off-diagonal elements on which the architectures fail 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.

Last but not least, one of the most apparent and common ways in order to get an insight into the training behaviour of any model, and thus also to get an impression about its performance, is to look at the development of the accuracy for both, training and validation data over a plenty of epochs. An according plot for all of the three in-depth investigated architectures is provided in section 7. Within the plot, the x-axis represents the training time in epochs, while the improvement is indicated as accuracy on the left and as loss indicated on the right y-axis. Overall, it can be summarized that the result of all architectures can handle the binary classification problem with a very high final accuracy and low loss value. Even the development of the curves show that there is a continuous improvement without any sudden nor 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.

**Failure Analysis:** In order to obtain a more accurate assessment of the quality of the AMC dataset and the performance of the deep learning architectures, the models, out of the validation dataset, incorrectly predicted by InceptionNet\_V3 were split according to the different defect types and visualized. As table 1 shows, the most incorrectly classified models have a non-printable defect in the middle of the model.

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.

Examples of these models are visualized in fig. 18. Since this missclassifications 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 fig. 18). It is also interesting to note that nine models that had no defect were incorrectly classified as non-printable (compare to table 1). 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 fig. 18). Therefore, the assumption that all selected models are printable does not hold and as a consequence the label for these 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 fig. 18). 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 fig. 18). 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 fig. 18).

