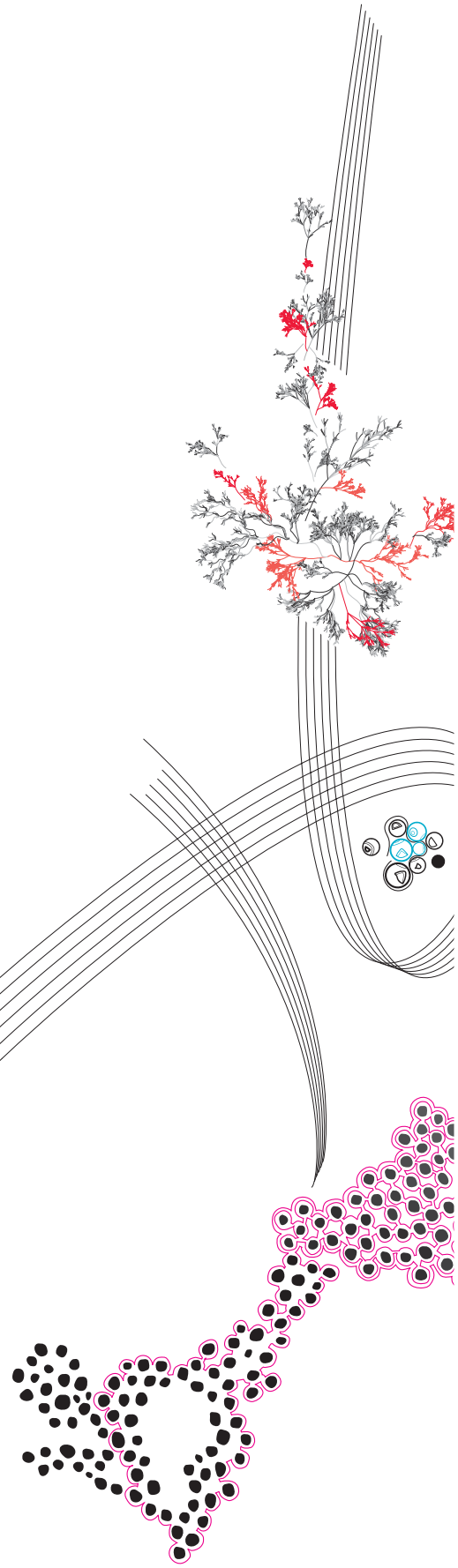


BSc Thesis Creative Technology

Point cloud based semantic segmentation for catenary systems using deep learning: Compressibility of a PointNet++ network

Nils Rutgers

Supervisor: F. Ahmed

July, 2022

Department of Creative Technology
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Point cloud based semantic segmentation for catenary systems using deep learning: Compressibility of a PointNet++ network

Nils Rutgers[1]

July, 2022

[1]Email: n.rutgers@student.utwente.nl

Abstract

The current signalling system used to manage train traffic is outdated. The entire railroad system is divided into sections. As only one train is allowed to be in a specific section, the track is used inefficiently. ProRail is collaborating with StruktonRail to modernise the signalling system. To get an overview of the current catenary systems, a digital twin needs to be created that digitally resembles the physical situation. A deep neural network was trained using point clouds, in this case, a PointNet++ network, to identify catenary arches. To deploy such a model on a microcontroller, it needs to be compressed. There are numerous algorithms to compress a computer vision model, however, there is no method available to compress a network trained on point clouds. This research aims to investigate the feasibility of applying available compression methods on a PointNet++ network while preserving performance.

The research will follow the CRISP-DMME cycle with a focus on the technical understanding and realisation phases. Literature research is done to determine a suitable method for compressing a PointNet++ network. This method leverages pruning, quantization and Huffman encoding to compress the size of a model. A digits classification and a PointNet++ model were compressed using this method. By comparing the compression results in terms of weight loss, size reduction and change in accuracy, the effectiveness of the compression method is evaluated. It is found that compression without performance loss is possible for computer vision networks. However, when compressing a PointNet++ using conventional compression methods, the accuracy will drop drastically. On the other hand, the models show similarities in terms of weight reduction and quantization. There are similarities between the compression results of both models. This indicates that compression without performance loss might be possible. However, further research needs to be done on the similarities between image and point cloud trained models on theoretical and software levels.

Acknowledgements

First and foremost, I would like to thank Dr. Faizan Ahmed for providing and supervising this project. The help and honest feedback I was provided with aided the project to stay on the right course. Next to that, I would like to thank Ir. Nacir Bouali for the critical supervision of the project. I would also like to thank Ir. Bram Ton and fellow graduation student Plamen Bozov for their technical expertise. Lastly, a thanks goes out to my family and friends, with a special note to the Line-Up, a group of fellow graduating students helping me through the toughest of times.

Contents

List of Figures

Chapter 1

Introduction

1.1 Situation

A signalling system is in place to ensure that two trains are never on the same part of the train track in opposite directions. The currently used technology and infrastructure stem from the 1950s. The entire Dutch railway system is divided into sections. The signalling system makes sure that there is not more than one train in a section, independent of the speed and direction of a train. Due to this implementation, the tracks are used in a relatively inefficient manner. ProRail, which is responsible for the management and maintenance of the Dutch railroads, is working together with StruktonRail to modernise the signalling system. This new system is still in development. However, to be able to speed up the implementation of a new system, Strukton needs to create digital twins of the current infrastructure. A digital twin is an exact virtual copy of reality. A 3D dimensional object of the real world is converted into a digital 3D model. Changes to this object are converted in real time into the digital counterpart. This creates digital counterpart that is always up to dat with the status of the physical object. In this case, a perfect digital representation of the currently used catenary arches and signposts. Using this twin, StruktonRail can analyse which parts are reusable for the new system or need to be removed/replaced. To create such a digital twin, a LiDAR scanner is mounted on a train to capture high-density point cloud data of the surroundings of the train. To distinguish different parts of the arches, a deep neural network (DNN) was trained for semantic segmentation [5]. For this, a PointNet++ [6] model was used.

As mentioned by Ton et. al. [5], the current method of data acquisition has two disadvantages [5]. First, is the vast amount of data that is captured. Second, is the unstructured nature of the data. Next to this, the data is captured in an inefficient way. On average, the catenary arches are 70 meters apart [7]. This makes roughly 65 meters, per 70 meters, uninteresting data. This comes down to a data acquisition efficiency of approximately seven per cent. Currently, the captured data has to be pre-processed to remove the uninteresting data from the data set. Next to this manual labour, it puts a strain on the storage of data during collection. When it would be possible to only collect data on the catenary systems, the storage overhead of the captured point clouds can also be reduced. This way, the data can either be stored more efficiently during acquisition or the point clouds could be collected with a higher resolution.

The proposed solution is to use a microcontroller to control the LiDAR sensor. This microcontroller should be able to detect incoming and passed catenary arches. This ensures that only the arches are captured, without the uninteresting surroundings. This requires the deployment of a neural network on a microcontroller. However, trained neural

networks are often too large to deploy on a microcontroller, resulting in long inference times (if even possible to deploy at all) [8][9][10]. With the foreseeable end of Moore's Law, it is not an option to wait for more suitable processors [11]. This creates a need for compressing the model. Due to this need, it is important to understand the consequences of compressing a PointNet++ network. In general, deep neural architectures specifically designed to handle point clouds are a recent invention. Consequently, there is little to no research on compressing such a network. Therefore, there is very limited knowledge of compressing a PointNet++ network. In this research the feasibility of applying available compression techniques to this type of network.

