Point Cloud Scans - Artefacts Filtering

Lukáš Gajdošech

January 15, 2020

1 Introduction

In this project, we evaluate several models and approaches for solving the problem of author's master's thesis. We have already presented the *problem definition* in the **project proposal** document. For a recap, we are given a scan from the *Photoneo 3D scanner*, which is a point-cloud organized into a 2D matrix. It contains the 3D position, intensity and normal of each point. Our goal is to filter artefacts, caused by lightning and reflections, see Fig 1.

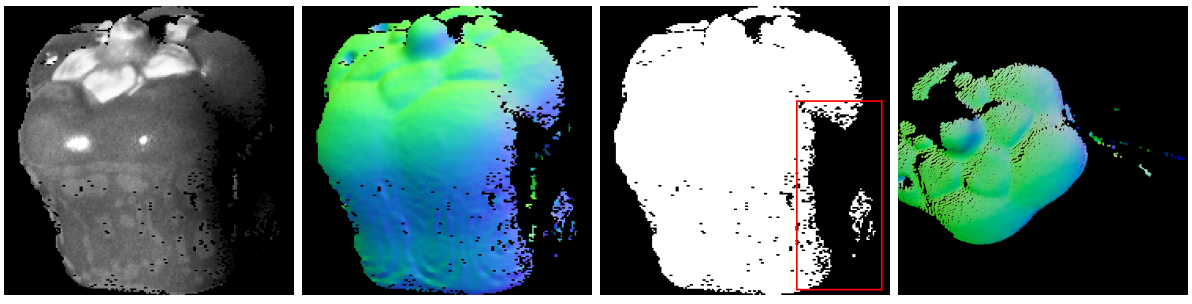
Ground truth is made using an existing algorithm, which needs the full reconstructed object, as it uses the redundancy between scans (overlays). Proposed machine learning approaches should capture the characteristics of the artefacts, enabling the filtering from a single-view scan, which is a notable advantage over the conventional solution.

We planned to only use scans of objects from an uniform material (like glossy ceramics), but after some testing, it seemed like a pointless simplification. The prepared dataset therefore contains also other objects, like a metallic model of a car with translucent plastic parts.

This document describes two different approaches for solving this problem. In the first approach, we generate fixed $n \times n$ (n is a modifiable hyper-parameter) neighborhoods for each point. This gives us the feature vector, which is then inputted into a binary classification model (**SVM, Random Forest** [4]), that decides, whether the center point of the $n \times n$ window is an artefact or not. Main drawbacks of this approach are the ambiguity of the n parameter and a low performance, as we need to run the prediction individually for every single point.

In the second approach, we generate feature maps of the input scan (2D images of normals, intensities and depths) and supply them into a convolutional neural network. The output labeling is a binary segmentation mask, dividing points of true geometry from artefacts. This approach seems to be more scale-able and definitely needs more investigation beyond this project. While many **CNN** architectures might work [3], **UNET** is evaluated here [5].

Neither of the approaches are implemented in a final state and even the dataset is too small and limited. The main objective here is to try out these models and discuss results. All the code is available at `https://github.com/gajdosech2/pc-filtering`.



(a) intensity map (b) normal map (c) ground truth (d) different view

Figure 1: Points look correct on the single-view scan (b), but from a different angle, we see they are in fact artefacts, outside of the object's geometry (d).

2 Data Preparation

We are working with a custom data, obtained as the part of the assignment. Generating the dataset was unfortunately the most time consuming part of the project. After acquiring a set of scans with clearly visible artefacts, we had to preprocess the point cloud data individually for each of the approaches. The actual data preparation was done in a separate `C++` project `PCFiltering`, which can be found in the mentioned **github** repository.