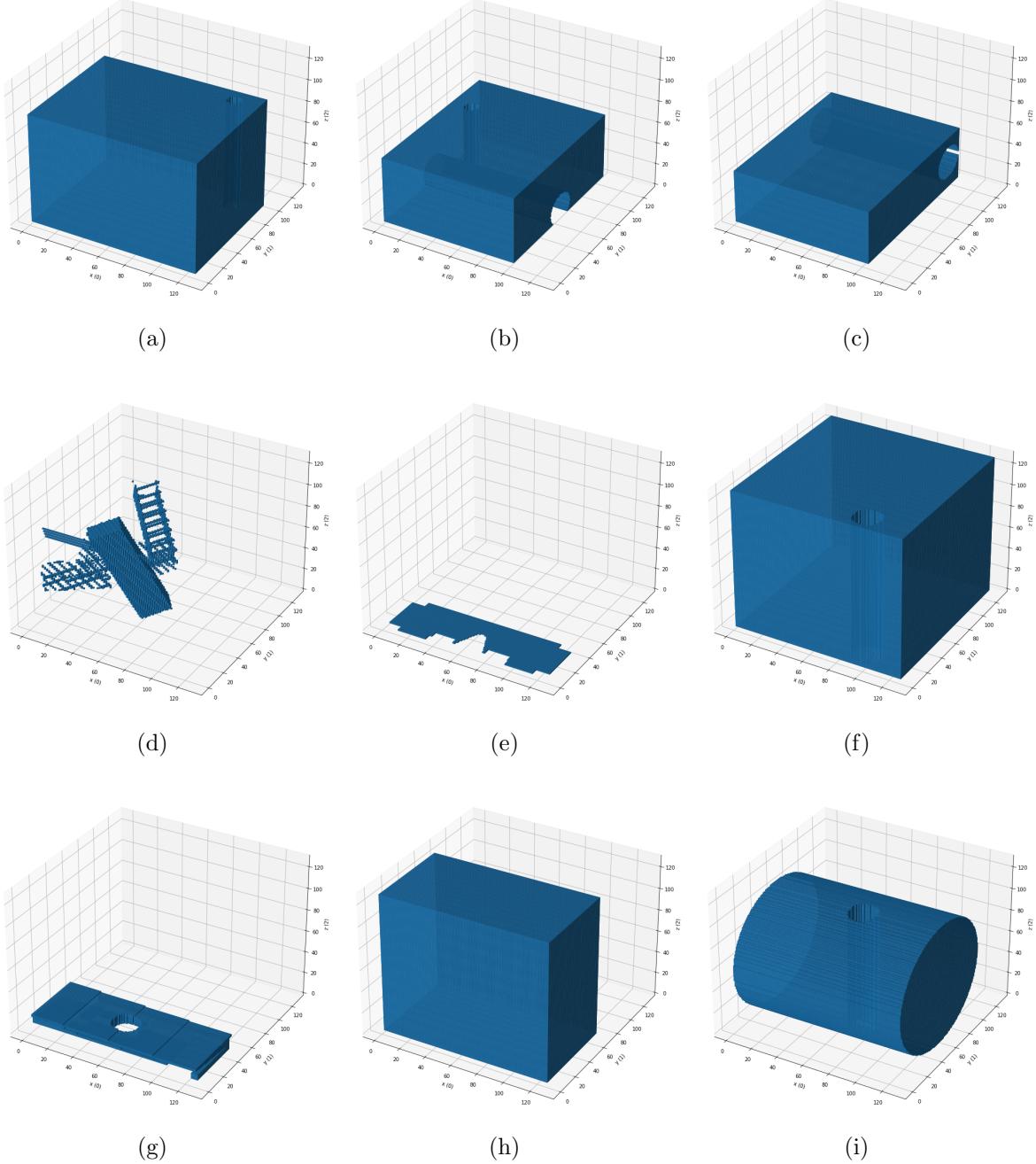


Figure 18: Visualizations of missclassified models out of the validation set by the trained InceptionNet\_V3 model. (a) Example of a model having a non-printable defect in the middle. (b) Example of a model having a non-printable defect in the middle and additionally a printable hole that was already 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 fails to classify it correctly.

## 7 Discussion

Starting from Mesh Models in the format of .stl files, we designed a data processing pipeline consisting of a series of transformations: normalization, alignment, cleaning and voxelization. We put an assumption that these models are 3D printable. To construct the non printable models we designed 3 defector algorithms. The goal of these approaches is to insert holes in the printable models and label them as printable or non printable by following some rules regarding the radius and the border. The final approach that we used for the synthetic data generation was the Defector-top-down-view. Actually, it allows us to have more flexibility regarding the insertion of the holes and builds more sophisticated defects. Thus, the resulting Dataset consists of the processed original models, the models with defects having a printable radius, the models with defect having a non printable radius and the models with defects having a non printable border. We can clearly see that our final Dataset is balanced and has approximately the same number of examples per class.

The Deep Learning pipeline consisted of implementing different architectures. The first model that was tested is the Vanilla CNN and was already giving good results: accuracy of [...]. The second model that we have tried was the ResNet (In this part we still need to discuss how to tell the story). However, even better results are achieved when using InceptionNetv1 and InceptionNetv3.

The evaluation of the Models was done by only using the validation set. It would be highly interesting to use a test set from another chunk of the ABC Dataset or from the Thingi10K dataset. This will give a precise idea about how our model is behaving ...  
We also recommend future work focused on the development and testing of a more sophisticated defector algorithm. For example using the third approach that we developed: Defector-top-down-view with randomly rotated holes. Try different settings of the defector, border and radius(to discuss because the significance is not big).

(Add more propositions of future work)

"Layout: Two paragraphs, one is considering the conclusion of all work done, the other one is referencing to possible future work."

### ”conclusion”

- development of data pipeline: starting with .stl files and ending with normalized, aligned, cleaned and voxelized 3D models
- defector approach: development and evaluation of different methods in order to introduce defects
- Defector-top-down-view approach delivers most flexible and best results
- end-to-end deep learning pipeline using different kinds of architectures
- InceptionNets and ResNet: outstanding performance

**"future work"**

- evaluation on a specific test set like other chunk of ABC dataset or samples of the Thingi10K dataset
- Solve data problems mentioned in the failure analysis part in the results chapter such as removing models already having a hole and too complex models, f.e. by using other criterions for model selection f.e. the compactness of the models (see Appendix 7).
- Further develop DefectorTopDownView such that it uses all informations given in the top-down-view by finding a suitable similarity measure for the area where the hole will be added.
- Further develop the approach of adding defects from another axis-aligend directions and non-axis aligend directions to the 3D models as already started in 7
- evaluate parameter setting of defector, play around with "radius printable", "radius non-printable", "boarder" and the type of defect
- Use other voxel representations such as tsdf or sdf and use
- change the kernel size of the used deep learning models
- evaluate neural network performance using explainable AI techniques
- Self-supervised approach: Train on original 3D models without artificial defect using an autoencoder. Once the autoencoder can reconstruct the 3D models, only take encoder part and stack final layer for binary classification. Take 100 models, each labeled by an expert, of different difficulty, 50 printable, 50 non-printable and apply those models for training the encoder.