1.2 Preliminaries

Point clouds are lists of unordered data points that represent shapes in a digital 3D space. These points are captured by a LiDAR sensor (Light Detection And Ranging) [12]. By sending out laser beams and capturing their reflection, the sensor can determine the distance to an object or surface. The time between sending the beam and capturing its reflection is used to calculate this distance. When this information is combined with directional information, the location of the reflection point can be plotted in 3D space.

Deep learning (DL) is a subset of machine learning (ML) see figure 1.1, they are distinguished by how an algorithm learns. The main difference between ML and DL methods is how features are learned. In ML, the learning features are manually crafted, whereas DL methods learn these features automatically. DL algorithms can determine which features are most important to distinguish the data, wherein ML this hierarchy is established manually by a human [13]. A deep neural network (DNN) is the result of a DL algorithm, this is an artificial neural network with multiple layers between the input and output layers. Artificial neural networks are designed to mimic the human brain through a set of algorithms. The 'deep' part refers to the depth of layers in a neural network [13]. Before training, the amount of hidden layers and their respective amount of neurons is defined by a human. The more complex the data is to predict, the larger the number of needed layers and neurons. The weights of these neurons are fine-tuned using the process of backpropagation [14]. This method is used to calculate derivatives in the network. A loss function is calculated which represents how far the network's predictions are from the ground truth. Then, the gradient of the loss function with respect to the weights of the network can be determined. Using this, each weight can be updated individually. With these elaborate networks comes a large storage requirement. Compressing a model can, among others, reduce the storage requirement and inference time of a network [8][9][10].

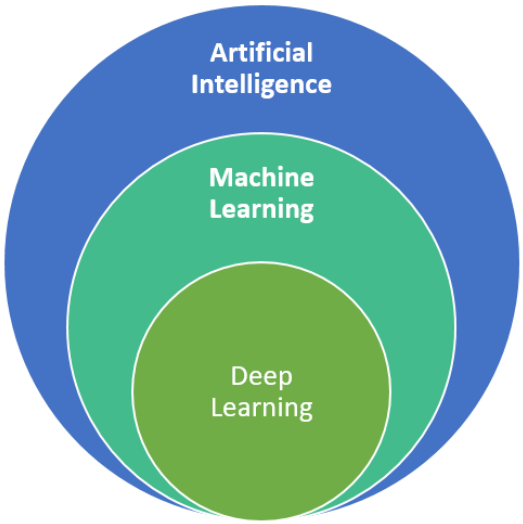


FIGURE 1.1: Diagram of the relation between AI, ML and DL [1]

1.3 Research Objectives

The primary focus of this research is to investigate the feasibility of applying compression techniques available for other deep learning architectures to the case of architectures based on point clouds. In the research, the PointNet++ architecture will be used as an example to compress using available techniques. A trained PointNet++ network will be compressed using the available techniques. The influence of compression on the performance of the network will be investigated. The aim is to reduce the size of the network to such an extent that it can fit on a microcontroller. The primary research question is the following:

How can a trained PointNet++ model be compressed to fit on a microcontroller while preserving performance?

The literature research consists of two parts. One, determining suitable state-of-the-art methods for compression. Two, investigating what constraints deployment on a microcontroller put on the network. The research will focus on answering the following sub-research questions:

- Which methods are available for compressing a deep neural network?
- What are the characteristics of a PointNet++ network?
- What constraints does deployment on a microcontroller put on a neural network?

By implementing a suitable compression method, the relation between the compression rate and level of accuracy can be investigated.

1.4 Thesis Structure

A literature review will be presented in chapter 2. The available literature will be used to answer the proposed research questions. Chapter 3 will highlight the Crisp-DMME model proposed by Huber et. al. [2]. The methodology that will be followed throughout the preparation and execution of the experiment is based on this model. In chapter 4, the

preparation phases of the project will be executed. Chapter 5 continues on this by conducting the experiment. The limitations encountered during the realization of the project will be discussed here. The chapter that follows will discuss the results of the experiment and create and understanding from this. The chapter will conclude with answering the research question and making suggestions for potential avenues of future research.

Chapter 2

Background Research

2.1 Literature Research Questions

The following research questions were investigated for the background research of the project. The focus of the first one is to create an overview state-of-the-art compression approaches and their respective methods. The second focus is to get an overview of the structure of a trained PointNet++ model, as this could influence the choice of compression. The focus of the third is to get an overview of currently used microcontrollers and their limitations that need to be taken into account when deploying a DNN on one.

- Which methods are available for compressing a network?
- What are the characteristics of a PointNet++ network?
- What constraints does deployment on a microcontroller put on a network?

2.2 Methodology

For the acquisition of sources, the databases of Google Scholar [15] and ACM digital library [16] were used. As the field of machine and deep learning is rapidly growing and evolving, the main criteria for including papers is the publication date. However, some compression methods have very few publications on the respective topic. Therefore, some papers are included unless their earlier publication date. An explanation for the lack of publications is given in the results section of this chapter. In the context of this literature research, two terms are used, approach and method. An approach refers to a theoretical concept of compression, whereas a method refers to a specific implementation (often an algorithm) that leverages a described approach to compress a network.

On the topic of constraints for deployment on a microcontroller, literature was used to determine relevant information and characteristics to deploy neural networks. A short list of widely used microcontrollers was compiled, for which the relevant information was gathered. This short-list consists of three microcontrollers which are significantly different from a technical perspective.

2.3 Results: Compressibility

In the literature, it was found that compression approaches can be divided into four categories. In the following sections, these categories of compression approaches will be highlighted. The described categories are compact model approach (CMA), tensor decomposi-

tion (TD), quantization, and network sparsification (NS) [10]. Cheng et. al. [8] use similar categories, however, some are split into separate subcategories. As this literature research is executed to get an overview of different approaches, the first four are sufficient to gain an overview of available approaches. This information can then be used to choose a specific method applicable to the trained network. The methods can be distinguished into two categories, during- and post-training. CMA is a during-training compression approach, as it creates a pool of new compressed models, evaluates them and picks the best performing. The other three approaches modify the original base model to reduce the memory and computational costs. Therefore, all three fall in the post-training compression category.

2.3.1 Compact Model Approach

The first category of compression approaches is the compact model approach. This approach aims at creating a compressed network from a pool of candidates [10]. One method that falls in this category is highlighted by Choudhary et. al. [9]. This method is called knowledge distillation (KD). This method aims at transferring the knowledge learned by a bigger network (teacher network) to a smaller and lighter network (student network). This way, a student with similar generalization capabilities as the teacher can be created.

This approach has two major downsides that need to be taken into account when implemented. One, Choudhary et. al. [9] bring up the point that the teacher needs to be trained on a large dataset based on which it can generalize well on unseen data. When such a dataset is not available, the risk of overfitting the student model to the training data is increased significantly. Another drawback of CMA is that this type of approach generally achieves less competitive performances compared to other approaches [8].

2.3.2 Tensor Decomposition

The second category of compression method is tensor decomposition (TD). TD aims to split the original tensors into smaller ones. Due to this TD is mainly used to reduce the size of the convolutional layers [9]. Although tensors are a fundamental building block in deep learning models, the use of TD for compression is still in its early stages [17]. Bacciu and Mandic [17] describe three different methods of how TD can be leveraged for compressing DNNs. However, only one of these methods is mentioned by Choudhary et. al. [9]. This method is called Low-Rank Factorization (LRF). This method focuses on factorizing the weight matrix of a layer or filter into two matrices with lower dimensions [9]. LRF can be applied to the dense layer matrices, this application focuses on reducing the storage requirement of the model. Whereas applying LRF on the convolutional layers speeds up the inference process. Choudhary et. al. [9] mention an implementation challenge of LRF. The decomposition process results in harder implementation and is computationally more intensive. In addition to this, it is difficult to decompose the models in which an average pooling layer replaces the dense layer [9]. Cheng et. al. [8] agree with this and add to this by stating that LRF also requires extensive model retraining to achieve convergence compared to the base model.

