Color3DNet: Neural Network-Based Vehicle Classification Using Commercial Off-The-Shelf LiDAR and Camera Sensors and Machine Learning

by

Sebastian Landl

Student ID: 01524483

Submitted to the Department of Computer Science

University of Salzburg

in Partial Fulfillment of the Requirements

for the Degree of Diplom-Ingenieur

Supervisor: Assoc.-Prof. Dipl.-Ing. Dr. Stefan Resmerita

October 2021

Statement of Authentication

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Salzburg, August, 2021 _____

Sebastian Landl

Acknowledgements

With the completion of this project the end of my Master's programme is in sight. I want to take this moment to thank several people who helped me along my way to get to this point.

First, I want to thank my parents *Ulrike* and *Thomas* for supporting me. Without them my studies would not have been possible.

Furthermore, I want to thank *Professor Christoph Kirsch*, the supervisor of my Bachelor's thesis, and *Professor Stefan Resmerita*, the supervisor of this thesis for their excellent support and guidance.

Next I want to thank the eMundo GmbH and especially *Wolfgang Brauneis* for supporting this project and helping us complete it. Also a big thanks for the great internships at eMundo, where I really learned a lot. I am looking forward to tackling many more exciting projects together!

Finally, I want to thank my dear friend and colleague *Christian Edelmayer*. I feel highly privileged to have had the honor of meeting you during my studies. I will never forget the challenges we faced and overcame, our great cooperation and all the good times we had together and I hope there is a lot more to come!

Thank you!

Abstract

LiDAR (**Li**ght **D**etection **A**nd **R**anging) is a distance measuring technology. This thesis investigates, whether it is viable to classify a 3D car scan made by an off-the-shelf LiDAR and camera sensor with more detail, such as identifying the car make or even the car model. To this, end custom preprocessing and augmentation methods, tailored to this kind of data, as well as already available filters for 3D data were used to prepare the data for training and classification by our neural network, the *Color3DNet*. It is based on the PointNet, which can classify point clouds, and extends it such that it can use color information about each point as well. The Color3DNet is able to learn car makes and even models, when using color values as well as the 3D data. We also investigated the importance of the color values. Our findings show that using RGB color information, even if the color is normalized, yields vastly superior performance compared to greyscale, black and white or no color information at all.

Contents

Chapter 1

Introduction

LiDAR (**Li**ght **D**etection **A**nd **R**anging) is a distance measuring technology. One of its many applications is in autonomous driving, where it is used to recognize the surroundings of an autonomous vehicle and possible obstacles [37] [23]. This thesis evaluates to what extent the LiDAR sensor of an off-the-shelf consumer device, the iPhone 12 Pro, can be used to classify scans on a more fine grained level, namely recognize car makes or models. It is part of a larger project done in collaboration with my colleague Christian Edelmayer, who wrote an accompanying *Data Management and Infrastructure* thesis [10]. Furthermore, this project is a cooperation with the eMundo GmbH Salzburg[1].

The 360 degree car scans themselves are stored as point clouds, which are a set of points that represent the surface of the scanned car. Each 3D point is also augmented by color information from the phone camera..

The classification is performed with machine learning techniques, more specifically neural networks. Therefore, this thesis has a chapter on machine learning background, including the necessary terminology and concepts.

The topic of classifying 3D data is not a new idea. There already exist many approaches, some of them using different types of convolution on different representations of 3D data [14] [27] [42] [21] [22]. The network developed for this thesis, the *Color3DNet*, is based on the *PointNet* [33], which works on point clouds. The PointNet was extended, such that the network can also work with color values.

[1] `https://www.e-mundo.de/de/subsidiaries/salzburg`

Another important part of this project is the data preprocessing and augmentation. Several simple methods were developed for the data we are working with. Additionally some methods from the *point cloud library* [38] were used. The preprocessing focuses on extracting only the car from a scan and therefore making it easier for the network to focus on the important features. In terms of augmentation, we use geometric as well as color augmentations. The latter allow us to pinpoint how important the color is for the network to learn the data.

The experiments performed show that 3D LiDAR scans with color information can be learned by the Color3DNet, when using color values. As the color is reduced, going from the original color to normalized color to greyscale to black and white, the performance reduces. Notably the performance is still pretty good with normalized values, but falls drastically when going to greyscale. Car models can be classified as well, even though not as well as car makes and the training takes longer. Generalization to never before seen car models is currently not possible, as the network tends to overfit in that case. We tried many classical methods to combat overfitting, however, one measure we could not try was to get more data. This may help, as currently the data set comprises of just under 400 scans. Another measure would be to increase the batch size, the batch size of 20 we used is very low, however, due to hardware limitations, we could not increase that number by a meaningful amount.

In this thesis first the LiDAR technology will be briefly introduced, along with an introduction to different ways of storing 3D data. A short machine learning background will be provided as well. Furthermore, existing work on both vehicle classification and specifically classification of 3D data will be presented. The next chapters will then detail the preprocessing and augmentation methods that were used, as well as the pipeline used to prepare the data. Then the PointNet and Color3DNet will both be described in detail and the design choices for the latter will be discussed. Finally, experiments demonstrating the viability of using LiDAR scans for vehicle recognition using the Color3DNet will be presented and conclusions will be drawn.

Chapter 2

LiDAR Data

2.1 What Is LiDAR?

LiDAR (**Li**ght **D**etection **A**nd **R**anging) is a technology that measures one or multiple distances. It works by emitting rays of light and measuring the time it takes for an emitted light ray to return by being reflected off of something. This measurement technique is called time-of-flight (TOF), as the time is measured the light takes until it returns to the sensor. The distance can then be calculated as the speed of light in air is known, which yields a 3D point. Ultimately, the result of such a measurement is a *point cloud*, which is a collection of points in 3D space [37].

Use cases for LiDAR range from any sort of distance measurements, mapping areas, measuring atmospheric aerosols to uses in autonomous driving, where it complements existing perception systems like cameras or RADAR [37] [23].

2.2 What Data Is Used?

The data used in this project are 360 degree 3D scans of cars that were collected with the iPhone 12 Pro, which is equipped with a LiDAR sensor, using a custom app we developed [4]. This enables any consumer to go out and buy a mobile device that is (relatively) affordable and combines a LiDAR sensor with a user interface, in this case the touch screen. Industrial LiDAR scanners on the other hand may not be designed to just be used by any consumer and they may be a lot more expensive.

2.3 How to Store Representations of LiDAR Data

The 3D data created by a LiDAR sensor can be stored and represented in different ways, each with its own advantages and drawbacks.

2.3.1 Point Cloud

The most obvious way to store the 3D points from a LiDAR scan is a *point cloud* (see Figure 2.1). It is simply a set of 3D points, which implies that they are unordered [33]. The upside is that this way of storing the collected data is very close to the raw sensor data, as a point cloud contains the 3D points collected by the LiDAR sensor. This is one of the reasons this data format was chosen for this thesis [37]. A downside is that the data can get huge. When two points are very close to one another both are stored even though one of the points might not add that much information. The raw point clouds that were collected for this thesis contain millions of points. The bottleneck is mostly reading the point cloud from disk, which can take a while with these amounts of data. This was fixed by preprocessing, which will be explained in more detail in chapter 5.

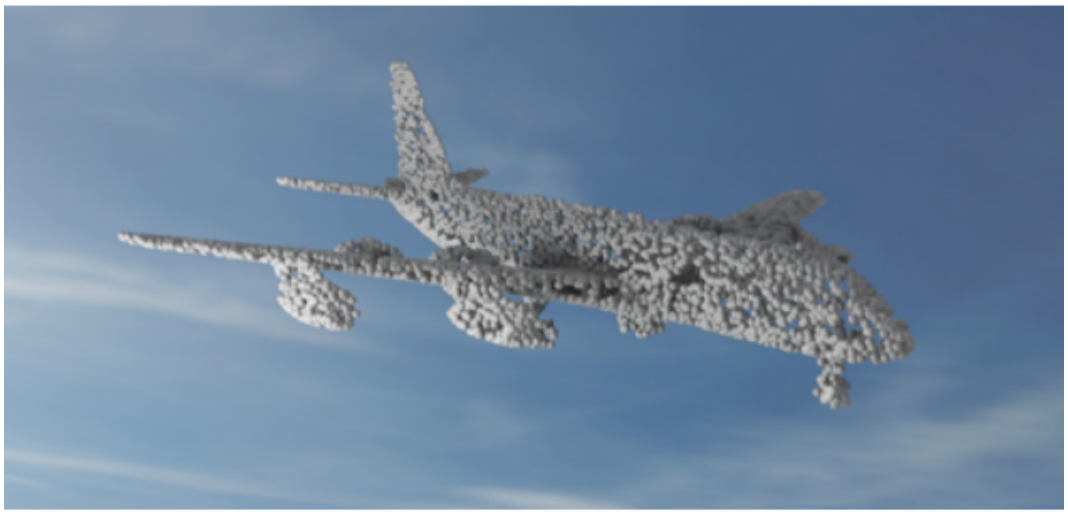


Figure 2.1: A plane represented by a point cloud (taken from [17]).

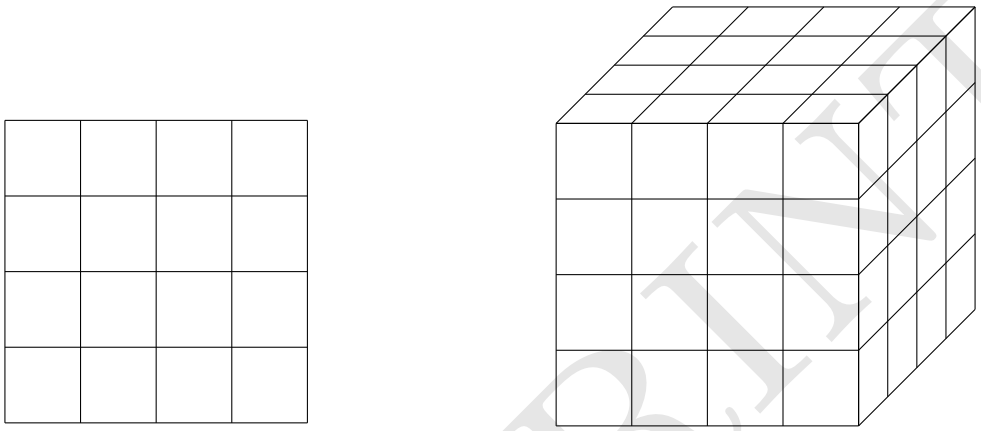
2.3.2 Mesh

Another way to store the 3D data gathered by a LiDAR sensor is a *mesh*. A mesh is a collection of vertices and faces, which are inferred from the 3D points (see Figure 2.2) collected by the LiDAR sensor. In simple terms a grid of points is fitted around the 3D points and the faces are inserted between the edges of that grid. If the resolution of that grid is low and we have fewer and larger faces we can store the data more efficiently. If the resolution is higher and we have many small faces, we have to store more vertices and faces, but are able to capture more detail. Another property of a LiDAR scan stored as a mesh is that a form of smoothing of surfaces is already applied, as the faces are inferred from the 3D points [36] [3].

Figure 2.2: A plane represented by a mesh (taken from [17]).

2.3.3 Voxel Grid

The final approach mentioned here is the *voxel grid*. First we need to establish what a *voxel* is. Simply put a voxel is a 3-dimensional pixel. An image is formed by pixels in the x and y axis. Voxels extend this concept by the z axis, which allows for the depiction of volumes in a voxel grid (see Figure 2.3). Voxels can be either filled or empty; filled means the voxel lies within the represented 3D shape, empty means it is outside (see Figure 2.4) [47].



pixel plane voxel grid

Figure 2.3: A pixel plane compared to a voxel grid.

A voxel grid only allows for a fixed resolution, and a LiDAR scan depicted by it may actually have a higher resolution, which results in loss of detail.

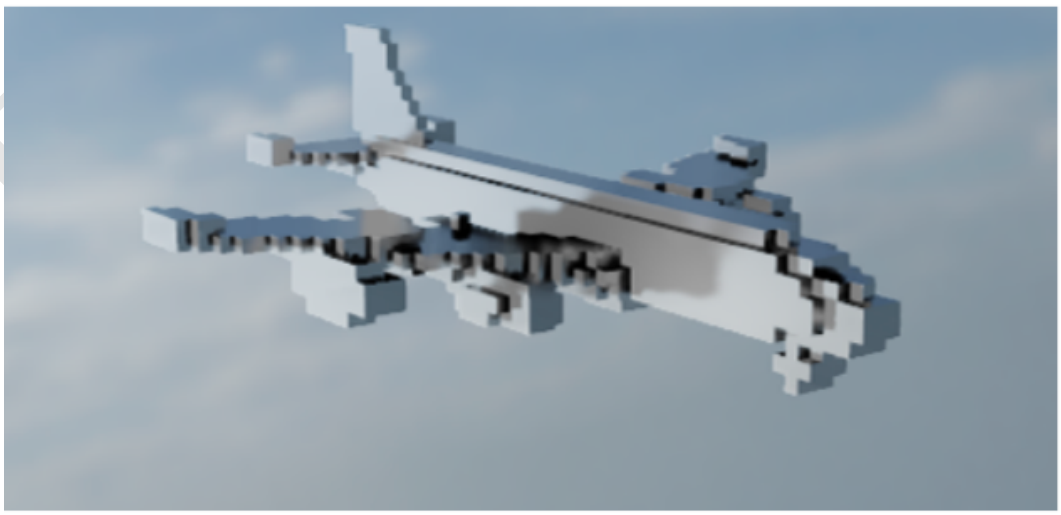


Figure 2.4: A plane represented by a voxel grid (taken from [17]).

Chapter 3

Machine Learning Background

As this thesis is about machine learning, some background knowledge is provided in this chapter. The concepts explained here were chosen, as they are relevant when explaining the neural network in chapter 7.

3.1 Supervised Learning

Supervised learning is an approach to learn how to predict a random variable $Y \in \mathcal{Y}$ given an input $X \in \mathcal{X}$. For example, if we are given a 3D scan of a car, which comprises 10 000 three-dimensional points, and want to predict the make of the scanned car, $\mathcal{X}$ would contain the car scans, each of dimension three by 10 000 and $\mathcal{Y}$ would contain the car makes we want to predict. The challenge now is to find a function called a *predictor*:

$$\begin{aligned} f : \mathcal{X} &\to \mathcal{Y} \\ X &\to f(X) = Y \end{aligned}$$

This function assigns the correct label $Y \in \mathcal{Y}$ to each input $X \in \mathcal{X}$. It is a function from the set of all functions $\mathcal{F} = f : \mathcal{X} \to \mathcal{Y}$. To find a function that is good at predicting the correct labels a performance metric has to be defined, namely the *loss* function, denoted by L. It depends on a given predictor f, an input X, and an output Y and it allows us to compare the performance of different predictors. Given two predictors f and g, if $L(f, X, Y) < L(g, X, Y)$ then predictor f performs better than predictor g. Therefore, the loss function should be minimized, which is done during training, as will be explained in section 3.4 [8].

3.2 Multi-Layer Perceptron (mlp)

In order to explain what a *multi-layer perceptron* (mlp) is, the concept of a single artificial *neuron* must be explained first, as such neurons are the building blocks for multi-layer perceptrons. Simply put a neuron takes inputs and calculates an output according to the inputs. The inputs may be weighted differently, as may be the outputs that lead to other neurons (see Figure 3.1). These weights are the parameters that are tuned during training (which will be explained in section 3.4) in order to find a good predictor [11].

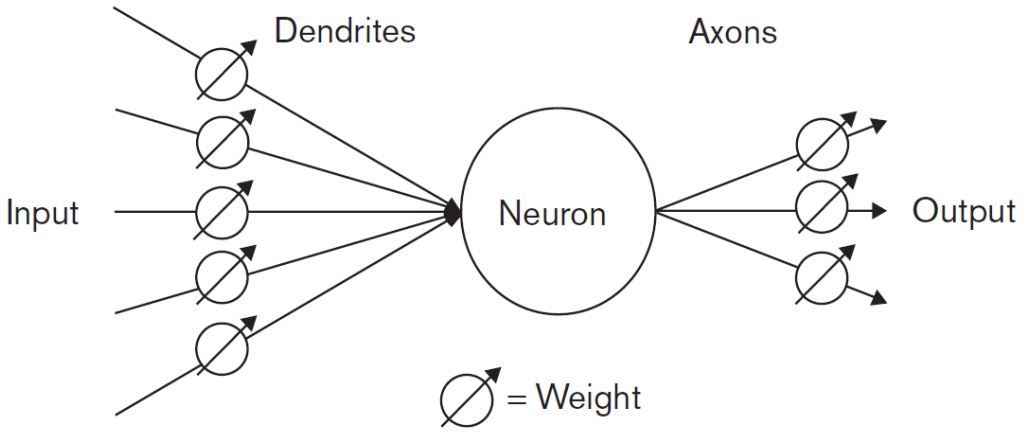


Figure 3.1: A single neuron (taken from [11]).

The calculation that happens in a neuron is the following:

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

with:

y...output of the neuron

ϕ...activation function

$\{w_1, ..., w_n\}$...weights of the inputs (tunable)

$\{x_1, ..., x_n\}$...inputs

b...bias (tunable)

The output y is calculated by first summing the weighted inputs $x_1...x_n$ multiplied with weights $w_1...w_n$. Then a *bias* b is added, which is tunable just like the weights of the connections between the neurons. Finally the calculated value is the input for the so called *activation function* ϕ, which ultimately determines the output of the neuron.

The choice of the activation function plays a crucial role, which will be explained in a bit [5] [24] [16].

These neurons can then be connected in several layers. There is always an input and an output layer and there may be hidden layers in between. The neurons of each layer are connected to all neurons of the layers before and after. Figure 3.2 illustrates this.

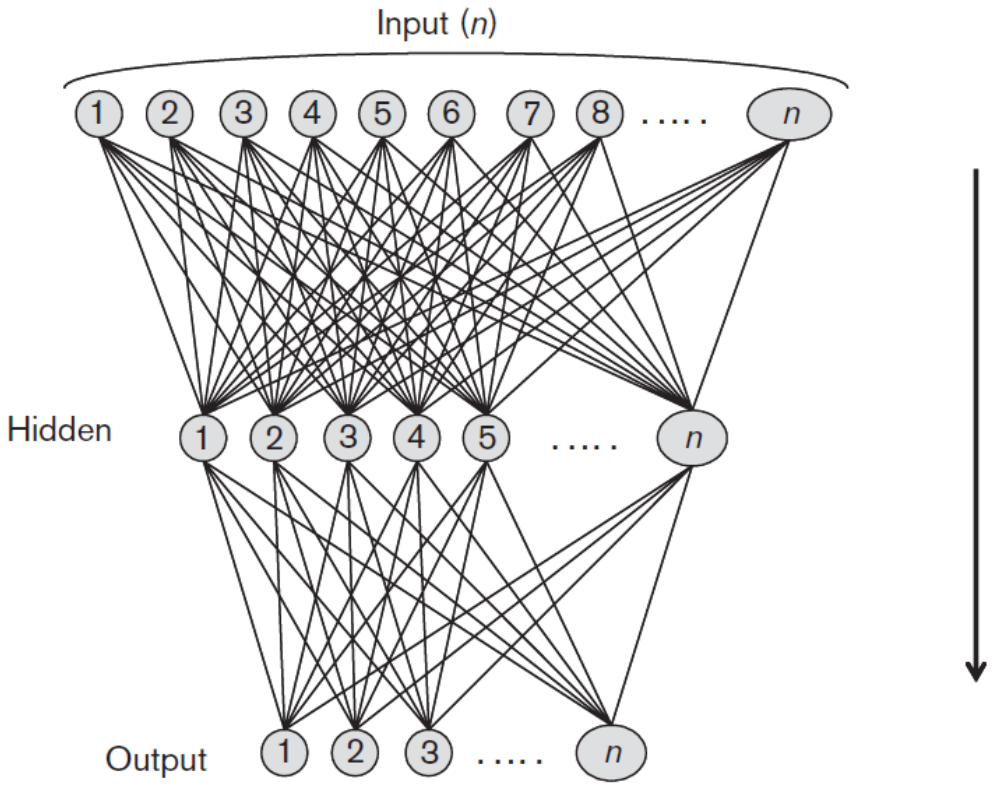


Figure 3.2: Three fully connected layers of neurons (taken from [11]).