## References

- [1] Open3d: A modern library for 3d data processing. URL <http://www.open3d.org/docs/release/index.html>.
- [2] 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#>.
- [3] Der additive manufacturing campus: Fahrzeugteile aus dem drucker., 2020. URL <https://www.bmwgroup.com/de/news/2020/additive-manufacturing.html>.
- [4] Aaron. Find rotation matrix to align two vectors. URL <https://stackoverflow.com/questions/67017134/find-rotation-matrix-to-align-two-vectors>.
- [5] 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.
- [6] Richard Aston. 3d printing done right. URL <https://www.boeing.com/features/innovation-quarterly/nov2017/feature-thought-leadership-3d-printing.page>.
- [7] 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.
- [8] 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.
- [9] Vivienne Brando. 3d printing - decarbonisation meets digitalisation. URL <https://www.airbus.com/public-affairs/brussels/our-topics/innovation/3d-printing.html>.
- [10] 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>.
- [11] 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>.
- [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](https://www.nasa.gov/centers_marshall/news/releases/2020/nasa-looks-to-advance-3d-printing-construction-systems-for-the-moon.html).

- [13] 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.
- [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*.
- [15] A. Elfes. Occupancy grids: A stochastic spatial representation for active robot perception, 2013.
- [16] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.
- [17] Andrew Gibiansky. Bringing hpc techniques to deep learning. URL <https://web.archive.org/web/20180128132031/http://research.baidu.com/bringing-hpc-techniques-deep-learning/>.
- [18] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- [19] 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.
- [20] 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>.
- [21] kctess5. Cuda c mesh voxelizer. URL <https://github.com/kctess5/voxelizer>.
- [22] 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>.
- [23] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [24] 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.
- [25] 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.

- [26] Anne Verroust-Blondet Mohamed Chaouch. Alignment of 3d models. *HAL Id:hal-00804653*, 2015.
- [27] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28, 1997.
- [28] Kevin P Murphy. *Machine Learning-A Probabilistic Perspective*. MIT press Cambridge.
- [29] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020. URL <https://developer.nvidia.com/cuda-toolkit>.
- [30] 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.
- [31] 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.
- [32] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.
- [33] shapeways. Tips for successful modeling.
- [34] David Stutz. A formal definition of watertight meshes. URL <https://davidstutz.de/a-formal-definition-of-watertight-meshes/>.
- [35] 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.
- [36] 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>.
- [37] 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>.
- [38] 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>.

- [39] 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>.
- [40] 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.

## Authors' contributions

*Bok, Felix*

*Bouziane, Nouhayla*

*Ebid, Ahmed*

*Kiechle, Johannes*

*Srinivas, Aditya Sai*

Example:

*AB carried out the molecular lab work, participated in data analysis, carried out sequence alignments, participated in the design of the study and drafted the manuscript; CD carried out the statistical analyses and critically revised the manuscript; EF collected field data and critically revised the manuscript; GH conceived of the study, designed the study, coordinated the study and helped draft the manuscript. All authors gave final approval for publication and agree to be held accountable for the work performed therein.*

## Compactness Parameter Model Selection

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 [34]. The mathematical formulation of the compactness calculation can be outlined as follows:

$$\text{compactness} = \frac{\text{volume}}{\text{boundingVolume}} \quad (16)$$

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) [30]:** 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:

$$SDF(x) = s : x \in \mathbb{R}^3, s \in \mathbb{R} \quad (17)$$

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,

$$x \in \mathbb{R}^3 = \begin{cases} \text{outside} & \text{if } f(x) > 0 \\ \text{surface} & \text{if } f(x) = 0 \\ \text{inside} & \text{if } f(x) < 0 \end{cases} \quad (18)$$

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.