2.3.3 Quantization

The third category of approaches is quantization. Quantization entails the reduction of the bit-width of parameters. With the vast amount of parameters in a DNN, the storage requirement can be decreased significantly using this kind of compression approach. Depending on the goal of quantization, either uniform or non-uniform quantization can be

applied. Uniform quantization means that quantization points are uniformly-separated. If the goal of compression is to speed up the inference phase on certain hardware, uniform quantization should be applied, as non-uniform quantization can only reduce the size of the DNN [9]. In 2011, Vanhoucke et. al. [18] showed that uniform quantization could be leveraged to speed up the inference time of a trained network.

Next to these benefits, the energy consumption of the DNN is reduced by applying quantization methods. The smaller the bit-width of parameters, the less energy it takes to add or multiply them [19]. Even though these factors sound very beneficial, the approach runs into one limitation that needs to be kept in mind. Cheng et. al. [8] mention that the relation between the level of quantization and accuracy loss needs to be kept in mind when compressing. When quantizing to such an extent that each parameter is represented by one bit, although very space-efficient, the accuracy of these binary networks is lowered significantly when dealing with large DNNs.

2.3.4 Network Sparsification

The last category of approaches is network sparsification. This concept entails all methods that compress a DNN by sparsifying the computational graph with fewer weights and/or connections. The most adopted approach that falls in this category is data pruning. This approach aims at reducing the storage requirement of a DNN. Secondly, it can be used to reduce the computation costs by pruning the parameters in the convolutional layer [9]. Cheng et. al. [8] disagree with this second part by stating that pruning is usually able to reduce the model size, but neither training nor inference time.

Within the data pruning approach, there are a variety of methods available to leverage the approach for compression. Liu et. al. [20] suggest a discrimination-aware channel pruning technique. This algorithm can focus the pruning on the channels in the model that have little discriminative power. Liu et. al. were able to achieve a 1.4x and 1.9x inference acceleration on a mobile device. On the other hand, Singh et. al. [21] developed a framework that uses an adaptive filter pruning module and a pruning rate controller. This allows for a specification in a desired error and tolerance, instead of a pruning level. The framework managed to reduce the parameters of the VGG-16 model by a factor of 17.5x. These two examples are proven to influence both the inference time and size of a network. This aligns with the previously mentioned statement made by Choudhary et. al. [9] and contradicts the point made by Cheng et. al. [8].

2.3.5 Selecting an approach

Selecting a method of compression that works best for a specific model is not as straightforward as one would have liked. As highlighted, the characteristics, application and requirements of a network play a large role in which approaches are applicable to the model. Next to this, there seems to be no golden rule for comparing methods of compression. When comparing methods to prior work, separate criteria could be used [10]. Cheng et. al. [8] also raise this point and propose some general suggestions on how to choose a method for compressing DNNs. The application and requirements of the compressed model are a driving factor in which method to use [8][9].

In addition to this, compression methods are often combined. Especially for the case of data pruning and quantization, where a DNN is first pruned, after which the remaining parameters are quantized [9]. A prime example of combining methods is the work of Tan and Wang [22], where three different methods were used to compress DNNs for speech

enhancement. With the combination of data pruning, quantization and TD, it was possible to reduce the size of different DNNs substantially, while preserving performance.

2.3.6 Conclusion

The goal of this literature review was to get an overview of the currently used concepts to compress a DNN and what factors play a role in choosing an appropriate method. From the literature, it is found that there is a variety of approaches which can be leveraged for compressing DDNs. For applying KD and TD there are major requirements for the original model and data. KD can only be used for applications with lots of varying training data to prevent the overfitting of the compressed model. As a second drawback, KD often performs significantly worse than other methods. The largest drawback for TD is that implementing such a method is often computationally expensive. Quantization and NS seem to be preferred methods of compressing. The majority of compression techniques use, at least, one of these methods. This is due to the few requirements they put on the original model and training data. Next to this, they can be used for a variety of goals of compression, for example reducing the storage requirement, inference time or energy consumption. However, when applying compression methods for these separate goals, a different part of the model needs to be compressed.

2.4 Results: PointNet++

PointNet++ [6] is the successor of PointNet [23], a neural network able to directly process point clouds. By building on the architecture of this network, PointNet++ was created. A hierarchical neural network was created that applies PointNet recursively on a nested partitioning of the input point cloud. The new network can learn features despite varying sampling densities in the training data. However, PointNet++ is not the only neural network that can work with point clouds efficiently. Other networks are Super Point Graph and Point Transformer, however, as mentioned before, the research of Ton et. al. showed that PointNet++ is the most accurate network [5].

The PointNet++ model uses methods similar to CNNs for feature extraction, by capturing geometric structures from small neighbourhoods. These local features are grouped into larger units and processed to produce higher-level features. Convolutional neural networks use hidden convolutional layers to convolve the input and pass it to the next layer. Each convolutional layer processes the data for its receptive field. The last layer is a fully connected one, where every neuron of one layer is connected to every neuron in another layer. This is used to classify the input data. PointNet++ applies PointNet recursively to learn features of a local region. By using sampling and grouping layers, the local regions are determined to pass on to the PointNet layer. Using this hierarchal feature learning, PointNet++ can learn from unordered input sets without losing information on the distance between points.

2.5 Results: Microcontroller constraints

Microcontrollers have a few characteristic parts that are of interest within the scope of this project; the Central Processing Unit (CPU), nonvolatile memory and volatile memory [24]. The CPU takes care of all arithmetic operations and manages the data flow. Nonvolatile memory is the memory that retains information when power is lost. Due to this characteristic, it is used to store the program on a microcontroller. The most frequently

used nonvolatile memory on microcontrollers is flash memory. On the other hand, will the volatile memory lose its information when power is lost. Therefore, it is only used to store non-critical data. For example, it can be used to temporarily store data during computation. A commonly used type of nonvolatile memory is RAM (Random Access Memory). As nonvolatile memory is often significantly quicker than non-volatile memory, it offers benefits when using nonvolatile memory for computation. [25][26]

Due to the characteristics and use cases of the different memory types (RAM and Flash) present on microcontroller boards, the following assumptions are made about deploying a neural network on a microcontroller:

1. The Flash memory is used to statically store the weights of a network. The size of the weights file can never exceed the available flash memory.

2. The RAM and CPU speed influence the inference time of a deployed network. Less available RAM or lower CPU speed will increase the inference time.

Within the world of IoT, microcontroller boards are commonly used. Due to their compact size, they are optimal for remote sensing and computing. Some boards are more widely used than others, a list of the more adopted board is composed. The relevant characteristics of these will be described below.