Such a construction of input, output, and any number of hidden layers is called a multi-layer perceptron. The advantage of mlps is that by increasing the complexity, i.e. by adding more hidden layers, more complex data can be classified. Figure 3.3 illustrates this.

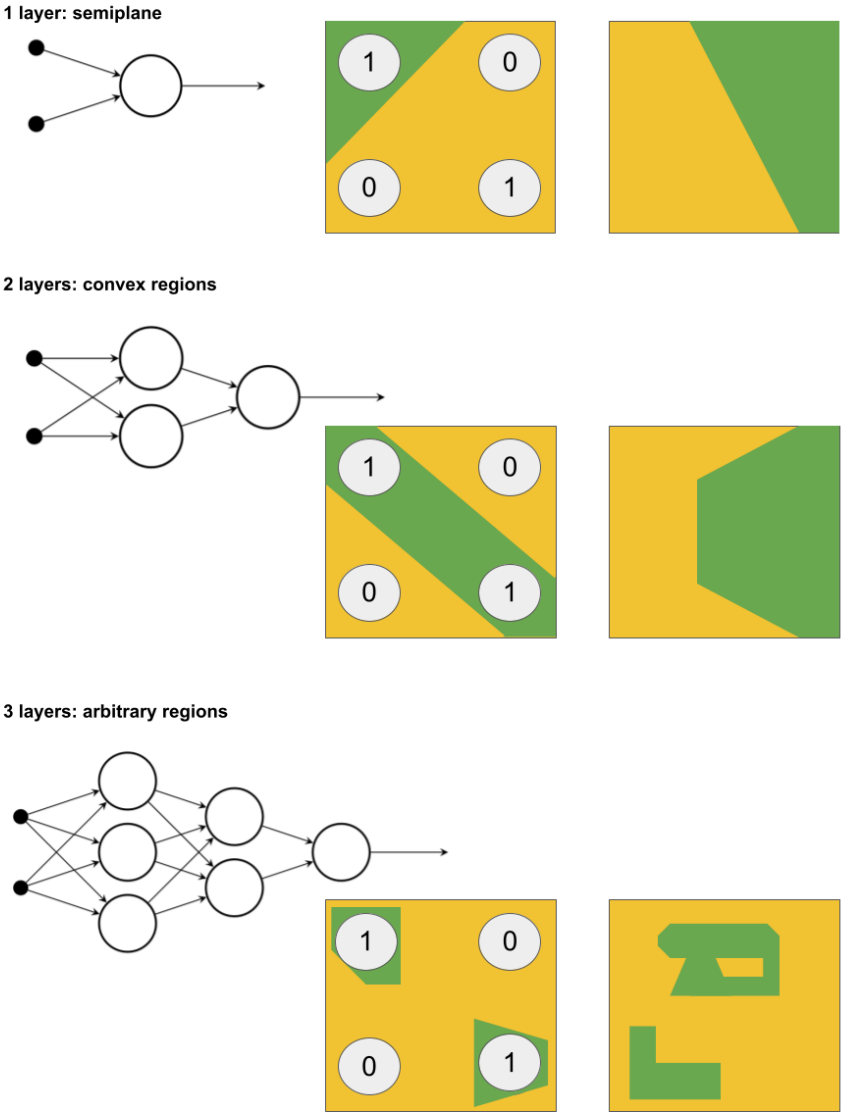


Figure 3.3: The more layers the mlp has, the more complex the learned regions can be [32].

The more layers are added, the more complex the regions that can be distinguished can become. In Figure 3.3 the data exists in a plane and each colored region represents a class learned by the model. Four data points are visible in the plane, each marked with their corresponding class (1 or 0). Through training the weights are adjusted such that the mlp learns the correct region for each class. This is also the point where the activation function becomes important. In order for the mlp to learn non-linear separations, the activation functions must also be non-linear, as a function of linear functions can also only be linear. Examples for non-linear functions used for this purpose are the *sigmoid* or the *Rectifier Linear Unit* (ReLu) functions (see Figures 3.4 and 3.5) [11] [35] [32] [43] [30].

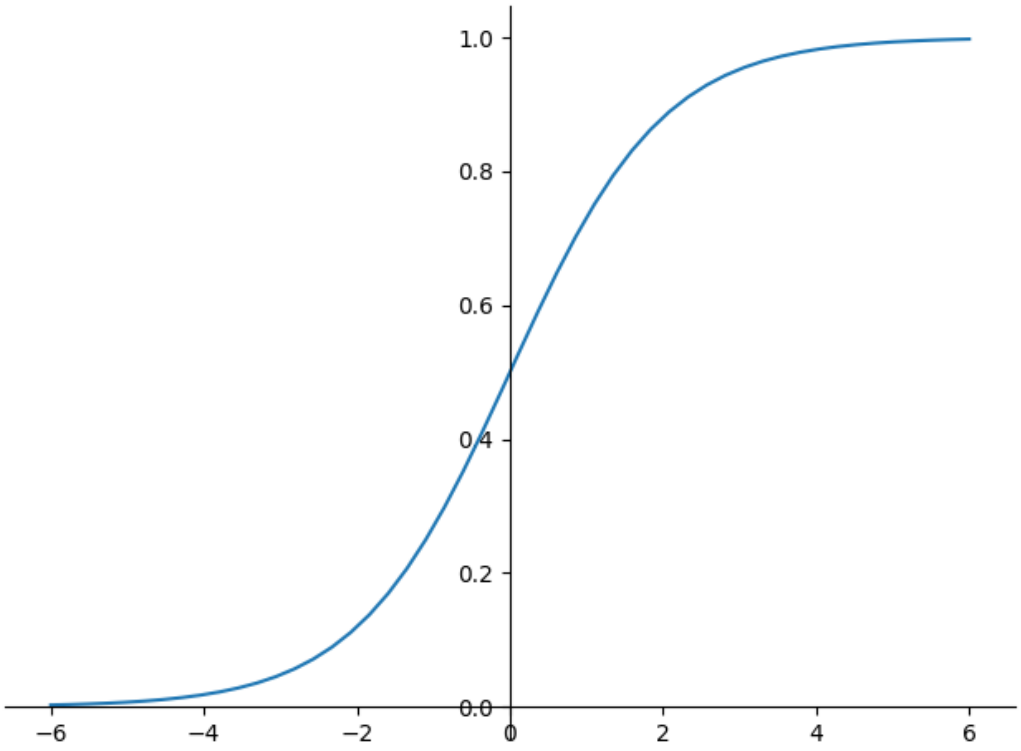


Figure 3.4: Sigmoid function: $sigmoid(x) = \frac{1}{1+e^{-1}}$

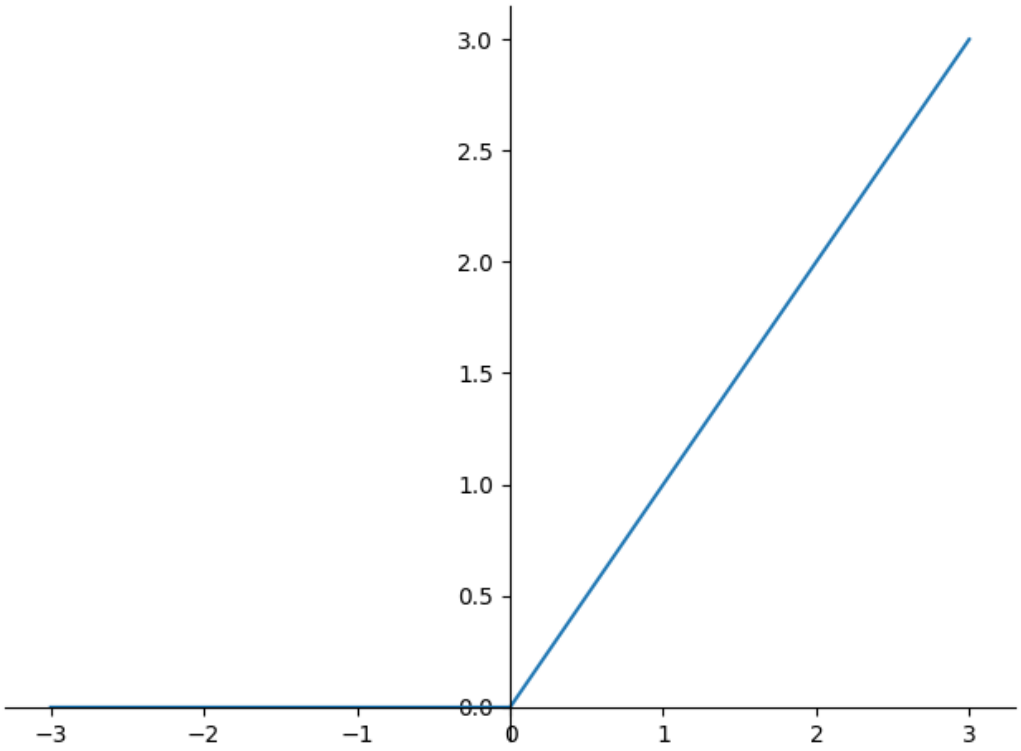


Figure 3.5: Rectifier Linear Unit (ReLu) function: $ReLu(x) = maximum(0, x)$

Regarding activation functions a special property is worth being noted, namely that usually in the output layer a different activation function than in the input layer or hidden layers is used, for instance the *softmax* function.

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

x_i is the input of a neuron with a softmax activation function. x_j are all the other neurons in the the same layer. It outputs a values from the range zero to one [43] [30].

3.3 Convolution

Convolution is a deep learning architecture that has been around for a long time and has been used successfully, especially in the field of image recognition [40] [9] [12]. In this section the basic concept of convolution as well as the convolution types relevant to this thesis will be explained.

3.3.1 2D Convolution

Convolution is very prominent in image recognition, where usually *2D convolution* is used. This type is explained first, as it is easy to illustrate and understand the basic concept, which makes understanding the other types easier.

A convolution layer is made up of multiple *filters*, which have a specific size, the *kernel size*. They are used to extract features and the values of these filters are adjusted during training. How convolution works is best explained with the help of Figure 3.6, which depicts a 2D convolution with a 4 x 4 input, a 3 x 3 convolution kernel/filter, and a 2 x 2 output. In this case the filter would have nine trainable parameters. During the actual convolution the filter is moved over the input and from the parameters of the filter (which are tunable during training) and the input values in the region the filter covers at the time, the output value is calculated. Then the filter is moved to the next position. Note that the illustration shows only one filter, while in practice usually multiple filters are used.

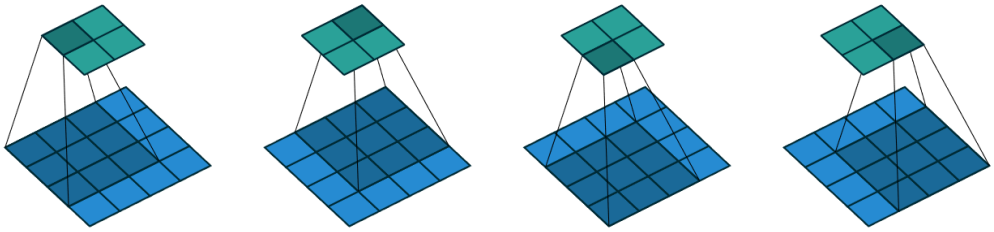


Figure 3.6: Illustration of a simple 2D convolution (taken from [9]).

Without going into too much detail the terms *stride* and *padding* will be mentioned shortly here as well. The stride is the value the filter is moved after every calculated output. In Figure 3.6 the stride is 1. Padding can be added to an input, which adds a border around it, as illustrated in Figure 3.7, where a padding of 2 is depicted. In the example before (Figure 3.6) the padding was 0 [9].

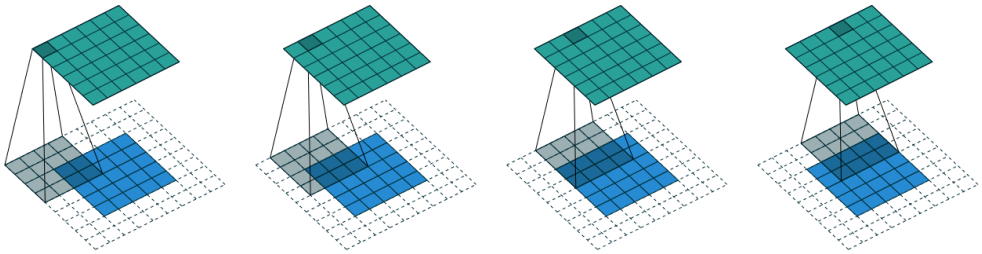


Figure 3.7: Illustration of a 2D convolution with a padding of 2 (taken from [9]).

3.3.2 3D Convolution

The *3D convolution* works similarly to 2D convolution, only that the input as well as the filters are three dimensional. The input is a 3D volume and the 3D filters are moved through it to calculate the 3D output. The concept of a 2D convolution is extended by one dimension. This convolution is used when working with 3D data, like voxel grids, and it can also be used to analyze video footage, where each frame is a 2D image and the third dimension is time [40] [12].

3.3.3 1D Convolution

The final type of convolution that will shortly be described here is *1D convolution*. It is again, similar to the other types of convolution discussed here already, except for the dimensions of the input and filters, which is 1. Thus any matrix operation is now an array operation, which not only allows this type of convolution to work on 1D data, like 1D signals from sensors (e.g. accelerometer) or audio signals, but also has a lower runtime than higher dimensional convolutions, as the operations are simpler [18] [2].

3.4 Learning Through Back Propagation

The actual learning process and tuning of the parameters (weights and biases), also called *training*, can be done with *back propagation*. A back propagation algorithm is half of the learning process. The other half is the forward algorithm, which is simply the input data propagating through the network. This is done by calculating the output of each neuron as described in section 3.2. Each neuron produces its output and sends it to the next layer until the output layer produces its outputs, which concludes the forward algorithm. After a forward propagation through the network the output of the network is compared with the desired output. According to that the parameters (weights, convolution filters) of the network are adjusted during the back propagation step. Once all training data has been propagated forward and backward trough the network, a so called *epoch* is concluded. Regarding when back propagation should be done, there are different approaches. One can perform this weight adjustment step after each example that has propagated through the network in the forward direction or only back propagate batch wise, meaning that after a fixed amount of examples (the *batch size*) have passed through the network in the forward direction, back propagation is performed [7] [26].

3.5 Overfitting

To explain what *overfitting* is, first the concept of training data and *test data* will be described shortly. Training data is the collection of examples used to train a network with back propagation. The test data is a collection containing different examples than the training data, which is used to evaluate the network and see how well it can classify new data it never saw before [28].

Overfitting describes the phenomenon that the training data can be classified very well by the neural network, however, the performance on the test data is very poor. In other words the model generalizes poorly, which is undesirable. This can happen, when a network has many parameters it can tune and can therefore learn the training set in such detail that is does not learn general features, but basically learns all examples in the training set by heart [48] [39] [49]. Figure 3.8 illustrates overfitting; as the epochs progress the performance on the training data increases, however, the performance on the test data does not.

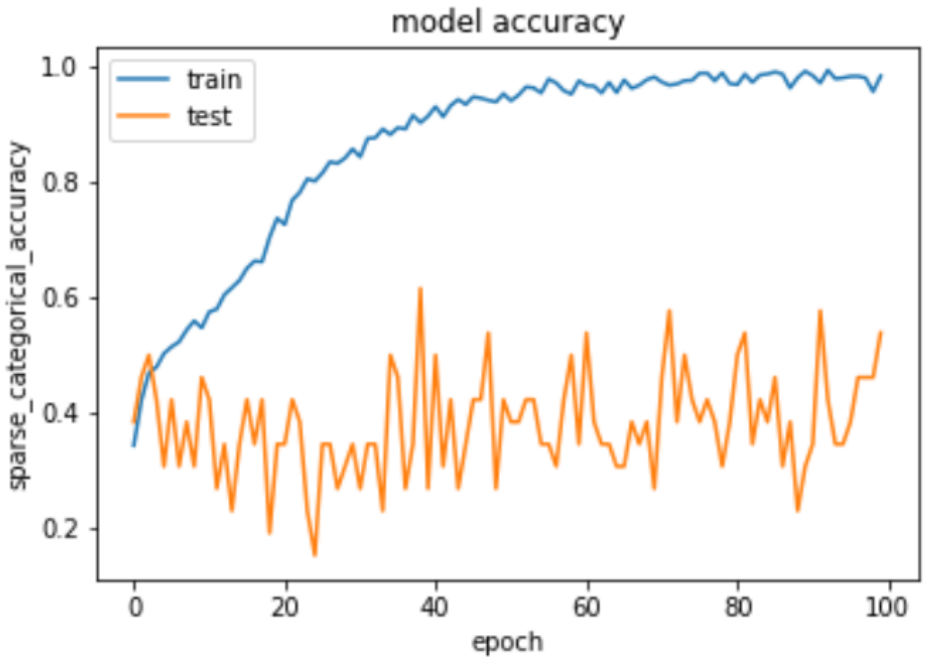


Figure 3.8: Illustration of overfitting. The training data is learned very well by the model, however, there is no generalization, meaning that the model performs poorly on the test data.

3.6 Batch Normalization

During training the weights of each layer may change after each back propagation step. This also means that the distribution of the signals passed to the next layer can change completely, which may make it difficult for a layer to learn good weights, if the distribution of its input data changes all the time. *Batch normalization* is a method of addressing this problem. It does so by normalizing the inputs of a layer for a given batch. The inputs are normalized such that the mean is zero and the variance is one. The calculations performed are the following:

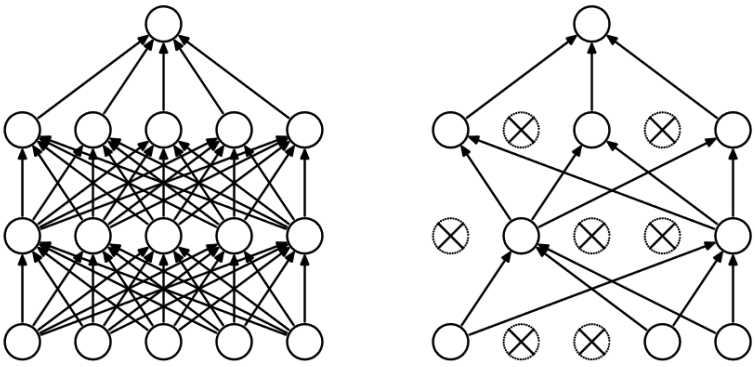
$$\text{Values } x \text{ over a mini-batch: } \mathcal{B} = \{x_1, ..., x_m\}$$
$$\text{Parameters to be learned: } \gamma, \beta$$
$$\text{Batch normalized results of the input values } x_i\text{: } \{y_i \equiv \mathrm{BN}_{\gamma,\beta}(x_i)\}$$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{mini-batch mean}$$
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{normalize}$$
$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{scale and shift}$$

Batch normalization allows for the use of higher learning rates, makes the performance of the model less dependent on the initialization of the weights, and has a regularizing effect, which can help to prevent overfitting [15] [48].