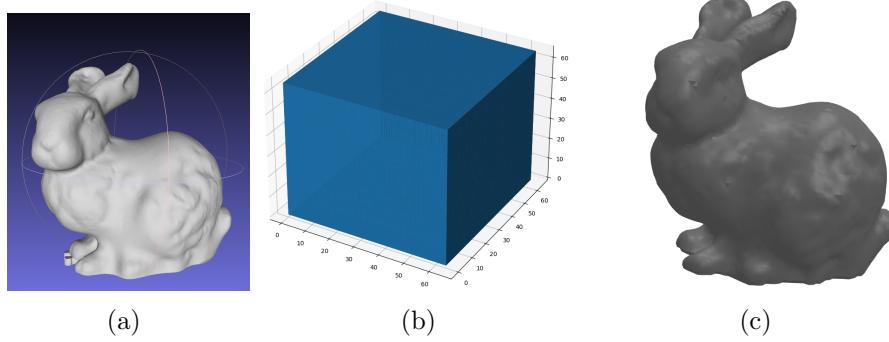


Figure 19: (a) Mesh: Stanford bunny (b) Signed Distance Function (SDF) (c) Reconstructed bunny using marching cubes algorithm.

- **Truncated Signed Distance Function (TSDF)** [30]: 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.

-0.9	-0.3	0.0	0.2	1	1	1	1	1
-1	-0.9	-0.2	0.0	0.2	1	1	1	1
-1	-0.9	-0.3	0.0	0.1	0.9	1	1	1
-1	-0.8	-0.3	0.0	0.2	0.8	1	1	1
-1	-0.9	-0.4	-0.1	0.1	0.8	0.9	1	1
-1	-0.7	-0.3	0.0	0.3	0.6	1	1	1
-1	-0.7	-0.4	0.0	0.2	0.7	0.8	1	1
-0.9	-0.7	-0.2	0.0	0.2	0.8	0.9	1	1
-0.7	-0.6	-0.1	0.1	0.3	1	1	1	1
0.5	0.3	0.2	0.4	0.8	1	1	1	1

Figure 20: TSDF Representation REFERENCE: SOURCE OF IMAGE

**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.

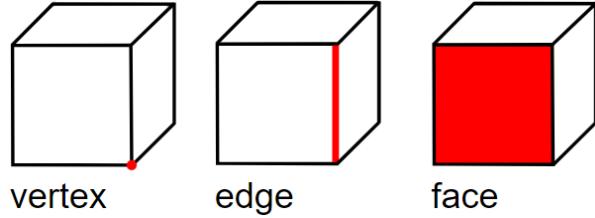


Figure 21: Example: Vertex, edge and face of a cube.

