3D CNN implementation for Object Recognition

Emerson Rodrigues Vero filho

Università degli Studi di Padova

June 2023

Abstract

A light transformation pipeline was implemented in Pythorch to transform polygonal mesh samples from the ModelNet10 dataset to point cloud format, and further to voxel grid format. Models of 3D Convolutional Neural Network architecture were proposed and implemented and study to learn classifying objects in the voxel grid format, on the benchmark of the ModelNet10 dataset. Good Performing results were obtained for the training and validation sets, and a best model chosen, but there was a significant drop on accuracy for the suggested test set. This behaviour was analysed on its possible causes and solutions.

1 Introduction

In recent years, the number of applications of 3D shapes representation has increased sharply and has posed 3D recognition as a fundamental problem in computer vision, computer graphics, and augmented reality. Convolutional Neural Networks are widely used to classify 2D images, having their architectures and results continuously enhanced by researchers. The relatively smaller sample size, if compared to 3D ones, often allows researchers to tune their resources with more flexibility to tackle the problem. The task of 3D objects recognition is often more computationally expensive as it has to deal with considerably larger sample sizes.

Some approaches consist of turning the problem into a 2D one, making projections of the samples, but often these methods encounter severe self-occlusions [1]. Point cloud-based methods have become widely applicable in recent years, using range sensors in cars and robots or depth cameras like the Kinect, but they suffer from harsh noises and waste substantial structural information of the 3D shapes [2]. Voxel Grid-based methods try to eliminate this problem and effectively present better results, even though they have the drawback of requiring large memory, as well as rendering voxel grids generates unnecessarily voluminous data and quantization artifacts. [3] This paper presents a light transformation pipeline to convert polygonal meshes in point cloud format and further in voxel grid format, customizing the samples resolution. Furthermore, it focus on the implementation of a 3D convolutional neural network to learn voxel grid objects extracted from the ModelNet10 Princeton Dataset.

2 Dataset

The ModelNet10 dataset is a widely used benchmark dataset for 3D object recognition and classification tasks [4]. It consists of 3D CAD (Computer-Aided Desing Software) models from 10 different object categories. Each category represents a different type of object commonly found in indoor environments. The CAD models are represented as a network of vertices connected by triangular meshes (faces), where each vertex is defined by its coordinates (x, y, z), and each face is defined by a set of three vertices.

2.1 Dataset Transformation

Table 1: ModelNet10 Classes Distribution

Name	**Train**	**Test**	**Total**
bathtub	106	50	156
bed	515	100	615
chair	889	100	989
desk	200	86	286
dresser	200	86	286
monitor	465	100	565
nightstand	200	86	286
sofa	680	100	780
table	392	100	492
toilet	344	100	444
Total	3991	908	4899

Figure 1: Polygonal Mesh Representation of a Chair

The dataset transformation process consists in the conversion of 3D CAD models from the ModelNet10 dataset into voxel grids. Voxel grids provide a volumetric representation of objects by discretizing the 3D space into a grid of voxels. Each voxel represents a small 3D unit and can be occupied or unoccupied by an object. The transformation enables the utilization of voxel-based deep learning approaches for object recognition. The entire composed pipeline was developed consisting of the following steps:

Mesh Representation: The initial step involves deploying the original 3D CAD models from the ModelNet10 dataset, as polygonal meshes. Each CAD model consists of a network of vertices connected by triangular meshes. The polygonal mesh is converted into an object that stores the vertices and faces, only.

Point Cloud Generation: A function able to generate points within the triangles was designed. Sets of random points of chosen quantity were generated within each triangle defined by three vertices. Given three vertices, ‘x‘, ‘y‘, and ‘z‘, a plane surface is well defined, as well as the triangle delimited by the vertices.