3.7 Dropout

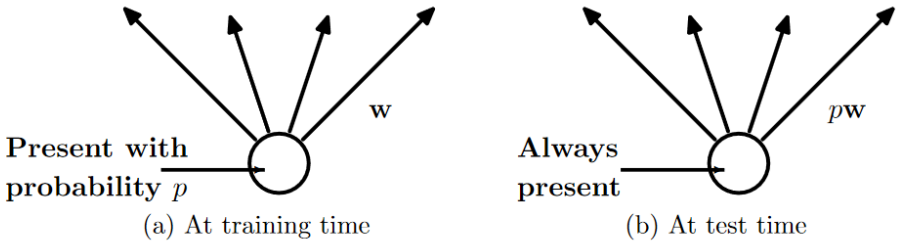
Dropout is a technique to prevent overfitting. During training a certain ratio, also known as the *dropout rate*, of randomly chosen neurons is switched off for each presented training example (see Figure 3.9).



(a) Standard Neural Net (b) After applying dropout.

Figure 3.9: Illustration of dropout. On the left a fully connected neural network is depicted. On the right the same network with dropout applied can be seen (taken from [41]).

This means that instead of the entire network being trained, many smaller networks which share their weights are being trained a bit. This prevents extensive co-dependencies of neurons on one another, which could lead to overfitting. During testing the entire network is used. Since during training all the neurons of the network never were active at the same time, the output of all the neurons of a layer combined would be too big during testing. Therefore the outputs are scaled down accordingly. If a neuron has a probability p of being turned on during training, the output of that neuron is multiplied by p in order to scale it accordingly (see Figure 3.10) [41].



(a) At training time (b) At test time

Figure 3.10: Illustration of the scaling of the neuron output during testing (taken from [41]).

Chapter 4

Existing Work

This chapter provides an introduction to related work. It is split into two parts; first an overview of existing work on vehicle classification and second an overview of techniques to classify 3D data.

4.1 Vehicle Classification

VMMR (**V**ehicle **M**ake and **M**odel **R**ecognition) using machine learning is a growing field with the main use case being vehicle identification. The ability to identify and also re-identify a vehicle allows for a detailed analysis of traffic flow, especially in large cities. Furthermore, vehicle identification can be used by law enforcement [29] [20] [25].

Vehicle recognition can be performed on different levels of detail. The most coarse-grained level is recognizing the vehicle *type* like for instance car, bus or truck. The next level of detail would be the *make*, which refers to the manufacturer of the car. Examples would be Audi, BMW or VW. The final level of detail is the *model*, which would be Audi A4, BMW i8 or VW Golf [25].

The are some special challenges that come with this kind of data. For instance the same car model can change shape over the years, which makes it difficult to recognize and basically introduces another level of detail, namely the year. Furthermore, vehicles even of different makes can look very similar. Such similarities can also exist between different models of the same make. Another big challenge are limited lighting conditions.

Low light, for instance at night, can make the features of the car less visible. All these factors can make accurate classification more difficult [45] [25] [6].

In general this task can also be done by a license plate recognition system, however, there are several downsides to such a system. Firstly, license plates can be ambiguous, for instance the letter "O" and the number "0" can look very similar. Moreover, a license plate can be difficult to recognize due to damage or be forged. To prevent fraudulent use of license plates a license plate recognition system can be complemented with a VMMR system, in order to check, whether the make and model of the car registered under a given license plate match the recognized make and model. Another advantage of VMMR over license plate recognition is that the camera angles required for the recognition are more flexible. For license plate recognition the actual license plate must be visible, which requires very specific camera placement. VMMR on the other hand can work from many camera angles [20] [45].

In terms of creating a neural network there are different approaches. Two common strategies for recognition are k-way classification and feature vector extraction. In the former approach a fixed number of classes (car types/makes/models) are chosen and the network is trained to recognize these classes. The latter approach works by having a feature vector as the output of the neural network and comparing that against a database of known, labeled feature vectors, which can be done with nearest neighbor search. The advantage of this strategy is that a new car model can always be added to the database without requiring the network to be retrained as it would be the case with the k-way classification strategy. The actual network can then be created either by hand-crafting the feature extractors or by using deep learning and letting the network create all feature extraction during the training process. The latter approach generally yields better results [29].

The actual recognition can be performed by extracting features like the shape, dimensions and textures of the vehicle. Furthermore, edges, contours and other high-level features can be extracted. Also local features, like the logo of the car make can be used to classify a vehicle. Modern neural networks can achieve very high accuracy, up to and over 99% [44] [6] [20].

Due to the success in VMMR recognition and the need for such systems, there exist several services offering exactly that. For instance eyedea[1] or CarNET[2], which claims an accuracy of 97%.

[1] `https://eyedea.ai/make-and-model-recognition/`
[2] `https://carnet.ai/`

The biggest difference between the approaches described in this section and the one presented in this thesis is the kind of data that was used, which is images as opposed to 3D LiDAR scans complemented by color values obtained from a camera sensor, which were used for this project. Furthermore, the dataset used here is much smaller. As for the machine learning approach, we perform k-way classification using deep learning.

4.2 3D Classification

Even though machine learning on 3D data is not as well researched as machine learning on images, there already exits a fair amount of research on that topic. This chapter serves to give an overview and examples for different approaches.

MeshCNN is an approach that works on meshes. It uses specialized mesh convolution and mesh pooling layers, which work on triangular meshes. This convolution works on an edge and the four additional edges of the two adjacent triangles. It tries to decide which edges to collapse in order to extract a useful feature [14].

The *3D ShapeNets* model works on 3D data represented by voxel grids. Due to the regular nature of this representation of 3D shapes it can use a Convolutional Deep Belief Network [19], taking advantage of convolutions which are traditionally used for image classification [47]. The *VoxNet* also works with voxel grids in order to use 3D convolutions on it. This type of 3D convolution is also used when working with video data, in which case time is the third dimension. In this use case this third dimension is simply the third spatial dimension. Furthermore, this model does not actually take voxel grids as input, but point clouds which are then transformed into voxel grids to use 3D convolution on them [27].

A completely different way of classifying 3D data is to transform it to 2D images which can already be classified very well. This is done by projecting the 3D shape onto an image, which is basically what happens when we view a digital 3D object or scene on a computer screen. Then the image is classified with a standard convolutional neural network. To boost the recognition performance even more, multiple views of the 3D shape can be used [42].

Point Clouds are a less structured representation of 3D shapes than for instance a voxel grid. To still use 3D convolution on a point cloud special types

of convolutions have been developed. One possibility is to use a spherical convolution kernel, where the center is one point and the spherical volume around it is split into discrete sections [21]. Another specialized convolution is the convolution on $\mathcal{X}$-transformed points, which uses a center point and its neighbors for convolution [22].

A special aspect of a point cloud is that it is an *unordered* set of points, which means that two point clouds can be identical even though the order of points a model sees can be different. The PointNet solves this by employing a transformation that takes all input points and outputs a single vector using symmetric operations such as addition or multiplication [33].

The network presented in this thesis, the Color3DNet, is based on the PointNet [33]. It is extended, such that it accepts a point cloud, where each point also has a RGB color value, meaning the input layer has size n x 6 instead of n x 3. Furthermore, the feature extraction pipeline used in the PointNet to extract the 3D features was duplicated, in order to have one extra feature extraction pipeline that processes the color values.

Chapter 5

Preprocessing

In order to work with the data collected by the LiDAR sensor preprocessing was a necessary step. The raw data comprises millions of 3D points, which is way too much as an input for our model. The preprocessing was also done to make the scans cleaner and enable the network to focus on the vehicle. This chapter describes what steps were taken with the focus on why they were taken. Details on how each step was implemented can be found in the complementary *Data Management and Infrastructure* thesis [10]. Furthermore, the code can be found on GitHub[1].

5.1 Custom Transformations

There are many ways to preprocess the data and try to make it easier to learn. Many implementations already exist, for instance in the *Point Cloud Library* (PCL) [38]. However, because the 3D data used here is quite specific, as every example is a 360 degree 3D scan of a car, some transformations that were implemented are custom tailored and may not work with other types of data.

[1] `https://github.com/ChrisEdel/vehicle-classification`

5.1.1 Offline Distance Filter

The *offline distance filter* removes points that are the furthest away from the centroid of the point cloud. As the name suggests it is an offline filter algorithm, meaning all points must be known upfront to perform this filtering step. This was done because when a car is parked close to another car, a wall, or another object, parts of that close object can end up on the scan. As the scanned car itself is in the center of the scan and comprises the most points of the scan, a removal of the outermost points is a good way of removing or at least reducing this type of artifact. See Figures 5.1 and 5.2 for a comparison of a scan with and without this filter applied. The data used in the experiments was preprocessed with a value of 5% for this filter.

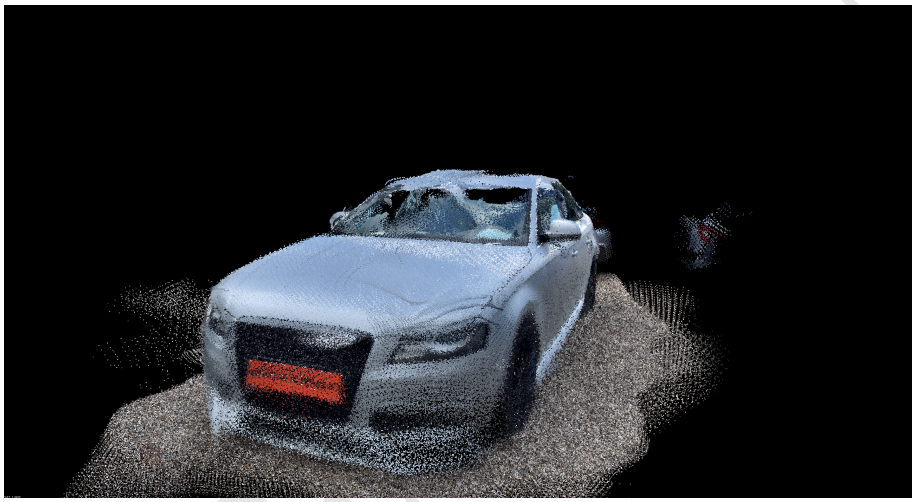


Figure 5.1: Without offline distance filter; on the right a part of another car is visible.

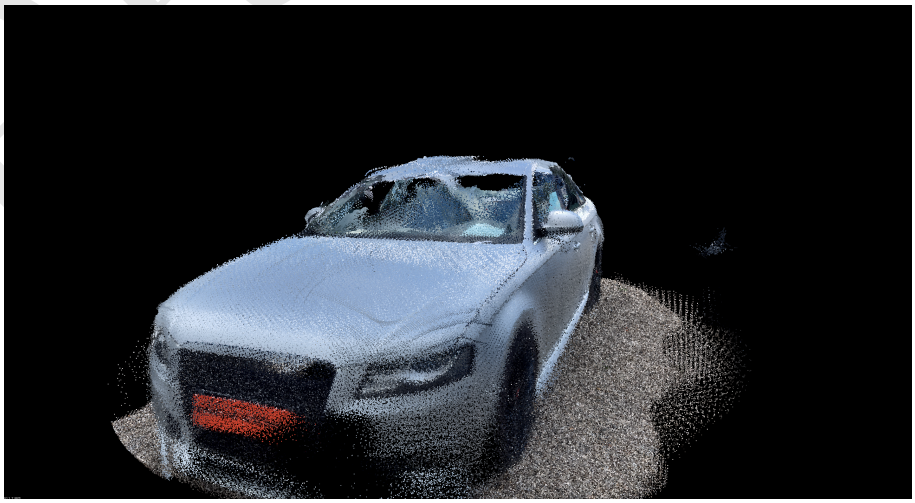


Figure 5.2: With offline distance filter (3.5%); the undesired part of the scan on the right has been reduced.

5.1.2 Floor Filter

The *floor filter* is another custom designed point removal technique. Similar to the artifacts the distance filter is designed to remove, this filter also removes an object close to the car which is the ground, see Figures 5.3 and 5.4. This filter works with a threshold, which is used to calculate a value along the height axis and every point below that value is removed. Values chosen for the particular scans used for this project are 0.3 and 0.275. The optimal values can vary depending on how the scans were made.

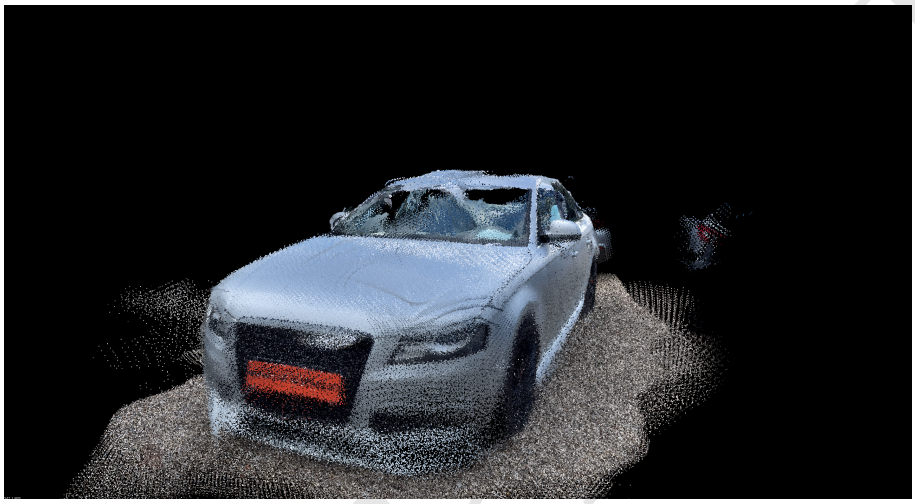


Figure 5.3: Without floor filter.

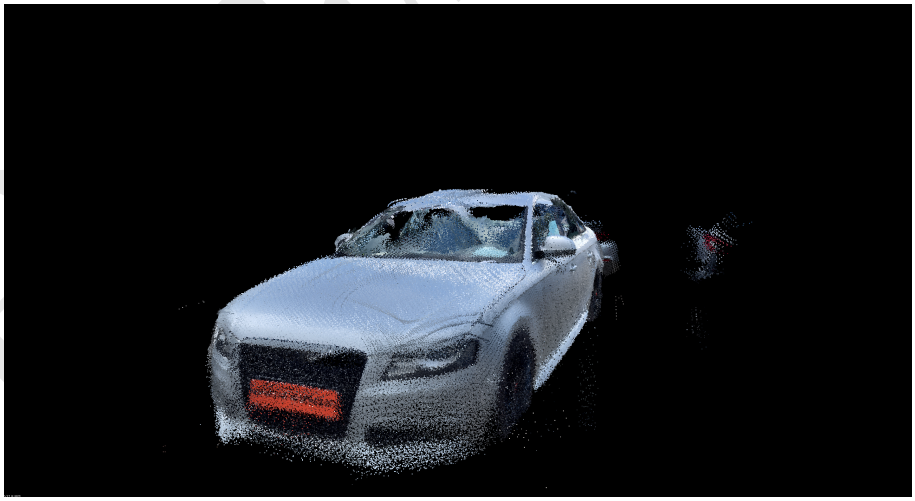


Figure 5.4: With floor filter (0.275).

5.1.3 Confidence Filter

The *confidence filter* is a filter that takes advantage of the confidence property associated with a 3D point measured by the sensorics of the iPhone 12 Pro (more details can be found in the complementary *Data Management and Infrastructure* thesis [10]). This property can be either 0, 1, or 2. The higher this value is, the more accurate the point. The filter simply eliminates all points that are below a certain confidence level, as illustrated in Figures 5.5 to 5.7. For training the network only points with confidence 2 were used.

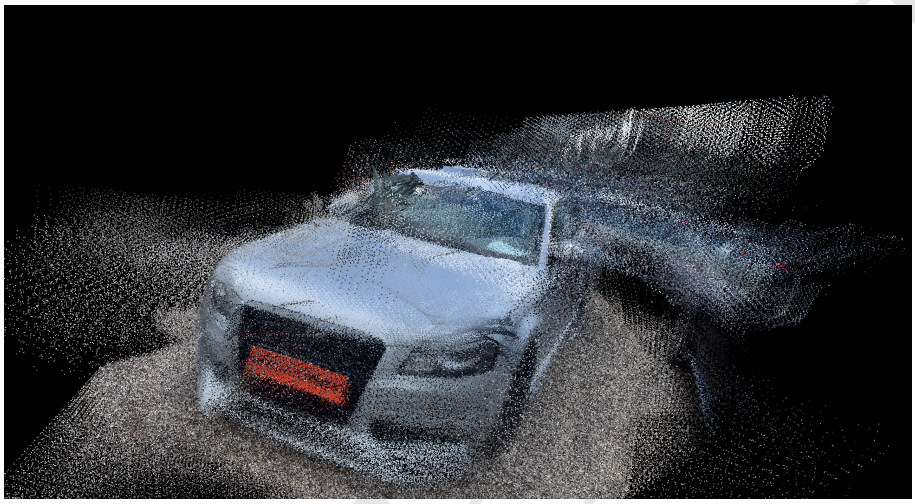


Figure 5.5: With confidence 0 (5 459 254 points).

Figure 5.6: With confidence 1 (4 649 717 points).

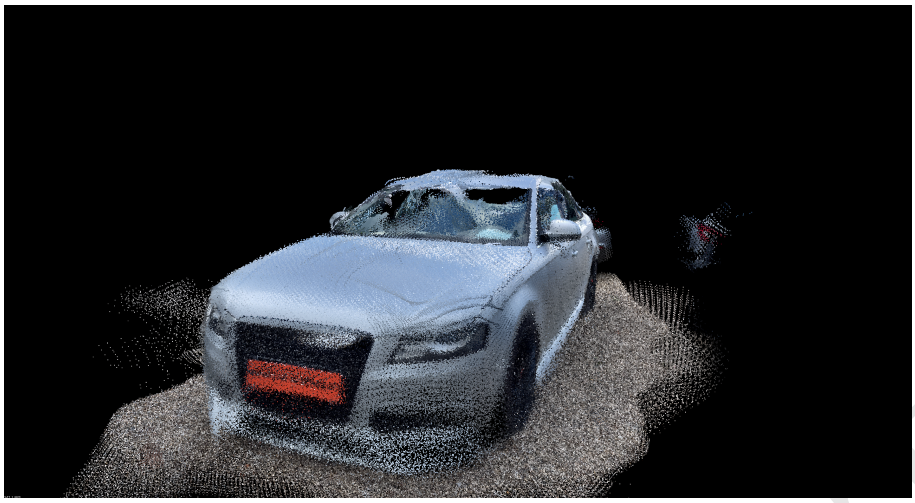


Figure 5.7: With confidence 2 (4 103 983 points).