**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.

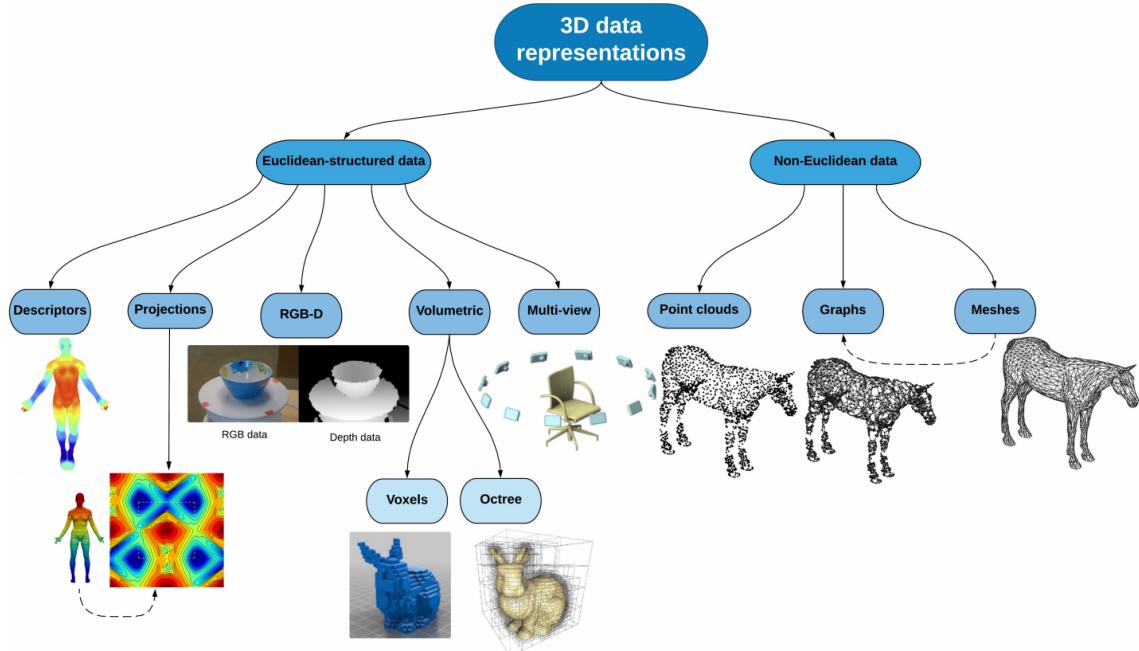


Figure 22: Overview: Different 3D data representations. Source: [5]