2.1 Approach I

Binary classification models such as the **SVM** and **Random Forest** accepts a feature vector as their input. In this approach, we look at patches of fixed $n \times n$ size. We have the middle point c, that we wish to classify and all its neighbors points $o^{(i)}; 0 \leq i \leq n \times n; o^i \neq c$, see Fig 2 (c). What *features* do we extract about these points? We consider the actual values of intensity, normal and depth to not be the best choice, as it might easily confuse the classifier.

The object is scanned from different angles, while being rotated on a rotary table. The normal values for points in the object are recorded relatively to some global coordinate system. This means, that two scans of the same object rotated by roughly π radians have opposite normals, as shown in Fig. (a) and (b). We definitely do not want our classification to consider normals in a certain absolute direction to signalize an artefact, as that is generally not true.

Therefore, for each point we instead record the **difference** of these values from the center point c. Formally, for each point $o^{(i)}$ we take (can be viewed as approximating the *derivatives* from discrete data):

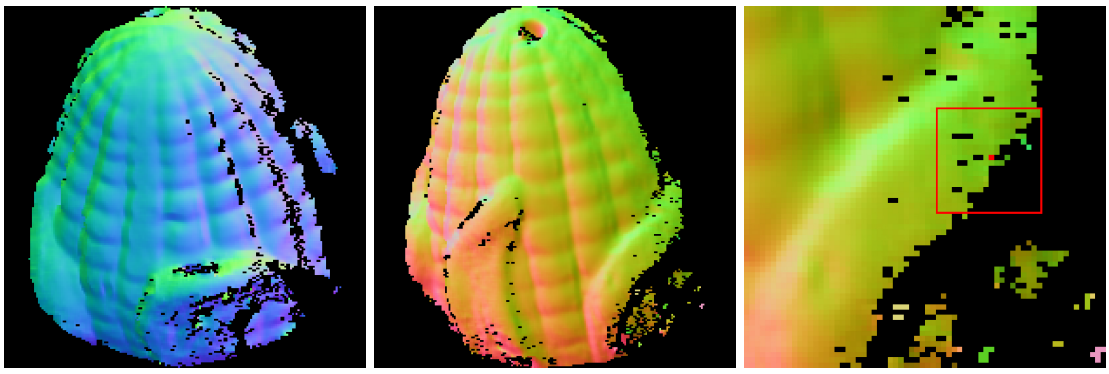
$$|c_{depth} - o^{(i)}_{depth}|, |c_{intensity} - o^{(i)}_{intensity}|, dist(c_{normal}, o^{(i)}_{normal})$$

where $dist(n_1, n_2)$ is an *euclidean distance* between the two vector normals. This gives us a matrix with $n \times n - 1$ rows and 3 columns. For the implementation in this project, we have chosen $n = 19$. The matrix is then unrolled into a vector with $(19 \times 19 - 1) \times 3 = 1080$ dimensions.

Together with a normalization, this approach should ensure that these features are invariant from transformations like rotation and scale. In other words, we are getting the information regarding how different the center point is from its neighborhood.

Using an unique *id number* for each point recorded in the header of the exported `.csv`, we can map each *neighborhood* file to its *ground truth* value, saved in a separate file. Similarly, the prediction of the classifier is saved into a file with *point id* $\rightarrow$ *prediction*, allowing a reconstruction of the filtered scan.

This approach have severe disadvantages. From a theoretical standpoint, the main drawback is the parameter n. In a scan of a bigger object, or with a finer resolution, it can easily happen that some artefact covers the whole $n \times n$ patch, forming an *uniform neighborhood* on its own. We are missing a global information and assuming that the size of artefacts is somehow bounded and that they are always surrounded with a homogeneous object's geometry. Additionally, increasing the n have drastic effect on the performance. This approach works for filtering small, random deviations consisting of just few points, but that is just a subset of possible artefacts. One possible improvement would be to *downsample/upsample* the scan to fixed resolution beforehand. Then it may be more realistic to find a window of a constant size, containing enough of both local and global information.