5.1.4 Online Distance Filter

As the name suggests the *online distance filter* operates at the time of data acquisition, without ever needing the entire point cloud. At the time of scanning all points that are further than x metres away from the sensor are omitted (Figures 5.8 and 5.9). The value for x was usually 2 to 4.

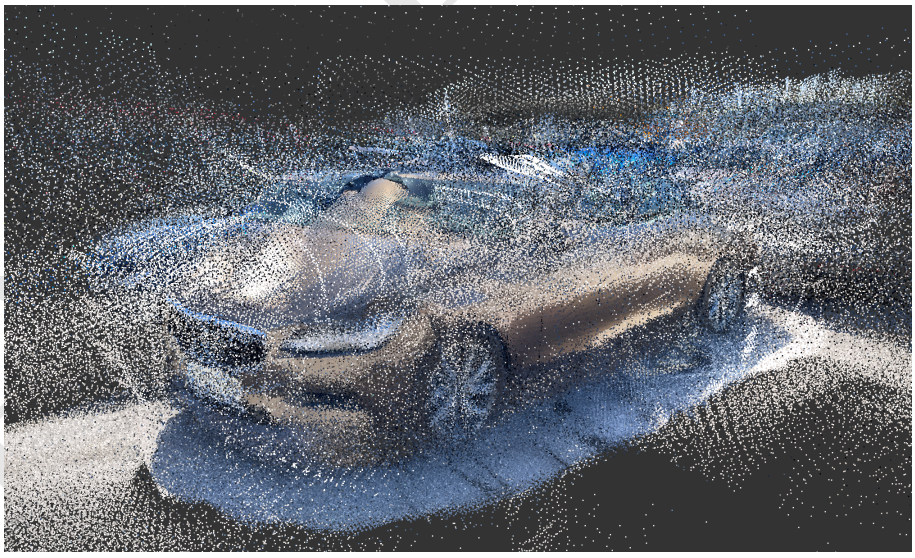


Figure 5.8: Without online distance filter (4 676 870 points).

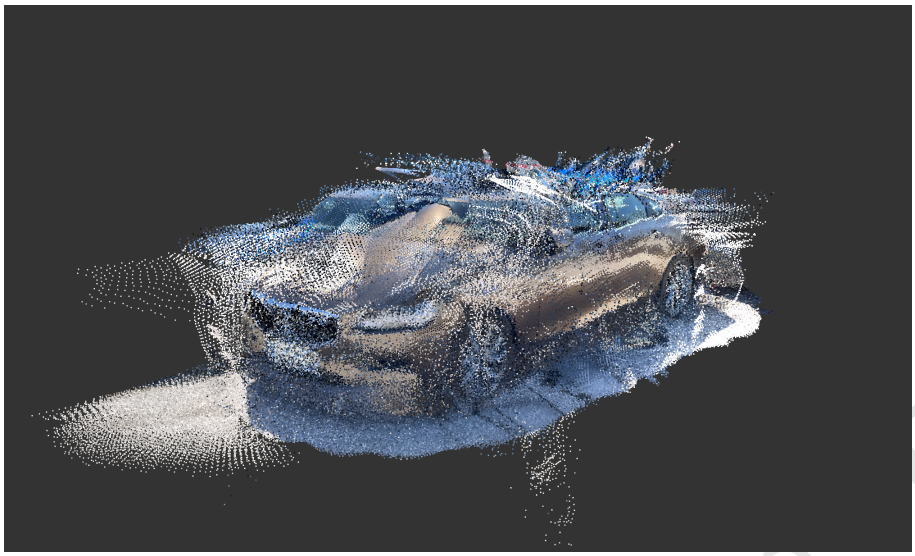


Figure 5.9: With online distance filter (4 028 255 points) (Confidence 0).

5.1.5 Subsampling

Even though *subsampling* is not called filter, like the techniques described before, it still removes points. However, as opposed to the other filters, the primary objective of this preprocessing step is not to remove artifacts and increase the focus on the scanned car, but to reduce the size of the point cloud. A point cloud orginally contains millions of points. If all points are used as an input for a neural network, huge amounts of memory would be necessary to accommodate all these points and the network would increase in size and complexity as well. To solve this, the subsampling step selects x points uniformly and randomly (Figures 5.10 to 5.12). The subsampling value used for the experiments later is 10 000.

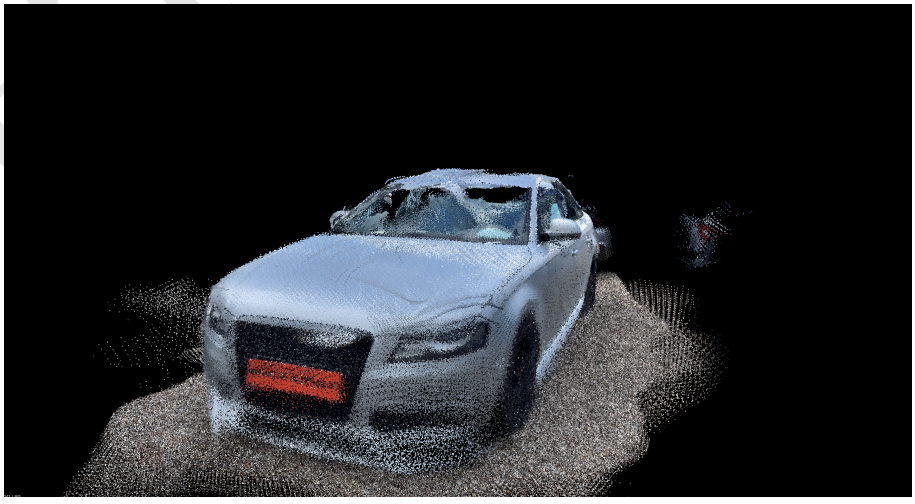


Figure 5.10: Without subsampling (4 103 983 points).

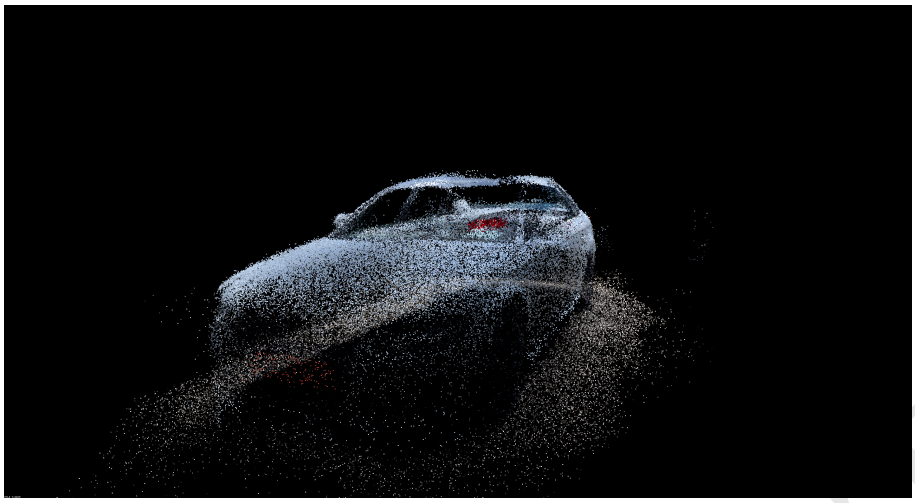


Figure 5.11: Subsampled to 100 000 points.

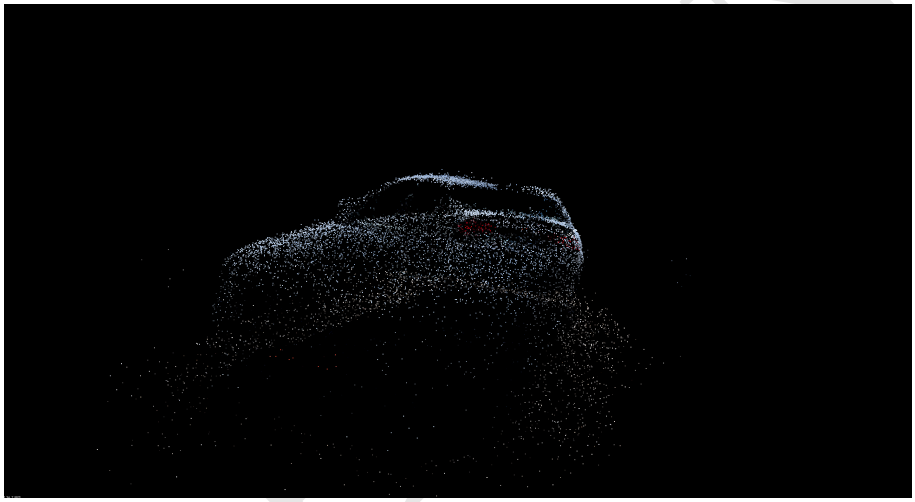


Figure 5.12: Subsampled to 10 000 points.

5.2 PCL Transformations

Apart from the custom preprocessing steps some functions offered by the PCL [38] were used, generally in order to make the point cloud smoother.

5.2.1 Voxel Grid Filter

The *voxel grid filter* aligns the point cloud by first transforming it to a voxel grid. The resulting point cloud comprises the centroids of all filled voxels, which results in a new point cloud that approximates the old one while being

aligned on a grid (Figures 5.13 and 5.14). This filter accepts three arguments, which are the resolutions along the x, y, and z axis. The arguments we experimented with were $0.015, 0.015, 0.015$. The goal of this preprocessing step is to make the point cloud smoother and it also has a subsampling effect. In the scans flat surfaces rarely appear flat, as the scan is not perfect. To mitigate that, a filter such as this one is used.

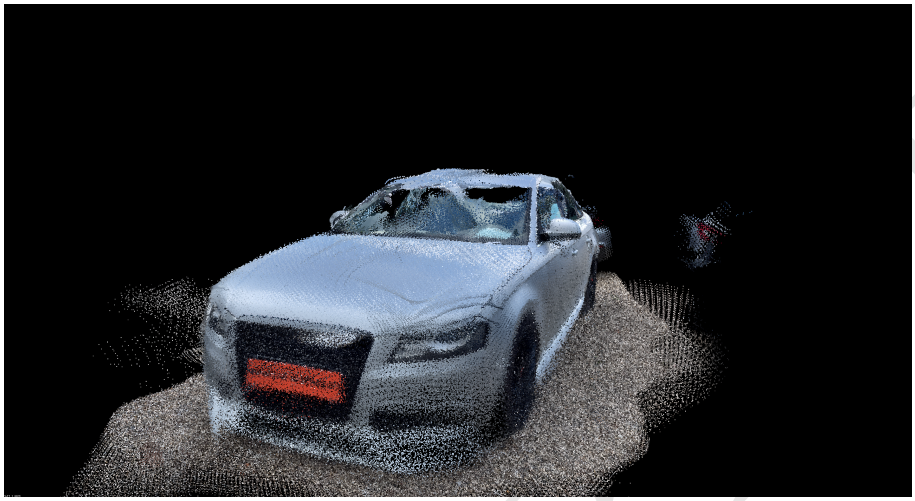


Figure 5.13: Without voxel grid filter.

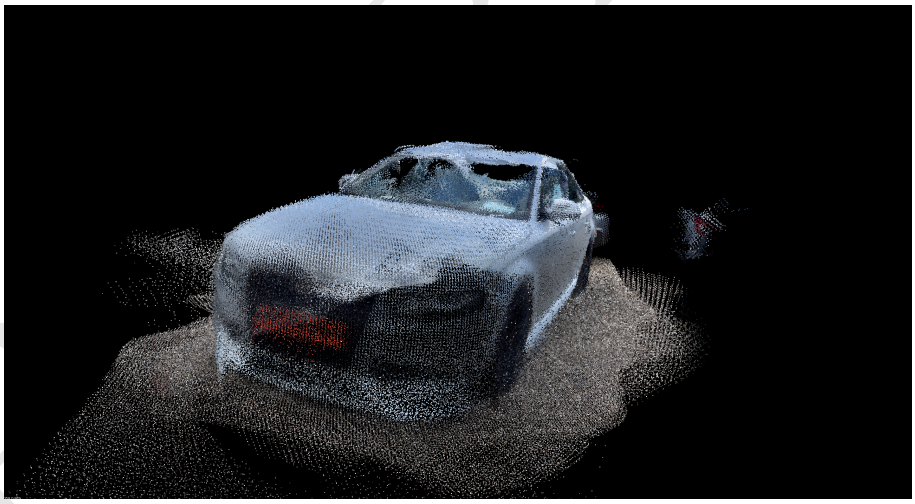


Figure 5.14: With voxel grid filter $(0.015, 0.015, 0.015)$.

5.2.2 Point Cloud Smoothing

Similar to the voxel grid filter, the *point cloud smoothing* serves to smoothen the scan and make flat surfaces that were not scanned perfectly appear flat (Figures 5.15 and 5.16). This filter takes one argument, which is a search radius. The smoothness of the resulting scan depends on the size of this search radius, where a bigger value corresponds to a smoother scan. However, a bigger value comes at the cost of longer runtimes. In the experiments the value 0.01 was used.

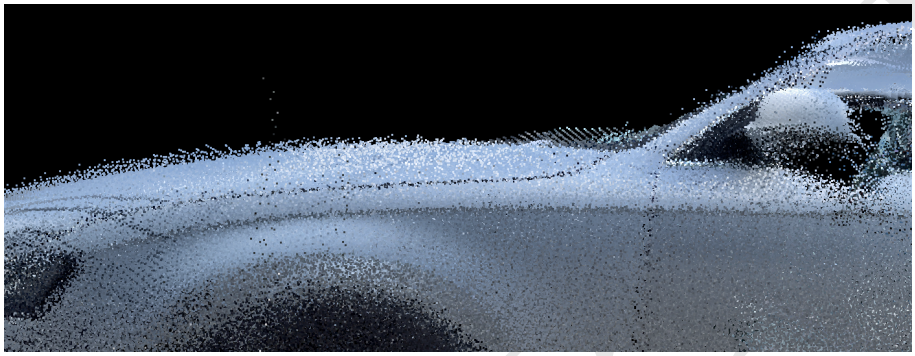


Figure 5.15: Without smoothing.

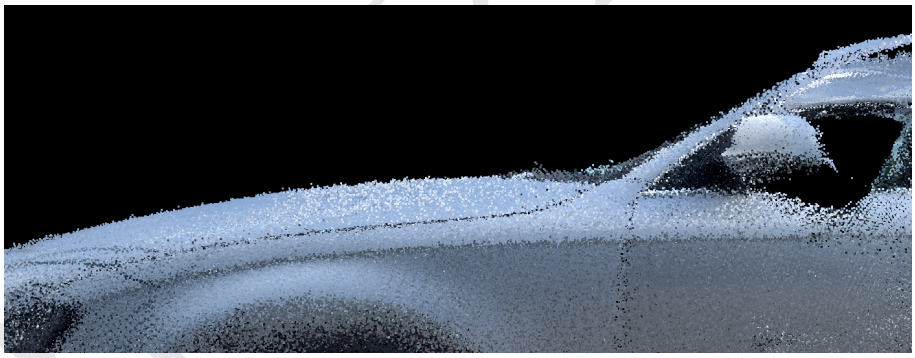


Figure 5.16: With smoothing (0.01).

5.3 Preprocessing Pipelines

The preprocessing pipelines used on the data consist of a combination of the filters described in the previous sections. Two different pipelines were used as illustrated in Figures 5.19 and 5.21. Examples of how a point cloud is transformed by these pipelines are given by Figures 5.17, 5.18, 5.20, and 5.22. The data used for the experiments was preprocessed with pipeline 2, as tests showed that it performed better than pipeline 1. This may be due to the voxel grid filter, as it aligns the points along a uniform grid, therefore making different point clouds resemble one another more. Even when the point clouds represent cars from different makes, the process of aligning the 3D points uniformly makes them geometrically more similar, which appears to hinder recognition performance.

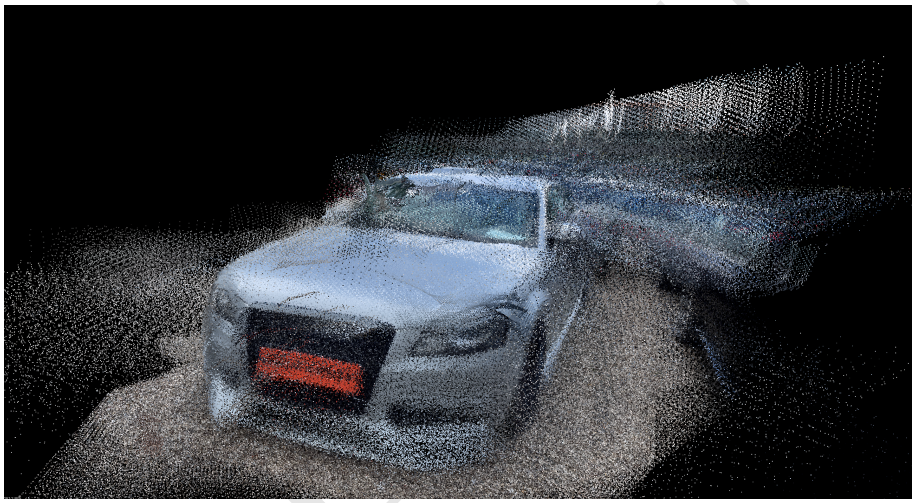


Figure 5.17: Before any preprocessing.

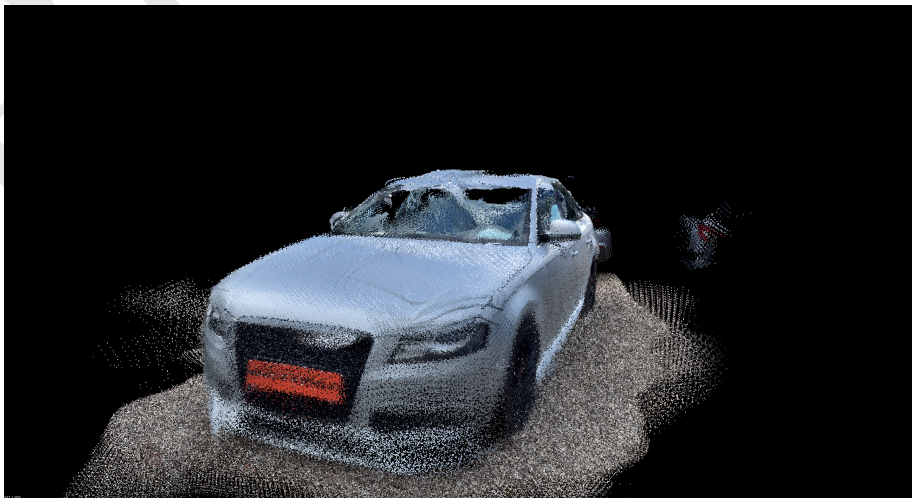


Figure 5.18: After only the online distance filter.