## AMC Dataset Examples

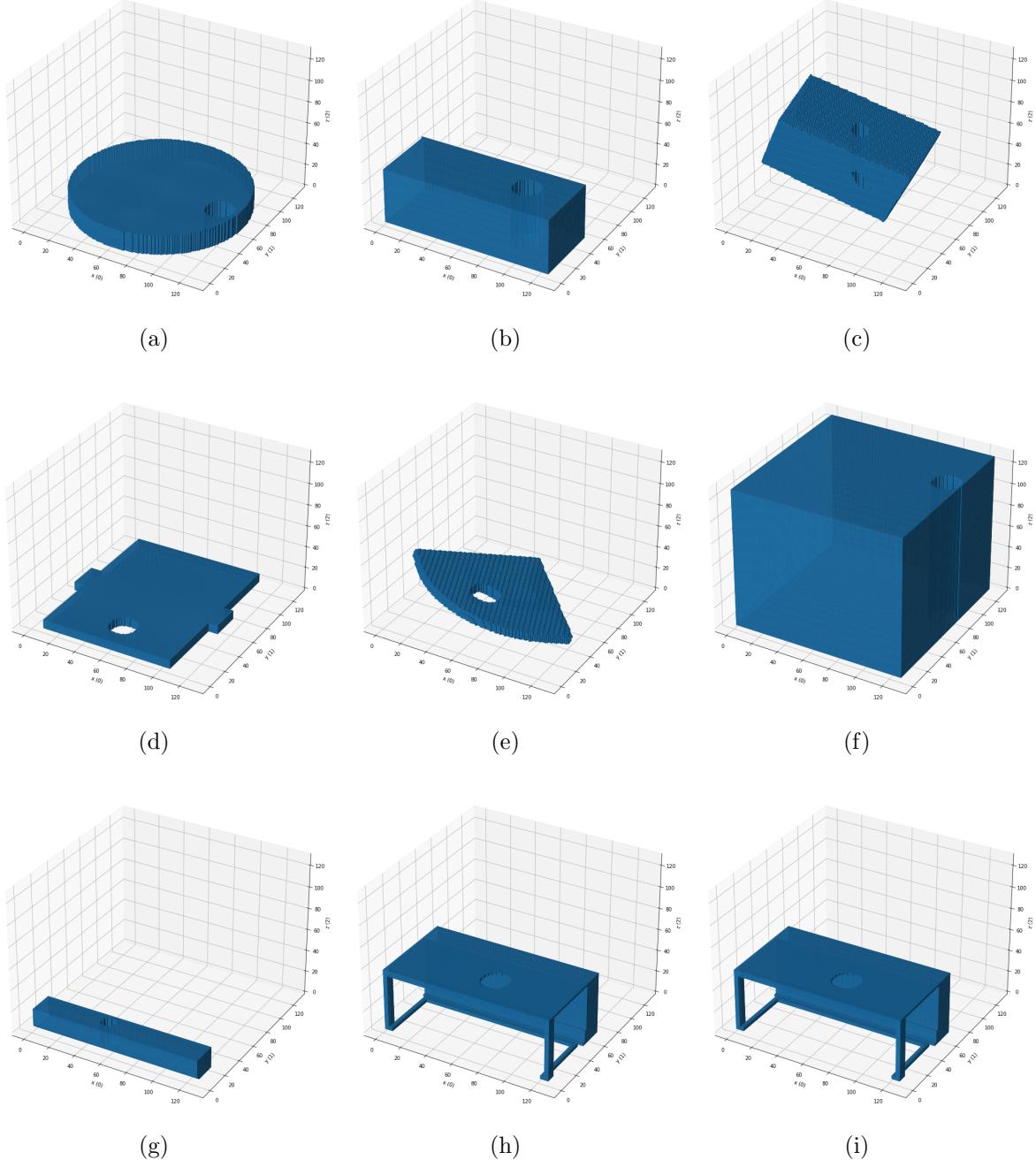


Figure 23: Examples of the AMC Dataset.

## 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 dimensionality 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 trials 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 24 shows some samples of the defects added by the ExhaustiveDefector.

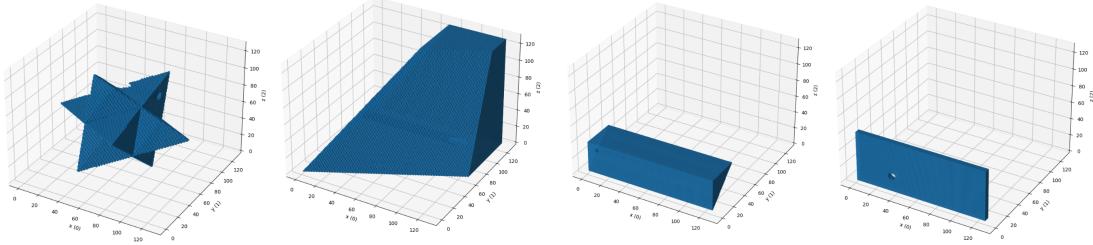


Figure 24: Samples of ExhaustiveDefector outputs.

## 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  $\phi_x, \phi_y, \phi_z$  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 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:

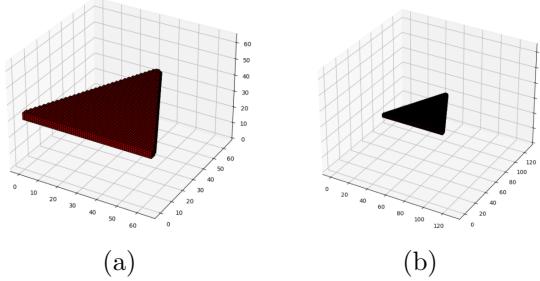


Figure 25: (a) 3D model with 64 resolution, (b) 3D model with 128 resolution.

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.

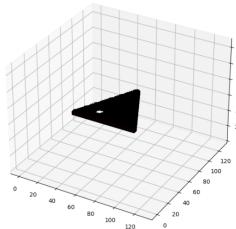


Figure 26: 3D model with a randomly rotated hole.

## Detailed Description of the Architectures Used

### Vanilla3DCNN

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 27: Detailed description of Vanilla3DCNN architecture used in the project.

### ResNet

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

InceptionNet_v1 Architecture									
Type	Patch_size/Stride	Output_size	Depth	#1x1x1	#3x3x3 reduce	#3x3x3	#5x5x5 reduce	#5x5x5	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 29: Detailed description of InceptionNet V1 architecture used in the project.

## InceptionNet V3

InceptionNet_v3 Architecture										
Type	Patch_size/Stride	Output_size	Depth	#1x1x1	#3x3x3 reduce_1	#3x3x3 reduce_2	#3x3x3	#3x3x3 reduce	#3x3x3	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 30: Detailed description of InceptionNet V3 architecture used in the project.