(a) corn saltshaker side A (b) corn saltshaker side A (c) point with neighborhood

Figure 2: Subfigures (a) and (b) show why the actual value of normals may not be a good feature, as it is skewed by transformations, (c) is the visual illustration of the neighborhood captured in the `.csv` file.

For our limited dataset, this approach showed some results, but we have also discovered how unbearably slow this per-point strategy is. A single scan has usually anywhere from 10^5 to 10^6 points and the fully reconstructed object can easily consist of 32 of these scans. now imagine the number of *.csv* files with neighborhood information for every point and the actual process of running the classification... For demonstration purposes, we have generated the data using small scans, each with the number of points in the order of 10^4. Works fine as a *proof of concept*, but is unfortunately far from a practical usability.

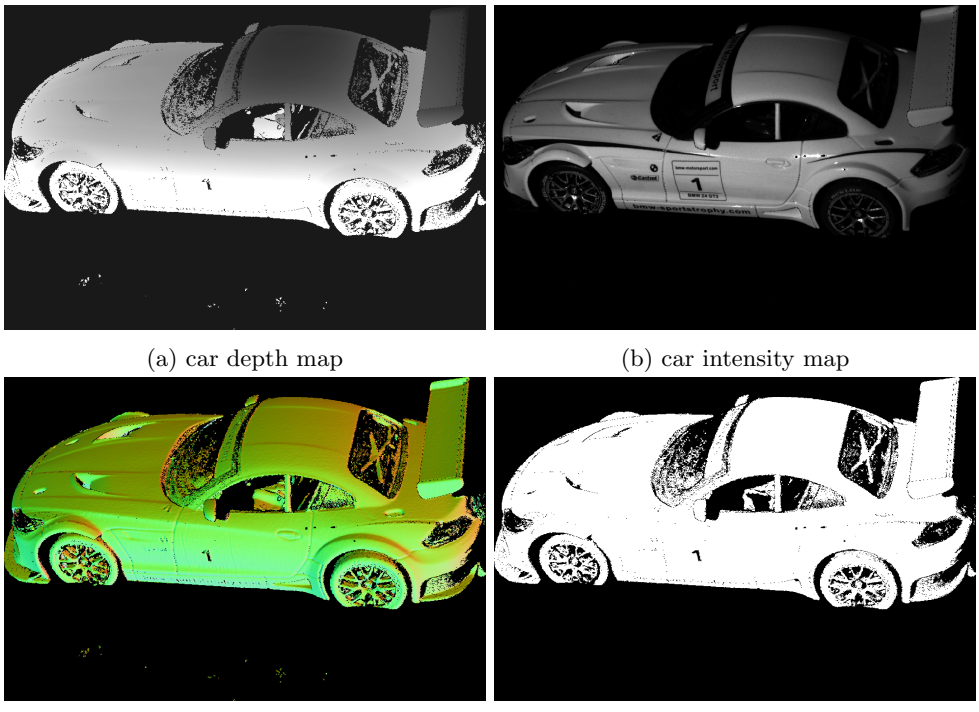
2.2 Approach II

In the second approach, we are preparing the data to be used in a convolutional neural network, with the *encoder-decoder* structure, such as the **UNet** [5]. One possibility would be to feed the actual 3D point cloud data into the network [1]. However, most of the existing architectures work with 2D images. Transforming organized point cloud into images is straight forward. Generated images were already shown throughout this document, but Fig. 3 shows the complete set of one scan. **CNN**s usually work with **RGB** data, but that is not enough for our purposes. Only the images with normals occupy all the 3 channels. We therefore stack the images together, resulting in a tensor with 5 channels.

As mentioned in the data preparation for **Approach I**, the problem with absolute values of these features is still present. Transformations like flipping the direction of normals should not affect the classification. In this approach however, we are not bothered by this and trust that given enough training samples, **CNN** will not be confused by this. After all, results show that these networks are able to classify an image of a cat, whether it is on sunlight or in a dark room, which is a similar situation.

