Data Management and Infrastructure for Vehicle Classification Using Commercial Off-The-Shelf LiDAR and Camera Sensors

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Science

by

Christian Edelmayer

Student ID 01617675

to the Department of Computer Sciences

at the Faculty of Natural Sciences

at the Paris Lodron University of Salzburg

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

Salzburg, 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, October, 2021

Christian Edelmayer

Danksagung

Mein Studium neigt sich mit dieser Arbeit dem Ende zu. Es waren fünf sehr spannende, interessante und lustige Jahre. Ich werde immer mit viel Freude auf diese Zeit zurückblicken.

Vielen Dank an meine Eltern, Sabine und Wolfgang, dass ihr mir dieses Studium ermöglicht habt und ich meinen vollen Fokus auf das Studium richten konnte!

Danke ebenso an meine Betreuer der Bachelorarbeit (Professor Christoph Kirsch) und dieser Arbeit (Professor Stefan Resmerita) für die durchgehende Unterstützung und die ausgezeichnete Betreuung!

Weiters möchte ich mich herzlich bei der Firma eMundo und insbesondere bei Wolfgang Brauneis bedanken. Danke dafür, dass du dieses Projekt ermöglicht hast und ich immer auf deine Unterstützung zählen konnte. Seit meinem ersten Praktikum 2018 ist eMundo ein stetiger Wegbegleiter gewesen und ich freue mich schon darauf, in Zukunft weiterhin gemeinsam an spannenden Projekten zu arbeiten!

Zu guter Letzt, ein riesiges Dankeschön an meinen Studienkollegen Sebastian Landl! Ohne dich wäre das Studium nicht dasselbe gewesen. Wir haben gemeinsam so viele Projekte und Herausforderungen gemeistert und dabei nie den Spaß verloren.

Danke für diese unvergessliche Zeit!

Abstract

The remote sensing technology LiDAR (Light Detection and Ranging) is used for measuring distances. A common use case is to detect obstacles (e.g. cars). Our goal was to use a combination of LiDAR and camera data in order to take that a step further and examine whether detailed classification of cars is possible, i.e. recognizing the car make and even the car type. 3D scans of different cars obtained by both the LiDAR and camera sensor of the iPhone 12 Pro are used in combination with machine learning. The focus of this thesis lies on data management and the infrastructure. The whole process of collecting, preprocessing and augmenting the data, as well as setting up the infrastructure is explained in detail. Our results show that the data (car makes and car types) is learnable even though the network tends to overfit, which nevertheless indicates that the data obtained is quite accurate. The results also show that our custom preprocessing and augmentation methods work well. Regarding the infrastructure, different experiments compare the usage of two different CPUs to the usage of three different GPUs for machine learning tasks and show the major benefits of using a powerful GPU.

Contents

Chapter 1

Introduction

LiDAR (Light Detection and Ranging) is a remote sensing technology for measuring distances. A lot of different use cases exist for that technology. The most exciting use case might be in the area of autonomous driving. An autonomous vehicle is often equipped with a LiDAR sensor (among other sensors) in order to detect obstacles [1]. The simple detection of obstacles works very well [2] [3], however, LiDAR (data) is normally not used for further classification. Its resolution is low compared to camera sensors and it is dependent on weather conditions [4]. The main goal of this project was to use data obtained from off-the-shelf LiDAR and camera sensors in order to examine what level of classification is possible. Since cars provide different granularities for performing classification, they were chosen as the classification subjects.

A vehicle can be classified based on different characteristics: the type it belongs to (in our case, we only consider one class, namely cars), the make (VW, Mercedes, Audi, ...) and the model in combination with the make (Audi A4, VW Golf, ...). Only classifying the type is called *Vehicle Type Recognition (VTR)*, whereas *Vehicle Make and Model Recognition (VMMR)* deals with classification based on make and model [5] [6].

The new LiDAR sensor of the iPhone 12 Pro (in combination with its camera sensor) was used for taking 3D scans of different cars. Using that data in combination with machine learning, we examined what granularity of the data could be correctly classified (i.e. different car makes or even different car types). I worked on this project with my university colleague Sebastian

Landl. The whole project was done in cooperation with the company eMundo GmbH Salzburg[1].

The main focus of this thesis lies on data management and infrastructure, which are both very important parts of machine learning projects. The work of my colleague Sebastian Landl deals with the machine learning part [7].

In this thesis, a basic explanation of the LiDAR technology along with its use cases and shortcomings will be given. Afterwards, the whole data collection process will be described in detail, with the main focus on our iOS app for collecting data. Our custom methods for data preprocessing and augmentation will be thoroughly presented. Moreover, different possibilities for setting up the infrastructure along with their advantages and drawbacks will be discussed. Our final bare metal infrastructure and the challenges we faced setting that up will be outlined. The experimental results show that data obtained from the iPhone 12 Pro is accurate enough for our neural network to learn different car makes and even the different car types with high accuracy. Not surprisingly, the neural network's ability to generalize is quite limited. The results and experiments further show that our custom preprocessing and augmentation methods work well for the considered data, as well as that a powerful GPU is very important for speeding up machine learning development.

Due to the lack of literature at the time of working on this project, using the LiDAR scanner of the iPhone 12 Pro for such a project was, at least to the best of our knowledge, kind of a novelty and could even be considered as a practical test checking whether the accuracy of the scanner proved to be good enough for real world applications. It is hoped that this work will contribute to the development of machine learning applications based on LiDAR sensors as they become available on more and more mobile phones.

[1] `https://www.e-mundo.de/de/subsidiaries/salzburg`, visited on 07/29/2021

Chapter 2

LiDAR Technology

LiDAR is an active remote sensing technology which stands for *Light Detection And Ranging*. In this chapter the basic functionality along with different use cases will be explained. Afterwards, the main limitations of the LiDAR technology will be described. At the end of this chapter, we will take a look at the LiDAR scanner of the iPhone 12 Pro and explain why this scanner is a good choice for our project.

2.1 Basic Functionality

The basic principle, also called *time-of-flight principle*, of LiDAR is very simple and easy to understand: light is emitted on to a target, then reflected back and by measuring the time it takes the reflected light to come back, (information about) the distance of the target (in relation to the source from which the light was emitted) can be calculated [8]. It is an active remote sensing technology because the energy (i.e. the light) is actively generated by the system as opposed to passive systems which detect the energy created by an external source. Actually, it is very similar to *RADAR* (Radio Detection and Ranging) with the main difference that LiDAR uses light waves instead of radio waves [9]. Here is an illustration of the basic principle:

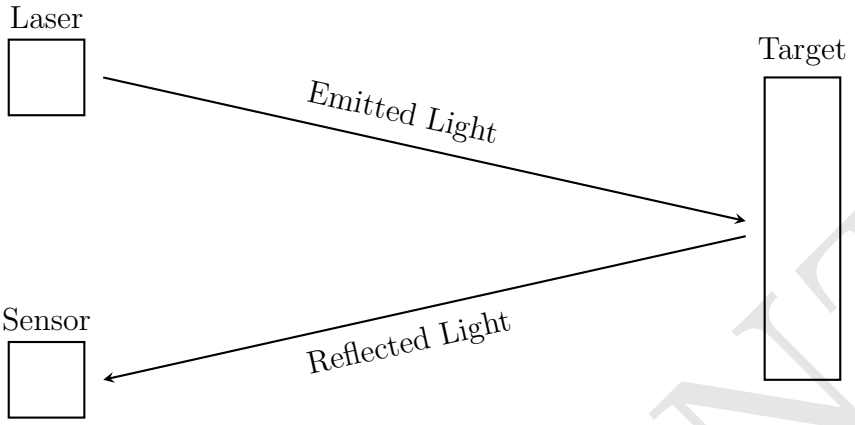


Figure 2.1: Illustration of the basic principle of LiDAR. The left side is considered as the source while the right sight is considered as the target.

One can also demonstrate this principle mathematically quite easily. The emitted light obviously travels at the speed of light, therefore the distance between the source and the target object can be calculated like this [9]:

$$D = c * (\frac{\Delta T}{2}) \tag{2.1}$$

where:

ΔT = the time it takes the light to travel from the source to the target and back to the source

c = speed of light

D = the (unknown) distance of the target object in relation to the source

To explain this in a little more detail, the speed of light is a known constant. ΔT is the information we get by using the LiDAR technology and measuring how long it takes the light to travel to the target and back. Obviously, the result needs to be divided by 2 since it includes the time for travelling forth to the target and back to the source. By simply multiplying that with the speed of light, we can obtain the distance of the target [9].

Of course, in reality things tend to get a little more complex than that simple principle. For example, modulation of frequency, phase and intensity might be needed [10]. Also, the resolution of the scan as well as the exact functionality depend on the scanner being used. However, the underlying principle is the same. It is important to note that the focus of this thesis does not lie on comparing different kinds of sensors or the explanation of the exact theoretical background of LiDAR, as there already exists a lot of literature on these topics, e.g. [11]. So, in order to understand this thesis and its results and implications, it is totally sufficient to understand the basic principle of LiDAR explained above.

2.2 Use Cases

First, we have to distinguish between two different types of LiDAR scanning: namely between **airborne** and **terrestrial** LiDAR. These two methods are quite self-explanatory. Airborne data is collected by scanning some target/area down below from up in the air (with the scanner being mounted on to a plane, for example), while terrestrial data is collected by scanning on the ground [8] [9].

LiDAR has a lot of different use cases. LiDAR can be used to scan unknown vegetation and create an accurate map of that area, for example [12]. Additionally atmospheric composition, structure, clouds, and aerosols can be studied by using LiDAR [8].

Some more applications are in the areas of agriculture, archaeology, law enforcement and military, as shown in [9].

However, one of the latest and most exciting beneficiary of LiDAR technology is the autonomous vehicle. The idea of fully autonomous vehicles is a topic humankind has dreamt about for quite some time now and that dream seems to become technologically feasible in the foreseeable future. LiDAR could be a key technology of that revolution. Since autonomous vehicles need to scan the surroundings for obstacles and threats in order to prevent accidents, a powerful scanning technology needs to be used. While there are still some obstacles ahead, LiDAR (in combination with other sensors and deep learning) is a likely candidate for that [10].

2.3 Limitations

While LiDAR offers quite some benefits like a long detection range, accuracy and being able to produce scans during night, the technology also has its limitations. One of the main drawbacks of LiDAR is its low resolution compared to camera sensors when it comes to classification of objects. Another one is that the performance of LiDAR may decrease severely in adverse weather conditions like heavy snow, rain or fog [4]. Especially in the autonomous vehicle area these are quite severe limitations, because a vehicle should be able to drive autonomously under almost every weather condition and not just on a sunny day.

Another limitation we personally experienced is that a LiDAR scan can be quite extensive and sometimes the scan contains too much of the environment. Especially if a specific object needs to be scanned, it often is not that helpful if the scan contains a lot of points of the environment, as that not only increases the size of the data enormously, but also makes the classification of the object a lot harder.

2.4 iPhone 12 Pro

Since 2020 some of the Apple products, namely the iPhone 12 Pro, iPhone 12 Pro Max, and 2020/2021 iPad Pro[1], include a LiDAR scanner. This is very interesting, since not too long ago LiDAR was a quite expensive technology and not that easily available on off-the-shelf devices.

As of now, the LiDAR scanner is mostly used for augmented reality apps and for improving the quality of images captured by the camera. However, the scanner can obviously be used for other applications as well, so we chose the iPhone 12 Pro for gathering our data. Here is how Apple describes its LiDAR scanner in the corresponding press release [13]:

[1] at the time of writing (July 2021); it is likely that there are more to come

> "The breakthrough LiDAR Scanner enables capabilities never before possible on any mobile device. The LiDAR Scanner measures the distance to surrounding objects up to 5 meters away, works both indoors and outdoors, and operates at the photon level at nano-second speeds. New depth frameworks in iPadOS combine depth points measured by the LiDAR Scanner, data from both cameras and motion sensors, and is enhanced by computer vision algorithms on the A12Z Bionic for a more detailed understanding of a scene. The tight integration of these elements enables a whole new class of AR experiences on iPad Pro."

Note that this press release is about the iPad Pro 2020, but it also applies to the iPhone 12 Pro which was released a little bit later.

Why is the iPhone 12 Pro a good choice for us? First, it might not be immediately clear from the quote of the press release since it is written quite compactly, but in other words: we do not only get information about the distance, but also about the color. Normally LiDAR data contains only geometric values, not color values. The fusion of data obtained by LiDAR and camera sensors is a well known topic, e.g. see [4] or [14]. By using the iPhone 12 Pro we do not have to worry about the fusion of these two sensors, as the fusion is done internally (more on that will follow in subsection 4.2.1). While this is a huge benefit and ultimately allows us to freely choose between using just the geometric data or both geometric and color data (in other words the fused data), this is not the main reason for choosing the iPhone 12 Pro as the scanner.

The main reason is that the iPhone 12 Pro has the advantage of being available to the regular consumer as opposed to being an extremely expensive device for industrial use only, which makes it easier to reproduce the results obtained in this project. What is more, the iPhone basically represents an 'all-in-one' solution, meaning that if we took the project further, we could in theory do everything locally on the iPhone, allowing the user to just download the app and use it out of the box. This would not have been possible if we had chosen another LiDAR scanner.

It is also interesting to note here that besides a comparison between the scanning capabilities of the iPad Pro versus an industrial 3D scanning solution [15] and a technical report on the quantification of the bias and repeatability of the iOS-based LiDAR scanner [16], not much literature on the (usage of the) LiDAR scanner of the iPhone/iPad could be found. Interestingly, the authors of the technical report also used a car for scanning. Even though they had a very different goal, their report offers details about the accuracy of the scanner, in case one wants to read more about that.

Chapter 3

Related Work

This chapter gives an overview of existing work regarding vehicle recognition (based on images) and 3D recognition, with the main focus on preprocessing, augmentation and hardware.

As far as we can tell from the existing literature, the topic of vehicle recognition from 3D data has received little to no attention so far. Most of the work on vehicle recognition that we could find is based on images, while recognition based on 3D data is scarce.