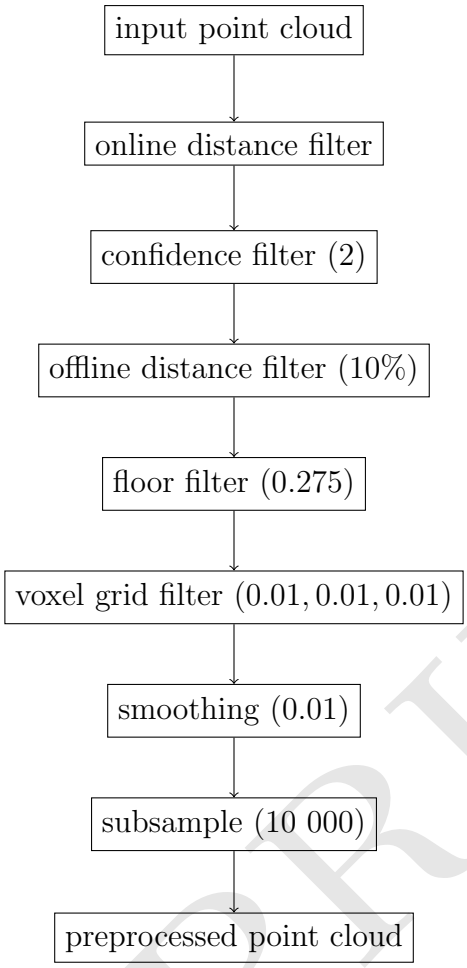


Figure 5.19: Preprocessing pipeline 1.

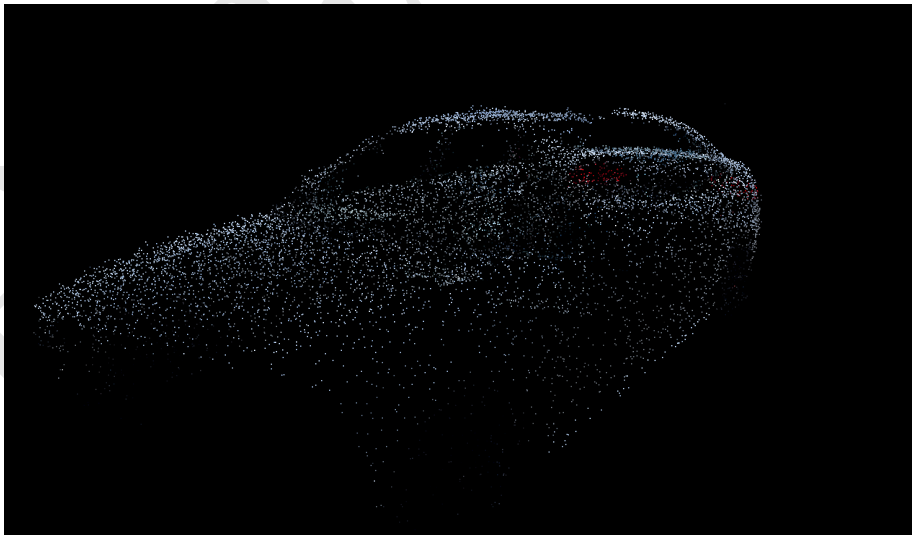


Figure 5.20: After pipeline 1.

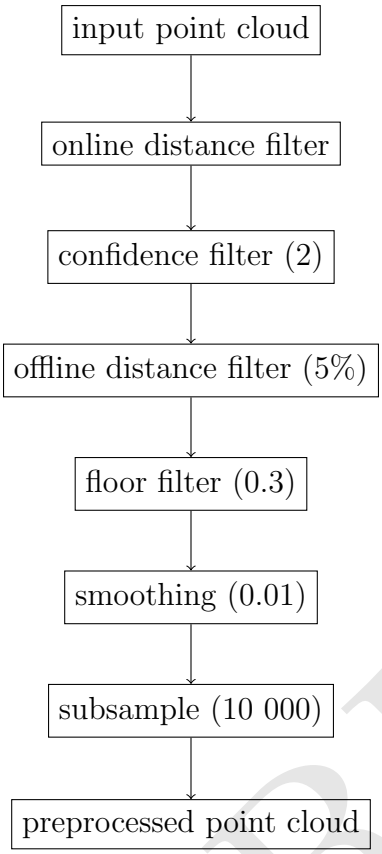


Figure 5.21: Preprocessing pipeline 2.

Figure 5.22: After pipeline 2.

Chapter 6

Data Augmentation

Data augmentation describes the process of manipulating data to create more and different examples for a neural network to learn from. Examples for data augmentation techniques in image classification are cropping, rotating, or flipping the images. These techniques have been used with great success [31]. Also data augmentation on 3D data has been used before [13]. This combined with the fact that a relatively low number of examples of car scans (only just under 400) were available made data augmentation an obvious choice for this project. This chapter explains the reasoning as to why each of the augmentation steps was chosen. A more detailed explanation of how these augmentations work can be found in the accompanying *Data Management and Infrastructure* thesis [10] and the code can be found on GitHub[1].

6.1 Geometric Augmentation

The first class of augmentations are the *geometric augmentations*. These apply a geometric transformation on the point cloud without destroying the general structure.

6.1.1 Translation

The *translation* augmentation step takes three arguments, which correspond to the x, y, and z directions. Then the entire point cloud is moved according to

[1] `https://github.com/ChrisEdel/vehicle-classification`

the given arguments. The structure of the point cloud and the positions of the points relative to one another remain unchanged by this augmentation, which is ideal, since the data has changed, but should remain recognizable for the neural network. In fact, if the network works properly the data must remain recognizable [33].

6.1.2 Random Translation

The *random translation* translates each point individually. Two arguments must be specified, an upper and lower limit. Then random, real numbers are uniformly generated from the range defined by the lower and upper limit and each coordinate of each point is translated by that random number. This artificially introduces some noise, see Figures 6.1 and 6.2. Again a new example is generated and, depending on the choices for the limits, a lot of the structure of the point cloud is preserved.

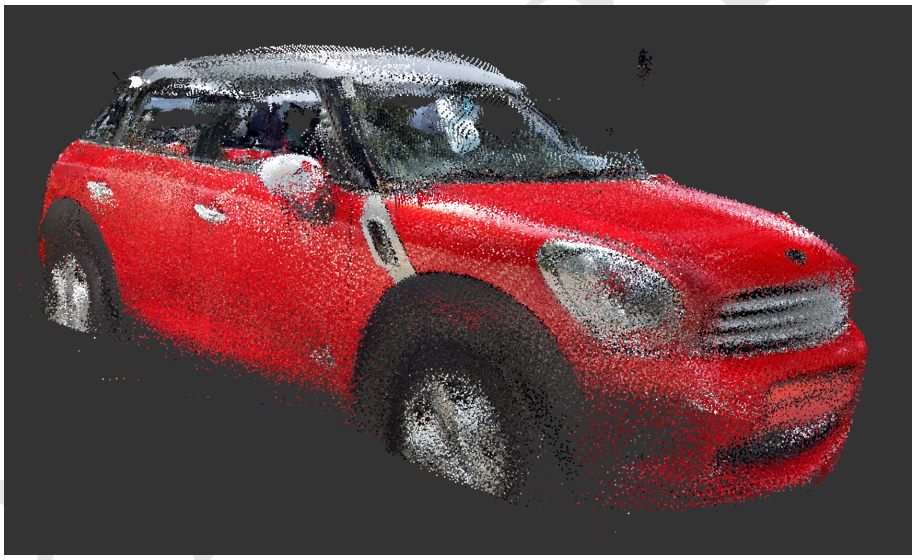


Figure 6.1: Without random translation.

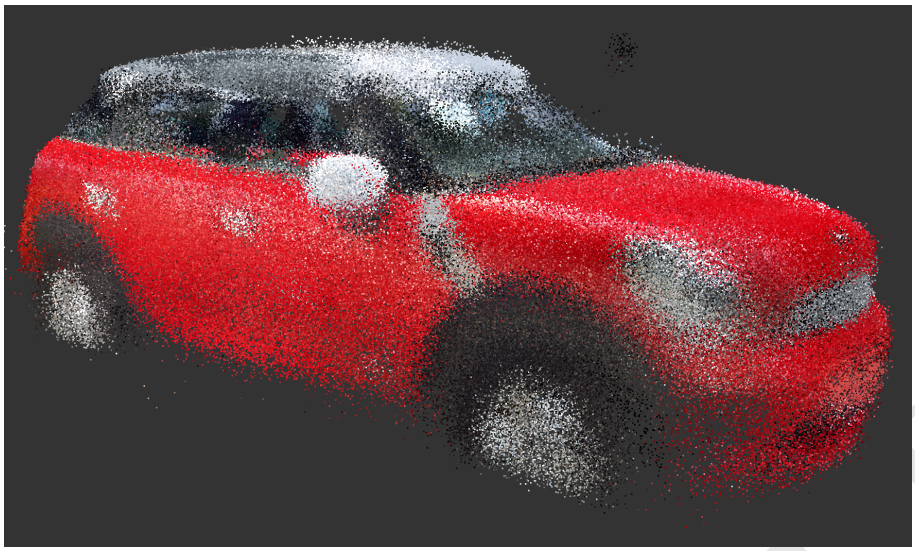


Figure 6.2: With random translation $(-0.05, 0.05)$.

6.1.3 Rotation

Rotation takes three arguments, which correspond to rotations around the x, y, and z axis in degrees. As the name of this augmentation suggests the entire point cloud is rotated according to the given arguments. Again, like the translation the structure of the point cloud remains unchanged and the network should still be able to classify it correctly [33].

6.2 Shuffle

Shuffle is a simple operation. A point cloud is a set of points and even though the collection is unordered, the neural network will see the points in one specific order. However, no matter which order the network sees the points of the point cloud in, it must appear as the same point cloud to the network [33]. In order to train the network accordingly this augmentation was used, where the points are simply shuffled. The appearance of the point cloud remains unchanged by this augmentation.

6.3 Color

This class of augmentations focuses on the color only. The classification of a car should not depend on what color the car has. Some methods take away color information and replace it with only greyscale or black and white values. This was done to explore how important the actual color values of a car scan are. Later on some of these augmentations were added as an additional final step in the preprocessing pipeline, in order to evaluate how well the network would perform, if it only has normalized color information available.

6.3.1 Hue

Changing the *hue* basically means changing the color. The colors black, white, and any shades of grey are preserved and only the main color of the vehicle is shifted. Figures 6.3 and 6.4 illustrate this. This transformation ensures that a car of any color can be recognized. Furthermore, the scans were made at car dealers and some of them mark all their cars with a colored label specific to that car dealer. If a car dealer then has almost exclusively one make for sale it could happen that the network simply looks for that specific label color in the scan and labels according to that, which is not really desirable. To mitigate such effects this augmentation was introduced.

Figure 6.3: Without hue shift.

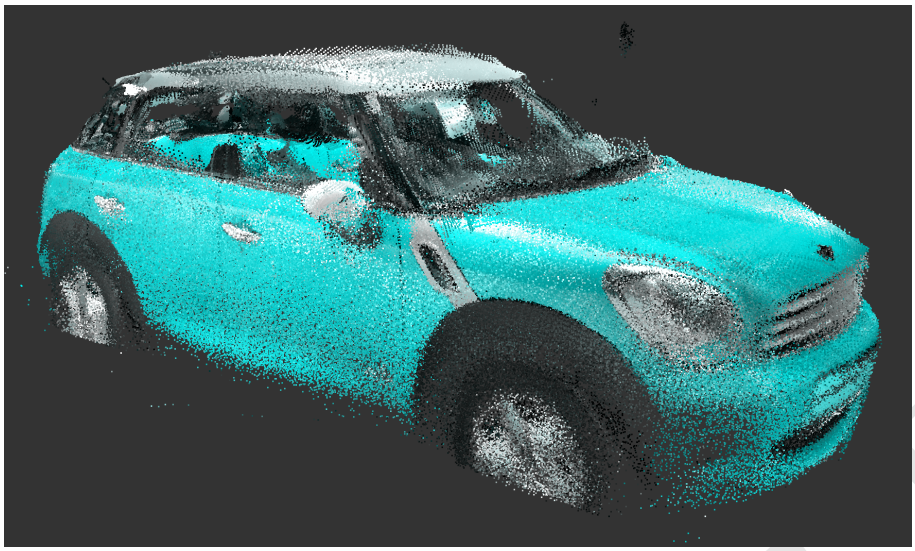


Figure 6.4: With hue shifted to 180.

6.3.2 Greyscale

Greyscale transforms the colors into shades of grey. Even though the actual color information is lost, information about the intensity is preserved. This can be seen in Figures 6.5 and 6.6. This step may be able to mitigate the same undesirable effect described in 6.3.1. In addition to that it may provide insight as to how much of the color information is actually necessary for classification.

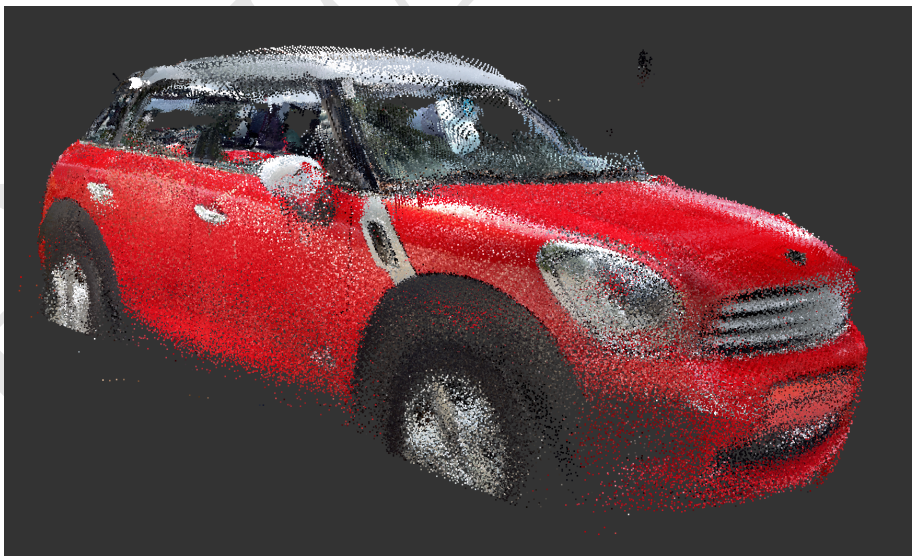


Figure 6.5: Without greyscale.

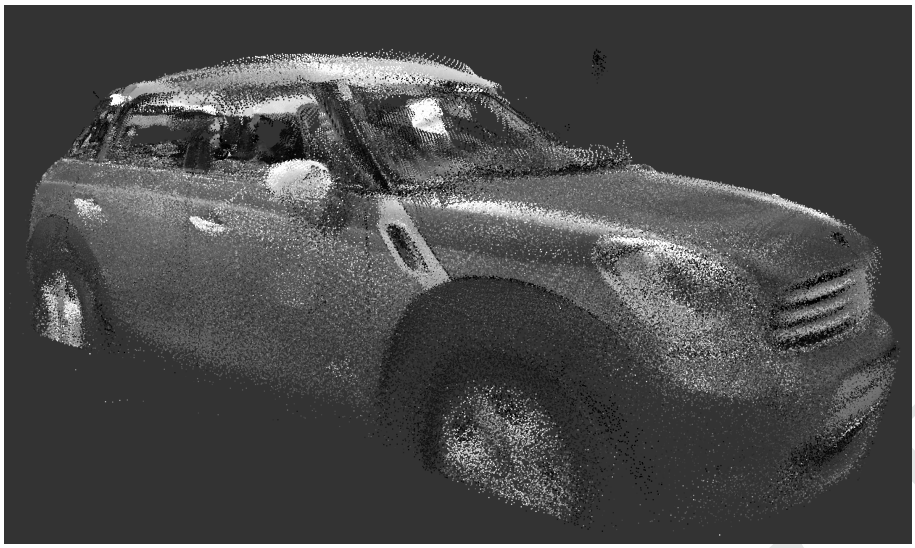


Figure 6.6: With greyscale applied.

6.3.3 Black and White

The *black and white* transformation transforms the color of every point to either black or white, see Figures 6.7 and 6.8. A lot of information is lost here, but not all. This step should completely eliminate the undesirable effect mentioned in 6.3.1 and may help to more closely determine the amount of color data necessary to classify successfully, similar to 6.3.2.

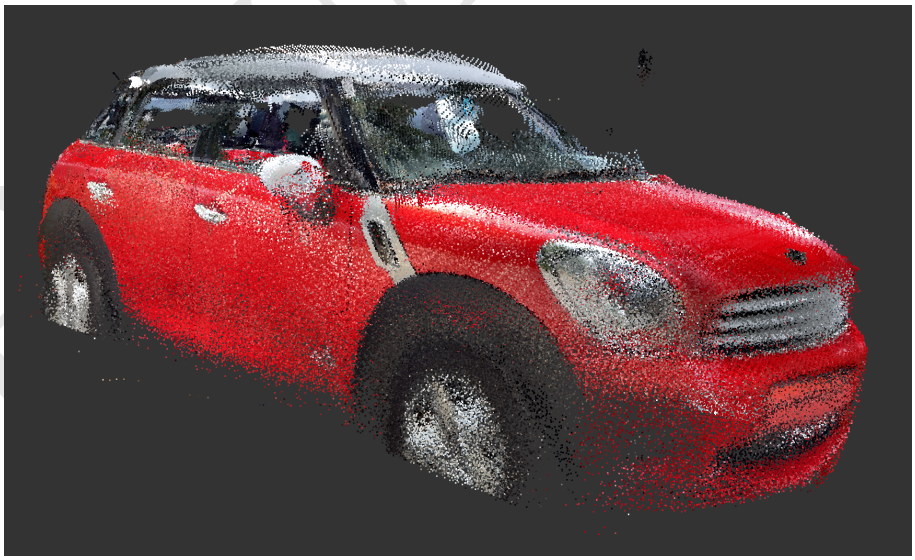


Figure 6.7: Without black and white transformation.

Figure 6.8: With black and white transformation applied.

6.3.4 Random Color Transform

Finally, the *random color transform* takes two values as arguments and shifts every RGB value of every point by a random integer drawn uniformly from the range given by the two arguments, the effects of which can be seen in Figures 6.9 and 6.10. This introduces some noise to the color data to make the network more robust against imprecise values and it may allow us to further pinpoint the importance of color for classification.

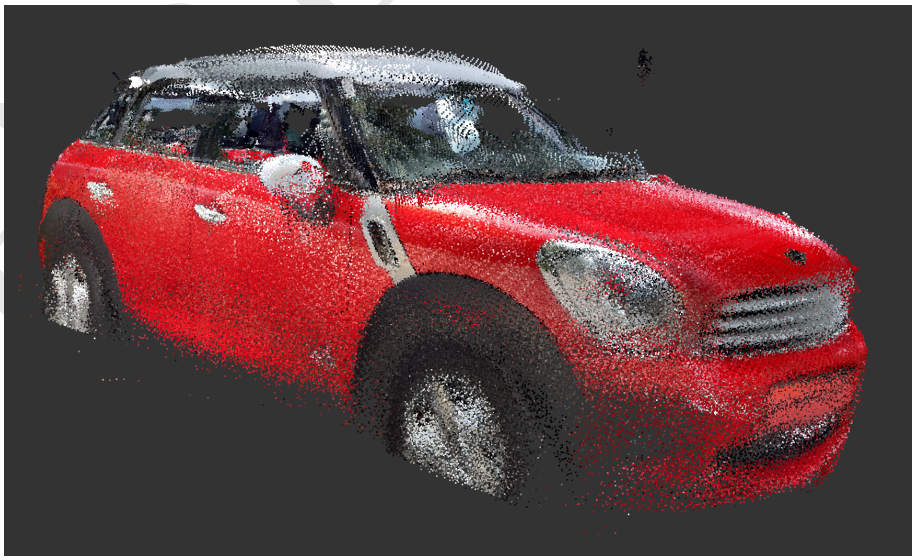


Figure 6.9: Without random color transform.