Note that the normalization by stretching values of each image into a $[0, 1]$ range (also known as *min-max scaling* [2]) is done already in this pre-processing step. Otherwise, visual inspection of the images would not be possible, as for example intensities are sometimes cramped up into a small range like $[0.02, 0.09]$, which would result in a mostly black picture.

One problem is the inconsistent size of the images, which will be solved by *tiling* them. This will be done in the program before the training. We only make sure that there is no dead space around the object in this stage, by trimming the scans, but other than that, images are saved in the resolution of the original scan. Again, *downsampling/upsampling* may increase generalization.



(a) car depth map (b) car intensity map

(c) car normal map (d) car ground truth

Figure 3: Set of images for the scan of a car model. Images (a) - (c) are stacked into a 5 channel image used as an input to the CNN and (d) is the ground truth classification mask.

3 Methods

After exporting the data for each of the approaches, we are ready to do the machine learning training. In the **github** repository, these can be found in the `Notebooks` folder, as `.ipynb` jupyter notebooks.

3.1 Approach 1

After the lengthy technical stage of preparing the data, we now load the neighborhood information for each point and split it into two sets for training and testing. First, we train the **SVM** from the `sklearn` library. After trying out few kernels and parameters, we decided to use the *Gaussian kernel* (referred to as **R**adial **B**asis **F**unction) with penalty parameter $C = 1000$ and $\gamma = \frac{1}{10000}$. Using these parameters, we achieved 100% training accuracy. Wait, what?

It is important to realize that we are working with a very *skewed* dataset. Only about 2% of the points are artefacts, which means that even a classifier which would mark every point as a true geometry, ignoring all the artefacts, would score 98% accuracy. Generally a solid performance, utterly useless in our case. Instead of evaluating the classification based on accuracy, we use a *confusion matrix* [2]:

$$\mathbf{M}^{(svm)} = \begin{bmatrix} 166 & 112 \\ 53 & 18508 \end{bmatrix} \quad \text{precision\_svm} \sim 0.994 \quad \text{recall\_svm} \sim 0.997$$

$\mathbf{M}^{(svm)}$ is the confusion matrix of our **SVM** model, obtained using `cross_val_predict()`. This function performs the *K-fold split* (similarly to *K-fold cross-validation*), but instead of returning the evaluation scores, it returns the predictions made on each test fold. This gives us a clean prediction for each instance in the training set. Prediction is made by a model that never saw the given sample during training.

Each row in a confusion matrix represents an actual class, while each column represents a predicted class. $\mathbf{M}^{(svm)}_{1,1} = 166$ gives us the number of points, that were correctly classified as being artefacts (*true negatives* = *TN*). $\mathbf{M}^{(svm)}_{1,2} = 112$ is the number of points, that are artefacts based on the ground truth, but were misclassified as being a true geometry (*false positives* = *FP*). In the second row, we have the points that are part of the true geometry. $\mathbf{M}^{(svm)}_{2,1} = 53$ of them were wrongly classified as being artefacts (*false negatives* = *FN*) and finally, $\mathbf{M}^{(svm)}_{2,2} = 18508$ points were correctly predicted as being part of the object's surface (*true positives* = *TP*).

Two more concise metrics can be calculated from the *confusion matrix*: *precision* and *recall*:

$$\text{precision} = \frac{TP}{TP + FP} \qquad \text{recall} = \frac{TP}{TP + FN}$$