Barycentric coordinates were used to represent the points with each tringles. Barycentric coordinates are a system of coordinates used to express the position of a point within a triangle relative to its vertices. These values represent the relative weights or proportions assigned to each vertex of the triangle to determine the position of the generated point within the triangle. [8]

Randomly generate values ‘u‘ and ‘v‘ were sampled from a uniform distribution [0,1] with the role of representing the Barycentric coordinates of the generated points within the triangle. By varying the values of u and v within the range [0, 1] and ensuring that u + v ¡= 1, it is guaranteed that the generated random points lie within the triangle defined by the three vertices. These coordinates were then, selected. The remaining coordinate ‘w‘ was calculated for each point inside the triangle. The remaining coordinate ‘w‘ is computed as ‘1 - u - v‘, ensuring that the sum of the barycentric coordinates ‘(u, v, w)‘ is equal to 1.

The last step of this part of the processing was to calculate the Cartesian coordinates of the points inside the triangle using the barycentric coordinates ‘(u, v, w)‘ and the corresponding coordinates of the triangle vertices ‘(x1, y1, z1)‘, ‘(x2, y2, z2)‘, ‘(x3, y3, z3)‘. The Cartesian coordinates ‘(x, y, z)‘ of each point inside the triangle are obtained using the following equations:

$$x = u \cdot x1 + v \cdot x2 + w \cdot x3$$

$$y = u \cdot y1 + v \cdot y2 + w \cdot y3$$

$$z = u \cdot z1 + v \cdot z2 + w \cdot z3$$

Defined this generating function, 3 coordinate points were generated by iterating over the mesh's faces. There was also an intermediate step of calculating the area of each face (triangle), and sampling points within triangles with an area greater than a specified threshold. At this point, the data is already in the point cloud format.

Rescaling Stage: The Rescale stage in the transformation pipeline performs a scaling operation on the vertices of the input mesh. The scaling is defined by a desired size and a scaling parameter.

The Rescale stage takes the output points of the previous step, denoted as $\mathbf{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, where $\mathbf{v}_i$ represents the i-th vertex in 3D space.

The Rescale operation rescales each vertex $\mathbf{v}_i$ to a new vertex $\mathbf{v}'_i$ based on the desired size s and the scaling parameter p. The rescaling equation is given by:

$$\mathbf{v}'_i = \frac{s}{2} \times \frac{\mathbf{v}_i}{p \cdot \max(\mathbf{V})}$$

Here, $\max(\mathbf{V})$ denotes the maximum value among the coordinates of all vertices in the mesh.

In this equation, each point $\mathbf{v}_i$ is divided by the scaling parameter p multiplied by the maximum value in the mesh to normalize the point coordinates. The normalized point is then scaled by $\frac{s}{2}$ to fit within the desired size s. The resulting output $\mathbf{v}'_i$ is the rescaled point. The Rescale stage returns a new set of rescaled points.

This Rescale operation ensures that the mesh vertices are scaled proportionally to fit within the desired size while maintaining the original shape and relative positions of the vertices.

Filtering Stage: a fixed amount of the input points are random chosen to continue in the pipeline, the others are excluded to reduce computational effort of the following stages.

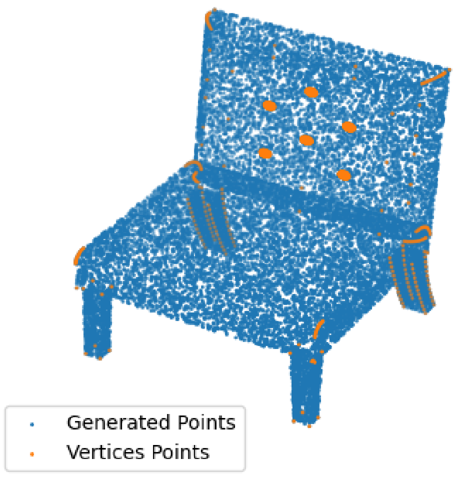


Figure 2: Point Cloud Representation - Bed

Broadcasting: The Broadcasting phase in the transformation pipeline performs a broadcasting operation on the vertices (points) of the input. This operation converts the point coordinates to integer values by truncating the decimal part, effectively "rounding down" the coordinates to the closest integer.