The Raspberry Pi is a well-known development board used for a wide variety of use cases. As it is advertised as a tiny desktop, it has the capabilities of one. The latest version, the Raspberry Pi 4 is available with 1, 2, 4, or 8 GB of RAM and uses a 64-bit processor. The flash memory is inserted in the form of an SD card. [27]

Arduino offers a wide range of easy-to-use microcontroller boards. However, the majority of them use the same microprocessor chip, the ATMEGA323P, an 8-bit processor. The difference between the boards is in the connection possibilities to sensors and other devices. This means that the microcontrollers offered by Arduino (for example the Uno or Nano) have the same RAM (2 KB) and flash memory (32 KB). [28]

A third widely used microcontroller is the ESP32. This board uses the 32-bit LX6 microprocessor, with 0, 2 or 4 MiB of flash memory. 1 MiB is equal to 2^{20} bytes, which comes down to 1,048,576 bytes. This is 48,576 bytes larger than an MB, which is $10^6 = 1,000,000$ bytes. This means that the maximum flash memory available on an ESP32 is 4,194,304 bytes, roughly 4.2 MB. Next to this, the ESP32 is equipped with 320 to 400 KB of RAM. [29]

Chapter 3

Methodology

3.1 CRISP-DMME

CRISP-DMME is an extension of the CRISP-DM model [30]. CRISP-DM is short for Cross Industry Process for Data Mining. The CRISP-DM methodology is a process model that serves as a base for a data science process. The methodology describes six phases that every data science process follows, these phases have specific tasks that fall in them. The CRISP-DMME adds three phases to the CRISP-DM cycle. These phases are shown as the grey boxes in figure 3.1. This extension is specifically tailored to engineering applications. It provides a communication and planning foundation for data analytics within the production domain. In the following sections, the different phases will be highlighted and explained. [2]

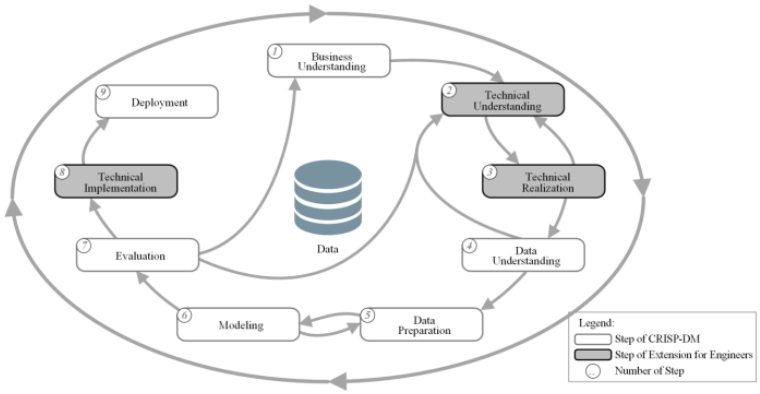


FIGURE 3.1: The nine phases of the CRISP-DMME methodology[2]

The initial phase, business understanding, has the goal to understand the project objectives from a business perspective. In this phase, the success criteria are defined along with the assessment of available resources. This knowledge is transformed into a problem definition and a preliminary plan to achieve the objectives. In the second phase, technical understanding the goal is to transform the business goals into measurable technical goals and develop an experiment plan. This phase occurs iteratively with the technical realization, where the goal is to select measuring concepts and conduct the experiment plan as described in the previous phase.

In the fourth phase, data understanding, the initial data identification and collection are executed. This proceeds the first analysis of data sets to gain first insights into the data and, possibly, identify data quality problems. If different data is needed, it can be needed

to go back to the technical understanding phase. The fifth phase, data preparation, covers all activities needed to construct a final dataset. This dataset is used as raw input for the training algorithm. These activities include selecting, cleaning and labelling the data to fit the input requirements. This phase often occurs iteratively with the sixth phase, modelling. In the modelling phase, the modelling techniques are selected and implemented. Typically, several techniques could be applied to the same problem. In the evaluation phase, a model has been built that appears to have a high quality. However, before deploying the model, it should be evaluated against the defined success criteria. At the end of this phase, a plan for technical implementation, phase eight, is to be made. The goal of this phase is to enable the evaluated model to be provided with run-time data during production. Finally, the evaluated model is deployed in a practical environment (phase nine).

The original project that is executed by Bram et. al. [5] is based on the CRISP-DM method. For this project, the extension that CRISP-DMME offers is useful in terms of the project description. For this reason, the phases described by the original CRISP-DM method are already executed within the scope of the project. Where needed, the differences between the results of the phases will be highlighted.

The data that was used to train the current model is available for evaluating the compressed model. Although the dataset is relatively small, it was leveraged in a specific way to prevent overfitting of the model on the training data. By applying a training method called 'leave-one-out-cross-validation', LOOCV. On each iteration of the training algorithm, one instance of the training data was left out. Then the model was evaluated using this last instance. This method was specifically developed for smaller datasets to overcome the problem of overfitting the model.

3.2 Methodology for Compressing

3.2.1 Compression Methods

As mentioned in the literature research section 2.3.6, the characteristics, application and requirements of the network are important factors in choosing a compression method. Next to this, compression methods are often combined to increase the efficiency of the algorithm. Using these factors, suitable compression methods are determined. Following, the paperswithcode database will be used to find a suitable algorithm with an open-source implementation. In the following chapter 4.3, the described method will be highlighted, resulting in a choice of compression method.

3.2.2 Evaluation of compression method

The compression method will be evaluated based on two conditions;

1. The achieved compression rate
2. The change in accuracy level

The achieved compression rate will be measured as a reduction in size. It is calculated by how many times the compressed model can fit in the original memory requirement. For example, if a model of 500 MB is compressed to 25 MB, the compression rate is 20 times (20x). A common heuristic to measure the accuracy of object detection models is the Intersection over Union (IoU). As the PointNet++ network is trained to predict multiple classes per input, the mean Intersection over Union (mIoU) is used. This is the average

IoU of all predicted classes. The change in accuracy level will be measured in a change of the Mean Intersection over Union (MIoU). IoU per class is calculated as follows:

$$TP/(TP + FP + FN)$$

[31]. This equation can best be explained using a confusion matrix 3.2. A confusion matrix shows prediction versus truth labels. T and F stand for true and false respectively, this refers to the actual class of the data input. P and N refer to the predicted class. As indicated in the confusion matrix by the red/green colour, TP and TN are correctly predicted cases. Whereas the FP and FN are incorrectly predicted cases. The IoU calculates the ratio between how many points were correctly predicted to be part of a class and the points that were incorrectly predicted. This way of measuring accuracy is often used as an evaluation metric for semantic (image) segmentation [31][32].

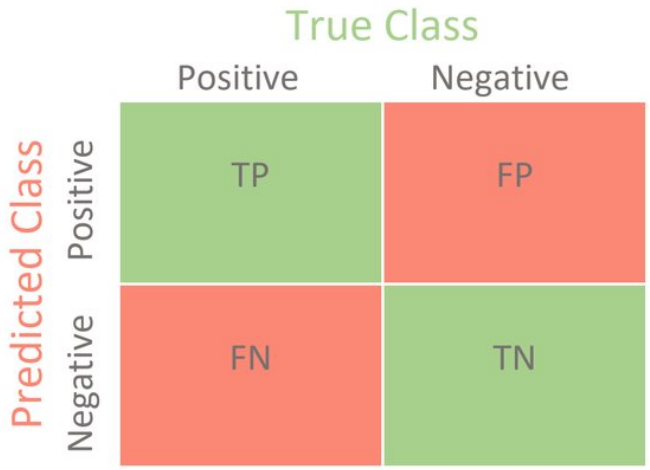


FIGURE 3.2: Confusion matrix for binary classification[3]

3.3 Development Environments

For the technical implementations needed in this project, Python was used as the primary development language. For ease of setup, a local virtual environment on a PC running a Windows operating system was used to compress the test model. As it was found that the used compression script could execute in a timely fashion (a matter of minutes), there was no need to do GPU-enhanced calculations. For compressing the trained PointNet++ network, there were some dependencies to compile which would not be possible on a PC with a windows operating system. Due to this, a Jupyter Notebook environment provided by the University of Twente (UT) was used. This virtual environment is Linux based and comes with conda pre-installed, capable of doing GPU-enhanced calculations. The environment is set up by the UT for students and researchers to execute data mining tasks. There is an option to run an environment on one of two servers, both of which have 256GB of RAM and use either an Nvidia Tesla T4 or Nvidia A10 for GPU. In both phases of the research, the Tensorflow library was used for deep learning implementations, as the original implementation of PointNet++, on which this project is based, makes use of the TensorFlow library. TensorFlow is an end-to-end open source platform for machine

learning. It consists of tools to easily and intuitively build and train models [33]. A part of this library is Keras, a high-level API, which allows for easy model iteration and debugging.