Figure 6.10: With random color transform $(-25, 25)$.

Chapter 7

The Color3DNet

As there already exist many architectures to classify 3D data and specifically point clouds, the first step was to use and evaluate an existing network and see how it performs on this data. For this the PointNet was chosen.

7.1 Starting Point: The PointNet

The *PointNet* is a segmentation and classification network. In this thesis only the classification network is considered, as the task evaluated is a classification task.

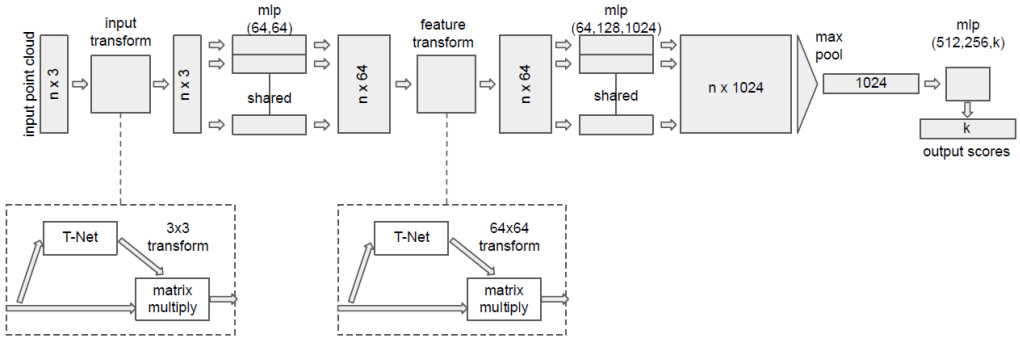


Figure 7.1: PointNet as classification network [33].

Figure 7.1 illustrates the architecture of the PointNet. The unordered input points are first transformed into a vector, using symmetric transforms such that the ordering the network received the points in does not affect the vector. Examples for symmetric operations are addition or multiplication. Next a

feature transformation is performed and the point features are aggregated by a max pooling layer. Both these transforms use mlps and 1D convolution and make use of a mini network called the *T-Net*.

Finally a multi-layer perceptron with batch normalization and dropout outputs the classification scores. The number of neurons of each layer of the mlps are denoted in parentheses [33].

The first step of evaluating the network was to run the data through the PointNet. As the data consists of 3D points plus color information in the beginning the color data was omitted and a baseline on the 3D data only was established.

7.2 Processing Bigger Point Clouds

The original PointNet was designed to classify point clouds of size 2 048 [33]. The car scans, however, are much bigger, the raw point clouds comprise millions of points. Reducing these to only 2 048 points would result in a huge loss of information, therefore the PointNet was modified to handle bigger point clouds of size 10 000. One necessary modification was increasing the size of the input layer to accept the bigger input. To then improve the performance of the network, which was originally designed to classify much smaller pointclouds, parts of the network were increased as well. Due to GPU memory constraints (a Nvidia Tesla K80 with 12GB VRAM per processor was used [46]) not the entire network could simply be blown up, but different configurations, where different parts of the network were increased in size, were tested.

The first part that was increased in size is the feature extraction. In this step the size of the T-Net was increased by adding more filters to the convolutional layers and making the layers of the mlps bigger.

Additionally the mlp at the end of the network was increased as well, simply by increasing the number of neurons in each layer.

7.3 Processing Color

Since the initial tests of the PointNet and slightly modified versions of the PointNet had not performed very well in classifying car makes using only the 3D data, the color information was added.

At first the simplest modification was tested, meaning the input size was increased to pass color values as input as well, which did not yield great results. This approach was tested, just in case it worked well, even though that was not expected. It would mean that the color features are extracted identically to the 3D features. However, as color information is quite different from the 3D information, the feature extraction must be different as well. This is a hypothesis to explain the results from the experiments.

Therefore, the part of the PointNet consisting of T-Nets has been duplicated, in order to extract the features from the 3D points and the color information separately. To make this work, the input layer has been enlarged, so that in can now accept input of size 10 000 by six, meaning each point cloud has 10 000 points and each point consists of three 3D coordinates and three RGB color values. Then the input is split, such that the 3D coordinates and color values are separated. Each of them goes through the feature extraction, which each includes two T-Nets. Finally, the outputs are concatenated and then classified by the remaining dense layers.

7.4 Final Architecture

In this section the final architecture of the *Color3DNet* is presented (see Figure 7.2). As the name suggests the final setup uses color values, as this additional information vastly improves the performance of the network. This means that the input layer is of shape $n \times 6$. The input is a pointcloud with n points with color information. Each point consists of x, y, and z 3D position values and three RGB color values representing the color of that point. The next step is to split the 3D position values and the RGB values. Then the 3D and the color data each get fed into their own feature extraction pipeline, which is the same as the feature extraction used for 3D points in the pointnet [33]. Finally, after the max pooling step both the data flows from the 3D and the color data are concatenated and fed to a mlp and ultimately gets us the output scores for the classes.

The activation function used is the ReLu function, except for the output layer where a Softmax function is used, like in the original PointNet [33]. The Python source code is included in the Appendix and made available on GitHub[1].

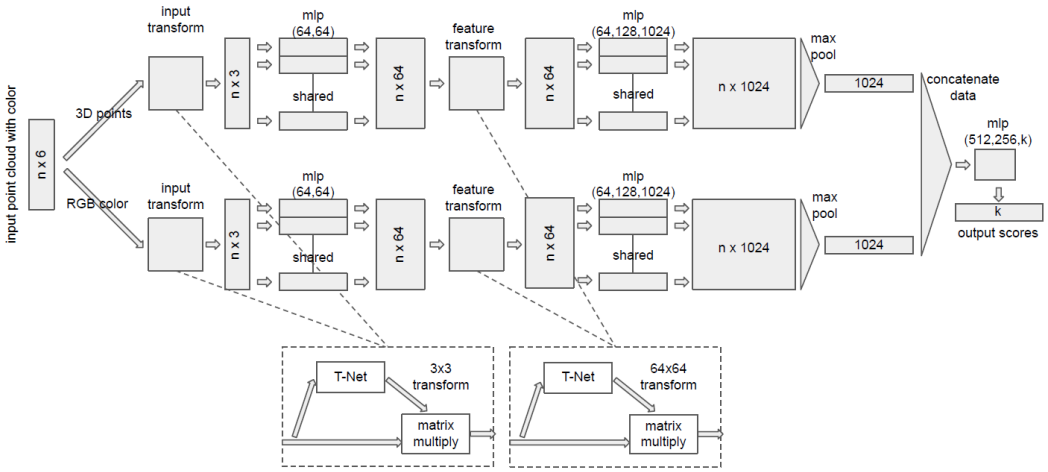


Figure 7.2: Architecture of the Color3DNet.

This network allows us to examine the impact of introducing color values to the process of classifying 3D data. As the Color3DNet is based on the PointNet, comparing the two seems reasonable, even though they are different networks. In the next chapter, the difference between classifying with and without color data will be highlighted.

[1] `https://github.com/ChrisEdel/vehicle-classification`

Chapter 8

Experiments

In this chapter the results of the experiments are presented. The data that was used was split into a training and test set randomly and the weights of the neural network were initialized randomly. The default configuration for the experiments is as follows: An average of three runs each with a different random seed (2 132 515 321, 1 861 562 392 and 913 172 673) for the data split and initial weights was taken. The batch size used is 20 and the point cloud size is 10 000.

The data to be classified are 360 degree car scans. The classes are: Audi, Volvo, Ford, Mercedes, and VW. As mentioned in a previous chapter only a limited number of car scans were available (just under 400). This rather low number was increased with the usage of the data augmentation techniques discussed in chapter 6.

In terms of how the data was split, two approaches were taken. The first approach was to split all the data, original and augmented, into a train (80%) and test (20%) set. This shows nicely how well the network can learn the data.

The second approach, the *strict split*, puts an emphasis on generalization to never before seen car models and makes it harder for the network. To explain this, let us consider the following example: We have only one Audi Q2 in the data set. In the first approach this car scan would get augmented and due to the random split the network has a good chance of seeing either the original or an augmented version of this car scan during training, which will help it to classify it correctly during testing. The strict split does not allow for such a situation; when splitting the data strictly, only the original data is split into a train (85%) and test (15%) set. Then the training data is augmented. This ensures that the test sets consists of scans that the network has never seen before, not even as an augmented version of scan.

If an experiment deviates from these default settings, it is explicitly stated.

8.1 Learnability of the Data

These experiments try to answer the question, whether this kind of data can be learned at all. To this end, the original PoinNet and the Color3DNet were both trained with the standard 20% split.

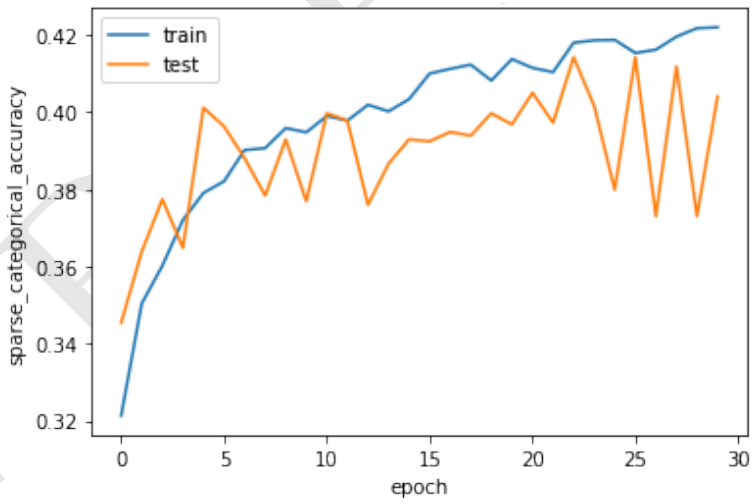


Figure 8.1: Accuracy graphs of the PointNet training for 30 epochs with a 20% split.

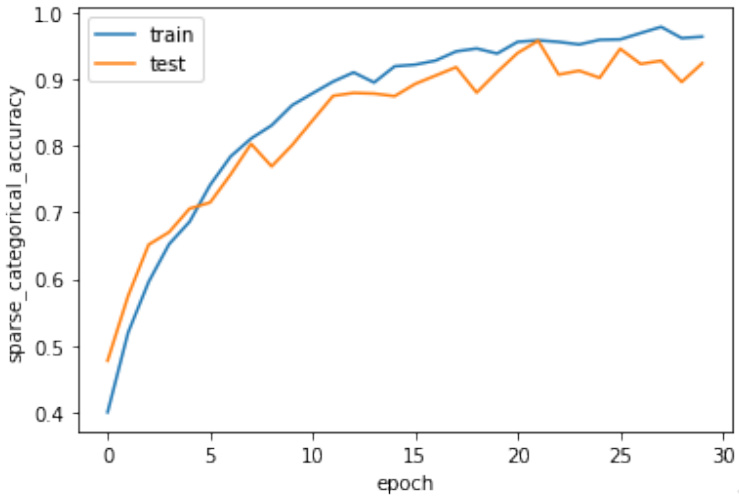


Figure 8.2: Accuracy graphs of the Color3DNet training for 30 epochs with a 20% split.

What can be observed here is that the PointNet has difficulty learning the data (Figure 8.1). The Color3DNet on the other hand can learn the data very well (Figure 8.2). That the Color3DNet performs much better than the PointNet at learning the data is not very surprising, when one considers that the Color3DNet uses much more data than the PointNet, as the RGB color values of each point are considered as well. Additionally, the data the PointNet was originally trained with (ModelNet40[1]) has much more coarse classes (chair, table, plane, car, ...), as opposed to only car makes.

8.2 Importance of Color

As seen in the last section, adding color information and modifying the PointNet such that it can handle the additional information makes the previously unlearnable data learnable. In this section results will be shown that put an emphasis on answering the question as to how important the color data is. To this end, the network has been trained with data that differs in the color information. The different types of color information used were:

- Original color
- Color normalized to a hue of 200, which is a cyan color
- Greyscale
- Black and white

[1] `https://modelnet.cs.princeton.edu/`

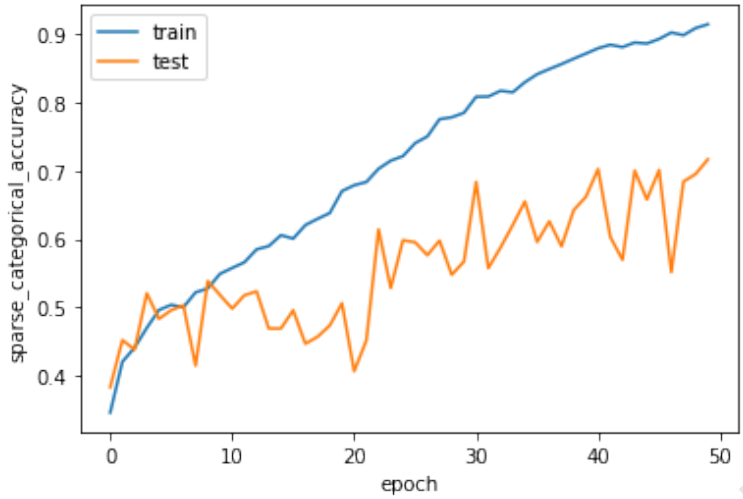


Figure 8.3: Accuracy graph of the Color3DNet training for 50 epochs with a 20% split on data normalized to hue 200.

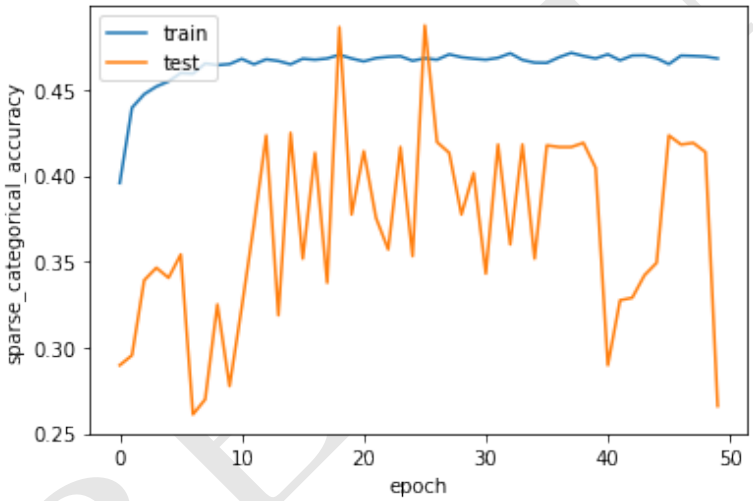


Figure 8.4: Accuracy graph of the Color3DNet training for 50 epochs with a 20% split on greyscale data.

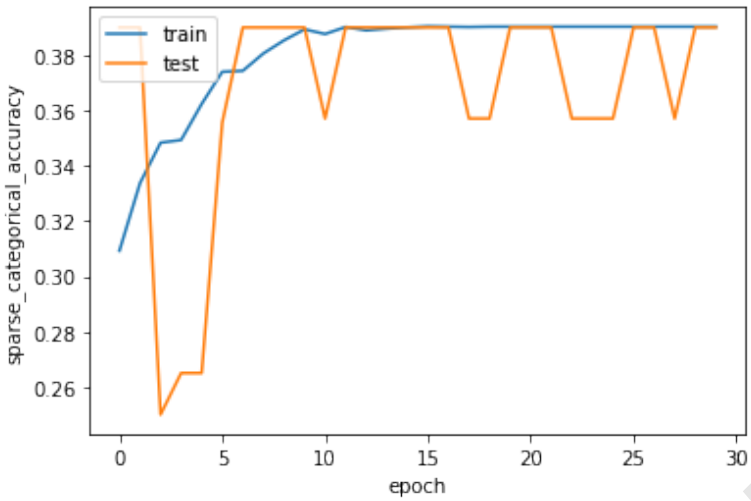


Figure 8.5: Accuracy graph of the Color3DNet training for 50 epochs with a 20% split on black and white data.

A general trend that can be seen here is that as we remove color information, the network has increasing difficulty learning the data. Figure 8.2 shows that the Color3DNet can learn the data with the original color very well. After normalizing the color values to a hue of 200 (Figure 8.3), we can already observe that the network takes longer to reach the same accuracy as before on the training set. Furthermore, generalizing becomes more difficult.

Taking it a step further, Figure 8.4 shows the training with only greyscale color values, which reduces performance even further. This is not really surprising, as going from color, even normalized, to greyscale removes quite a lot of information. One might argue that normalized color and greyscale both loose the color information, but keep relative color information (like which part is brighter or darker). However, greyscale removes even more information, which is explained in more detail in the accompanying *Data Management and Infrastructure* thesis [10].

Finally, Figure 8.5 shows the training results for black and white data, which performs slightly worse than greyscale data, which comes as no surprise, since even more information is removed.

8.3 Generalization

The following experiments focus on generalization by using the strict split. These results give us an insight on how the network handles data it never saw before, including completely new car models.

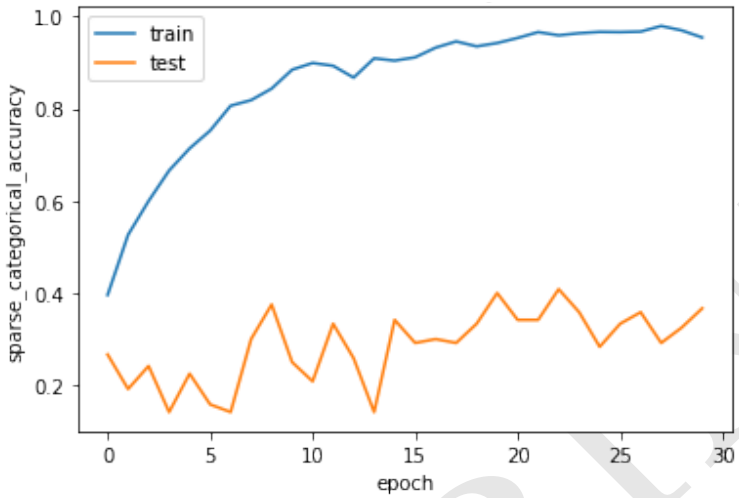


Figure 8.6: Accuracy graph of the Color3DNet training for 30 epochs with a 15% strict split on data with original color.