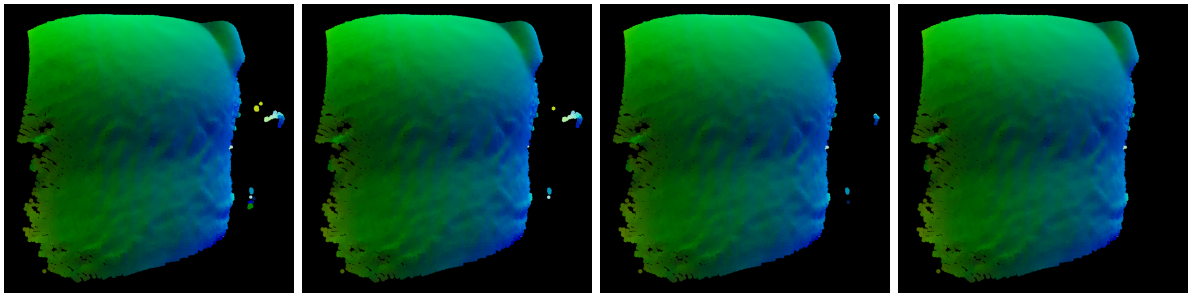
Precision tells us how precise the classifier is, when it marks some point as being a true geometry and *recall* is the ratio of true geometries that are correctly detected by the classifier. It can be shown, that there exist a trade-off between these two metrics, i.e. we maximize one at the cost of the other [2]. In our case, *recall* is the more important metric. We want a classifier, that correctly detects all the true geometries, albeit if it means that it is not very precise and some of those are in fact artefacts. That is definitely better than a highly precise classifier, which marks a point as a true geometry only when it is sure, resulting in lot of true points being lost, marked as artefacts. Classifier with a good *recall* and lower *precision* can always be re-run several times on the same scan, similarly to how median filtering is used for salt and pepper noise in image processing.

For a comparison, these are the metrics for the **Random Forest** classifier:

$$\mathbf{M}^{(rfc)} = \begin{bmatrix} 182 & 96 \\ 6 & 18555 \end{bmatrix} \quad \text{precision\_rfc} \sim 0.995 \quad \text{recall\_rfc} \sim 0.999$$

On the folds of the training data, it performs better than **SVM**, only 6 points of the true geometry have been misclassified as artefacts. Similar results are obtained on the test data, see Fig. 4. Both of the models seems to keep most of the true geometry intact, which is good. **SVM** filters less artefacts (0.998 recall), while **RFC** is more more aggressive and closer to the ground truth (0.999 recall). However, the test data sample is very similar to the training examples (all of them are ceramic saltshakers of different shapes).

So the message is clear - the training dataset is too small and limited to obtain any general conclusion. We most likely *overfit* the training data really hard. Unfortunately, as mentioned in the section about data preparation, using richer datasets consisting of more samples is painfully slow using this approach.



(a) original scan (b) **SVM** filtration (c) **RFC** filtration (d) ground truth

Figure 4: **RFC** filtration is close to the ground-truth. Images were intentionally captured from an angle, where artefacts are clearly visible.

3.2 Approach 2

Before training the **CNN**, we have to make sure all our images are of the same size. We satisfy this by *tiling* them into windows of size 64×64 with a step of size 32. Since the step is smaller than the window, same points will occur in multiple tiles, as Fig. 5 illustrates. This is beneficial for the training process, as it fully exploits the training set, gaining as much information from it as possible.

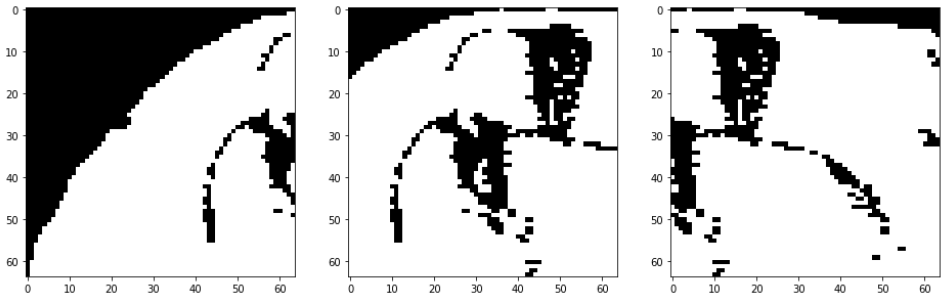


Figure 5: Tiling of the images, illustrated on the ground truth segmentation mask. Due to the smaller step size, some points are duplicated across multiple tiles.