## Train/Validation Loss & Accuracy Development

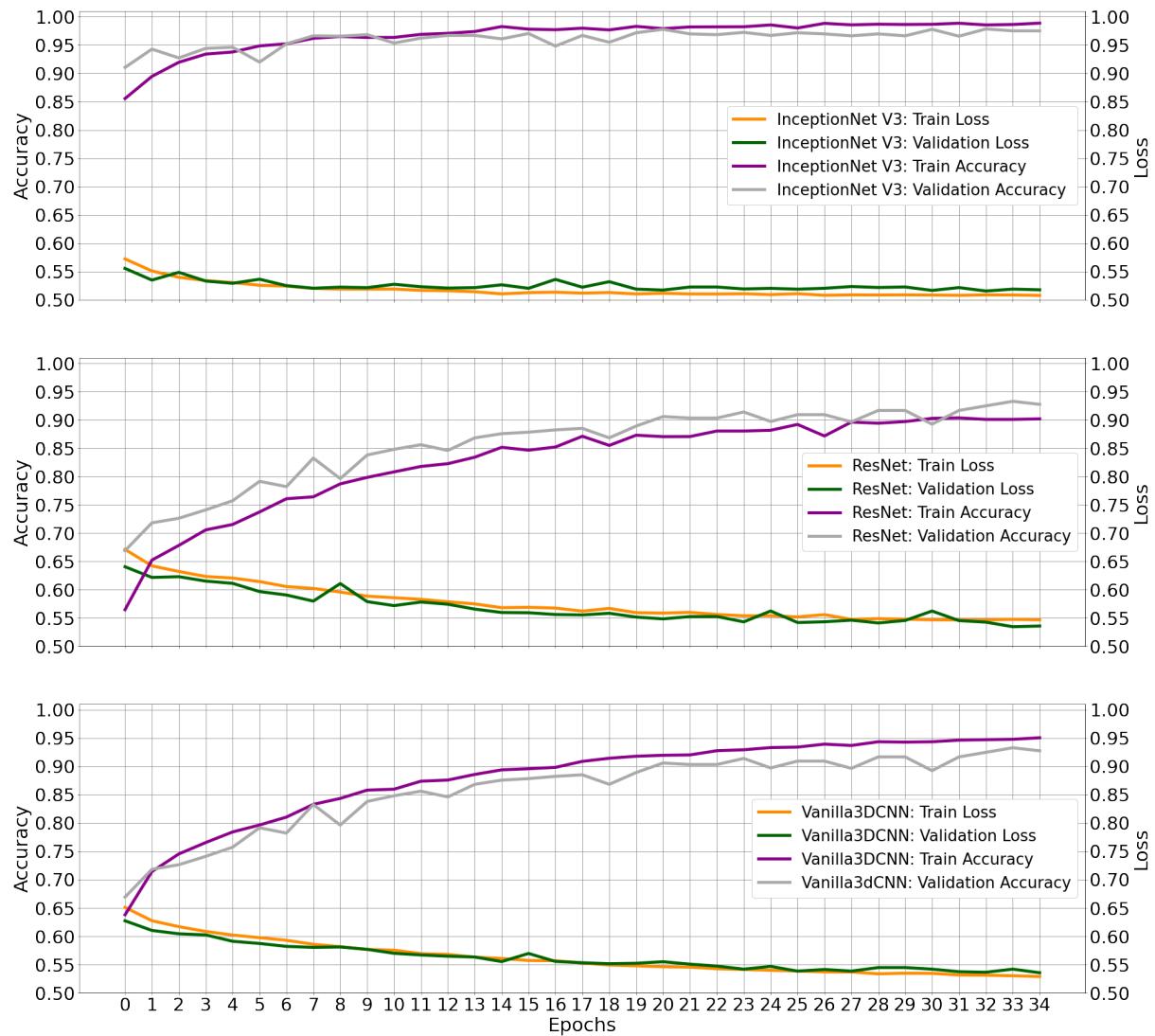


Figure 31: Development of loss & accuracy on the train & validation data depicted for the first 35 epochs. Top: InceptionNet V3, middle: ResNet, bottom: Vanilla3DCNN.