Chapter 4

Specification

In this chapter, the results of the different phases described by the CRISP-DMME model will be discussed. As this project builds on previously executed research, some of these phases have already been executed and will not be elaborately discussed in this report. Previous information can be found in the research by Ton et. al. [5].

4.1 Business Understanding

4.1.1 Objectives

The purpose of this project is to investigate the accuracy loss when reducing the size of the PointNet++ model to be deployed on a microcontroller. By compressing the model, it can be investigated what the accuracy levels are at different size reductions. The main goal is to reduce the size of the network. However, getting an inference close to real-time decision-making is needed for the use case described in the introduction. As trains travel at a speed of around 80 km/h [34], the decision should be made within 3 seconds, otherwise, the catenary arch might have passed before it is identified as one. A speed of 80 km/h comes down to 22 m/s, with an inference time of 3 seconds, the train travels roughly 70 meters before the prediction is given.

4.1.2 Resources

The data that was used to train the current model is available for evaluating the compressed model. Although the dataset is relatively small, it was leveraged in a specific way to prevent overfitting of the model on the training data. Using a training method called 'leave-one-out-cross-validation' LOOCV, the risk of overfitment is negated. On each iteration of the training algorithm, one instance of the training data was left out. Then the model was evaluated using this last instance. This method was specifically developed for smaller datasets to overcome the problem of overfitting the model. The dataset that has been used to train the original model consists of 13 labelled catenary arches. This dataset is publicly available at [35] and will be used in this project. It will be leveraged for two goals during the project. One, to retrain the model during and two, to test its performance after compression. As the model has already been trained on this dataset, the data understanding and preparation phases of this project are already described in the research by Ton et. al. [5].

4.2 Technical Specification

As the goal of this project is to reduce the size of the network to fit on a microcontroller while preserving performance, a minimum compression rate was calculated. As the weights of the network need to be stored on non-volatile memory, the flash memory of different microcontrollers is the initial parameter determining whether a network can be deployed. The amount of flash memory available is different for every microcontroller. For the Arduino microcontrollers, the size has to be reduced by roughly 162 times. The ESP32 has 2 or 4 MiB of flash memory, resulting in a needed compression rate of 2.4 - 1.2 times, respectively. As the flash memory on a Raspberry Pi is inserted using an SD card, this can be picked with a suitable size to deploy the model.

4.3 Technical Understanding

From the literature research, it is found that there are multiple characteristics of the project and network that put constraints on the choice of compression methods. First, an important factor is that there already exists a trained model. This eliminates the possibility of using a during-training compression method. Therefore eliminating the usability of the Compact Model Approach. This constraint does allow for the use of Tensor Decomposition, Quantization or Network Sparsification. Secondly, an important characteristic of the trained network is its architecture. As the model is trained using the structure proposed by Qi et. al. [6], it is a Convolutional Neural Network. From the literature, it is found that Pruning, a technique within Network Sparsification, is an efficient way to compress and accelerate CNN's [8]. The last important factor for determining a compression method is the goal of compression within this project. The goal is to reduce the size of the model to deploy it on a microcontroller, without reducing the accuracy of the network. Cheng et. al. [8] state the following on this topic: "Quantization and Pruning generally give reasonable compression rates while not hurting the accuracy. Applications that require stable models can benefit from these compression methods." Given the factors described above, it can be concluded that using Quantization or Network Sparsification methods are suitable approaches to compress the trained network. As it has been found that compression methods are often combined to increase the efficiency of compression [9][22], it is reasonable to combine the use of these two methods.

To choose a specific compression algorithm to test, the following database was used [36]. This database includes the latest Machine Learning research and the code to implement it. As creating a new algorithm to compress a neural network is beyond the scope of this project, it can be used to find a compression method that fits the specifications of this project. The used search terms "neural network compression pruning quantization" yielded 23 results, out of these 23 there were two that were much more favoured (roughly 23 and 4 thousand saves, as opposed to less than 650 saves). Of these two, only one proposes a compression algorithm, therefore the compression algorithm of this research was selected for this project. The selected research is by Song et. al. [4]. In this research, a three-phase compression algorithm is proposed. It first prunes the weights, then quantizes the weights and finally used Huffman encoding to save the remaining weights 4.1. The proposed compression algorithm managed to compress AlexNet by 35x and VGG-16 by 49x without loss of accuracy [4].

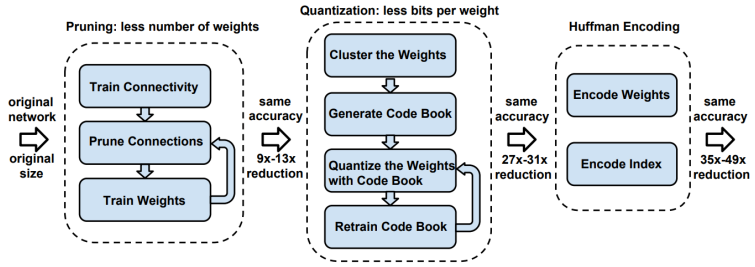


FIGURE 4.1: Flowchart of the compression algorithm proposed by Song et. al. [4]

4.3.1 Proof of concept

The focus of this phase is to get the compression script by Simon-lsy [37] working locally and understand how it can be altered to manipulate the extent of compression. The goal is to compress the model provided in the referenced repository. The open-source code is an implementation of the compression algorithm proposed by Song et. al. [4]. The flowchart of the algorithm is given in figure 4.1. The implementation by Simon-lsy makes use of a model trained on the MNIST dataset included in Keras [38]. Keras is a deep learning API that runs on top of the machine learning platform Tensorflow. The used MNIST dataset is a digits classification set that consists of 60 thousand grayscale 28x28 images of the 10 digits that can be used for training. Next to this, it provides a test set of 10 thousand images [38] on which the performance of the network can be tested. The working script will be used to understand the compression algorithm and control the rate of compression.

4.3.2 Code analysis of script

After correctly installing all necessary dependencies, the compression script could be run successfully. From the code, it is clear that there are a few declared constants that influence the compression process. These will be highlighted and their influence explained.

The first constant, `COMPRESSION_RATE` directly influence the rate of pruning. This constant is used to determine the amount of parameters that are dropped relative to the total amount of parameters in a layer. In order to prune the weights, all parameters from a layer are sorted based on their respective values. Then, based on a threshold value, the insignificant weights are dropped. In the following section, the code responsible for this is highlighted and elaborately explained. For complete code, refer to the GitHub repository of Simon-lsy [37].

```
tmp = deepcopy(weight[..., i])
tmp = np.abs(tmp)
tmp = np.sort(np.array(tmp))
threshold = tmp[int(tmp.shape[0] * COMPRESSION_RATE)]
weight[..., i][np.abs(weight[..., i]) < threshold] = 0
```

These lines of code are part of the `prune_weights` function in the compression.py script [37]. The list `tmp` is created by copying the weights of a single layer, after which this list is sorted. As `tmp` is a NumPy list [39], the shape function of this list returns the length of the list in all its dimensions. The first index, index 0, is taken from this list. The returned value corresponds to the number of entries in the list. Following, the amount of indices, which correspond to the number of weights, is multiplied by the `COMPRESSION_RATE` and gives an index value. This index is used to save the value in the `tmp` list as `threshold`

value. Following, in the last line of the code block, any weight smaller than this threshold is set to 0. All weights larger than the threshold are left as they are. After the highlighted code, the pruned weights are saved and set to the weights of the model, this can be found in the source code [37]. This concludes the pruning step of the algorithm.