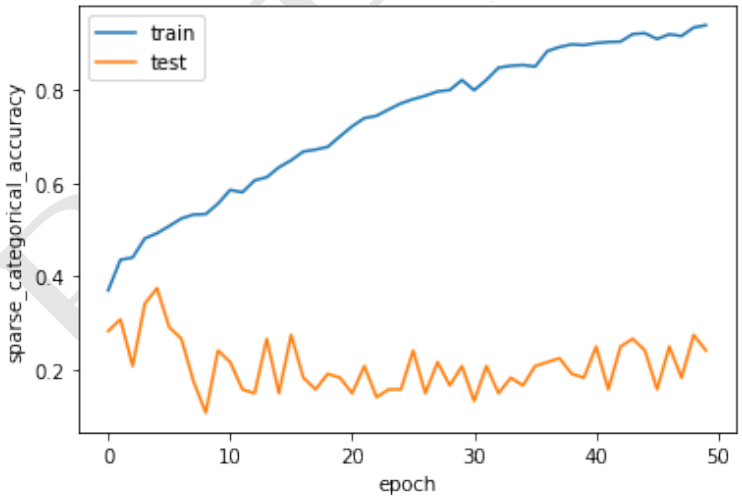


Figure 8.7: Accuracy graph of the Color3DNet training for 50 epochs with a 15% strict split on data normalized to hue 200.

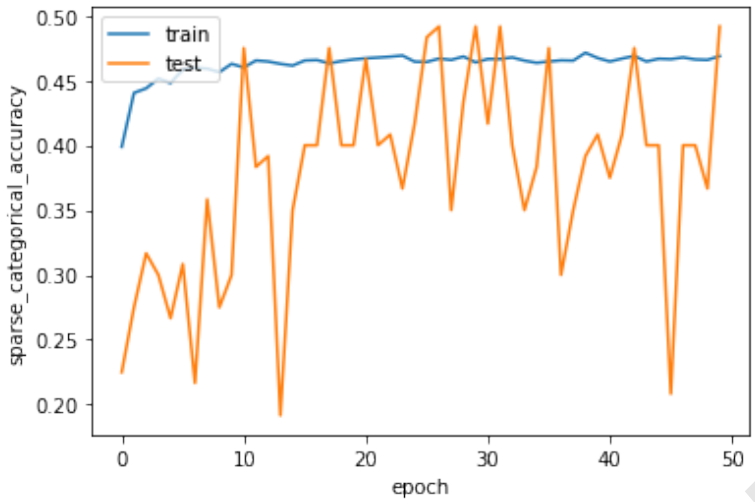


Figure 8.8: Accuracy graph of the Color3DNet training for 50 epochs with a 15% strict split on greyscale data.

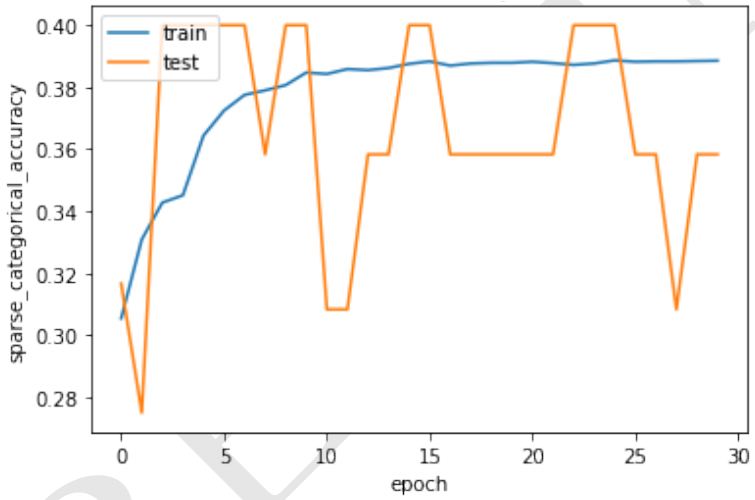


Figure 8.9: Accuracy graph of the Color3DNet training for 50 epochs with a 15% strict split on black and white data.

A notable observation that can be made from these results is the overfitting of the Color3DNet (Figure 8.6). This may occur due to the fact that the strict split reduces the size of the already limited data even more. Another factor making the network overfit may be the fact that when using the strict split a never before seen car model can be in the test set, which may give the network extra difficulty classifying it correctly.

Not surprisingly this problem of overfitting persists when training with data normalized to hue 200 (Figure 8.7).

When removing any color and reducing the color information to greyscale

(Figure 8.8) or black and white (Figure 8.9), the overfitting is not quites as visible, but that is only due to the fact that the network is unable to learn the data at all beyond a certain point.

We attempted to improve the generalization and reduce overfitting. We tried different approaches, like using dropout with different rates, reducing the size of the network and introducing L1 and L2 regularization. Unfortunately none of these helped. We also had ideas we were unable to try. The first would be to collect a lot more data. About 400 scans is a very small data set to work with, even though it was enough to at least show the viability of the approach to use LiDAR scans with color information for classification [39]. The other option would be to increase the batch size [34] by using more powerful hardware. We were constrained to the 12 GB of VRAM provided by our Nvidia Tesla K80 [46]. A bleeding edge Nvidia A100 would allow up to 80 GB of VRAM [1], however, such hardware was unfortunately not available to us.

8.4 Point Cloud Size

One idea to improve network performance was to give it more information to work with by using bigger point clouds. To that end an experiment was done with the same network, but a point cloud size of 25 000. This has one side effect; due to memory limitations, the maximum batch size was now constrained to eight when training on a GPU. Before with 10 000 points the maximum possible batch size was 20.

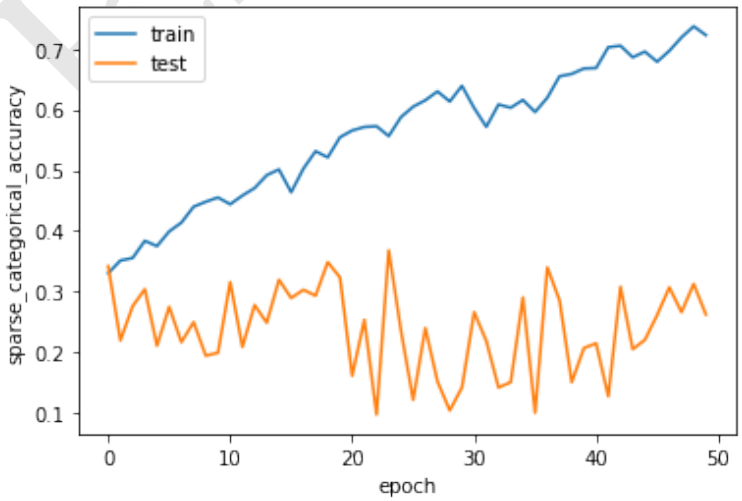


Figure 8.10: Accuracy graph of the Color3DNet training with point clouds of size 25 000 for 50 epochs with a 20% split and a batch size of 8.

These results show us several things. To begin with, the performance on the training set suffered; whereas the Color3DNet using a point cloud size of 10 000 (Figure 8.2) achieved an accuracy of almost 1, the Color3DNet trained with a point cloud size of 25 000 (Figure 8.10) only got just above 0.7, even with more epochs.

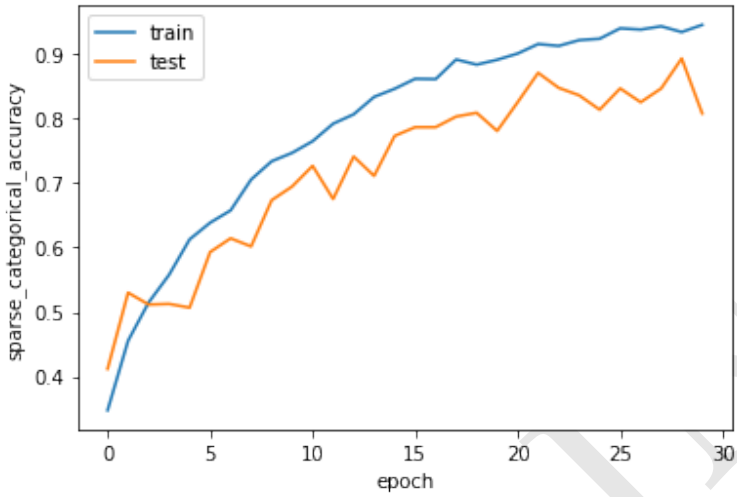


Figure 8.11: Accuracy graph of the Color3DNet training with point clouds of size 25 000 for 30 epochs with a 20% split and a batch size of 20.

Interestingly enough in Figure 8.11, which shows training with point cloud sizes of 25 000 and a batch size of 20 (training was performed on the CPU), we can observe that the accuracy on the test set recovers and overall a higher accuracy is reached faster.

This result is quite surprising, as [26] shows that smaller batch sizes do not in general have adverse effects on accuracy, in fact the contrary is shown.

However, as we can see the bigger batch size improves training in every way: The training is faster, the network performs better and overfitting is massively reduced. Regarding the point cloud size we cannot observe a real benefit of using 25 000 points instead of 10 000 points.

8.5 Car Models

To conclude this chapter we examine some experiments that show the performance of the network when going for the most detail and using all available car models as classes. This yields 145 classes, which include 144 car models and one default label, if the car scan is of no known car model.

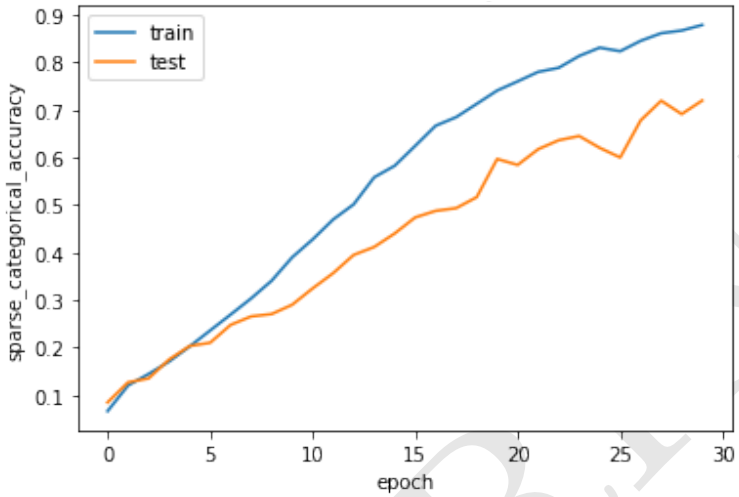


Figure 8.12: Accuracy graph of the Color3DNet training for 30 epochs with a 20% split classifying car models.

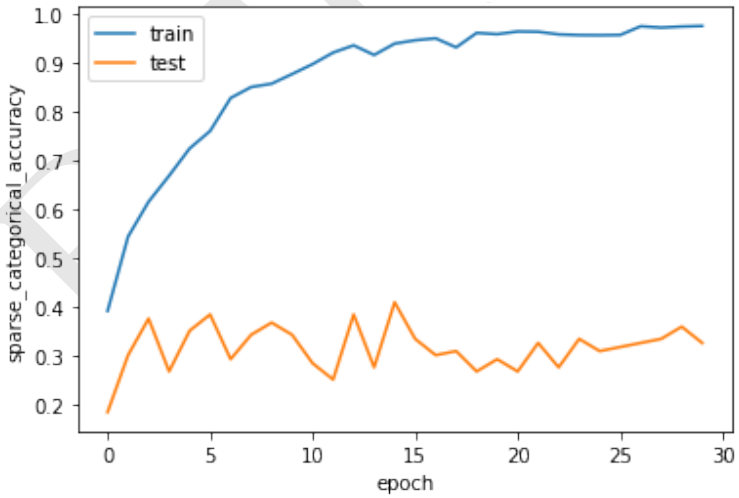


Figure 8.13: Accuracy graph of the Color3DNet training for 30 epochs with a 15% strict split classifying car models.

These experiments (Figures 8.12 and 8.13) show that even though car models have a lot more classes, the data can still be learned by the Color3DNet, although not as quickly as the car makes. Furthermore the problem of overfitting when applying the strict split still persists. None of this comes as a surprise. As the network now has to distinguish between many more classes and may have to extract more subtle features the learning process takes longer, as can be seen when comparing Figures 8.2 and 8.12. On the other hand, the amount of data used is still not much more, therefore the fact that the overfitting problem still occurs makes sense as well. Additionally, if a specific car model only occurs in the test set, the network has pretty much no chance of classifying that model correctly.

Chapter 9

Conclusion and Outlook

This project set out to evaluate, whether a LiDAR scan generated by an off-the-shelf sensor could be used for detailed classification; to be more precise, the classification of car scans regarding their make or even model. These scans consist of three 3D coordinates from the LiDAR sensor and three RGB color values from the camera sensor. In order to actually classify the data, a network was needed to work with this kind of data, so the Color3DNet was developed, using the PointNet [33] as a starting point.

What we discovered is that the such detailed classes can definitely be learned by a neural network, when the color values of the 3D points are used. Regarding the color, the original color of the scan works best, but even when the color is normalized the training set can be learned.

This project could be continued by further experimenting with techniques mentioned in section 8.3, in order to try to classify never before seen scans and car models. Furthermore, the size of the input could be increased to for example 100 000 points or even more with according modifications to the network. This would be possible on more powerful hardware. Another aspect that could be investigated further is the network itself; a different architecture could be chosen as a starting point. A different network could also enable the use of different representations of 3D data, which may include color information as well. Then transformations similar to the ones described in this thesis could be applied to the different 3D data representations.

Bibliography

[1] *A100 Datasheet*. NVIDIA. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf (visited on 07/27/2021).

[2] Sajjad Abdoli, Patrick Cardinal, and Alessandro Lameiras Koerich. "End-to-End Environmental Sound Classification using a 1D Convolutional Neural Network". In: *CoRR* abs/1904.08990 (2019). arXiv: 1904.08990. URL: http://arxiv.org/abs/1904.08990.

[3] Eman Ahmed et al. "Deep Learning Advances on Different 3D Data Representations: A Survey". In: *CoRR* abs/1808.01462 (2018). arXiv: 1808.01462. URL: http://arxiv.org/abs/1808.01462.

[4] Apple. *iPhone 12 Pro - Technical Specifications*. 2021. URL: https://support.apple.com/kb/SP831?locale=en_US (visited on 07/23/2021).

[5] Adriano Baldeschi, Raffaella Margutti, and Adam Miller. "Deep Learning: a new definition of artificial neuron with double weight". In: *CoRR* abs/1905.04545 (2019). arXiv: 1905.04545. URL: http://arxiv.org/abs/1905.04545.

[6] Noppakun Boonsim and Simant Prakoonwit. "Car make and model recognition under limited lighting conditions at night". In: *Pattern Analysis and Applications* 20 (Nov. 2017). DOI: 10.1007/s10044-016-0559-6.

[7] Massimo Buscema. "Back Propagation Neural Networks". In: *Substance use misuse* 33 (Feb. 1998), pp. 233–70. DOI: 10.3109/10826089809115863.

[8] Carolina Crisci, Badih Ghattas, and Gonzalo Perera. "A review of supervised machine learning algorithms and their applications to ecological data". In: *Ecological Modelling* 240 (Aug. 2012), pp. 113–122. DOI: 10.1016/j.ecolmodel.2012.03.001.

[9] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2018. arXiv: 1603.07285 [stat.ML].

[10] Christian Edelmayer. "Data Management and Infrastructure for Vehicle Classification Using Commercial Off-The-Shelf LiDAR and Camera Sensors". Preprint of Master's Thesis. University of Salzburg, 2021. URL: https://github.com/ChrisEdel/vehicle-classification/tree/main/theses/edelmayer.

[11] Enzo Grossi and Massimo Buscema. "Introduction to artificial neural networks". In: *European journal of gastroenterology hepatology* 19 (Jan. 2008), pp. 1046–54. DOI: 10.1097/MEG.0b013e3282f198a0.

[12] Jiuxiang Gu et al. "Recent Advances in Convolutional Neural Networks". In: *CoRR* abs/1512.07108 (2015). arXiv: 1512.07108. URL: http://arxiv.org/abs/1512.07108.

[13] Martin Hahner et al. "Quantifying Data Augmentation for LiDAR based 3D Object Detection". In: *CoRR* abs/2004.01643 (2020). arXiv: 2004.01643. URL: https://arxiv.org/abs/2004.01643.

[14] Rana Hanocka et al. "MeshCNN: A Network with an Edge". In: *CoRR* abs/1809.05910 (2018). arXiv: 1809.05910. URL: http://arxiv.org/abs/1809.05910.

[15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[16] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. "Artificial neural networks: a tutorial". In: *Computer* 29.3 (1996), pp. 31–44. DOI: 10.1109/2.485891.

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

[18] Serkan Kiranyaz et al. *1D Convolutional Neural Networks and Applications: A Survey*. 2019. arXiv: 1905.03554 [eess.SP].

[19] Honglak Lee et al. "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks". In: *Commun. ACM* 54 (Oct. 2011), pp. 95–103. DOI: 10.1145/2001269.2001295.

[20] Hyo Jong Lee et al. "Real-Time Vehicle Make and Model Recognition with the Residual SqueezeNet Architecture". eng. In: *Sensors (Basel, Switzerland)* 19.5 (Feb. 2019). s19050982[PII], p. 982. ISSN: 1424-8220. DOI: 10.3390/s19050982. URL: https://doi.org/10.3390/s19050982.

[21] Huan Lei, Naveed Akhtar, and Ajmal Mian. "Spherical Kernel for Efficient Graph Convolution on 3D Point Clouds". In: *CoRR* abs/1909.09287 (2019). arXiv: 1909.09287. URL: http://arxiv.org/abs/1909.09287.

[22] Yangyan Li et al. “PointCNN”. In: *CoRR* abs/1801.07791 (2018). arXiv: 1801.07791. URL: http://arxiv.org/abs/1801.07791.

[23] You Li and Javier Ibañez-Guzmán. “Lidar for Autonomous Driving: The principles, challenges, and trends for automotive lidar and perception systems”. In: *CoRR* abs/2004.08467 (2020). arXiv: 2004.08467. URL: https://arxiv.org/abs/2004.08467.

[24] Nitin Malik. “Artificial Neural Networks and their Applications”. In: *CoRR* abs/cs/0505019 (2005). arXiv: cs/0505019. URL: http://arxiv.org/abs/cs/0505019.

[25] Muhammad Asif Manzoor, Yasser Morgan, and Abdul Bais. “Real-Time Vehicle Make and Model Recognition System”. In: *Machine Learning and Knowledge Extraction* 1 (Apr. 2019), pp. 611–629. DOI: 10.3390/make1020036.

[26] Dominic Masters and Carlo Luschi. “Revisiting Small Batch Training for Deep Neural Networks”. In: *CoRR* abs/1804.07612 (2018). arXiv: 1804.07612. URL: http://arxiv.org/abs/1804.07612.

[27] Daniel Maturana and Sebastian Scherer. “VoxNet: A 3D Convolutional Neural Network for real-time object recognition”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 922–928. DOI: 10.1109/IROS.2015.7353481.

[28] Ramesh Medar, Vijay Rajpurohit, and B. Rashmi. “Impact of Training and Testing Data Splits on Accuracy of Time Series Forecasting in Machine Learning”. In: Aug. 2017, pp. 1–6. DOI: 10.1109/ICCUBEA.2017.8463779.

[29] Xingyang Ni and Heikki Huttunen. “Vehicle Attribute Recognition by Appearance: Computer Vision Methods for Vehicle Type, Make and Model Classification”. In: *Journal of Signal Processing Systems* 93.4 (Apr. 2021), pp. 357–368. DOI: 10.1007/s11265-020-01567-6. URL: https://doi.org/10.1007/s11265-020-01567-6.