After the *tiling*, we delete tiles that are empty and we perform additional *data augmentation*, adding horizontal and vertical flips. This not only gives us more training examples, but should also ensure better generalization and hopefully teach the network, that artefacts are invariant from transformations. By this process, we have generated 23313 training samples from 38 original scans.

Tiling has one disadvantage. After training the network, when we wish to make a filtration on a new data, we must first tile it into pieces and then reconstruct it back into the full scan. We did not bother with this technical step in this project, as it is clearly possible, but we did not want to waste time on it, as it is not important from the machine learning perspective. Instead, in the prediction step, we just use scans that are already of the 64×64 size for simplicity.

CNN with the *encoder/decoder* architecture is the obvious choice here [3]. We could make one from the ground up, but instead opted to use an implementation of the **UNet** found in `Segmentation Models` library, which is based on the `Keras` framework [6]. We are optimizing the *binary crossentropy loss function* and use the F_1 score *metric*, which is the *harmonic mean* of the *precision* and *recall* metrics described in the proceeding subsection.

After 10 epochs using a batch size of 64, we obtained training score $F_1^t = 0.9871$ and evaluation score $F_1^e = 0.9830$ on a 30% of previously unseen, validation data. The training was performed using *RTX 2060* and was fast, so there is definitely a reserve for more epochs and training data.

Finally, we can see how the network works in practice, on a data for which we do not have ground truth. As mentioned, the existing algorithm needs the whole reconstruction of an object. We purposely picked this solo scan from a larger set, to simulate the real world situation we are trying to solve here - filtering a single-view scan.

Fig. 6 shows the normal map and intensity map of the input scan. These two images are combined into a $64 \times 64 \times 4$ tensor (we decided to omit the use of depth map in the network). As Fig. 7 illustrates, this is a patch from a bigger scan. It is not clear from the the view of the scanner, whether the weirdly shaped areas of points are a part of the toilet's inner rim or not (yes, we are using a scan of a toilet, because it is from glossy ceramic, producing beautiful artefacts). But looking at the point cloud from a different angle clearly shows, that these are in fact artefacts. Fig. 8 is the prediction of the network. It does not produce the binary classification, but rather a probability prediction for each point. Using a *threshold* equal to the average gray value in the raw output, we obtain the final filtered image. Most of the artefacts were filtered, leaving out some small bits. We could filter these as well by setting a higher textitthreshold, at a cost of losing part of the probably true geometry in the lower right corner. *Threshold* is yet another hyper-parameter.

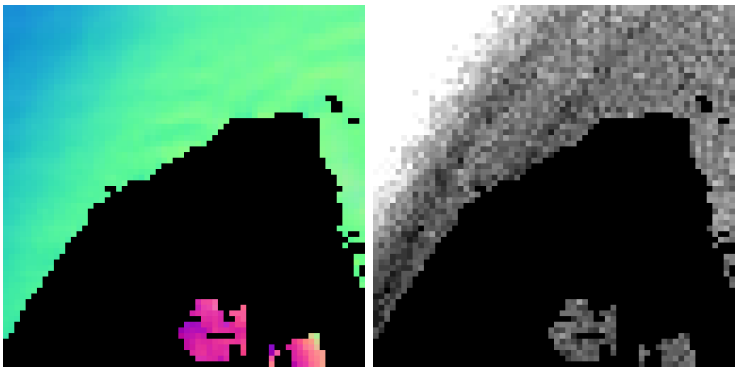


Figure 6: Normal map and intensity map of the test image.