The Broadcasting operation converts each point $\mathbf{v}_i$ to a new point $\mathbf{v}'_i$ by simply truncating the decimal part of each coordinate. This is expressed as:

$$\mathbf{v}'_i = int(\mathbf{v}_i)$$

Here, $int(\cdot)$ denotes the function that truncates the decimal part and keeps the integer part of a number.

By applying the Broadcasting operation, the point coordinates are converted to integer values, effectively representing the points as discrete positions in 3D space.

The Broadcasting phase returns a new set of points $\mathbf{V}' = \{\mathbf{v}'_1, \mathbf{v}'_2, \ldots, \mathbf{v}'_n\}$.

This Broadcasting operation simplifies the representation of the points by discretizing the positions and removing the decimal precision.

Unique Point Selection: Redundant and sparsely occurring points are removed from the point cloud, retaining only those that occur with a frequency greater than or equal to a minimum count threshold.

Voxel Grid Generation: Finally, the voxel grid representation is generated from the processed point cloud. An empty grid of the desired size is created, and each voxel corresponding to a point in the processed point cloud is assigned a value of 1.

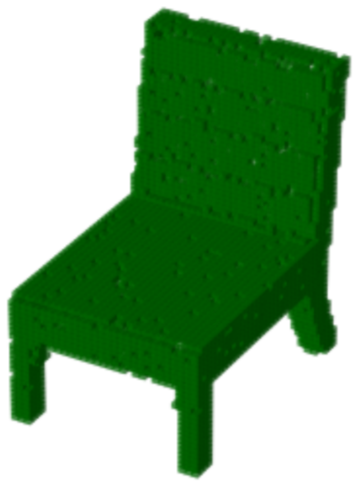


Figure 3: Voxel Grid Representation - Bed

The implemented broadcasting transformation, in which the point clouds are truncated to integers, is a significantly lighter alternative with respect to those used in the most widespread tools, in terms of computational effort. They often consist in calculating the distance of each point to the center of the voxel boxes, to perform the broadcasting by choosing the closest box.

The resolution for the outer box containing the object was chosen to be 60x60x60, higher than the one used in past implementations [5]. The higher the resolution is set, the less the effect of the lack of fineness of truncation is important.

Still, having the entire dataset transformed is a long and tiring task that could be more easily done using the Dask framework, in a Google Colab environment of a 35 GB RAM TPU (Tensor Processor Unit), and 40 available workers.

Storing tensors can be extremely memory demanding so the files were all compressed using the gzip library, and saved as ”pt.gz” files.

To begin, two 3D convolutional neural network were proposed with the following architectures:

2.2 Model 1

- Convolutional Layers:
  - `Conv1`: 3D Convolutional layer with 48 filters, kernel size of 6, stride of 2, and padding of 1, followed by ReLU activation.
  - `Conv2`: 3D Convolutional layer with 160 filters, kernel size of 5, and stride of 2, followed by ReLU activation.
  - `Conv3`: 3D Convolutional layer with 512 filters, kernel size of 4, and stride of 2, followed by ReLU activation.
- Global Max Pooling:
  - Global Pooling layer performing adaptive max pooling, resulting in a fixed-size feature representation of shape (batch_size, 512, 1, 1, 1).
- Fully Connected Layers:
  - `FC1`: Fully connected layer with input size 1024 and hidden size 128, followed by ReLU activation.
  - `FC2`: Fully connected layer with hidden size 128 and output size equal to the number of classes, followed by ReLU activation.

  changepage

2.3 Model 2

- Convolutional Layers:
  - `Conv1`: 3D Convolutional layer with 32 filters, kernel size of 5, stride of 2, and padding of 2, followed by ReLU activation.
  - `Conv2`: 3D Convolutional layer with 64 filters, kernel size of 5, and padding of 2 and stride of 2, followed by ReLU activation.
- Global Max Pooling:
  - Global Pooling layer performing adaptive max pooling, resulting in a fixed-size feature representation of shape (batch_size, 64, 1, 1, 1).