3.1 Vehicle Classification Based on Images

Since the classification (also called recognition) of vehicles based on images is a known and well researched topic, a lot of literature can be found on that. There exist VMMR systems that rely on license plate recognition [17], systems that use different machine learning methods [5] [6] [17] and even a large data set designed specifically for VMMR can be found [18].

In [19] a good explanation about the difficulties of VMMR can be found:

> "VMMR [...] is a challenging task due to the strong inter-class similarity between different makes of vehicles and significant intra-class variability among the models of the same make. Additionally, we are confronted with considerable number of car models,

including different car manufactures and models depending on the year."

Some VMMR systems rely on the recognition of license plates. However, license plates can be ambiguous (different letters or numbers can look quite similar), forged, damaged or even duplicated. These factors make a correct license plate recognition very hard under certain circumstances. Also, in order to recognize license plates from image data, very specific camera angles are needed [17] [18].

Other systems use an image of the car as the input and perform VMMR based on that given image. The most important aspects of the car (also called features) and not just the license plate are considered. Features can be extracted either manually (hand-crafted feature engineering) or automatically (deep learning). For the general recognition of the model and make, two different methods can be distinguished: the task can either be seen as a classification problem (*k-way-classification*, training a network to recognize these k different classes) or as a metric learning problem where a mapping from the images to feature vectors needs to be found. The network then calculates the corresponding feature vector of the given image and the calculated feature vector is compared to existing feature vectors via a *nearest neighbour search*, for example [6]. VMMR systems that use machine learning offer an amazing performance with an accuracy above 90% [5] [17].

There also exists work on performing VMMR during the night [20]. However, the goal described in that paper is a little different to our goal. Their goal is to recognize a specific car, which is given beforehand. For example, if model and make are given, the aim of their system is to recognize the car (e.g., when looking at CCTV footage).

Even commercial services offering VMMR exist, for example a service called CarNET[1] and a service called eyedea[2].

[1] `https://carnet.ai/`, visited on 01/09/2021

[2] `https://eyedea.ai/make-and-model-recognition/`, visited on 01/09/2021

3.2 3D Recognition

As already stated, not much literature on the recognition of vehicles using 3D data could be found. However, there exists literature on 3D recognition in general.

Preprocessing (filtering) the point cloud has been done in the past and different methods have been proposed for that. A compact overview including different filtering methods - like statistical-based filtering techniques or neighborhood-based filtering techniques - can be found in [21]. Their goal was to compare different filtering techniques in terms of runtime and quality of the respective filter by defining error metrics. Different objects, ranging from a chair and a sofa to a bowl, represented as 3D point clouds were used for that. Interesting to note here is that the authors of that paper have used the Point Cloud Library (which we have also employed as explained in chapters 5 and 6) for the implementation of some of their filters. Another approach, inspired by the so called guided image filter, has been introduced in [22]. The implementation of that filter also uses the Point Cloud Library.

Augmentation on 3D data is a known topic as well, for example, there exists work on data augmentation regarding the segmentation of 3D medical images [23]. There also exists work on automated data augmentation in order to improve 3D object detection [24]. Various augmentation methods regarding LiDAR based 3D Object Detection can be found in [25]. Even though some of these methods (especially the global augmentation methods) are quite similar to the methods (which will be explained in detail in chapter 6) we used, our methods have not been taken from this paper, as we found that paper only after already having implemented our own methods.

It is also interesting to examine the hardware used for experiments regarding 3D data and 3D recognition. For the filtering experiments in both the preprocessing papers mentioned before, [21] and [22], a PC with Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz and 16 GB memory was used. For deep learning in combination with 3D data, different GPUs have been used: the authors of the VoxNet used a NVIDIA Tesla K40 GPU in their experiments [26] while the authors of the PointNet used a NVIDIA GTX 1080 GPU [27].

Chapter 4

Data Collection

Collecting the data was a huge part of this project. We needed very specific data, more precisely we needed a lot of 3D scans of different car types from various car makes. Of course, we also needed labelled data, i.e. data with a label containing information about the car type and car make. While there exist some data sets in that area - like the *Sydney Urban Objects Dataset*[1] or the *ModelNet40*[2] - they did not give us the exact data we needed, so we had to collect the data ourselves. However, in the end that proved to be a good decision, because that allowed us to have full control over the data collection process, meaning we could fully decide how and what we wanted to collect.

In this chapter the data format of point clouds will be explained. Afterwards the iOS App, which we developed in order to scan the cars, will be introduced. Finally, the scanning process and its obstacles and challenges will be described.

[1] `http://www.acfr.usyd.edu.au/papers/SydneyUrbanObjectsDataset.shtml`, visited on 07/29/2021

[2] `https://modelnet.cs.princeton.edu/`, visited on 07/29/2021

4.1 Data Format

We use a very simple data format for storing LiDAR data, a so called *point cloud*. A point cloud is just a set of points. The reason we chose this format is because it is one of the simplest ways to store our data without any loss of information. Also, it allows us to store a variety of different information. A `ply` (*Polygon File Format*) file storing a point cloud consists of a header (containing additional information about the point cloud) and the set of points. In our specific case, we store some meta information in the header. Here is an example header of our data:

```
ply
format ascii 1.0
element vertex 4676870
max number of points: 10000000
grid size: 5000
distance threshold: 2
property float x
property float y
property float z
property float red
property float green
property float blue
property float quality
element face 0
property list uchar int vertex_indices
end_header
```

Figure 4.1: Example header of a `ply` file.

The first two lines give information about the format of the file. The third line stores the size of the point cloud. The following three lines give information about internal app parameters, which will be explained in more detail in the next section (4.2.1). Afterwards, the properties of the points are defined. For the points, we store the x, y and z coordinates as well as the r, g, b color values and a confidence (here called quality) value. The values of the x, y and z coordinates are calculated in relation to the initial position of the phone when the scanning process was started. So, the origin - i.e. point

$(0, 0, 0)$ - represents the initial position of the phone. The set of points looks like this (note that only a very small part of the whole set is shown here):

```
-8.018113 -0.11770714 5.3685446 0.7051917 0.7527006 0.78911865 0.0
-8.027219 -0.12849717 5.359598 0.60244024 0.6515469 0.7173275 0.0
-8.052642 -0.12541449 5.3248134 0.35278037 0.4420207 0.5805314 0.0
-8.073795 -0.12489437 5.296474 0.33915097 0.43156227 0.5626304 1.0
-8.090582 -0.12742327 5.2748523 0.36695296 0.45131883 0.5836314 1.0
-8.109696 -0.12706573 5.249278 0.35126668 0.43563256 0.5679451 1.0
-8.125629 -0.12897524 5.2286115 0.3592662 0.43495837 0.56333536 2.0
-8.140535 -0.13126753 5.209429 0.31445393 0.3977825 0.53044516 2.0
-8.153701 -0.13479106 5.192926 0.28504705 0.36453852 0.49065298 2.0
-8.166668 -0.13802357 5.1766024 0.2825 0.35991687 0.4867314 2.0
-8.178714 -0.14179605 5.1616654 0.28715685 0.3659231 0.48443925 2.0
-8.189876 -0.14616086 5.148083 0.26019606 0.33896232 0.45747846 2.0
-8.202048 -0.14886123 5.1326613 0.30000785 0.3718251 0.4848432 2.0
-8.212201 -0.15363798 5.120544 0.27868432 0.35050154 0.46351963 2.0
-8.221776 -0.15882896 5.1093183 0.27094707 0.33581525 0.44333532 2.0
```

Figure 4.2: Example set of points.

Note that the ordering of the points does not make a difference and any information can be encoded, meaning that more or less information could be used. Our maxim was to always store as much data as possible, just in case we needed it afterwards.

4.2 iOS App

Since the LiDAR scanner of the iPhone 12 Pro is relatively new, not a lot of apps use it yet. Also, the main purpose of most of the apps using the LiDAR scanner is augmented reality (with quite amazing results). Interestingly, there exists an app called *3d Scanner App™* [3] which allows to make 3D scans of objects and export the data in various formats. Theoretically we could have used this app, as it allows to export the data in point cloud format. However, we decided against that since we did not want to become

[3] `https://apps.apple.com/de/app/3d-scanner-app/id1419913995`, visited on 07/29/2021

dependent on an external app. We wanted to have full control over our scanning process. So, we started developing our first iOS app.

The main challenge was getting into the whole iOS infrastructure and finding information about how data from the LiDAR sensor can be accessed. Because this was a relatively new topic, not a lot of information existed at the time of development. As a starting point we used a sample project offered by Apple called "Displaying a Point Cloud Using Scene Depth"[4]. After doing a lot of research on 3D graphics, rendering and Apple's ARKit framework, we were able to understand the code. We modified and extended the code quite a lot. Creating the app was a lot of work and many different concepts (especially in the 3D graphics area) were used, so in this section only the most important aspects of the app will be introduced, as covering all material would be out of scope for this thesis. Also, we will focus more on what the app can do and not so much on how the app does that, as that would dive into areas like Swift programming and 3D graphics. Nevertheless, some details about the internal workings of the app are important to understand, which will be explained next.

4.2.1 Functionality

Apple's *ARKit* framework provides a so called `depthMap` instance property and a `confidenceMap` instance property. The `depthMap` contains information about the estimated distances, while the `confidenceMap` contains information about the accuracy of those estimates, with 0 being the lowest and 2 being the highest accuracy. As of now[5], there is no way to access the raw LiDAR data, Apple explains in the Explore ARKit 4 video [28] the internal process like that:

> "The colored RGB image from the wide-angle camera and the depth ratings from the LiDAR scanner are fused together using advanced machine learning algorithms to create a dense depth map that is exposed through the API."

[4] `https://developer.apple.com/documentation/arkit/environmental_analysis/displaying_a_point_cloud_using_scene_depth`, visited on 07/29/2021

[5] July 2021

More precisely, we can get the following values for a point:

- x, y, z values (geometric)
- r, g, b values (color)
- confidence value

It should be emphasized again here that it is not possible to access the raw LiDAR data, but only the fused data. For our specific case, that can actually be considered a great advantage, since more information (i.e. geometric and color values) can be accessed and there is still the possibility to just ignore the color values and only use the geometric values. That is also the reason why no additional fusion of the data was done, as the iPhone takes care of that internally on a low level. Nevertheless, an additional fusion of data would have been possible, since the app also takes pictures. However, there is not much sense in doing that if fused data is provided out of the box by the API of the iPhone.

So, how are these points obtained? The camera captures an image, onto which a grid is placed, with very high resolution. The grid has a given size (i.e. the number of points to be sampled), which can be modified in the app, and is calculated uniformly, meaning that the points to be sampled are placed uniformly on the image. In other words, not every pixel of the image is considered as this would lead to an enormous amount of points, but only the number of points on the grid. For the sampled points one can access the depth (via the `depthMap`) and its corresponding accuracy (via the `confidenceMap`). Whenever the camera is moved/rotated by a certain threshold, that process is repeated and the newly sampled points are stored. Note that there exists an upper limit of the number of points that are stored. This upper limit can be modified in the app, if it is exceeded, the oldest points are simply overwritten by the newest points.

Figure 4.3 shows an actual example of how a scan of a car is built, in order to illustrate that explanation above.

Figure 4.3: Illustration of how the scanning functionality works. As explained before, points on the grid are sampled and that process is repeated every time the camera is moved/rotated by a certain threshold. In this example one can see how the scan is built from the beginning until the end, resulting in a complete 360° scan of the car. Note that the number plate has been blurred.

This was a quite compact and slightly superficial explanation, however, a detailed explanation of the internal functionality of the app would need a lot of background information on other topics, which would again be out of scope for this thesis, as the app is a tool to collect the data and not the only focus of this thesis. Since the source code of the app is too big to be included in the thesis, it is made available in a dedicated GitHub repository[6].

For the following parts of the thesis, these key points should be understood and remembered:

- For each sampled point we get the following values:
  - x, y, z values (geometric)
  - r, g, b values (color)
  - confidence value
- A grid is used to sample the points.
- The size of the grid as well as the upper threshold of stored points can be modified in the app.

4.2.2 Features

Now, let us take a look at what features the app offers. It is important to keep in mind that the main purpose of the app is to collect data and to make that data collection process as easy as possible. The user interface of the app looks like this:

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

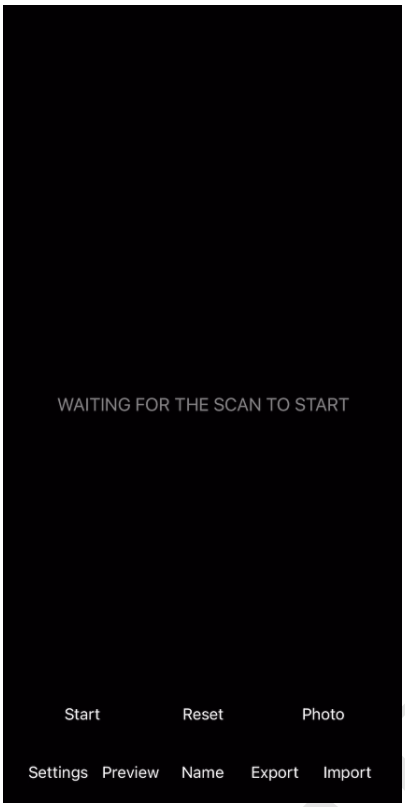


Figure 4.4: Start screen.

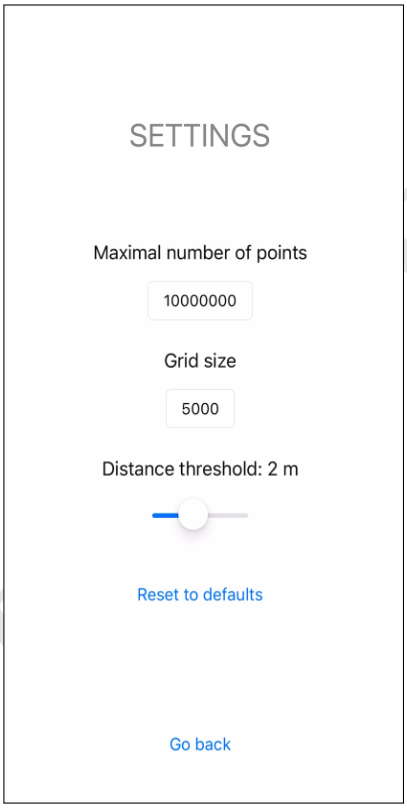


Figure 4.5: Settings screen.