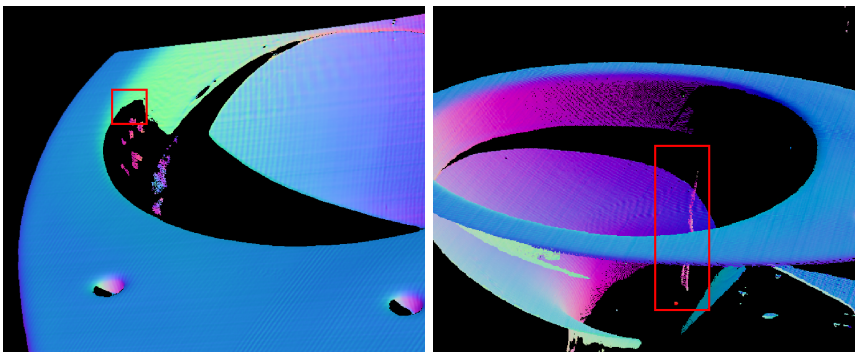


Figure 7: First image demonstrates the position of the patch in the bigger scan. On the second image, we can see that the area of points form artefact, caused by reflection.

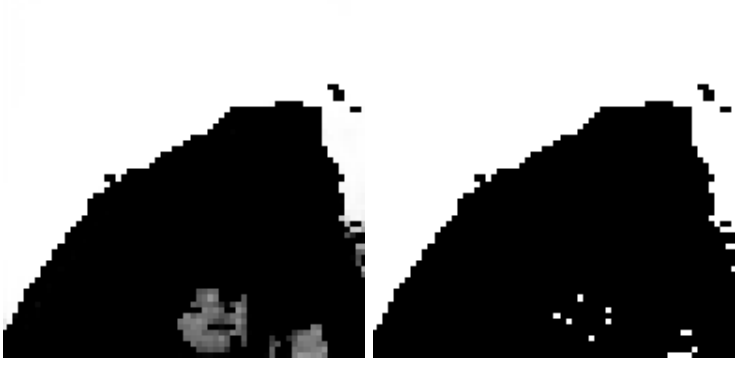


Figure 8: Raw prediction of the network and the final one, produced by *thresholding*.

4 Discussion

In this project, we scratched the surface of the underlying problem and presented some simple machine learning algorithms trained on a *toy* dataset. Everything in here can be questioned and discussed, from the choice of architectures, followed by the way we processed the data, to the way we have set-up hyper-parameters. Any kind of feedback regarding any of these areas will be highly appreciated.

We are aware of the fact that some conclusions and assumptions made here were *informal* and a more thorough theoretical analysis is needed. From a practical standpoint, trying out different hyper-parameters in both of the approaches and a wider set of possible **CNN** architectures would be worthwhile. Additionally, finding the values of parameters using an automated process like *Grid Search* is definitely more bullet-proof and faster, than doing it by hand [2].

The second presented approach is considered to be much more scale-able. Due to the performance limitations and weak generalization, the first approach probably showed the maximum of its potential. In the future, we therefore plan to build on the **CNN** strategy. Both from the standpoint of modifying the architecture, as well as trying other, non-ML techniques, like mathematical morphology for processing the data.

Even though all the drawbacks and limitations, we still think that notable results were achieved here, being *proof of concepts* at minimum. We are confident to say, that this is the one trial-error-analysis development cycle of a machine learning project, as mentioned in the assignment.

References

[1] David de la Iglesia Castro. 3D MNIST. `https://www.kaggle.com/daavoo/3d-mnist`, 10. 2016.

[2] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc., 1st edition, 2017.

[3] Divam Gupta. A beginner's guide to deep learning based semantic segmentation using keras. `https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html`, 6. 2019.

[4] Andrew Ng. CS229 lecture notes - supervised learning. `https://see.stanford.edu/Course/CS229`, 2012.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: convolutional networks for biomedical image segmentation. volume 9351, pages 234–241, 10. 2015.

[6] Pavel Yakubovskiy. Segmentation models. `https://github.com/qubvel/segmentation_models`, 2019.