After the pruning step, the network is retrained on the entire dataset to recover possibly lost accuracy. Next, the weights in the resulting network are quantized. For the quantization and weight sharing process, KMeans clustering is used. Within the python library scikit-learn, there exists a class for KMeans clustering [40]. The KMeans algorithm can cluster data in a specified amount of groups with equal variance. Using these clusters and their centroids (the mean of a cluster), the weights of the network can be quantized. The BITS constant determines the number of clusters in which the data is divided. Following is a more elaborate explanation of the code responsible for this process.

```
space = np.linspace(min_weight, max_weight, num=2 ** BITS)
kmeans = KMeans(n_clusters=len(space),
                init=space.reshape(-1, 1),
                n_init=1,
                algorithm="full")
kmeans.fit(nonzero_weight.reshape(-1, 1))
layer_cluster_index = kmeans.labels_
layer_centroids = kmeans.cluster_centers_.flatten()
cluster_index[layer_id] = layer_cluster_index
cluster_centroids[layer_id] = layer_centroids
new_weight = kmeans.cluster_centers_[kmeans.labels_].flatten()
for idx in range(len(nonzero_index)):
    index = nonzero_index[idx]
    weight_array[index] = new_weight[idx]
w[0] = weight_array.reshape(shape)
layer.set_weights(w)
```

This block of code is executed for every layer in the network that is being compressed. Right before the highlighted code, all nonzero weights of a layer are stored in an array and the minimum and maximum weights are determined. Using these extreme values, a linear space is created. The function `.linspace()` returns a linearly distributed series using a start value (`min\_weight`), end value (`max\_weight`) and an amount of values (`num`). The length of this array and its values are used to initialize a KMeans [41] object. Following, all values of the nonzero weights are clustered in the initialized object `kmeans` by the `.fit()` function. Then, all indices and mean values of the clusters are stored in the `cluster_index` and `cluster_centroids` arrays respectively based on the current layer that is being compressed. Based on the different clusters that are made (`kmeans.labels_`), the new weights are set to the means of the respective clusters (`clusters_centers_`). Then all these values are set as new weights of the current layer, this is done by the for loop and following two lines of code. This results in a layer with weights that are the means of the determined clusters.

Lastly, the quantized weights are encoded in the Huffman tree to make the storage more efficient. As it only groups and stores the weights in a smart way, this stage doesn't influence the accuracy of the model. It is a lossless data compression algorithm. The concept is to assign unique codes to each value based on the frequency it occurs. Therefore, the algorithm consists of two phases.

1. Build a Huffman tree.

2. Traverse the Huffman tree and assign codes to each character.

The first phase begins by creating a min-heap for all the values in the model, they are compared based on the frequency they occur. Then, two nodes with the minimum frequency are merged into an internal node. The frequency of this node is the sum of the frequencies of the two children. This node is added to the minheap with the left child having the lowest frequency. This is repeated until there is only one node present in the minheap. This concludes the first phase. The codes are assigned in the second phase. This is done by traversing the created Huffman tree, starting from the root. When moving to the left child, assign a 0, otherwise (when moving to the right) assign a 1. The sequence of 0's and 1's encountered when traversing to a leaf node is the code of a specific leaf (in this case the value of a weight). [42]

4.3.3 Experiment description

To make a comparison between compression and accuracy loss, the respective accuracy statistics will be measured at four different moments; before compression, post pruning (PP), post retraining (PR) and post quantization (PQ). Next to the accuracy values, the influence of pruning on the number of weights per layer and their minimum and average values. This statistic will be gathered for both the original and PP networks. To measure the number of weights left in the network, the total number of non-zero weights is counted. This statistic will be expressed as Relative Weights Reduction (RWR), the number of weights after pruning over the number of weights after pruning. The accuracy after the Huffman encoding will not be measured as this is the same as before this stage, since this only influences the way the weights are stored and leaves the values untouched. To make a comparison between compression and reduction in size, the size will be measured after the compression algorithm is finished, post-compression (PC). As the retraining step has a fair share of probability in it, the script will be run five times for different when varying `COMPRESSION_RATE` and `BITS`. This is done to get an average impact of the compression stage on the accuracy and size. Between the measurements, the `COMPRESSION_RATE` will be varied from 0.95 to 0.75, with a linear 0.05 decrease. The amount of compression is expected to be insignificant for lower compression rates, therefore the lowest value used is 0.75. The `BITS` will be varied for the integer values from 5 down to 1. As it is not possible to quantize more than 1 bit, this is the lowest value used. The effect of quantization is expected to be negligible for values higher than 5, therefore this is the highest value used. As mentioned, the script will be run 5 times for each combination. This comes down to 125 runs of the script. These 125 runs will be executed for both the models, coming to a total of 250 compressed weights.

Chapter 5

Technical Realisation and Evaluation

In this chapter, the experiment will be executed and its results evaluated. First, the compression algorithm described by Song et. al. [4] will be implemented on both the digits classification and PointNet++ network. Between different stages of compression, the relevant accuracy statistics will be gathered and documented. The limitations encountered during implementation will be discussed, along with potential limitations in realization, sources of errors and quality of the resulting data. Following, the retrieved data will be evaluated for both models. By comparing the factual results of both models, an understanding can be created from it.

5.1 Implementing compression on PointNet++ network

With the compression script functioning properly locally, the script was implemented on the PointNet++ network. This required that everything related to the MNIST model needs to be altered to the PointNet++ model. Next to this, it needs to be checked whether all used functions still work properly.

Using scripts that were created during previous stages of the project, the loading of training/testing data and the semantic segmentation model could be implemented. This ran into one major issue, as this model required a custom tf_ops file to be compiled. As this file was originally implemented using TensorFlow version 2.3.0 and the oldest version available was 2.5.0, it could not be compiled. By changing the tf_ops file, it could be compiled with the most recent TensorFlow version (2.9.0). As the PointNet++ model is an extension of the Tensorflow library, most of the functions to access weights and layers were still usable without needing extra attention.

As a different accuracy measured was used between the two networks, this needed to be altered for the implementation on the PointNet++ network. The function model.evaluate() returns multiple statistics of the network, among which the accuracy. On the contrary, the mIoU is used to evaluate the performance of the PointNet++ network. For this, the UpdateMeanIoU class from the Keras API is used. When initializing an object of this class, the number of classes that need to be predicted is set. With this information, combined with the current predictions and truth labels, the mIoU score is calculated. However, this ran into one major limitation of the produced data. As stated in the research by Ton et. al. [5], the mIoU score of the trained PointNet++ network is 71%. When loading the weights into a model and computing the mIoU score (without compression), the best performing set of weights reaches a mIoU score of 0.199 (19.9%). The only difference between the scripts of the previous and this project is the file format of the data on which predictions are made. Originally, .laz files were used however, these could not be loaded

into the Jupyter Notebook environment. Therefore, the files were converted to .las files. As these files contain the same information, only stored in a different way, it should not influence the outcome of this statistic [43]. As the issue with a lower mIoU could not be solved, a relative accuracy reduction (RAR) score was calculated. To compute the RAR, the resulting mIoU is divided by the original mIoU and multiplied by 100. This gives the percentage of the original mIoU that is left.