Table 2: Model 1 Architecture

Layer (type)	Output Shape	Param #
Conv3d-1	[-1, 48, 29, 29, 29]	10,416
ReLU-2	[-1, 48, 29, 29, 29]	0
Conv3d-3	[-1, 160, 13, 13, 13]	960,160
ReLU-4	[-1, 160, 13, 13, 13]	0
Conv3d-5	[-1, 512, 5, 5, 5]	5,243,392
ReLU-6	[-1, 512, 5, 5, 5]	0
AdaptiveMaxPool3d-7	[-1, 512, 1, 1, 1]	0
Linear-8	[-1, 128]	65,664
Linear-9	[-1, 10]	1,290
Total params		6,280,922
Trainable params		6,280,922
Non-trainable params		0
Input size		0.82 MB
Params size (MB)		23.96 MB
Forward/backward pass size		24.21 MB
Estimated Total Size		48.99 MB

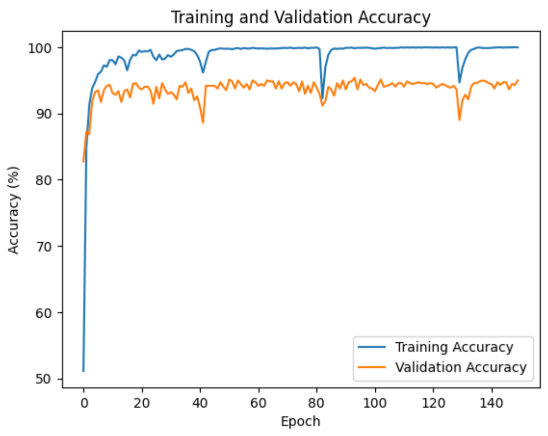


Figure 4: Train and Validation Accuracy for Model 1, Split Training Set

- Fully Connected Layers:
  - `FC1`: Fully connected layer with input size 1024 and hidden size 128, followed by ReLU activation.
  - `FC2`: Fully connected layer with hidden size 128 and output size equal to the number of classes, followed by ReLU activation.

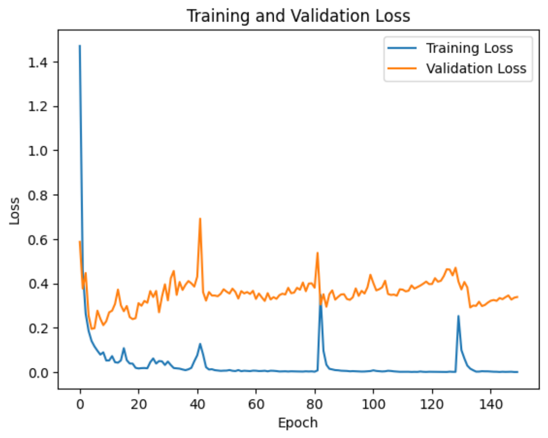


Figure 5: Train and Validation Loss for Model 1, Split Training Set

The final output for both models is the predicted class probabilities, so there are 10 possible outcomes.