In Figure 4.4 it can be seen that the user has several options. Starting the scan, resetting the scan and manually taking pictures. Note that the app does take pictures automatically when scanning, this feature was implemented in order to get as much data as possible, in case we needed it afterwards. What is more, when the scan is started, the current progress of the scan is shown on the screen, meaning the user can see how the scan is building (up), as illustrated in Figure 4.3. Scanning would also work without this feature, however, it makes the scanning process a lot easier if it can be clearly seen what has already been scanned.

The scan can (and should) also be named in order to collect labelled data. The export option allows to export the point cloud as a `ply` file and the taken pictures as a `zip` file.

In the settings (Figure 4.5), the upper threshold of points as well as the grid size can be modified. Note that the bigger these two parameters get, the bigger the resulting point cloud gets. Also, the distance threshold (in subsection 5.2.1 called online distance filter) can be adjusted within a range of zero to five metres in steps of one metre. The exact meaning of the distance threshold will be explained in detail in the preprocessing chapter (5).

A `ply` file can be imported and previewed directly in the app, as shown in Figure 4.6. This is a helpful feature if one wants to quickly check a previous scan.

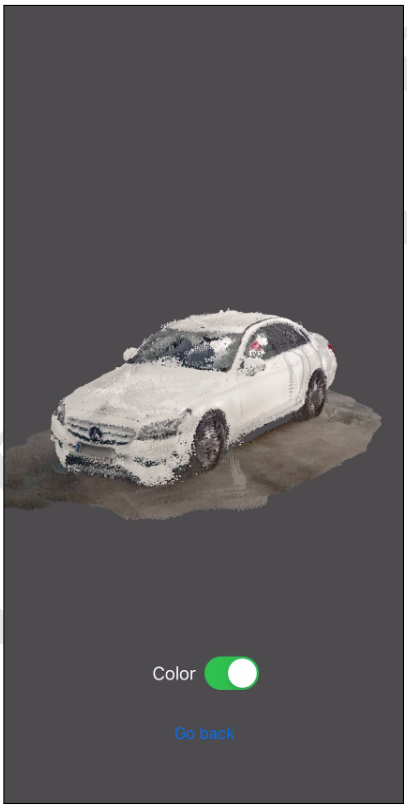


Figure 4.6: Preview of an example scan. The user has the option to toggle the color on and off. Note that the number plate has been blurred.

To sum up, the purpose of this app is to make the data collection process as easy and as fast as possible. The app has been modified and extended several times, especially after gaining some experience during the first few scans. The above described features proved to be very helpful in the scanning process, which will be described next.

4.3 Scanning Process

We scanned a lot of different car types and car makes at different car dealerships. Basically our scanning process was:

1. Asking the car dealer for permission to scan their cars.
2. Using the iPhone app to take 360° scans of the cars.
3. Using AirDrop (a service offered by Apple which allows sharing data between Apple products) to send the exported `ply` files (along with the captured pictures) to the MacBook of my colleague.
4. Labelling the data directly on the MacBook.

Even though the data could have been labelled via the naming option of the app, in reality this proved to be much slower than using AirDrop to send the scans over to my colleague and labelling it on the MacBook. The possibility to use AirDrop to send files quite seamlessly between the iPhone and the MacBook made this whole scanning process much easier and faster.

4.4 Obstacles & Challenges

During data collection, we encountered some obstacles and challenges, most of these during the development of the app, as described before. The main challenge in this part of the project was definitely to develop the app. Working with Swift and XCode was not that difficult, but the lack of tutorials and literature on the new LiDAR scanner of the iPhone made the development quite hard, since it involved a lot of trial and error. Also, finding a reasonable default configuration of the parameters took time, as we wanted to find a good balance between getting a lot of data and having reasonable scan sizes. In the end, we chose the following default values:

- Maximal number of points: 10 000 000
- grid size: 5 000
- distance threshold: 2 m

We also encountered some (minor) issues during scanning. The first one was the battery life of the iPhone and the MacBook. Especially the battery of the iPhone did often not last for the whole scanning process at a car dealer. However, this was expected, as scanning the cars is a very work intensive job for the iPhone. The simple fix for that was to bring a power bank.

Another more interesting issue was not really an issue of the app/scanner, but more of an issue of the LiDAR technology. The darker a car is, the harder it is to take an accurate scan. Since darker colors absorb more light and are therefore less reflective, the scan is less accurate [29]. Figure 4.7 shows an example from our scans, comparing the scan of a white car to the scan of a dark car. It can be clearly seen that the scan of the white car is more accurate.

Aside from that, the scanning progress went surprisingly well without any major obstacles.

Figure 4.7: Comparison of the scans of a dark and a white car.

Chapter 5

Preprocessing

Preprocessing is always an important topic in machine learning projects [30], so we developed our own preprocessing tool. At first, we used Python, which worked well in the beginning. However, after doing some research we found that there already exists an extensive open source project called *Point Cloud Library (PCL)* [31] offering a lot of functionality regarding point clouds. In order to use that and to combine our custom filters with filters offered by PCL, we switched from Python to C++, while also getting benefits in terms of speed in the process.

In this chapter a short motivation will be given why preprocessing is important in our case. Afterwards, various preprocessing methods will be explained with the focus on how each preprocessing method is implemented. In the complementary thesis about machine learning the focus lies on why these methods are useful [7].

5.1 Motivation

In order to give a short motivation about preprocessing, Figure 5.1 and Figure 5.2 show the difference that preprocessing can make.

Figure 5.1: Original scan.

Figure 5.2: After preprocessing.

Figure 5.1 shows the original scan before preprocessing. The red part in the middle is the car. However, there is a lot of unwanted noise in the scan, which should be removed. The goal of preprocessing is to remove that noise and ideally to extract only the car.

Figure 5.2 shows the scan after preprocessing. That is a huge difference. The scan illustrated in Figure 5.2 represents the object (i.e. the red car) much better than the scan illustrated in Figure 5.1. In other words, by preprocessing our data, we make it easier for the neural network to learn our data as we remove unnecessary and unwanted noise [30].

The corresponding filtering order to achieve such an extraction of the car is illustrated in Figure 5.2. The complete preprocessing pipeline used for the neural network contains one extra step at the end, namely the subsampling operation (which will be explained later in section 5.4).

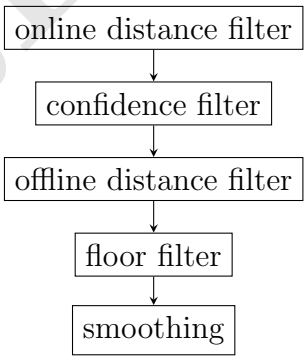


Figure 5.3: Filtering order for extracting the car.

5.2 Custom Filters

In this section our custom filters will be introduced. These are filters we thought of and implemented ourselves. The main goal of these filters is to remove as much noise as possible, especially points representing the floor and outliers. While removing the floor and outliers from 3D data is a known and well researched topic, e.g. see [32], [33] and [21], we were not able to find methods that were easy to implement. So, we decided to create our own filters with the aim to keep their implementations as simple as possible. The main benefit of our custom filters is their simple implementation, so after reading the subsequent sections, one should be able to implement these filters (more information can again be found in our GitHub repository). And not only that, but due to their simplicity, their runtimes are also quite fast. It is important to note here that these custom filters work well for our project, as we specifically developed them for our own data, so there is no guarantee that these filters will have a similar performance on other types of data.

5.2.1 Online Distance Filter

The *online distance filter* is applied on a lower level than the rest of the filters. Also, as it is an online filter it does not need access to all of the points beforehand. While in the code (of the iPhone app) it comes down to a simple condition check, this filter uses the fact that we developed the iPhone app ourselves, making it somewhat of a unique filter.

The online distance filter works like this:

1. A value $t \in \{1, 2, 3, 4, 5\}$ is given as input. This value represents the threshold in metres.
2. During scanning, the points are sampled as explained in the chapter before.
3. For every point the distance in relation to the source (i.e. the iPhone) is checked.
4. If this distance exceeds the threshold t, the point gets a confidence value of -1 (i.e. is 'filtered').

This is possible, because at the time when the points are sampled both the position of the source (i.e. the iPhone) as well as the (estimated) positions of the points are known. It would not be possible doing this one level higher, i.e. on the point cloud. That is the specialty of this filter and the huge advantage of having control over the data collection process. The reason we do not completely discard the points is that at that time we did not know whether or not we might need them afterwards, so we decided to keep them, just in case.

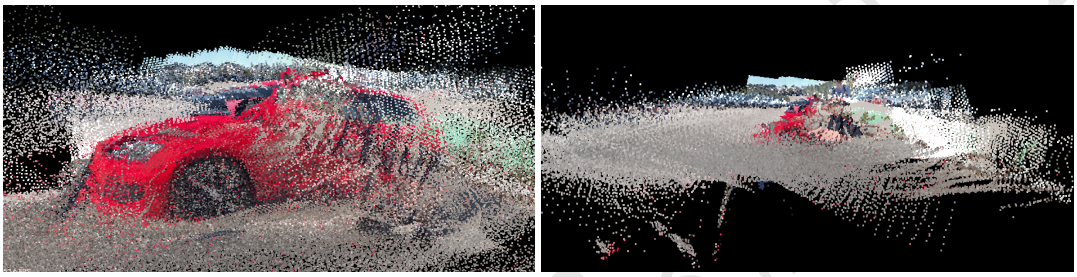


Figure 5.4: No online distance filter applied.

Figure 5.5: Online distance filter applied.

Figure 5.4 shows how a scan looks before applying the online distance filter, while Figure 5.5 shows how the same scan looks after applying it. It can be seen that this filter alone removes a lot of unwanted background noise.

5.2.2 Confidence Filter

The *confidence filter* is again a result of the data collection process. As explained before, for each point we have information about the confidence given by the scanner. A high confidence value means that the values of this point should be quite accurate, while a low confidence value means exactly the opposite. By default, 0 is the lowest confidence value and 2 is the highest confidence value. Note that we introduced a confidence value of -1 for the points being filtered by the online distance filter. Actually, we kind of misuse the confidence value for that, as we just overwrite the confidence of these points and the values of a point with confidence -1 could still be very accurate. Nevertheless, this is not much of an issue, since in the end we do not use these points anyway.

Technically the confidence filter is again a simple condition check, however, the check is done on a higher level, i.e. on the exported point cloud and not during the scanning process. We parse the confidence value for each point and if it is below the given threshold, the point is discarded, i.e. filtered out.

The following Figures show an example with three different confidence thresholds:

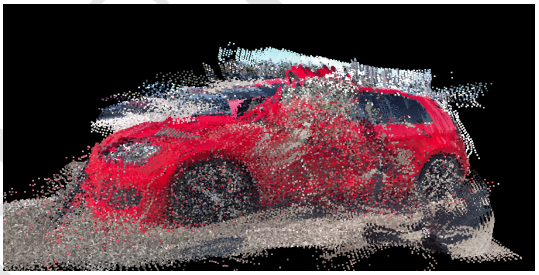


Figure 5.6: Confidence filter applied with threshold 0.

Figure 5.7: Confidence filter applied with threshold 1.

Figure 5.8: Confidence filter applied with threshold 2.

By setting the confidence threshold equal to or higher than 0, the online distance filter is implicitly applied. If only points with a confidence value of 2 are used, the car seems to be extracted quite well. However, the points on the floor should still be removed.

5.2.3 Offline Distance Filter

The *offline distance filter* removes a given percentage of points which have the largest distance to the centroid, with the centroid being the mean over all x, y and z values. In order to do that, it does need access to all of the points beforehand.

The offline distance filter works like this:

1. A threshold $t \in (0, 1]$ is given as the input.
2. The centroid of the point cloud is calculated (PCL offers a method for doing that).
3. Also using PCL, for each point a tuple with the first entry being the euclidan distance to the centroid and the second entry being the point itself is created.
4. This new list (called vector in C++) of tuples is sorted according to the first element (i.e. the distance to the centroid) in ascending order.
5. We iterate over this sorted list and add the points to a new (filtered) point cloud. However, we do not iterate over the entire list, we stop

early to omit the given percentage of points. More concretely, if the original point cloud is of size s, we only add the first $s * (1 - t)$ points.

The main purpose of this filter is to remove unwanted background noise. Since this noise is per definition in the background, it is reasonable to assume that these points tend to have the largest distance to the centroid.

Figure 5.9: No offline distance filter applied.

Figure 5.10: Offline distance filter with threshold 0.1 applied.

Figures 5.9 and 5.10 show the difference that applying the offline distance filter makes. We can see that the offline distance filter removed the noise a bit. While there is still some noise (especially the floor) visible, the car is extracted pretty well, especially compared to the unpreprocessed scan.

The offline distance filter could be improved by somehow taking the shape of the car into account, which is currently not the case. In order to do that, one has to first carefully think about how to automatically get information about the shape of a car, as cars can have very different shapes. This is an idea for future work in order to improve the custom filters.

5.2.4 Floor Filter

Removing the floor or the ground from 3D data is, as already stated, a common problem, so there exists literature on that, e.g. [32]. However, since we know exactly what our data looks like, we decided to implement a simple solution by ourselves, as shown in Figures 5.11 and 5.12. Our *floor filter* works like this:

1. A positive real value is given as the input. This value represents the threshold in metres. Let us call this value t.

2. The lowest value of all y coordinates in the point cloud is calculated. Let us call this value y_{min}.

3. All points with a y coordinate lower than $y_{min} + t$ are removed from the point cloud.

Figure 5.11: No floor filter applied.

Figure 5.12: Floor filter with threshold 0.3 (metres) applied.

Even though the concept of this filter is quite simple, it almost completely removes the floor. We implemented this filter under the assumption that the floor has the lowest height (i.e. the lowest values on the y axis) and that through some trial and error a suitable threshold can be found.