The third limitation that was encountered during the research is the available memory in the Jupyter Notebook environment. When the complete compression script was executed, the environment gave an error on memory usage. The memory physically present as a resource for the environment is more than enough to run the script. However, as separate threads are created per user to execute their data mining algorithms, the amount of memory used per thread is limited. For the script to not greedily start using all (physically) available memory, it had to be limited within the code. When limiting the memory access to such an extent allowed by the Jupyter environment, the script had too little available memory to execute in one go. Due to time constraints, multiple scripts were created that use the output of the previous as input. The pruning, retraining and quantization/encoding phases were all implemented using different scripts.

Due to time constraints, there was no possibility to retrain the model similar to the implementation by Simon-lsy [37]. In this script, a sparse network is created during pruning, consisting of only zeros and ones for every weight in the original network. By retraining the pruned network, the zero value (pruned) weights could become non-zero. Therefore, the retrained network is multiplied by this sparse network to set weights to zero where needed. For the PointNet++ model, the retraining takes much longer than for the MNIST model. Therefore, a different implementation was chosen. A different set of weights (with a similar mIoU score) was used and multiplied with the pruned model. As the pruned model has zero values present, the sparse identity stays intact.

5.2 Evaluation: Compression of MNIST

The original size of the digits classification model is 3,430 KB (or 3.43 MB) with an accuracy of 0.99119997. This accuracy value is the output of the function `model.evaluate()` provided by the Keras API [38]. The float output resembles the number of times the prediction matches the label assigned to the data point in the test set. A value of 1 would mean that the model has perfect accuracy, where every case in the test set is predicted correctly. A value of 0.5 means that the model predicted half (50%) of the test cases correctly. The lower the returned value, the more accuracy was lost by the respective compression phase. Following, the results of the three stages, pruning, retraining and quantization, will be highlighted in terms of accuracy and weight loss. Finally, the size of weights after compression will be evaluated.

5.2.1 Pruning

The pruning phase removes weights from the network. As seen in the code implementation [37], the first layer is untouched, as this layer sets the shape of the data that is handled by the network. Removing weights from this layer could result in a malfunctioning model. Next to this, it is found that layers 3, 4, 5 and 7 have zero weights in them. This is due to the layer type they are initialized with. These layers are set as maxpool2D, flatten and dropout layers [38]. All these layers influence either the input or shape of the network and are not used for predictions, therefore they do not have weights.

Due to the implementation of the network, all layers are pruned at the same rate, apart from the first one, which is left untouched. From following non-zero layers a percentage of weights are dropped, based on the compression rate. Therefore, it is expected that the amount of weights increases linearly for the lower compression rates. However, the amount of weights in the second layer stays the same for all compression rates. The only exception is a compression rate of 0.75, for this rate the amount of weights in layer 2 is increased by 290. This can be seen in figure 5.1a. As the most significant weights are kept by the pruning algorithm, the max weights don't change as a result of the compression rate. The minimum weight value per layer does decrease as the compression rate decreases. As the number of weights in layers 1 and 2 does not change by compression rate, their respective minimum weight value does not change either. The minimum value decreases exponentially for both layers 6 and 8, as visible in figure 5.1b. From table 5.1, it can be seen that the size reduction is indeed insignificant for a compression rate smaller than 0.75. The averages of the size of the compressed model and the intermediate accuracies for each compression value are given in table 5.1. The `BITS` constant is set to 5, as this is the largest value used in the next section. For the complete dataset, refer to the GitHub repository [44].

5.2.2 Retraining

In figure 5.2a, the relation between the compression rate and average accuracies can be seen. What becomes clear from this is the importance of the retraining step with respect to the accuracy for higher compression rates. At a compression rate of 0.9, the average increase in accuracy as a result of retraining is 0.093. Whereas this difference reduces exponentially to 0.011 for a compression rate of 0.75. Next to this, it is visible in figure 5.2a that there is very little difference in the average accuracies of PR and PQ. This is highlighted in figure 5.2b, where the accuracies PR and PQ are graphed against the corresponding compression rates. The difference in average accuracy is at most 1.2E-4, which is negligibly small with respect to the average accuracies. The effect of quantization on the accuracy will be more elaborately described in the following section.

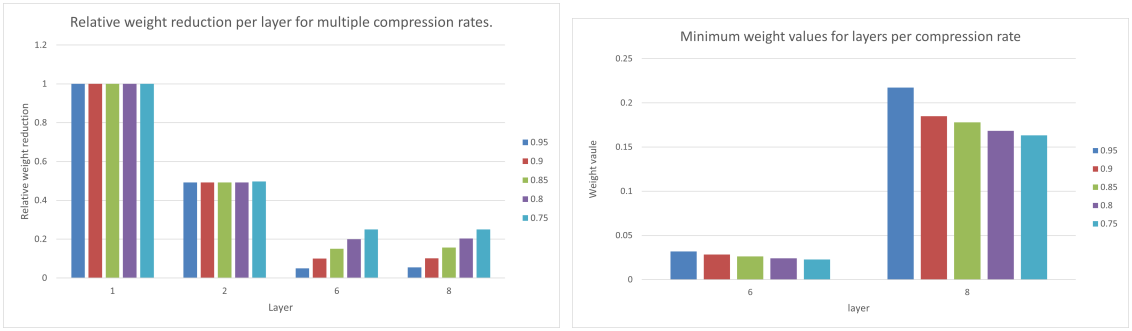
5.2.3 Quantization

As described in section 4.3.2, a bit value of 5 results in a linear space with $2^5 = 32$ intermediate values. For almost all compression rates it can be observed that the PR and PQ accuracy are nearly the same for the quantization parameter of 3 or higher. The difference in accuracy is for bit values of 1 in the range of 2E-2 - 4E-4 for the different compression rates. Whereas for a bit value of 3, the difference in accuracy is in the range 2E-4 - 6E-5. The only exception for this observation is the compression rate of 0.75. For this case, the accuracy PR is significantly smaller than the accuracy PQ. This effect is visible in the respective graphs in figure 5.3. Even though the accuracies remain similar, the size is still reducing. From figure 5.4 it can be seen that the size still decreases for BITS $=>$ 3, although at a lower rate than for BITS $=>$ 2.

Another observation that becomes clear from the graphs in figure 5.3 is that for compression rates of 0.95 and 0.9 the average PR accuracy is approximately the same for different quantization values (maximum difference of 0.001). Where the average accuracy for different quantization values varies significantly more for lower compression rates.

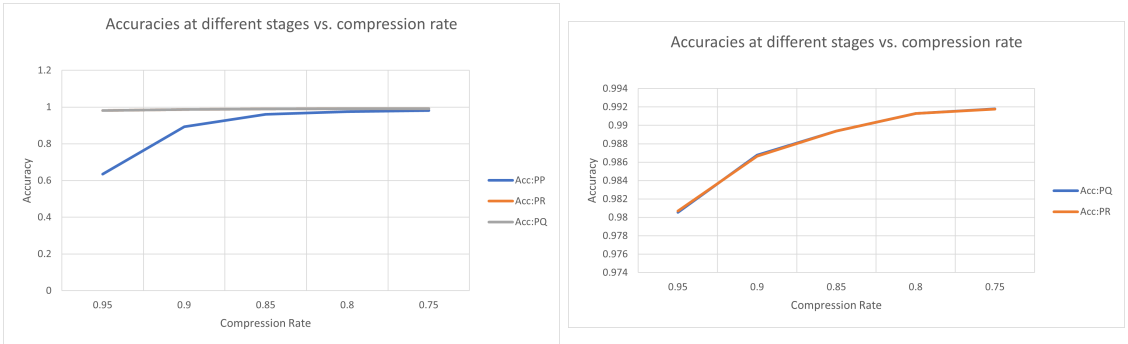
COMPRESSION_RATE	Acc: PP	Acc: PR	Acc: PQ	Size: PC (KB)
0.9	0.893400013	0.986659992	0.98678	959
0.85	0.961199999	0.989380002	0.989380014	1289
0.8	0.975799978	0.991280007	0.991280007	1625
0.75	0.980499983	0.991760004	0.991799998	1963.4

TABLE 5.1: Table displaying the averages of the accuracies at different stages and eventual size at different compression rates.