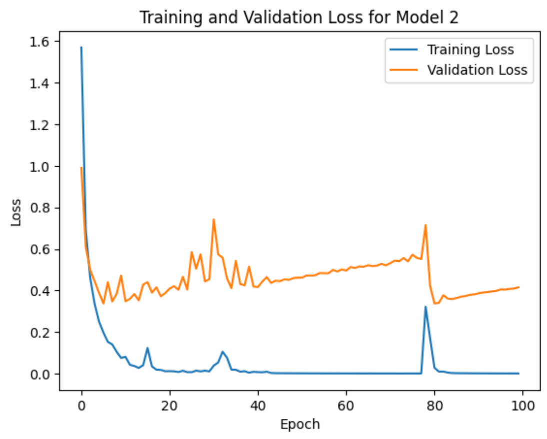


Figure 6: Train and Validation Accuracy for Model 1, Split Training Set

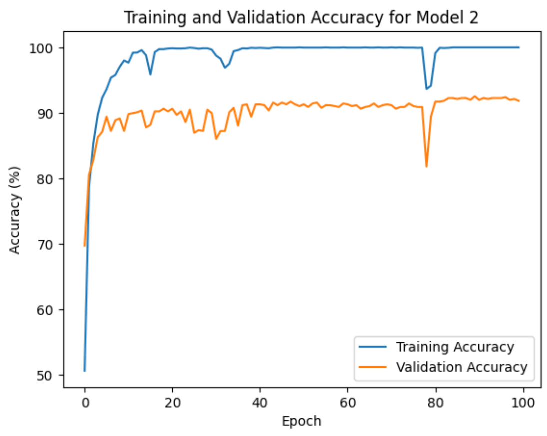


Figure 7: Train and Validation Loss for Model 2, Split Training Set

changepage

3 Training the Models

For the first training, the suggested training set of ModelNet10 was split into training and validation sets using a 80:20 ratio. The training set, comprising 80% of the data, was used for model training, while the remaining 10% formed the validation set

Table 3: Model 2 Architecture

Layer (type)	Output Shape	Param #
Conv3d-1	[-1, 32, 30, 30, 30]	4,032
ReLU-2	[-1, 32, 30, 30, 30]	0
Conv3d-3	[-1, 64, 15, 15, 15]	256,064
ReLU-4	[-1, 64, 15, 15, 15]	0
MaxPooling	[-1, 64, 1, 1, 1]	0
Linear-6	[-1, 10]	650
Total params		260,746
Trainable params		260,746
Non-trainable params		0
Input size		0.82MB
Params size (MB)		0.99MB
Forward/backward pass size		16.48MB
Estimated Total Size		18.30MB

for performance evaluation and hyperparameter tuning. This approach ensures a fair assessment of the model's generalization ability and helps prevent overfitting.

To ensure efficient training, we utilized the available hardware resources by setting the device for training. An 85 GB Google Colab GPU (Graphics Processing Unit) was used.

For training both, the loss function was set as the CrossEntropyLoss. Cross entropy loss is commonly used in multi-class classification tasks, such as image recognition, to measure the difference between predicted class probabilities and the true class labels.

To optimize the parameters of both models, Adam optimizer was chosen. Adam combines the benefits of two popular optimization methods, adaptive gradient descent and RMSprop. It adapts the learning rate dynamically during training, enabling faster convergence and better generalization.

The learning rate for the Adam optimizer was set to 0.001, which determines the step size for parameter updates during optimization. This value is suggested by empirical and is often used for similar applications.

For Model 1, the training process was conducted over 150 epochs throughout the training, both the loss and accuracy were monitored to evaluate the model's performance.

A convergence behaviour was observed in the end of the training, as the training accuracy already reached the 100% mark and the validation accuracy depicts a convergence trend around 94.3%.

The performance of Model 2 was measured through 100 epochs of its training. Again, it is perceived full convergence of the training accuracy with 100 %, whereas a convergence trend was observed around 92.3%