5.3 Point Cloud Library Filters

Preprocessing the data using only our custom filters does extract the car pretty well in most of the cases, however, there are still some missing operations like a smoothing operation, for example. In this section a few methods offered by PCL will be explained.

5.3.1 Voxel Grid Filter

The main purpose of the *voxel grid filter* is to downsample the point cloud or in other words to decrease the number of points. This is actually very important as our data tends to get quite large. Some of our scans have a size of around 500 MB and contain millions of points.

A voxel grid is a basically just a grid of 3D boxes, which are placed over the point cloud [34].

The voxel grid filter works like this [34]:

1. The filter takes three positive reals as input arguments. These represent the resolutions in the x, y and z directions of the 3D boxes.
2. For each voxel (3D box) the centroid is calculated.
3. The points inside a voxel are replaced (i.e. downsampled) by the centroid of the voxel.

Note the important detail in step 3: the points are replaced by the centroid, meaning their geometric values are changed. This could potentially result in a loss of (geometric) information.

Figure 5.13: No voxel grid filter applied.

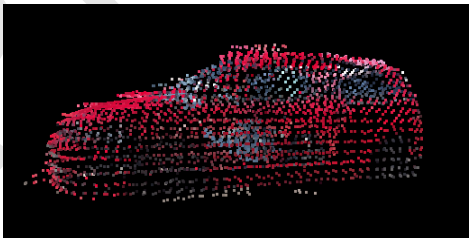


Figure 5.14: Voxel grid filter with 0.1 for x, y and z directions applied.

It is clearly visible that the scan illustrated in Figure 5.14 (with the voxel grid filter applied) contains fewer points than the scan illustrated in Figure 5.13. The benefit of this is that the data becomes smaller and easier to handle, however, it is important to keep a good balance between downsampling the data, while also keeping enough information about the shape of the object. The main drawback of this method is that it does change the geometric values of points. Also, the number of points of the resulting (filtered) point cloud cannot be defined. More information on that issue will follow at the end of this chapter in section 5.4.

5.3.2 Smoothing

Due to minor measurement uncertainties, surfaces can get a bit noisy. That is also the case for some surfaces in our car scans. In order to fix that, we use the *smoothing* operation offered by PCL. The inner workings of this method are quite complex and rely on higher order polynomial interpolation. For our use case, it is sufficient to just use the operation with the default values used in the example of the documentation [35], with the only difference that the search radius can be set dynamically. The higher the search radius, the smoother the scan gets. However, that also increases the runtime of the smoothing operation.

Figure 5.15: No smoothing operation applied.

Figure 5.16: Smoothing operation with a search radius of 0.01 applied.

The surface of the scan illustrated in Figure 5.16 looks much smoother and less noisy than the surface of the scan illustrated in Figure 5.15.

5.3.3 Radius Outlier Removal Filter

Since no scan is perfect, there is always the chance of having outliers in the data. While these might not be a huge issue, it is still a good idea to have a way of getting rid of those. Actually, that is also what the offline distance filter, which was explained before, does. The *radius outlier removal filter* offered by PCL uses a different method for that. One can specify how many neighbors each point needs to have within a specific radius. If a point contains less neighbors in the radius, the point is removed [36].

5.3.4 Statistical Outlier Removal Filter

Similar to the radius outlier removal filter, the purpose of the *statistical outlier removal filter* is to remove outliers, too. However, another method is used.

The statistical outlier removal filter works like this: first, the number of neighbors to analyze for each point is specified (the first argument). Then the standard deviation of the mean distance to the query point is calculated. If the distance of a neighbour to the query point is larger than the standard deviation multiplied by a so called standard deviation multiplier (the second argument), the point is removed [37].

5.3.5 Pass Through Filter

The purpose of the *pass through filter* is to filter along a specified dimension. The principle is very simple, one can specify the dimension (x, y or z) and a corresponding range. If the value of the specified dimension is outside of the corresponding range, the point is removed. For example, if the user specified the y dimension with a range of $[0, 2]$, then a point with a y coordinate of 1 is not removed, while a point with a y coordinate of 5 is removed [38].

5.4 Subsampling

The final part of preprocessing deals with *subsampling*. As already explained, one big issue with our 3D data is that it tends to get quite large. Our unpreprocessed point clouds contain millions of points. This would be just too much to handle in terms of data management and also for the training phase of the neural network. Therefore, the data needs to be subsampled. This was already mentioned in the explanation of the voxel grid filter. One problem of the voxel grid filter is that we cannot specify the number of points of the resulting point cloud. However, it is important to have a normalized input size, meaning that the input data should be of the same size. Therefore, we implemented a subsampling method which allows us to define the exact number of points to subsample to.

It works like this:

1. The number of points to subsample to is specified. Let us call this number s.
2. If s is greater than or equal to the size of the point cloud, no subsampling is performed.
3. Else, s points are randomly and uniformly chosen from the point cloud.

Figure 5.17: No subsampling applied.

Figure 5.18: Subsampling to 10 000 points applied.

This scan illustrated in Figure 5.17 consists of 2 291 934 points, while the subsampled scan illustrated in Figure 5.18 consists of only 10 000 points. This is quite a big reduction of points, however, the overall structure of the vehicle is still recognizable, at least to the human eye. A good trade-off between data size and accuracy needs to be found, since reducing the data size ultimately also means reducing the accuracy of the scan. As always in computer science, there is no free lunch.

Chapter 6

Data Augmentation

In order to explain what data augmentation is and why it is important, let us take a look at what data and how much of it we collected.

We collected about 400 (3D) scans of different cars under the following weather conditions:

- sun
- clouds
- snow
- light rain

The scans consist of the following car makes:

Car make	**Number of scans**
Ford	118
VW	76
Audi	39
Mercedes	36
Volvo	24
Citroen	17
Seat	14
BMW	13
Porsche	12
Others (20 makes)	45

We used that data for training and evaluating the neural network. Unfortunately, the network overfitted, meaning that it could learn the training set almost perfectly, but was not able to generalize well. One approach for mitigating overfitting is to expand the data set [39]. This was kind of a problem, because we did not have any more and getting more would have meant to repeat the scanning process described in the second chapter. Of course, this would have been a possibility, but our guess was that if 400 scans are not enough, we would have to collect well over 1 000 scans in order to make a great difference and that would have been very time intensive. Instead, we chose to try data augmentation.

By augmenting the existing data set, more examples can be generated from the original set without manually collecting new data. In other words, 'new' scans are artificially created. Data augmentation on images is a known topic and can for example be done via geometric or color transformations [40]. Despite having a different format (3D instead of 2D data), we decided to also try geometric and color transformations.

The different methods we used for data augmentation will be explained in this chapter. Again, the focus lies on how these operations are implemented, while an explanation of why these augmentations are useful can be found in the corresponding machine learning thesis [7].

Similar to preprocessing, data augmentation is also a well researched topic [40]. While data augmentation is mostly done on 2D data, there exists some literature about data augmentation on 3D data as well, e.g. [23]. So, data augmentation on 3D data is not a completely new topic. There even exists work on LiDAR based 3D object detection [25] with the focus on detection of cars. However, their goal is to detect cars, while our goal is to classify them even further. Also, as far as we know, doing data augmentation on 3D scans of cars (i.e. on point clouds representing cars) with tansformations on both the geometric and color values is a novelty.

For the geometric transformations we mostly used the functionality offered by PCL. Regarding the color, we implemented custom transformations. Note that initially we implemented these for augmentation only, however, in the end we used some of the color transformations also for preprocessing. The focus of our own implementations lies again on simplicity. Important to note

is that our own transformations (the color transformations) are tailored to our data, i.e. we wanted to augment our car scans in such a way that we do not destroy the shape, color or main characteristics of the car. More details on that will follow in the subsequent sections.

6.1 Geometric Transformations

Geometric transformations on our point clouds modify the x, y and z (geometric) values of the points while the r, g and b (color) values remain unchanged. The idea is that by rotating/translating the geometric values of the points, the whole point cloud is modified and therefore new data is generated.

In order to understand these transformations in more detail, some concepts need to be explained. In our case, the geometric information of points is represented by vectors with three coordinates (x, y, z). These vectors can be transformed by using a transformation matrix. The simplest matrix would be the identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The identity matrix represents no transformation at all, it leaves the vector unchanged:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

In our specific case, we have a $4x4$ transformation matrix, which looks like this:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So, what parts of this transformation matrix represent the translation and the rotation? The red part defines the rotation:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The green part defines the translation:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The main benefit of using PCL for these transformations is that it takes care of creating the corresponding transformation matrix. Only the desired values for the translation and/or rotation need to be specified and the transformation matrix is automatically generated by PCL [41].

6.1.1 Translation

A *translation* of the point cloud can be defined on the x, y and z axis. It is also possible to define the translation on two or even all three axes. This transformation takes three values as input, which correspond to the desired translation on the x, y and z axes. So, the relative positions of the points to each other stay the same, however, the absolute values of all points are changed.

6.1.2 Rotation

Similar to the translation, a *rotation* can also be defined on the x, y and z axis. Internally (i.e. in the transformation matrix) the unit used for rotation is radians. However, since for the user it is easier to define the rotation in degrees and degrees can be easily converted into radians, the unit of the input (in our preprocessing/augmentation tool) is degrees.

Figure 6.1: Without rotation.

Figure 6.2: With rotation (180 degrees) on the y axis.

The difference between the scan illustrated in Figure 6.1 and the scan illustrated in Figure 6.2 is a little more visible compared to the translation, even though the structure of the point cloud remains unchanged, as the same transformation is again applied to every point.

6.1.3 Random Translation

This transformation is quite different compared to the translation and rotation because it changes the structure of the point cloud. *Random translation* works like this:

1. It takes two values a and b as input. a defines a lower limit and b defines an upper limit, i.e. a range $[a, b]$ is defined.

2. For each point of the point cloud, three random values $x_r, y_r, z_r \in [a, b]$ are chosen.

3. These values are used to transform the x, y and z coordinates of the point:

$$x = x + x_r$$
$$y = y + y_r$$
$$z = z + z_r$$

The interesting thing is that the points and even the geometric values of a single point are transformed randomly and independently of each other, resulting in a change of structure of the point cloud, as illustrated in Figures 6.3 - 6.5.

Figure 6.3: Without random translation.

Figure 6.4: With random translation with range [-0.05, 0.05].

Figure 6.5: With random translation with range [-0.15, 0.15].

It can clearly be seen that the bigger the input range is, the more structure is lost. When applying this transformation, one has to find a good trade-off between keeping enough structure of the data and modifying the data via a big enough range in order to create data that is not too similar to the original data.

6.2 Shuffle

Shuffle is a very simple operation and corresponds to just randomly changing the order of the points in the point cloud. An attentive reader might now be a bit confused as it was stated earlier in this thesis that the order of points in a point cloud does not matter. This is still true, which is why there is no difference visible when looking at the illustration of the original scan and a shuffled scan. However, for the network the order of the points might make a difference. If that is the case, 'new' data can be generated very easily by just shuffling the existing data.

6.3 Color Transformations

As stated earlier, not only the geometric values can be transformed, but the color values can be transformed as well. The challenging part here is to transform these values realistically, i.e. the number plate should not turn out to be red in the end. We implemented four different methods, which will be explained down below.

6.3.1 Hue

The purpose of this transformation is to (randomly) change the color of the car without changing the color of the tires or the number plate, for example. The RGB color model is what we use to store our data, however, there is also the HSV color model:

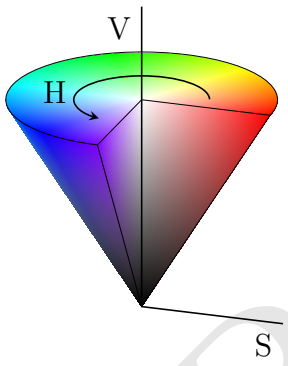
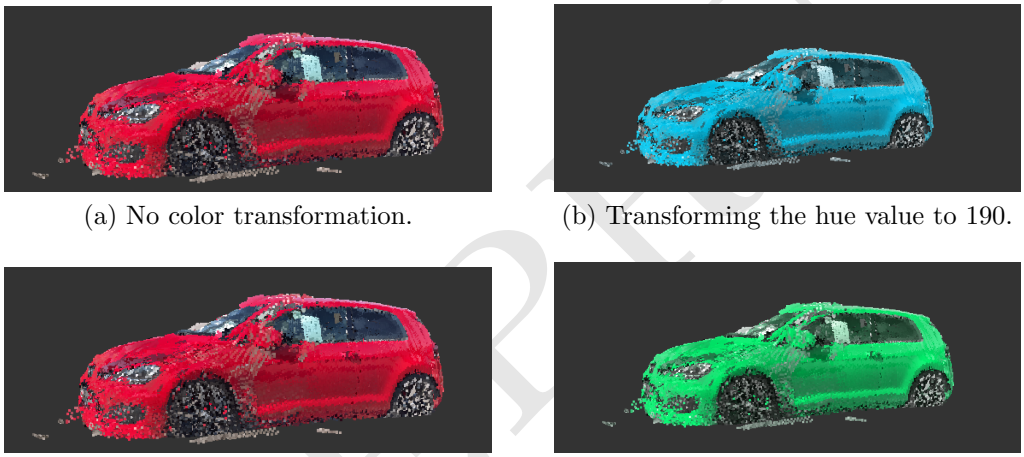


Figure 6.6: HSV color model.

From this illustration it can be seen that this color model is much better suited to perform the desired transformation, i.e. by changing the hue value. HSV is a cylindrical representation of the color space, meaning by just changing the hue value most of the visible colors will be shifted to different colors. However, colors like black, white and grey will remain the same, i.e. the tires, as we do not want to change the color of the tires, but only the color of the car. If the car has multiple colors, all colors get changed accordingly and the color composition of the car stays the same.

Internally this transformation works like this:

1. Either a specific hue value is provided or a random hue value in the range of $[0, 359]$ is chosen.
2. The RGB values are converted to HSV values.
3. The hue value is changed to the value obtained in step 1.
4. The (new) HSV values are converted back to RGB values.



(a) No color transformation.

(b) Transforming the hue value to 190.

(c) No color transformation.