[30] Chigozie Nwankpa et al. “Activation Functions: Comparison of trends in Practice and Research for Deep Learning”. In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[31] Luis Perez and Jason Wang. “The Effectiveness of Data Augmentation in Image Classification using Deep Learning”. In: *CoRR* abs/1712.04621 (2017). arXiv: 1712.04621. URL: http://arxiv.org/abs/1712.04621.

[32] M. Popescu et al. “Multilayer perceptron and neural networks”. In: *WSEAS Transactions on Circuits and Systems archive* 8 (2009), pp. 579–588.

[33] Charles Ruizhongtai Qi et al. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *CoRR* abs/1612.00593 (2016). arXiv: 1612.00593. URL: http://arxiv.org/abs/1612.00593.

[34] Pavlo Radiuk. "Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets". In: *Information Technology and Management Science* 20 (Dec. 2017), pp. 20–24. DOI: 10.1515/itms-2017-0003.

[35] H. Ramchoun et al. "Multilayer Perceptron: Architecture Optimization and Training". In: *Int. J. Interact. Multim. Artif. Intell.* 4 (2016), pp. 26–30.

[36] Fabio Remondino. "From point cloud to surface: The modeling and visualization problem". In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 34 (Mar. 2004). DOI: 10.3929/ethz-a-004655782.

[37] Santiago Royo and Maria Ballesta-Garcia. "An Overview of Lidar Imaging Systems for Autonomous Vehicles". In: *Applied Sciences* 9.19 (2019). ISSN: 2076-3417. DOI: 10.3390/app9194093. URL: https://www.mdpi.com/2076-3417/9/19/4093.

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

[39] Shaeke Salman and Xiuwen Liu. "Overfitting Mechanism and Avoidance in Deep Neural Networks". In: *CoRR* abs/1901.06566 (2019). arXiv: 1901.06566. URL: http://arxiv.org/abs/1901.06566.

[40] Satya P. Singh et al. *3D Deep Learning on Medical Images: A Review.* 2020. arXiv: 2004.00218 [q-bio.QM].

[41] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[42] Hang Su et al. "Multi-view Convolutional Neural Networks for 3D Shape Recognition". In: *CoRR* abs/1505.00880 (2015). arXiv: 1505.00880. URL: http://arxiv.org/abs/1505.00880.

[43] Tomasz Szandala. "Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks". In: *CoRR* abs/2010.09458 (2020). arXiv: 2010.09458. URL: https://arxiv.org/abs/2010.09458.

[44] Faezeh Tafazzoli and Hichem Frigui. "Vehicle Make and Model Recognition Using Local Features and Logo Detection". In: Nov. 2016. DOI: 10.1109/ISIVC.2016.7894014.

[45] Faezeh Tafazzoli, Hichem Frigui, and Keishin Nishiyama. "A Large and Diverse Dataset for Improved Vehicle Make and Model Recognition". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2017, pp. 874–881. DOI: 10.1109/CVPRW.2017.121.

[46] *Tesla K80 Board Specification*. NVIDIA. 2015. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf (visited on 07/05/2021).

[47] Zhirong Wu et al. "3D ShapeNets for 2.5D Object Recognition and Next-Best-View Prediction". In: *CoRR* abs/1406.5670 (2014). arXiv: 1406.5670. URL: http://arxiv.org/abs/1406.5670.

[48] Xue Ying. "An Overview of Overfitting and its Solutions". In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. DOI: 10.1088/1742-6596/1168/2/022022. URL: https://doi.org/10.1088/1742-6596/1168/2/022022.

[49] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *CoRR* abs/1611.03530 (2016). arXiv: 1611.03530. URL: http://arxiv.org/abs/1611.03530.

Appendix

color3Dnet.py

The following is the main Python script. It contains the code for the neural network as well as the code to record all results and metrics and a simple command line interface.

```
import argparse
import os
import random
import sys
from pprint import pprint
from time import time
from uuid import uuid4

import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

import matplotlib.pyplot as plt
import numpy as np

import LiDARCarDataset
from utils import get_time_str, check_gpu, plot_confusion_matrix, write_data_distribution, labels_iter


def main(paths, granularity='manufacturer_main', split=0.2, augmentation_split=False, strict_split=False, shuffle=True,
         random_seed=None, shuffle_pointcloud=False, normalize_distr=False, color=True, augmented=True,
         num_points=10000, batch_size=20, epochs=30, dropout=0.3, device='/device:CPU:0', tensorflow_data_dir=None,
         artifacts_dir=None):
    uuid = str(uuid4())
    experiment_name = 'color3Dnet-{}'.format(uuid)
    os.mkdir(experiment_name)
    print('Experiment name:', experiment_name)

    if not tensorflow_data_dir:
        tensorflow_data_dir = os.path.join(experiment_name, 'tensorflow_data')
    if not os.path.exists(tensorflow_data_dir):
        os.mkdir(tensorflow_data_dir)

    if not artifacts_dir:
        artifacts_dir = experiment_name
    if not os.path.exists(artifacts_dir):
        os.mkdir(artifacts_dir)

    if not random_seed:
        random_seed = random.randint(0, sys.maxsize)
    print('Using random seed:', random_seed)

    arguments = locals()

    # ensure compatibility with RTX cards:
```

```
config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)

tf.random.set_seed(random_seed)

builder_kwargs = {'path': paths,
                  'granularity': granularity,
                  'split': split,
                  'augmentation_split': augmentation_split,
                  'strict_split': strict_split,
                  'shuffle': shuffle,
                  'random_seed': random_seed,
                  'shuffle_pointcloud': shuffle_pointcloud,
                  'normalize_distr': normalize_distr,
                  'color': color,
                  'augmented': augmented
                  }

classes = LiDARCarDataset.GRANULARITY[granularity]
num_classes = len(classes)

(ds_train, ds_test), ds_info = tfds.load('LiDARCarDataset', split=['train', 'test'],
                                         data_dir=tensorflow_data_dir,
                                         shuffle_files=True, as_supervised=True, with_info=True,
                                         builder_kwargs=builder_kwargs)

# train pipeline:
ds_train = ds_train.cache()
ds_train = ds_train.batch(batch_size)
ds_train = ds_train.prefetch(tf.data.experimental.AUTOTUNE)

# test pipeline:
ds_test = ds_test.batch(batch_size)
ds_test = ds_test.cache()
ds_test = ds_test.prefetch(tf.data.experimental.AUTOTUNE)

### NEURAL NETWORK ###
class OrthogonalRegularizer(keras.regularizers.Regularizer):
    def __init__(self, num_features, l2reg=0.001):
        self.num_features = num_features
        self.l2reg = l2reg
        self.eye = tf.eye(num_features)

    def __call__(self, x):
        x = tf.reshape(x, (-1, self.num_features, self.num_features))
        xxt = tf.tensordot(x, x, axes=(2, 2))
        xxt = tf.reshape(xxt, (-1, self.num_features, self.num_features))
        return tf.reduce_sum(self.l2reg * tf.square(xxt - self.eye))

    def get_config(self):
        return {'num_features': self.num_features,
                'l2reg': self.l2reg}

def conv_bn(x, filters):
    x = layers.Conv1D(filters, kernel_size=1, padding="valid")(x)
    x = layers.BatchNormalization(momentum=0.0)(x)
    return layers.Activation("relu")(x)

def dense_bn(x, filters):
    x = layers.Dense(filters)(x)
    x = layers.BatchNormalization(momentum=0.0)(x)
    return layers.Activation("relu")(x)

def tnet(inputs, num_features):
    # Initalise bias as the indentity matrix
    bias = keras.initializers.Constant(np.eye(num_features).flatten())
    reg = OrthogonalRegularizer(num_features)

    x = conv_bn(inputs, 32)
    x = conv_bn(x, 64)
    x = conv_bn(x, 512)
    x = layers.GlobalMaxPooling1D()(x)
    x = dense_bn(x, 256)
    x = dense_bn(x, 128)
    x = layers.Dense(
        num_features * num_features,
        kernel_initializer="zeros",
        bias_initializer=bias,
        activity_regularizer=reg,
    )(x)
    feat_T = layers.Reshape((num_features, num_features))(x)
    # Apply affine transformation to input features
```

```
        return layers.Dot(axes=(2, 1))([inputs, feat_T])

    def split_data(x):
        # add 1 dim for cropping layer
        x = layers.Reshape((num_points, 6, 1))(x)
        # separate points and colors
        points = layers.Cropping2D(cropping=((0, 0), (0, 3)))(x)
        colors = layers.Cropping2D(cropping=((0, 0), (3, 0)))(x)
        # get rid of the extra dim again
        points = layers.Reshape((num_points, 3))(colors)
        colors = layers.Reshape((num_points, 3))(colors)
        return points, colors

    def make_model(dropout=0.3):
        inputs = keras.Input(shape=(num_points, 6))

        # split points
        x, xc = split_data(inputs)

        # processing of normal points
        x = tnet(x, 3)
        x = conv_bn(x, 32)
        x = conv_bn(x, 32)
        x = tnet(x, 32)
        x = conv_bn(x, 32)
        x = conv_bn(x, 64)
        x = conv_bn(x, 512)
        x = layers.GlobalMaxPooling1D()(x)

        # processing colors
        xc = tnet(xc, 3)
        xc = conv_bn(xc, 32)
        xc = conv_bn(xc, 32)
        xc = tnet(xc, 32)
        xc = conv_bn(xc, 32)
        xc = conv_bn(xc, 64)
        xc = conv_bn(xc, 512)
        xc = layers.GlobalMaxPooling1D()(xc)

        x = layers.Concatenate()([x, xc])
        x = dense_bn(x, 512)
        x = layers.Dropout(dropout)(x)
        x = dense_bn(x, 256)
        x = layers.Dropout(dropout)(x)
        x = dense_bn(x, 128)
        x = layers.Dropout(dropout)(x)

        outputs = layers.Dense(num_classes, activation="softmax")(x)

        model = keras.Model(inputs=inputs, outputs=outputs, name="color3Dnet")

        return model

    model = make_model()
    model.compile(loss='sparse_categorical_crossentropy', optimizer='Adam', metrics=['sparse_categorical_accuracy'])

    checkpoint_filepath = os.path.join(artifacts_dir, 'best-weights-{}'.format(experiment_name))
    model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath=checkpoint_filepath,
        save_weights_only=True,
        monitor='val_sparse_categorical_accuracy',
        mode='max',
        save_best_only=True)

    checkpoint_path_all = os.path.join(artifacts_dir, 'checkpoints-{}'.format(experiment_name), 'cp-{epoch:04d}.ckpt')
    checkpoint_dir_all = os.path.dirname(checkpoint_path_all)

    # Create a callback that saves the model's weights
    model_checkpoint_callback_all = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path_all,
                                                                       save_weights_only=True,
                                                                       verbose=1,
                                                                       save_freq='epoch')

    start = time()
    with tf.device(device):
        history = model.fit(ds_train, epochs=epochs, batch_size=batch_size, validation_data=ds_test, shuffle=True,
                            callbacks=[model_checkpoint_callback, model_checkpoint_callback_all])
    stop = time()
    time_seconds = stop - start

    ### log artifacts ###
    print('LOG ARTIFACTS')
    print('Save ds_info')
```

```
    with open(os.path.join(artifacts_dir, 'ds_info-{}.txt'.format(experiment_name)), 'w') as file:
        file.write(str(ds_info))

    print('Create and save accuracy graph')
    # accuracy graph:
    plt.clf()  # clear figure
    plt.plot(history.history['sparse_categorical_accuracy'])
    plt.plot(history.history['val_sparse_categorical_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('sparse_categorical_accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    filename = os.path.join(artifacts_dir, 'accuracy-{}.png'.format(experiment_name))
    plt.savefig(filename)

    print('Create and save loss graph')
    # loss graph:
    plt.clf()  # clear figure
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    filename = os.path.join(artifacts_dir, 'loss-{}.png'.format(experiment_name))
    plt.savefig(filename)

    print('Save raw history')
    # raw history
    with open(os.path.join(artifacts_dir, 'history-{}.txt'.format(experiment_name)), 'w') as file:
        file.write(str(history.history))

    print('Restoring weights from best checkpoint')
    # get best checkpoint
    model.load_weights(checkpoint_filepath)
    print('Evaluate model with best weights')
    with tf.device(device):
        train_loss, train_acc = model.evaluate(ds_train)
        val_loss, val_acc = model.evaluate(ds_test)

    # confusion matrix train
    print('Predict training data for confusion matrix creation')
    with tf.device(device):
        train_pred = model.predict(ds_train)
    train_pred = np.argmax(train_pred, axis=1)
    cm = np.asmatrix(tf.math.confusion_matrix(list(map(int, labels_iter(ds_train))), train_pred))

    print('Create and save confusion matrix for training data')
    plt.clf()
    plot_confusion_matrix(cm, classes, normalize=False, title='CM - train')
    plt.savefig(os.path.join(artifacts_dir, 'cm-train-{}.png'.format(experiment_name)))

    print('Create and save normalized confusion matrix for training data')
    plt.clf()
    plot_confusion_matrix(cm, classes, normalize=True, title='CM - train - norm')
    plt.savefig(os.path.join(artifacts_dir, 'cm-train-norm-{}.png'.format(experiment_name)))

    # confusion matrix test
    print('Predict test data for confusion matrix creation')
    with tf.device(device):
        test_pred = model.predict(ds_test)
    test_pred = np.argmax(test_pred, axis=1)
    cm = np.asmatrix(tf.math.confusion_matrix(list(map(int, labels_iter(ds_test))), test_pred))

    print('Create and save confusion matrix for test data')
    plt.clf()
    plot_confusion_matrix(cm, classes, normalize=False, title='CM - test')
    plt.savefig(os.path.join(artifacts_dir, 'cm-test-{}.png'.format(experiment_name)))

    print('Create and save normalized confusion matrix for test data')
    plt.clf()
    plot_confusion_matrix(cm, classes, normalize=True, title='CM - test - norm')
    plt.savefig(os.path.join(artifacts_dir, 'cm-test-norm-{}.png'.format(experiment_name)))

    print('Save data distribution in training set')
    # data distribution train
    write_data_distribution(ds_train, classes,
                            os.path.join(artifacts_dir, 'data-distr-train-{}.txt'.format(experiment_name)))

    print('Save data distribution in test set')
    # data distribution test
    write_data_distribution(ds_test, classes,
                            os.path.join(artifacts_dir, 'data-distr-test-{}.txt'.format(experiment_name)))
```

```
298
299         print('Save general information (arguments provided, runtime, ...)')
300         # other info
301         with open(os.path.join(artifacts_dir, 'info-{}.txt'.format(experiment_name)), 'w') as file:
302             file.write('Experiment name: {}\n'.format(experiment_name))
303             file.write('\n')
304             file.write('best_train_loss: {}\n'.format(train_loss))
305             file.write('best_train_acc : {}\n'.format(train_acc))
306             file.write('best_val_loss  : {}\n'.format(val_loss))
307             file.write('best_val_acc   : {}\n'.format(val_acc))
308             file.write('\n')
309             file.write('Random seed    : {}\n'.format(random_seed))
310             file.write('Training time  : {} ({}s)\n'.format(get_time_str(time_seconds), time_seconds))
311             file.write('Arguments      :\n')
312             pprint(arguments, stream=file)
313
314         print('All done')
315
316
317     if __name__ == '__main__':
318         parser = argparse.ArgumentParser(description='Train the color3Dnet.')
319         parser.add_argument('paths', metavar='PATHS', nargs='+',
320                             help='one or multiple data source path(s)')
321         parser.add_argument('--granularity', metavar='GRANULARITY', default='manufacturer_main',
322                             help='Granularity levels available: ' + str(LiDARCarDataset.GRANULARITY.keys()))
323         parser.add_argument('--split', metavar='SPLIT', type=float, default=0.2, help='data split from range (0, 1)')
324         parser.add_argument('--augmentation-split', metavar='AUGMENTATION_SPLIT', type=bool, default=False, help='If set to '
325                     'True, training is performed on augmented data only and testing is performed on non-augmented data only')
326         parser.add_argument('--strict-split', metavar='STRICT_SPLIT', type=bool, default=False, help='Applies strict split; '
327                                                                                                     'refer to thesis for more information')
328         parser.add_argument('--shuffle', metavar='SHUFFLE', type=bool, default=True, help='If set to True, the data set is '
329                                                                                           'shuffled')
330         parser.add_argument('--random-seed', metavar='RANDOM_SEED', type=int, default=None,
331                             help='Sets random seed for the run, if not provided a random number is generated')
332         parser.add_argument('--shuffle-pointcloud', metavar='SHUFFLE_POINTCLOUD', type=bool, default=False, help='If set to '
333                                                                                                                  'True, the points in each point cloud are shuffled')
334         parser.add_argument('--normalize-distr', metavar='NORMALIZE_DISTR', type=bool, default=False, help='If set to True, '
335                             'the distribution of the data is normalized, such that the number of examples used for each '
336                                                                                                 'class is the same for all classes')
337         parser.add_argument('--color', metavar='COLOR', type=bool, default=True, help='If set to True, color values are '
338                                                                                       'used')
339         parser.add_argument('--augmented', metavar='AUGMENTED', type=bool, default=True, help='If set to False, no augmented '
340                                                                                               'data is used')
341         parser.add_argument('--num-points', metavar='NUM_POINTS', type=int, default=10000, help='Specifies the number of '
342                                                                                               'points in each pointcloud')
343         parser.add_argument('--batch-size', metavar='BATCH_SIZE', type=int, default=20)
344         parser.add_argument('--epochs', metavar='EPOCHS', type=int, default=30)
345         parser.add_argument('--dropout', metavar='DROPOUT', type=float, default=0.3, help='Specifies the dropout rate from '
346                                                                                           'the range (0, 1)')
347         parser.add_argument('--device', metavar='DEVICE', default='/device:CPU:0', help='Specifies the device to be used '
348                                     'for training; some examples are "/device:CPU:0", "/device:GPU:0" or "/device:GPU:1"')
349         parser.add_argument('--tensorflow-data-dir', metavar='TENSORFLOW_DATA_DIR', default=None, help='Specifies the '
350                             'directory to store the tensorflow data generated when parsing the data set; this directory can '
351                             'be reused to save some time when reusing the same data (with same split, etc.) for a different '
352                                                                                                                 'run')
353         parser.add_argument('--artifacts-dir', metavar='ARTIFACTS_DIR', default=None, help='Specifies the directory where '
354                                                                                           'the artifacts are saved')
355
356         args = vars(parser.parse_args())
357
358         main(paths=args['paths'], granularity=args['granularity'], split=args['split'],
359              augmentation_split=args['augmentation_split'], strict_split=args['strict_split'], shuffle=args['shuffle'],
360              random_seed=args['random_seed'], shuffle_pointcloud=args['shuffle_pointcloud'],
361              normalize_distr=args['normalize_distr'], color=args['color'], augmented=args['augmented'],
362              num_points=args['num_points'], batch_size=args['batch_size'], epochs=args['epochs'], dropout=args['dropout'],
363              device=args['device'], tensorflow_data_dir=args['tensorflow_data_dir'], artifacts_dir=args['artifacts_dir'])
```