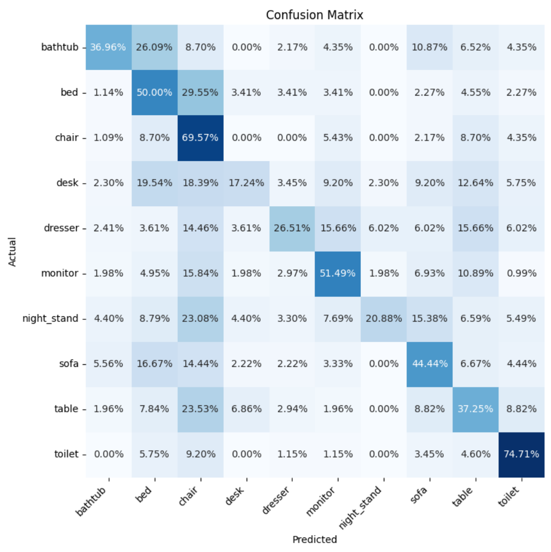
At first glance, these were promising results, since 94.3% and 92.3% would pose both models among two of the highest marks on the ModelNet10 qualification benchmark [5]. The high accuracy rate indicates that no over-fitting occurred, despite the 100% training accuracy. Both models perform well on the classification of this particular Dataset (split training set 80/20 % of ModelNet10 Dataset)

4 Analyzing Robustness

To ensure the power of generalization of the models and their capacity of performing well given different datasets, the suggested test set of ModelNet10 was classified.

There was a significant and very similar drop in accuracy for both models, for Model 1, the accuracy went down to 43.37% and for Model 2, 42.21%. Both results dropped similarly on the different test sets. Even if Model 1 is much simpler, the difference in accuracy due to deepening the network and making it more complex (increasing the number of parameters), in this case, is around 1%.

A confusion matrix, that displays the mismatch map of the samples for all the classes was computed. The confusion matrix for Model 1, is displayed. The 3 best classified objects were chairs (69.57%) and toilets (74.71%). The worst classified objects were desks (17.24 %), and night-stands (20.80%). The good performance in classifying chairs is likely due to the large amount of provided data belonging to this class (889/3991). The same logic applies to the opposite case for bathtubs (106/3991), because only a few bathtub data was provided to the model (106/3991), it performed poorly on their classification. Some classes never get mistaken by others, and that could give a glance on the features the model is learning.



	bathtub	bed	chair	desk	dresser	monitor	night_stand	sofa	table	toilet
bathtub	36.96%	26.09%	8.70%	0.00%	2.17%	4.35%	0.00%	10.87%	6.52%	4.35%
bed	1.14%	50.00%	29.55%	3.41%	3.41%	3.41%	0.00%	2.27%	4.55%	2.27%
chair	1.09%	8.70%	69.57%	0.00%	0.00%	5.43%	0.00%	2.17%	8.70%	4.35%
desk	2.30%	19.54%	18.39%	17.24%	3.45%	9.20%	2.30%	9.20%	12.64%	5.75%
dresser	2.41%	3.61%	14.46%	3.61%	26.51%	15.66%	6.02%	6.02%	15.66%	6.02%
monitor	1.98%	4.95%	15.84%	1.98%	2.97%	51.49%	1.98%	6.93%	10.89%	0.99%
night_stand	4.40%	8.79%	23.08%	4.40%	3.30%	7.69%	20.88%	15.38%	6.59%	5.49%
sofa	5.56%	16.67%	14.44%	2.22%	2.22%	3.33%	0.00%	44.44%	6.67%	4.44%
table	1.96%	7.84%	23.53%	6.86%	2.94%	1.96%	0.00%	8.82%	37.25%	8.82%
toilet	0.00%	5.75%	9.20%	0.00%	1.15%	1.15%	0.00%	3.45%	4.60%	74.71%

Figure 8: Confusion Matrix for Model 1

To further analyse robustness, the ModelNet10 suggested test set was modified by applying horizontal permutations on the samples, by exchanging the order of the coordinates axes (z vertical ax excluded) and their orientation. So, for example, a chair could be found in four different positions, each 90 degrees twisted from the other. This transformation on the test set did not display a significant change on the classification of the samples, apart from slight random shifts around the before mentioned values for accuracy. That fact indicates that the model is learning a function invariant to horizontal permutations (shifts on 90 degrees). So, this kind of data augmentation probably would not have any effect if applied to the training set, as it was once thought by the author. In fact, the results were not enhanced by feeding Model 1 with randomly horizontally permuted data.