(d) Transforming the hue value to 147.

Figure 6.7: Illustration of the transformation of the hue value.

It can be seen in Figure 6.7 that the change of color still looks quite realistic and the color of the tires does not change.

6.3.2 Greyscale

This transformation transforms the colored scan to a greyscale scan, as shown in Figure 6.8. The internal functionality works like this:

1. The RGB values are added together: $sum = r + g + b$
2. The average is taken: $avg = \frac{sum}{3}$
3. The RGB values are updated:
   - $r = avg$
   - $g = avg$
   - $b = avg$

(a) No color transformation.

(b) Transforming the colors to greyscale.

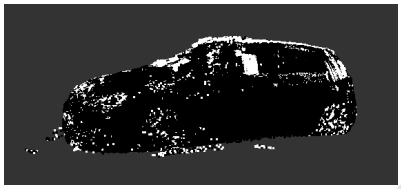
Figure 6.8: Illustration of transforming the colors to greyscale.

6.3.3 Black & White

This transformation transforms the colored scan to a black and white scan, as shown in Figure 6.9. Similar to the greyscale transformation, it works like this:

1. The RGB values are transformed to greyscale (as described before).
2. For each of those greyscale values it is checked whether the value is closer to 0 or closer to 255, i.e. if the value is less than 128, it is set to 0, otherwise it is set to 255.

(a) No color transformation.



(b) Transforming the colors to black and white.

Figure 6.9: Illustration of transforming the colors to black and white.

6.3.4 Transform Random

This transformation changes the color values randomly and independently, as shown in Figure 6.10. It works almost like the random translation explained in subsection 6.1.3, only on color data:

1. It takes two values a and b as input. a defines a lower limit and b defines an upper limit, i.e. a range $[a, b]$ is defined.

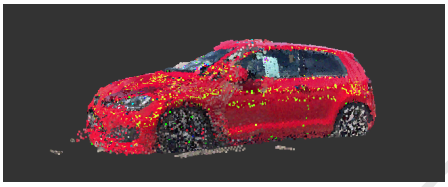
2. For each point of the point cloud, three random integer values $r_r, g_r, b_r \in [a, b]$ are chosen.

3. These values are used to transform the r, g and b values of the point:

$$r = r + r_r$$
$$g = g + g_r$$
$$b = b + b_r$$

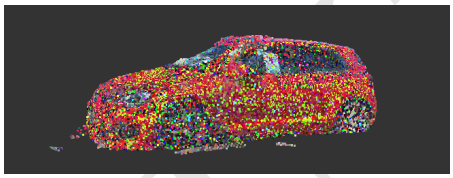
4. For each new color value it is checked whether the value is in the range of 0 and 255 (as this is the allowed range for RGB values). If the value is less than 0, it is set to 0. If the value is greater than 255, it is set to 255. A wrap around here could lead to very drastic color changes, which is not what we want. We want to have control over the randomness with the range that is defined.

(a) No color transformation.



(b) Random transformation with range [-10, 10] applied.

(c) No color transformation.



(d) Random transformation with range [-50, 50] applied.

Figure 6.10: Illustration of the random transformation.

It can again be seen that the bigger the input range is, the greater the loss of (color) information is.

Chapter 7

Infrastructure

Machine learning is tightly coupled to a corresponding infrastructure. Finding a suitable configuration of the infrastructure is essential and can make or break the whole project [42] [43].

In this chapter the needed requirements regarding the infrastructure for our project will be explained. Different infrastructure options will be presented and compared to each other. Afterwards, our own experience with setting up the infrastructure will be described, as well as the major obstacles. The chapter will end with a detailed overview of our final setup.

7.1 Requirements

At the beginning our infrastructure consisted simply of Colab notebooks offered by Google. For the very first test runs that worked fine, however, as the size of both the network and the data increased, the limitations of these notebooks were soon reached. So, for the next steps we had to think carefully about the needed infrastructure and its requirements. The main requirements were:

- Storage space
- A powerful GPU
- Availability
- A platform for visualising/storing results

Let us take a look at these requirements in more detail.

7.1.1 Storage Space

Since the gathered data is quite big (one scan can get as big as around 500 MB without preprocessing applied), a lot of storage space is needed. Of course, there is always the alternative to store the data remotely and not directly on the server. However, the main drawback of that is that if we need the data, which we obviously do when training the network, we first have to download it temporarily on the server, which might be a bottleneck and therefore increase the overall time of training. In our certain use case it makes more sense to store the data locally on the server. It is sufficient for us to use a HDD instead of a SSD, which would have increased the cost of the infrastructure.

Note that managing the data is also an important topic which needs to be taken care of. As different preprocessing and augmentation methods are applied, a lot of 'new' data is generated. New data needs to be kept separate from the original data and labelled accordingly. It is very important not to mix different kinds of data (i.e. data that has been preprocessed differently). More on that will follow in the section 7.5.

7.1.2 Powerful GPU

GPUs (Graphics Processing Units) were originally introduced and used for computer graphics. While they are still used for that, nowadays they are also used for general purpose applications. When it comes to numerical computations, GPUs tend to outperform CPUs by quite a large margin [44]:

> "The reason why GPUs have floating point operations rates much better even than multicore CPUs is that the GPUs are specialized for highly parallel intensive computations and they are designed with much more transistors allocated to data processing rather than flow control or data caching".

Because of that, a powerful GPU is needed for the training phase of the neural network. As the network becomes bigger and bigger, i.e. a deep neural network, the training time increases significantly. That is the reason why a

powerful GPU is very useful. What is more, the GPU preferably should offer a lot of RAM, because more RAM allows for either a bigger network or for a bigger batch size.

7.1.3 Availability

Availability is important in almost any project. Even though we do not offer a time critical application, the infrastructure should still offer a high amount of availability since we want to be able to start and access runs at any time without unexpected down times.

As already stated, we used Colab notebooks in the beginning. The main problem with that was that we did not have an unlimited amount of runtime, i.e. our runs got cancelled after a certain amount of time. At first this was not a problem, however, with time our network became bigger and bigger and at some point we were not able to finish the training phase anymore. Therefore, availability is a very important requirement and should allow us to complete training phases that might even take days.

7.1.4 Platform for Visualising/Storing Results

Machine learning is a process that requires a lot of iterations [45]. Therefore it is important to be able to analyse and to interpret the results. We wanted to have a platform included in our infrastructure which allows to store and visualise the results. The added benefit of such a platform is that if the configurations of the runs are saved as well, all the results become reproducible, which is always of the utmost importance in machine learning projects.

7.2 Options

When thinking about a suitable infrastructure for a machine learning project, there are various possibilities available. Each one has its advantages and drawbacks and the best option always depends on the specific project, in other words an optimal general solution does not exist. It is important to note here that these explanations are a result of our own experience.

7.2.1 Software as a Service (SaaS)

Software as a Service (SaaS) offers basically the whole application/platform out of the box. Nowadays there also exists the term *Machine Learning as a Service (MLaaS)*. Services like these are offered by Google, Amazon, IBM and Microsoft, just to name a few. As a user, one does not need to configure the infrastructure or even think about it. This can obviously be a huge advantage, especially if there is a lack of knowledge about that. However, on the other side one becomes completely dependent on the provider and also sacrifices a lot of flexibility. If there is the need for a specific feature, it might not be possible to implement it in the desired way, if the underlying infrastructure (which often cannot be modified) does not allow that. We did not use SaaS for our project, because it does abstract away almost all of the infrastructure and part of this project was to set up and manage the infrastructure, so we preferred setting up the infrastructure by ourselves.

7.2.2 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) is somewhat of a trade-off between SaaS and bare metal. While it does not provide the application out of the box, the infrastructure is provided. Again, the main providers are Google, Amazon, IBM and Microsoft.

We started with a **p2.xlarge** instance provided by Amazon EC2[1] (which is part of AWS). We chose this particular instance because it offers a powerful NVIDIA Tesla K80 GPU (more on that will follow in subsection 7.7.1). The initial settings of the server can be configured on the AWS website. It is important to note here that we did not have direct access to that configuration interface. At our company, we asked for that specific instance and afterwards we were provided with `ssh` access to the machine. In other words, we could not perform any advanced configurations on the machine. If we wanted to do these, we needed to ask the person (at our company) responsible for managing the AWS instances.

After getting access (via `ssh`) to the machine, everything was working just fine out of the box. So obviously a great benefit of that option is that the

[1] `https://aws.amazon.com/ec2/instance-types/p2/`, visited on 07/29/2021

infrastructure is provided in such a way that it can be immediately used. Another benefit is that some parts of the configuration are quite easily customizable, if one has access to the configuration site. For example, in order to increase the storage space of the server, one only needs to specify that on the configuration page available in the browser, and the storage space is increased almost immediately. This is a huge benefit, especially if the infrastructure needs to be scalable. Also, when using IaaS, one does not need to worry about the functionality of the hardware. For example, if for some reason the GPU breaks, the infrastructure provider (in our case Amazon) takes care of that, not the user.

On the other hand, we also encountered some drawbacks with this option. Firstly, it does come with a price, which should not be much of a surprise. Another thing was the missing flexibility. Whenever we wanted to open a port for example, we had to get in touch with the person at our company in charge of managing the AWS instances. Since we only had `ssh` access to the machine, these problems were quite tedious. One time we did a simple reboot and the server did not start afterwards. Without having direct access to the server or the corresponding interface, we were not able to fix that problem by ourselves. Note that this experience might have been different with complete access to the configuration interface.

We also had some issues with restrictive firewall settings of the server. What is more, initially we only had about 100 GB of storage space. In order to fulfill the requirement regarding the storage space stated before, we would have needed at least around 1-2 TB of storage space. While it would have been possible to increase the storage space very easily, it would again have been quite expensive.

In summary, this option offers some excellent benefits, especially if one does not want to worry about the infrastructure and the underlying hardware. On the other hand, if for whatever reason direct access to the infrastructure/hardware is needed, it might get tedious.

7.2.3 Bare Metal

Bare metal means setting up everything by oneself: all the infrastructure including power and networking as well as the actual location of the server, since a suitable location needs to be found.

Let us start with the disadvantages here: it obviously is a lot of work and quite some knowledge about the topic is needed. Also, suitable hardware needs to be bought and configured. One has to make sure that all the different parts work in combination with each other. For example, we had problems using our GPU in combination with the main board, because the GPU needed to allocate more memory than the main board supported. The solution for that problem was to use another main board. Such problems would never arise in the other two options, at least not for the user, with a bare metal setup they are quite common. Scalability is also not provided. If the capacity of the infrastructure needs to be increased, one has to take care of that manually. And of course, if some part of the hardware breaks, there is no infrastructure provider to turn to, as the infrastructure is provided by oneself.

While it is quite a challenge to get the bare metal infrastructure up and running with high availability, once a stable setup is reached, it does offer some amazing benefits. In our case, the main benefit was the complete control over the infrastructure. Whenever we needed to open a port, change firewall settings or add an additional hard drive, we could do that without any delay or missing authorization. Also, we had no more monthly costs like we had with the IaaS option. However, it is important to note here that we already owned a lot of the hardware we decided to use for the setup. If we had to buy everything from scratch, it would have been much more expensive. One thing we personally learned from the shift from IaaS to bare metal is that it is dangerous to think that bare metal is much cheaper than IaaS. This might be true, if one already owns a good amount of hardware (like we did) or in the long run (after months/years), but is definitely not always the case (especially in the short run).

To sum up, this option only comes into question if enough knowledge about that topic is available. When thinking about switching to bare metal (or also starting with this option), it is important to first carefully think about the

needed hardware and also the amount of work the project takes. By choosing bare metal one sacrifices a lot of the comfort the cloud options provide, on the other hand, it offers much more control over the infrastructure.

7.3 Kubernetes & Kubeflow

As explained earlier in this chapter, one requirement of the infrastructure was to have a tool for visualising and storing the results of different machine learning runs. After some research, we discovered that Kubeflow on top of Kubernetes might be a good choice.

It is important to note here that Kubernetes is a really big platform and covers a lot of different topics, an introduction and detailed explanation of Kubernetes would probably require a thesis for itself. Also, in the end we decided not to use Kubernetes, as we encountered a lot of problems. Therefore, this section will only shortly explain the experience we had with Kubernetes and Kubeflow, as a more detailed explanation would be out of scope and also not relevant, since the final setup does use another tool for that.

In [46] a compact summary of Kubernetes can be found:

> "Kubernetes is an open source platform that defines a set of building blocks which collectively provide mechanisms for deploying, maintaining, scaling, and healing containerized microservices."

There exist a lot of different tools that can be used in order to set up a Kubernetes environment (also called *cluster*). We tried `microk8s`[2], `minikube`[3] and `k3s`[4]. Normally one can set up the initial configuration quite easily, an explanation for that can be found in the corresponding documentation of the used tool. However, in order to manage a stable Kubernetes cluster, a lot of knowledge is required.

Kubeflow builds on top of Kubernetes and tries to simplify machine learning tasks. It offers support for Jupyter notebooks, pipelines, hyperparameter

[2] `https://microk8s.io/`, visited on 07/29/2021
[3] `https://minikube.sigs.k8s.io/docs/start/`, visited on 07/29/2021
[4] `https://k3s.io/`, visited on 07/29/2021

tuning and much more.

In the end, we decided against Kubernetes and Kubeflow, because we always encountered new problems and just could not focus on the tasks which we would have used Kubernetes and Kubeflow for. We tried setting up a suitable cluster for quite some time and in the end we even had a somewhat working cluster consisting of a few computers (also called *nodes*) using `k3s`. Sometimes that cluster was working fine, but new problems arose almost daily. Suddenly a pod failed or a service was not accessible anymore. Since we are no Kubernetes experts, it always took a lot of effort fixing these problems. These delays and down times contradicted the requirement of availability and therefore we decided against using Kubernetes and Kubeflow.

What is more, Kubernetes is a big piece of software consisting of many parts, so by installing Kubernetes one also introduces a lot of external dependencies. We encountered that particular problem: at one point we had a working setup of Kubernetes and Kubeflow running. It was the end of the week and we decided to start a few machine learning runs over the weekend and to look at the results on Monday. During the weekend we left the setup untouched, so nothing was changed. Starting on Monday, out of the blue we were suddenly not able to access the Kubeflow dashboard anymore, meaning we also could not access the results of the runs. It took us a few days of research until we found a likely reason for that: it turned out that the implementation of Kubeflow we used had a dependency on an external DNS service and this service just disappeared over the weekend. After some more research, it turned out that this DNS service was provided by a company that faced some sort of scandal shortly before that, causing a third of the employees to suddenly leave [47]. Our guess is that one of these employees that quit was responsible for that service.

While we do not know if our guess is true, the problem described above stays the same: having a lot of external dependencies can suddenly become an issue, disrupting the workflow and stopping the progress almost completely. Kubernetes is a great tool for sure, if one gathers enough knowledge about it. For us, the drawbacks and problems definitely outweighed the benefits, so we decided not to use it.

7.4 MLflow

After that troublesome experience with Kubernetes and Kubeflow, we looked for an alternative and quickly found MLflow[5], which is also an open source project with the main purpose of simplifying machine learning development.

The reason that convinced us to use MLflow was that it just worked out of the box. After the installation, the user interface was available and working. This was quite an opposite experience compared to Kubernetes and Kubeflow. Using MLflow we got further in about two days than we got after about a month trying to use Kubernetes and Kubeflow. It should be noted here that this statement just reflects personal experience and does in no way mean that Kubernetes is bad or not recommendable. Kubernetes offers a vast amount of functionality, while MLflow is targeted at machine learning only. In our case, that was exactly what we needed, so in the end MLflow just proved to be the better choice for us.

MLflow offers a lot of functionality tailored to machine learning development, the feature we used the most was the tracking feature. That allowed us to track all the necessary parameters like accuracy, weights, confusion matrices and much more. By doing that, we also ensured that our runs can be reproduced at any time, which is very important.

Another thing that should be stated here is that the usage of MLflow does in no way influence the results of the runs. It is a platform that simplifies parts of the machine learning process, however, the same runs could in theory be started without any platform like MLflow, therefore it was just shortly introduced in this thesis.

7.5 Data Management

As already explained, our data is quite big, so we had to carefully think about how to manage the data on the server. The first decision was simple, we wanted to store the data directly on the server. The unpreprocessed data has a size of about 125 GB. Our infrastructure setup contains a 2 TB HDD

[5] `https://mlflow.org/`, visited on 07/29/2021

and a 1 TB HDD for storing the data.

The next decision we had to make was how to handle the preprocessing and augmentation of data. Depending on the respective preprocessing operations, preprocessed data might become a lot smaller. Here we also decided to do that directly on the server, as we had enough storage space and other options, like doing that remotely, would have caused a bottleneck whenever the server needed access to the data.

However, since we tried a lot of different preprocessing and augmentation methods, we needed an efficient pipeline for that. Therefore, we wrote a Python script which internally uses the C++ tool explained in the preprocessing (5) and augmentation (6) chapters. In the end, the pipeline looked like this:

1. The path of the source data is specified (i.e. the data that should get preprocessed/augmented).

2. The target directory is specified (i.e. where the preprocessed/augmented data should be stored).

3. The arguments for preprocessing/augmentation are specified. Since for some cases random numbers are useful, the script allows input arguments like `{a,b}` where a defines the lower bound, b the upper bound and internally a random number in this range is generated.

4. The tool parses all this information, uses the preprocessing/augmentation tool which was explained in the chapters before and also encodes all the necessary information in the corresponding file names.

5. Finally, the newly preprocessed/augmented data is stored at the target location. The name of each file contains all information needed in order to understand the operations that were performed. This information is also available in the header of the `ply` file.

This pipeline for data management proved to be quite efficient and fast. If certain data is needed for the neural net, just the above described arguments are needed in order to start the script. Another huge benefit of that management pipeline is that it also works on directories and maintains the structure.

For example, if the original data is preprocessed, afterwards augmented with 10 different augmentation methods and each augmentation is stored in a corresponding directory, then one can just specify the parent directory as the source path and apply a new augmentation method on all the already augmented data. This proved to be very useful when experimenting with color, i.e. it was very easy to apply a color transformation on all the existing data.

If a lot of different preprocessing/augmentation methods are applied, then one has to first think about a suitable management structure, especially if these methods are mixed. Names of archives and folders are really important in order to keep these operations comprehensible and reproducible.

In case additional information is needed, the output of the tool can also be written to an external log file. By encoding the operations in the file names and by additionally storing the log files, reproducibility of preprocessing/augmentation is ensured.

7.6 Obstacles & Challenges

Before explaining the final setup of our infrastructure, let us take a look at the major obstacles and challenges we faced regarding the infrastructure. Basically we faced two major challenges.

The first big obstacle we encountered was definitely the usage of Kubernetes and Kubeflow. Reflecting back on this, a lot of time was wasted trying to use these tools even though they were not a perfect fit for our project. Since we did not need most of the functionality offered by Kubernetes, we should have stopped trying after the first few days. However, at that time we also did not know that MLflow would just work out of the box and completely fulfill our needs for the machine learning development.

The other major challenge we faced was the integration of the NVIDIA Tesla K80 into our bare metal infrastructure. At first, we started with the infrastructure provided by Amazon. That offered the NVIDIA Tesla K80 GPU out of the box for us. As we shifted towards our bare metal setup, we also wanted to have access to a NVIDIA Tesla K80 GPU. That is a quite specialized server GPU. We bought one at an auction on eBay from a dealer in

Denmark. That process worked just fine and the GPU arrived shortly after. However, the first problem was that the case of the computer was too small for the NVIDIA Tesla K80. We fixed that problem by using another case. Shortly after that the next problem was encountered: the GPU needed to allocate more than 4GB of memory, while our main board did not support that. The main board was also kind of old and did not offer the option to enable that setting. An update of the BIOS was also not possible. So, we switched to another main board and luckily that worked.

Another minor challenge to note was the already mentioned structure of data management. At the beginning, we did the preprocessing with a simple shell script and had no naming conventions for the corresponding archives. The more data we created with that method, the more confusing the whole data management process became. So, in the end, we decided to write a script for that and also decided on a naming convention which was applied automatically by the script. This whole process helped a lot and made things far more reproducible.

7.7 Final Setup

As already hinted at in the previous sections, for the final setup we used a bare metal server that was located in my basement.

First, the general specifications of our server:

```
OS: Ubuntu 20.04.2 LTS (Focal Fossa)
Kernel Version: 5.4.0-74-generic
CPU: Intel Core i7-6700K@4.6GHz
RAM: 24GB DDR4@2666Mhz
Mainboard: Asus Z170-P
GPU_0: NVIDIA Geforce GTX 970
GPU_1: NVIDIA Tesla K80
GPU_2: NVIDIA Tesla K80
```

We will now take a closer look at the most important details.

7.7.1 NVIDIA Tesla K80

In order to satisfy the requirement of a powerful GPU for the training phase of the neural network, we bought a NVIDIA Tesla K80 GPU at an auction on eBay. A nice description can be found in [48]:

> "The NVIDIA® Tesla® K80 graphics processing unit (GPU) is a PCI Express, dual-slot computing module in the Tesla (267 mm length) form factor comprising of two NVIDIA Tesla K80 GPUs. The NVIDIA Tesla K80 GPU Accelerator is designed for servers and offers a total of 24 GB of GDDR5 on-board memory (12 GB per GPU) and supports PCI Express Gen3. The NVIDIA Tesla K80 is only available with a passive heat sink, which requires externally generated airflow for cooling."

The NVIDIA Tesla K80 consists of two Tesla GK210 GPUs based on the Kepler architecture, each of them offering 2 496 processor cores at a base core clock of 560 MHz. More information can be found in [48] and [49]. For us, a very important detail is that the NVIDIA Tesla K80 is per default not used as a single GPU with 24 GB of memory, but actually as two 'separate' GPUs each with 12 GB of memory. The drawback of that obviously is the smaller amount of memory per GPU, however, on the other hand that allows us to use both of them separately and independently, meaning we can just start two runs at the same time without any (concurrency) problems.

It actually is not that common to integrate a NVIDIA Tesla K80 GPU into a bare metal setup. It took a lot of time and effort. Also, to the best of our knowledge, there is no literature on how to exactly do that, at least we were not able to find something aside from the NVIDIA Tesla K80 Board Specification [48] and the NVIDIA Tesla K80 Datasheet [49]. While these offer a lot information, they do not offer an explicit guide on how to integrate the GPU into a bare metal setup, which made this part particularly hard.

From our own experience, these are the most important things that need to be considered:

- The NVIDIA Tesla K80 needs to be cooled, so additional fans might be needed.
- The main board needs to support allocating 16 GB per GPU.
- The case of the computer has to be big enough.
- Enough power is needed, so an additional power supply might be necessary.
- The NVIDIA Tesla K80 uses an 8-PIN CPU power connector.

The NVIDIA Tesla K80 GPU looks like this:

Figure 7.1: NVIDIA Tesla K80.

In addition to the NVIDIA Tesla K80, the server also contains a NVIDIA GeForce GTX 970 GPU with 4 GB of memory. So, using `nvidia-smi` on our server, we get the following output:

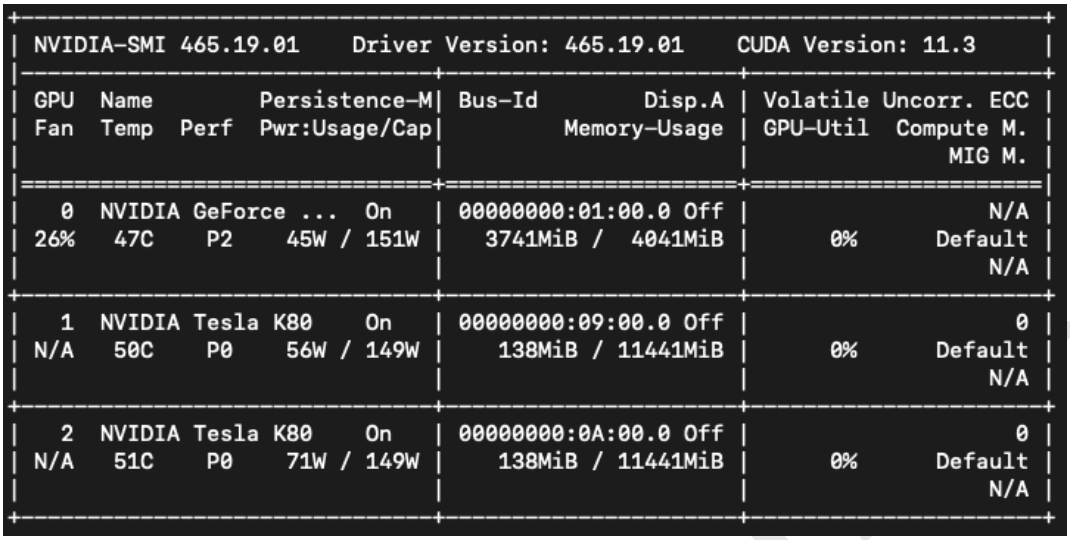


Figure 7.2: Output of `nvidia-smi`.

This setup allows us to start three different runs at the same time with each run accessing its corresponding GPU. Of course, the NVIDIA Tesla K80 offers far more RAM than the NVIDIA GeForce GTX 970, more on that in the next chapter (8).

7.7.2 Storage

For storing the data (original, preprocessed and augmented) we simply use two HDDs with 2 TB and 1 TB of storage respectively. The results of the machine learning runs and artifacts are also stored on these two HDDs. The operating system runs on a SSD with 128 GB of storage. This storage layout proved to be totally sufficient for our needs.

7.7.3 SSH Access

The server was located in my basement and therefore per default only accessible from within my private network. So, in order to make it externally accessible for authorized users, we did the following:

1. Open a (randomly generated) port number via the router settings for external access.
2. Change the default port of `ssh` to this new port number.
3. Whitelist the IPs of authorized users. Disallow all other IPs.
4. Allow `ssh` access via key access.

This proved to be a secure and efficient way to allow authorized users access to the bare metal server.

7.7.4 Development Environment

In order to make the machine learning development easier, we chose MLflow as the framework. It was installed locally on the server and could be accessed at port `5000`. For storing the models and artifacts, a simple SQLite database was used.

For the development of the machine learning code, Jupyter notebooks were used. By installing Jupyter locally on the server, these notebooks could be accessed on port `8888`. The combination of MLflow and Jupyter proved to be a very efficient development environment.

Results of the runs were automatically sent to a corresponding Slack channel in order to keep us up to date. For the monitoring of various parameters of the server, Prometheus[6] in combination with Grafana[7] was used.

Overall, this development environment was quite tailored to our exact needs and proved to be a very suitable environment for us, however, for other use

[6] `https://prometheus.io/`, visited on 07/29/2021
[7] `https://grafana.com/`, visited on 07/29/2021

cases it might not be a perfect fit. As always, there is not the one right general configuration. One needs to first specify the corresponding requirements and needs, and afterwards optimise the infrastructure into the right direction.

7.8 Summary

To sum up, we started out with Google Colab notebooks, then used the IaaS option and ultimately shifted towards the bare metal option. While this was a very work intensive and challenging step, in the end it proved to be the right decision, as the form fitted infrastructure allowed us to focus on the machine learning part without having to worry too much about the underlying hardware once it had been set up. Also, all the four key requirements were satisfied with this setup.

The main challenges were finding (and configuring) a suitable machine learning platform and the integration of the NVIDIA Tesla K80 into our bare metal server.

Chapter 8

Experiments

The main purpose of the infrastructure - at least in our specific use case - was to provide a suitable environment for machine learning development. The neural network is trained on our data, which is split into a *train set* and a *test set*. The train set is used for the training phase and the test set is used for evaluating the performance of the neural network. Detailed information about the structure of the corresponding neural network can be found in the companion machine learning thesis [7]. The infrastructure should provide a fast and efficient environment for training the neural network, as this is the phase that takes the longest and also evaluation can obviously only be done after the training phase is completed. In other words, the main purpose of the infrastructure was to make the training phase as fast as possible. A neural network can be trained on the CPU or on the GPU (if there is one available).