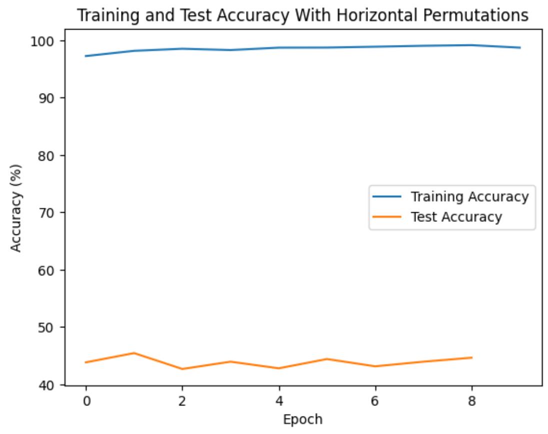


Figure 9: Training and Test Accuracy for model 1 trained with horizontal permutations

5 A Glance At Resolution

A dataset with higher resolution was also generated (200,200,200) along with a new instance of the same archictecture of Model 2. It was split in the same trend as before (80/20 %). A few epochs were enough to perceive the poor results on the validation set (oscilating around 50%), probably indicating an underfitting situation. On test set, it classifies most of the samples as chairs, with an overall accuracy around 15. It is likely that the model loses track of the biggest features, whereas pooling the finer ones. A more complex architecture should be used for this dataset, resulting in an extremely computationally demanding task. In fact, the instance with architecture of Model 2 (simpler than Model 1) took a considerable long time, and an early-stopping before convergence had to be done to retrieve the parameters of the model. In this context, the choice for 60x60x60 samples was proved to be wiser.

6 Conclusion

In Spite of the excellent performance on the training set (train and validation), both models proved a significant drop in accuracy, and demonstrated to be weaker than the first inspection, being model 1, more complex, only slightly better. The confusion matrix showed some evidence that this difference is due to the unbalance is samples distribution in training and data set. Inter-class similarity and intra-class variance issues can be potentially mitigated with additional data, specially on the most misleading classes. An alternative to that would be a suitable kind of Data augmentation for the context of voxel grids. The most obvious data augmentation, adding random noise to point clouds, may not have a significant effect because the final result will be very similar to the current one after the broadcasting phase. Rotating the data horizontally by 90 degrees showed to be a non-effective transformation, A proposed augmentation (not implemented) would be to stretch some of the coordinates of the point clouds, in a way that it deforms an object and make it look almost like a new one. This alternative, that applies a multiplication to each single coordinate of each single point, was also proved and abandoned, as it seemed too computationally and time consuming,

7 Bibliography

References

[1] Hang Su, Subhransu Maji, Evangelos Kalogerakis, Erik Learned-Miller. *Multi-view Convolutional Neural Networks for 3D Shape Recognition.* In Proceedings of the IEEE International Conference on Computer Vision (ICCV), (2015).

[2] Saifullahi Aminu Bello, Shangshu Yu, Cheng Wang. *Review: Deep Learning on 3D Point Clouds.*

[3] Nima Sedaghat, Mohammadreza Zolfaghari, Ehsan Amiri, Thomas Brox. *Orientation-boosted Voxel Nets for 3D Object Recognition.*

[4] ModelNet. *ModelNet - A 3D Model Repository.* Available online: https://modelnet.cs.princeton.edu/

[5] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. *3D ShapeNets: A Deep Representation for Volumetric Shapes.* In Proceedings

of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (2015).

[6] Dask Development Team. *Dask: Parallel Computing with Task Scheduling*. Available online: https://dask.org/ (Accessed: Month Day, Year).

[7] Wolfram MathWorld. Barycentric Coordinates. Retrieved from https://mathworld.wolfram.com/BarycentricCoordinates.htmlhttps://mathworld.wolfram.com/BarycentricCoordinates.h