Let us take a look again at the specifications of our server:

```
OS: Ubuntu 20.04.2 LTS (Focal Fossa)
Kernel Version: 5.4.0-74-generic
CPU: Intel Core i7-6700K@4.6GHz
RAM: 24GB DDR4@2666Mhz
Mainboard: Asus Z170-P
GPU_0: NVIDIA GeForce GTX 970
GPU_1: NVIDIA Tesla K80
GPU_2: NVIDIA Tesla K80
```

Additionally, we had access to another system with the following specifications:

```
OS: Windows 10 Professional 64 Bit (Version 10.0.19041.1237)
CPU: Intel Core i9-10850K@3.6GHz
RAM: 32GB DDR4@2400Mhz
Mainboard: ASUS TUF Z490-PLUS
GPU_0: NVIDIA GeForce RTX 3070
```

So, the following different options will be compared:

1. Intel Core i7 (CPU) on Ubuntu
2. Intel Core i9 (CPU) on Windows
3. NVIDIA GeForce GTX 970 (GPU) on Ubuntu
4. NVIDIA Tesla K80 (GPU) on Ubuntu
5. NVIDIA GeForce RTX 3070 (GPU) on Windows

A detailed description of the NVIDIA Tesla K80 can be found in subsection 7.7.1. The NVIDIA GeForce RTX 3070 is based on the Ampere architecture. It offers 5 888 CUDA cores at a base core clock of 1.5 GHz [50].

For a comparison of these options and their respective advantages and disadvantages, we will look at different experiments. In order to understand these experiments, first a few basics about training a neural network need to be explained. Training a neural network works like this (note that this is a very high level explanation) [45]:

1. Input (i.e. training) data is given to the neural network in so called *batches*. The *batch size* gives information about how many data samples one batch contains. For example, if the batch size is 10, then one batch contains 10 data samples.
2. This batch (of input/training data) is propagated through the neural network resulting in some kind of output of the neural network.
3. The output is compared to the actual labels of the data.

4. The error (difference between the output and the actual labels) is calculated.

5. The parameters of the neural network are modified (tuned) in order to minimize the error.

Doing the above described steps for the whole training set results in a so called *epoch*, i.e. in one epoch the neural network is trained on the complete training set. The training phase consists of lots of epochs [45]. The number of epochs is a parameter of the network which the developer needs to tune. Obviously, the more epochs the training phase consists of, the longer the training phase takes. Important to note here is that the choice of the batch size does influence the accuracy of our network. More information about that can again be found in the corresponding machine learning thesis [7].

In the following experiments the average runtimes of an epoch and the possible batch sizes for each option will be examined. Fast runtimes of epochs and large batch sizes are desired. In addition to that, further experiments will be discussed which give information about how long it takes each option to reach an accuracy of over 90%.

8.1 CPU

Training the neural network on the CPU is the simplest and easiest solution. Sometimes this is also the only solution, in case there is no GPU available. As already explained before, the CPU is expected to be (much) slower than a GPU on machine learning tasks.

The Intel Core i9 has 32GB of RAM available while the Intel Core i7 has 24GB of RAM available, so both options allow for a larger batch size than the GPUs. The system with the Intel Core i9 allows for a maximum batch size of 150 and the one with the Intel Core i7 for a maximum batch size of 40. The runtimes shown in Figure 8.1 are the average of 10 epochs.

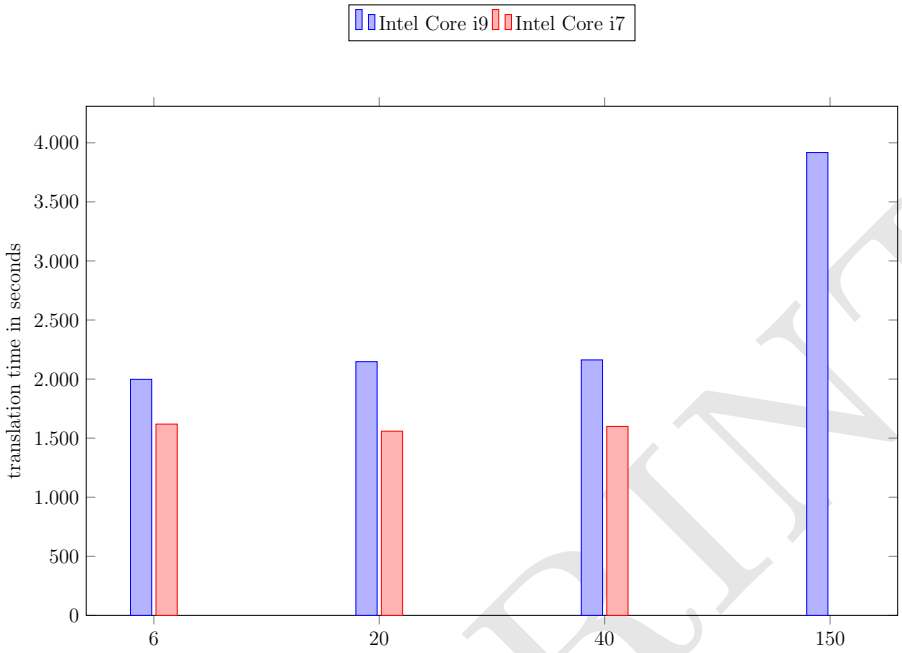


Figure 8.1: Average runtimes of an epoch using different batch sizes on two different CPUs.

Interesting to note here is that the more powerful Intel Core i9 CPU performs worse than the Intel Core i7 CPU when looking at the average runtimes of an epoch. This is quite unexpected. One hypothesis to explain this behaviour would be the different operating systems. Different memory management may cause this huge difference.

The great benefit of using the CPU is obviously that one does not need a GPU, which does require work and money. Also, if the CPU is combined with a lot of RAM, large batch sizes are possible. Finding the right batch size is not an easy job, but ultimately it is always a good thing if a large batch size is possible. The downside of using a CPU is that it is much slower than a GPU, which will be shown next.

8.2 GPU

In this section different experiments using the three available GPUs will be discussed.

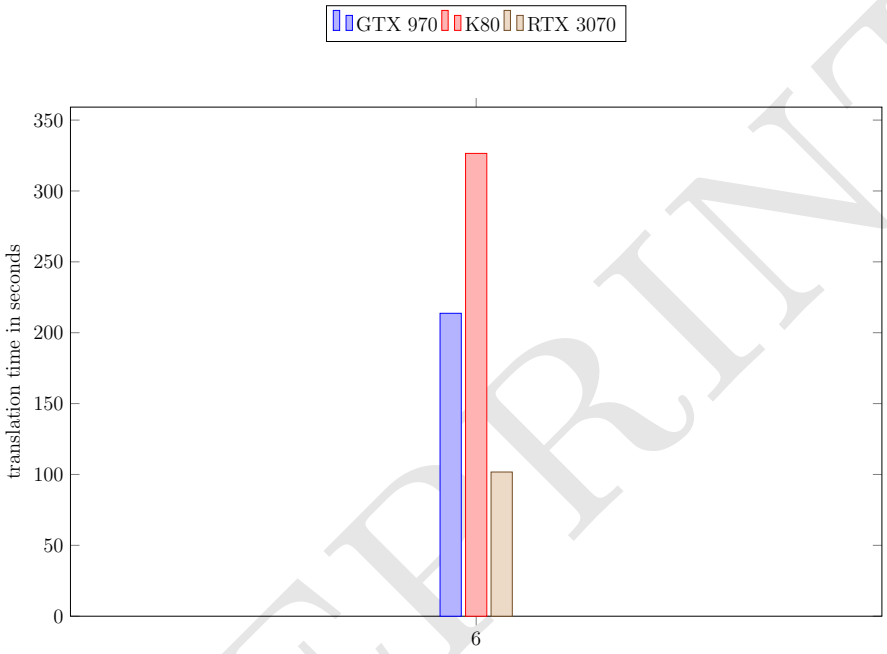


Figure 8.2: Average runtimes of an epoch using a batch size of 6 on three different GPUs.

8.2.1 NVIDIA GeForce GTX 970

Let us now take a look at the average runtime of an epoch when using the NVIDIA GeForce GTX 970 with 4GB of RAM. Since the RAM is quite low compared to the other options, we are not able to use a large batch size. So, we are only able to examine the runtime with a low batch size. Again, the runtimes shown in Figure 8.2 are the average of 10 epochs.

Here the great difference between the CPU and the GPU can already be seen. While the average runtime of an epoch on both CPUs is over 1500 seconds, on the NVIDIA GeForce GTX 970 it is about 200 seconds, which is

a difference by a factor of almost eight.

The benefit of using a GPU is obviously the fast runtime. However, on the other hand, a GPU might offer less RAM than a CPU, so there is the possible drawback that a large batch size might not be possible.

Also, GPUs tend to be quite expensive and according to the specific device, they might not be that easy to include into a bare metal setup.

8.2.2 NVIDIA Tesla K80

We mostly used the NVIDIA Tesla K80 which offers 12GB RAM. This GPU provides a good trade-off between a fast runtime and a considerable amount of RAM. Also, as described in the Infrastructure chapter (7), we had access to two of those. The 12GB of RAM allows us to increase the batch size. The runtimes shown in Figure 8.3 are the average of 10 epochs.

Figure 8.3: Average runtimes of an epoch using two different batch sizes on the NVIDIA Tesla K80.

Both batch sizes result in a quite similar runtime. However, much more

importantly, due to the larger batch size fewer epochs are needed for reaching a high accuracy (see Figure 8.7). Otherwise the same statements about GPUs explained before apply here as well.

8.2.3 NVIDIA GeForce RTX 3070

The NVIDIA GeForce RTX 3070 has 8GB of RAM available, which allows for a maximum batch size of 8. The runtimes shown in 8.4 are the average of 10 epochs.

Figure 8.4: Average runtimes of an epoch using two different batch sizes on the NVIDIA GeForce RTX 3070.

When compared to the NVIDIA Tesla K80 the average runtimes of an epoch are faster by a factor of about three, which is quite a large difference. However, the more interesting question is how long each of these options takes to reach a satisfying accuracy, since that depends on both the average runtime of an epoch and the possible batch size. This question will be answered in the next section.

8.3 CPU vs GPU

Let us compare the five different options with each other.

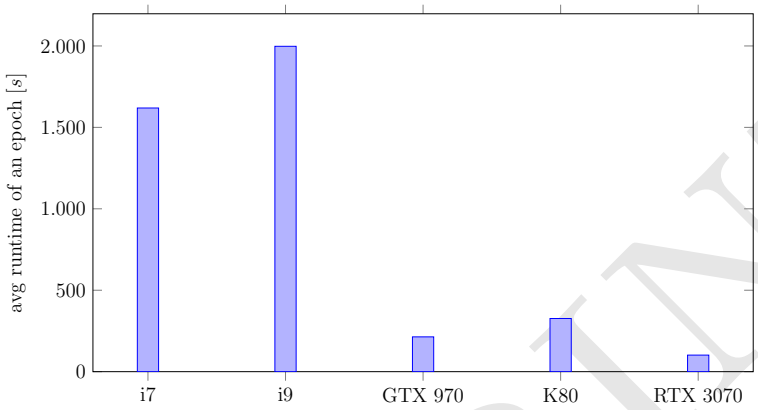


Figure 8.5: Average runtimes of an epoch with a batch size of 6.

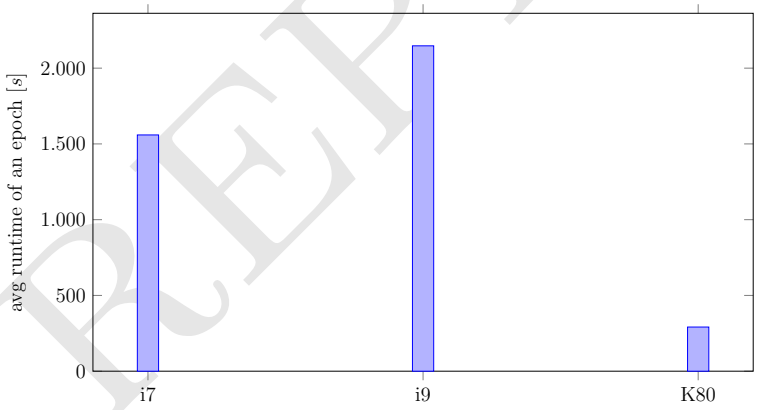


Figure 8.6: Average runtimes of an epoch with a batch size of 20.

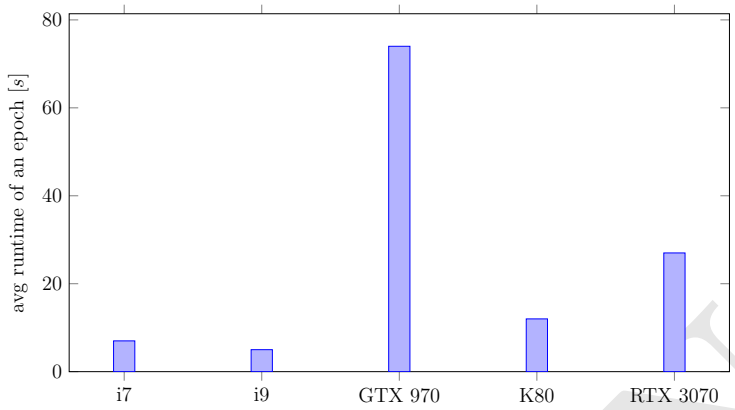


Figure 8.7: Number of epochs needed to reach an accuracy of over 90%.

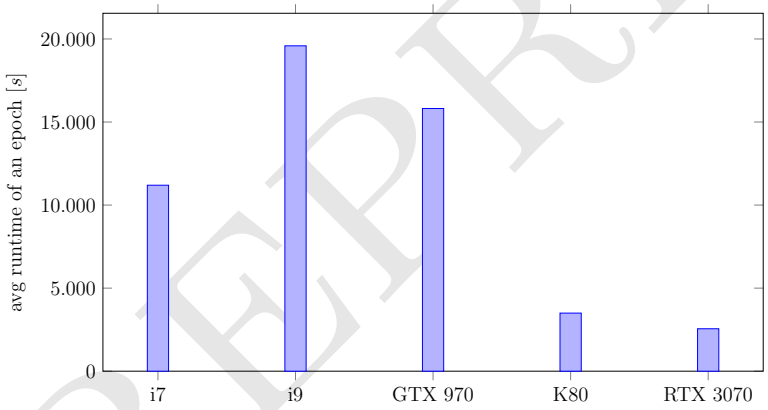


Figure 8.8: Training time needed to reach an accuracy of over 90%.

The difference of the average runtimes of an epoch shown in Figures 8.5 and 8.6 between CPU and GPU is immense. It is surprising that the NVIDIA Tesla K80 is slower than the NVIDIA GeForce GTX 970 in terms of average runtime of an epoch. Still, in the end the NVIDIA Tesla K80 proves to be the better option compared to the NVIDIA GeForce GTX 970, since it allows for a larger batch size which in turn results in needing fewer epochs and therefore reducing the overall training time. Note that we only had access to the NVIDIA GeForce RTX 3070 late in the project only for testing purposes, which is the reason why our infrastructure used the NVIDIA Tesla K80 and not the NVIDIA GeForce RTX 3070.

Figure 8.7 shows how many epochs it takes to reach an accuracy of over 90% with the maximal batch size, while Figure 8.8 shows how much time this takes. The maximum batch sizes are the following:

- Intel Core i7: 40
- Intel Core i9: 150
- NVIDIA GeForce GTX 970: 6
- NVIDIA Tesla K80: 20
- NVIDIA GeForce RTX 3070: 8

While both CPUs need fewer epochs (due to their large batch sizes), overall they still need a lot more time than the NVIDIA K80 and the NVIDIA GeForce RTX 3070. Interestingly, the NVIDIA GeForce GTX 970 takes the longest overall, due to the small batch size. It seems like a batch size of 6 is just too small and therefore a lot of epochs are needed. The NVIDIA Tesla K80 needs fewer epochs than the other two GPUs, but it is still slower than the NVIDIA GeForce RTX 3070 in the end. In conclusion, the fastest option would be the NVIDIA GeForce RTX 3070.

To sum up, the CPU is the simplest (and slowest) solution, but might offer the largest amount of RAM. On the other hand, a GPU is expensive and setting it up might be a challenge. However, a powerful one (with a reasonable amount

of RAM) outperforms the CPU in terms of runtime by a large margin, the potential downside might be the lack of RAM. Speaking from our experience, it pays off to use a GPU if one needs to perform intensive training. Using the NVIDIA Tesla K80 was the right choice for our project, since it offered both fast runtimes and a considerable amount of RAM, which allowed for a larger batch size compared to the NVIDIA GeForce GTX 970.

Chapter 9

Conclusion & Future Work

To sum up, the overall goal of the project was to explore machine learning-based classification of car makes and types from LiDAR and camera data offered by newest generation smartphones. The basic functionality of LiDAR was explained and afterwards the iOS application, which we developed for collecting the scans of the cars, was introduced. Different preprocessing and augmentation methods were defined and explained, especially our custom methods should be understood after reading the thesis and one should also be able to implement these. Different options regarding the infrastructure were discussed, as well as our personal experience setting up the bare metal server and the challenges we faced doing that. Our final setup was explained in detail. The experiments showed the advantages and disadvantages of using a CPU versus a GPU for training our neural network.

As the focus of this thesis lies on data management and infrastructure, the main findings regarding these two topics will be discussed.

The exact results of the neural network can be found in the corresponding machine learning thesis. Shortly summarized, we can say that the neural network is able to learn the data (both car make and even car types) very well, but the neural network's ability to generalize is quite limited. Nevertheless, showing that the data is learnable is very interesting and ultimately also shows that the combination of LiDAR and camera is able to produce very accurate and useful data for machine learning tasks. More concretely, this finding shows that the quality of the data provided by the new LiDAR sensor of the iPhone 12 Pro is - at least for the learnability of our certain use

case - completely sufficient.

What is more, these results show that our preprocessing and augmentation methods work well. Our network would not be able to learn the unpreprocessed data, where each individual scan consists of millions of points. The experiments presented in the previous chapter show that the integration of the NVIDIA Tesla K80 into our bare metal setup proved to be a good idea, as it offers a good trade-off between fast runtimes of epochs and enough RAM for larger batch sizes. Also, the experiments show that by using a powerful GPU the runtime of machine learning experiments can be greatly reduced when compared to a CPU.

Overall, the results are good, however, there is always room for improvement. One idea would be to collect more data and check whether that would improve the network's ability to generalize. Another one would be to just use the geometric values in combination with the pictures and use another fusion method. Doing that, one could even compare other existing fusion methods with the internal fusion on the iPhone. Improving the preprocessing and augmentation methods would also be an option, especially our custom ones. For example, one could improve the distance filter by taking the shape of the car into account. Also, other (more complicated) methods described in existing literature could be examined. Finally, an even more powerful GPU that offers more RAM could be used in order to be able to use a larger batch size. While there are certainly more improvement possibilities, these are the main ideas for future work regarding data management and infrastructure.

Bibliography

[1] You Li and Javier Ibanez-Guzman. "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems". In: *IEEE Signal Processing Magazine* 37.4 (2020), pp. 50–61. DOI: 10.1109/MSP.2020.2973615.

[2] A. Asvadi et al. "DepthCN: Vehicle detection using 3D-LIDAR and ConvNet". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6. DOI: 10.1109/ITSC.2017.8317880.

[3] Xinxin Du et al. *A General Pipeline for 3D Detection of Vehicles*. 2018. arXiv: 1803.00387 [cs.CV].

[4] Limin Guan et al. "Real-Time Vehicle Detection Framework Based on the Fusion of LiDAR and Camera". In: *Electronics* 9.3 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9030451. URL: https://www.mdpi.com/2079-9292/9/3/451.

[5] 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.

[6] Xingyang Ni and Heikki Huttunen. "Vehicle Attribute Recognition by Appearance: Computer Vision Methods for Vehicle Type, Make and Model Classification". In: *CoRR* abs/2006.16400 (2020). arXiv: 2006.16400. URL: https://arxiv.org/abs/2006.16400.

[7] Sebastian Landl. "Color3DNet: Neural Network-Based Vehicle Classification Using Commercial Off-The-Shelf LiDAR and Camera Sensors and Machine Learning". Preprint of Master's Thesis. University of Salzburg, 2021. URL: https://github.com/ChrisEdel/vehicle-classification/theses/landl.

[8] Jamie Carter et al. *Lidar 101: An Introduction to Lidar Technology, Data, and Applications*. 2012. URL: https://coast.noaa.gov/data/digitalcoast/pdf/lidar-101.pdf (visited on 07/29/2021).

[9] Ninad Mehendale and Srushti Neoge. *Review on Lidar Technology*. Available at SSRN: https://ssrn.com/abstract=3604309 or http://dx.doi.org/10.2139/ssrn.3604309. May 18, 2020.

[10] 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.

[11] Pinliang Dong and Qi Chen. "Basics of LiDAR Data Processing". In: Dec. 2017, pp. 19–39. ISBN: 9781351233354. DOI: 10.4324/9781351233354-2.

[12] Pinliang Dong and Qi Chen. "Basics of LiDAR Data Processing". In: Dec. 2017, pp. 63–105. ISBN: 9781351233354. DOI: 10.4324/9781351233354-4.

[13] Apple Press Release. *Apple unveils new iPad Pro with breakthrough LiDAR Scanner and brings trackpad support to iPadOS*. March 18, 2020. URL: https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/ (visited on 07/29/2021).

[14] G Ajay Kumar et al. "LiDAR and Camera Fusion Approach for Object Distance Estimation in Self-Driving Vehicles". In: *Symmetry* 12.2 (2020). ISSN: 2073-8994. DOI: 10.3390/sym12020324. URL: https://www.mdpi.com/2073-8994/12/2/324.

[15] Maximilian Vogt, Adrian Rips, and Claus Emmelmann. "Comparison of iPad Pro®'s LiDAR and TrueDepth Capabilities with an Industrial 3D Scanning Solution". In: *Technologies* 9.2 (2021). ISSN: 2227-7080. DOI: 10.3390/technologies9020025. URL: https://www.mdpi.com/2227-7080/9/2/25.

[16] B. Heinrichs and M. Yang. "Bias and Repeatability of Measurements from 3D Scans Made Using iOS-Based Lidar". In: *SAE Technical Paper 2021-01-0891* (2021). DOI: https://doi.org/10.4271/2021-01-0891.

[17] Hyo Jong Lee et al. "Real-Time Vehicle Make and Model Recognition with the Residual SqueezeNet Architecture". In: *Sensors* 19.5 (2019). ISSN: 1424-8220. DOI: 10.3390/s19050982. URL: https://www.mdpi.com/1424-8220/19/5/982.

[18] 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.

[19] 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.

[20] Noppakun Boonsim and Simant Prakoonwit. "Car make and model recognition under limited lighting conditions at night". In: *Pattern Analysis and Applications* 20.4 (2017), pp. 1195–1207. ISSN: 1433-755X. DOI: 10.1007/s10044-016-0559-6. URL: https://doi.org/10.1007/s10044-016-0559-6.

[21] Xian-Feng Han et al. "A review of algorithms for filtering the 3D point cloud". In: *Signal Processing: Image Communication* 57 (May 2017). DOI: 10.1016/j.image.2017.05.009.

[22] Xian-Feng Han et al. "Guided 3D point cloud filtering". In: *Multimedia Tools and Applications* 77 (July 2018). DOI: 10.1007/s11042-017-5310-9.

[23] Ju Xu, Mengzhang Li, and Zhanxing Zhu. *Automatic Data Augmentation for 3D Medical Image Segmentation*. 2020. arXiv: 2010.11695 [eess.IV].

[24] Shuyang Cheng et al. *Improving 3D Object Detection through Progressive Population Based Augmentation*. Apr. 2020.

[25] Martin Hahner et al. *Quantifying Data Augmentation for LiDAR based 3D Object Detection*. 2020. arXiv: 2004.01643 [cs.CV].

[26] 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.

[27] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV].

[28] Apple WWDC. *Explore ARKit 4*. 2020. URL: https://developer.apple.com/videos/play/wwdc2020/10611/?time=1114 (visited on 07/29/2021).

[29] Chris Deziel. *Which Colors Reflect More Light?* 23 April 2018. URL: https://sciencing.com/colors-reflect-light-8398645.html (visited on 07/29/2021).

[30] Sotiris Kotsiantis, Dimitris Kanellopoulos, and P. Pintelas. "Data Preprocessing for Supervised Learning". In: *International Journal of Computer Science* 1 (Jan. 2006), pp. 111–117.

[31] 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, 2011.

[32] Yongtao Yu et al. "Automated Extraction of Urban Road Facilities Using Mobile Laser Scanning Data". In: *IEEE Transactions on Intelligent Transportation Systems* 16 (Aug. 2015), pp. 1–15. DOI: 10.1109/TITS.2015.2399492.

[33] André Carrilho, Mauricio Galo, and Renato Dos Santos. "STATISTICAL OUTLIER DETECTION METHOD FOR AIRBORNE LIDAR DATA". In: vol. XLII-1. Sept. 2018. DOI: 10.5194/isprs-archives-XLII-1-87-2018.

[34] Radu B. Rusu. *Downsampling a PointCloud using a VoxelGrid filter*. URL: https://pointclouds.org/documentation/tutorials/voxel_grid.html#voxelgrid (visited on 07/29/2021).

[35] Zoltan-Csaba Marton and Alexandru E. Ichim. *Smoothing and normal estimation based on polynomial reconstruction*. URL: https://pointclouds.org/documentation/tutorials/resampling.html#moving-least-squares (visited on 07/29/2021).

[36] Gabe O'Leary. *Removing outliers using a Conditional or RadiusOutlier removal*. URL: https://pointclouds.org/documentation/tutorials/remove_outliers.html#remove-outliers (visited on 07/29/2021).

[37] Radu B. Rusu. *Removing sparse outliers using StatisticalOutlierRemoval*. URL: https://pointclouds.org/documentation/tutorials/statistical_outlier.html#statistical-outlier-removal (visited on 07/29/2021).

[38] Radu B. Rusu. *Filtering a PointCloud using a PassThrough filter*. URL: https://pointclouds.org/documentation/tutorials/passthrough.html#passthrough (visited on 07/29/2021).

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

[40] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[41] Victor Lamoine. *Using matrixes to transform a point cloud*. URL: https://pointclouds.org/documentation/tutorials/matrix_transform.html#matrix-transform (visited on 07/29/2021).

[42] Peter Rutten and David Schubmehl. *Serverinfrastruktur für künstliche Intelligenz stößt an Ihre Grenzen*. 2017. URL: https://www.themsphub.com/app/uploads/2018/11/GERMAN-systems-hardware-power-systems-po-white-paper-external-pow03210usen-20180125_POW03210USEN-1.pdf (visited on 07/29/2021).

[43] D. Sculley et al. "Hidden Technical Debt in Machine Learning Systems". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf.

[44] Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. *GPGPU Computing*. 2014. arXiv: 1408.6923 [cs.DC].

[45] Daniel Schlegel. *Deep Machine Learning on GPUs*. 2015. URL: https://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/seminar/2014-Daniel_Schlegel.pdf (visited on 07/29/2021).

[46] Leila Abdollahi Vayghan et al. *Kubernetes as an Availability Manager for Microservice Applications*. 2019. arXiv: 1901.04946 [cs.SE].

[47] Casey Newton. *Inside the all-hands meeting that led to a third of Basecamp employees quitting*. May 3, 2021. URL: https://www.theverge.com/2021/5/3/22418208/basecamp-all-hands-meeting-employee-resignations-buyouts-implosion (visited on 07/29/2021).

[48] *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 07/29/2021).

[49] *Tesla K80 Datasheet*. NVIDIA. 2014. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf (visited on 07/29/2021).

[50] *GeForce RTX 3070 Family*. NVIDIA. 2021. URL: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3070-3070ti/ (visited on 10/12/2021).