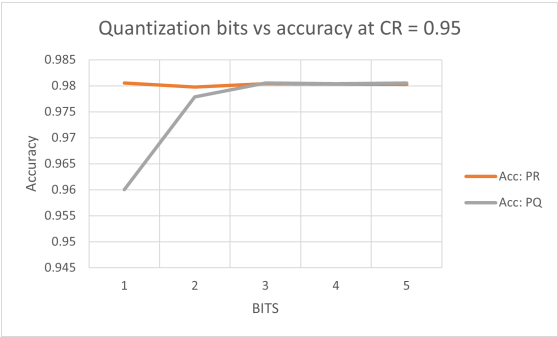
(A) Relative Weight Reduction (B) Minimum weight values

FIGURE 5.1: Graphs the influence of different pruning rates on the weights in layers 6 and 8.

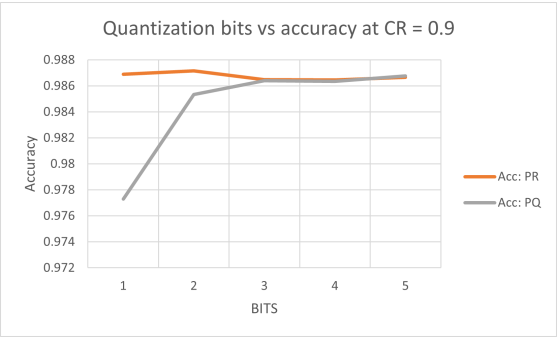


(A) Accuracies PP, PR and PQ. (B) Accuracies PR and PQ

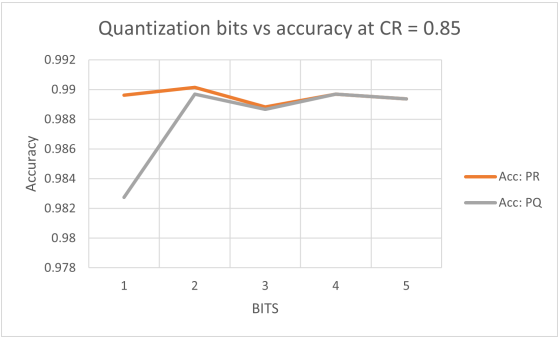
FIGURE 5.2: Plots of accuracies at different stages versus the compression rates.



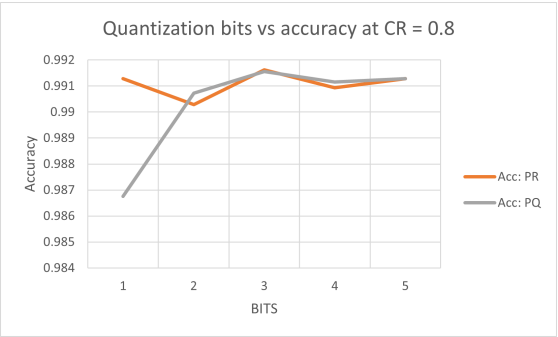
(A) COMPRESSION_RATE: 0.95



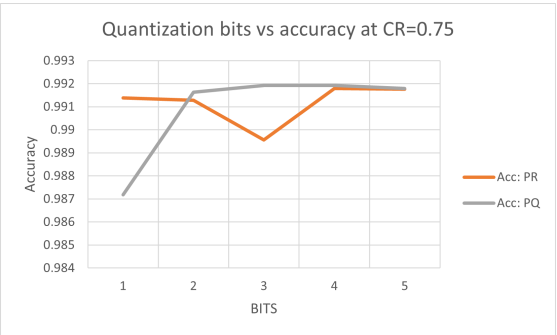
(B) COMPRESSION_RATE: 0.9



(C) COMPRESSION_RATE: 0.85



(D) COMPRESSION_RATE: 0.8

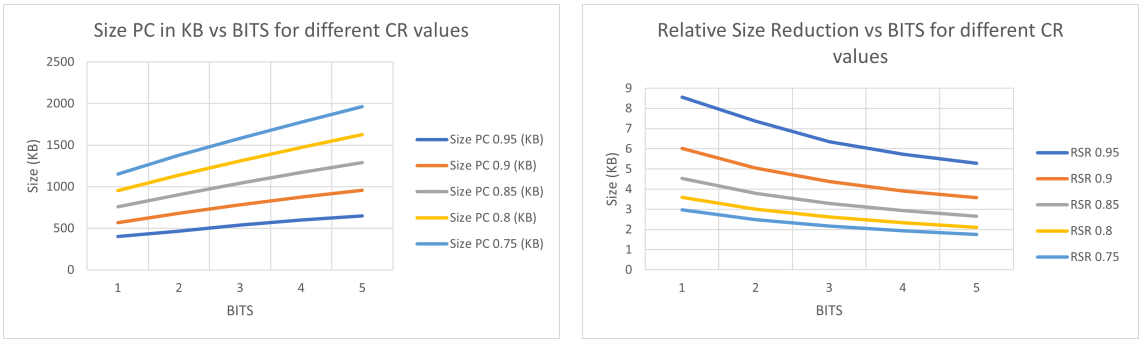


(E) COMPRESSION_RATE: 0.75

FIGURE 5.3: Graphs displaying relation between accuracy PR and PQ for different values of CR.

5.2.4 Size compression

The most important result of compression is the size of the output model. The aboslute and relative size compression are plotted in figure 5.4. In figure 5.4a it is visible that the effect of quantization is larger for a lower compression rate. This is to be expected as there are more weights present to quantize. In figure 5.4b the relative size reduction (RSR) is plotted versus the different compression rates for all quantization values. It is visible that the RSR linearly decreases (resulting in a larger size) with two different rates. The pivot point between the different rates is at a quantization value of 3 bits.



(A) Size vs Q

(B) RSR vs Q

FIGURE 5.4: Graphs of the reduction in size with respect to quantization at different CR values.

5.3 Evaluation of PointNet++ model

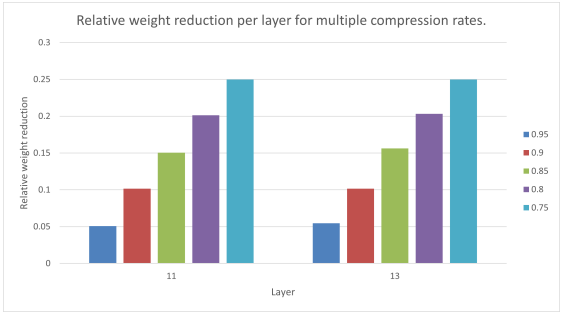
The original size of the PointNet++ model is 5,190 KB (or 5.19 MB) with a mIoU statistic of 0.199. Following, the results of the three stages, pruning, retraining and quantization, will be highlighted in terms of accuracy and weight loss. Finally, the size of the weights after compression will be evaluated.

5.3.1 Pruning

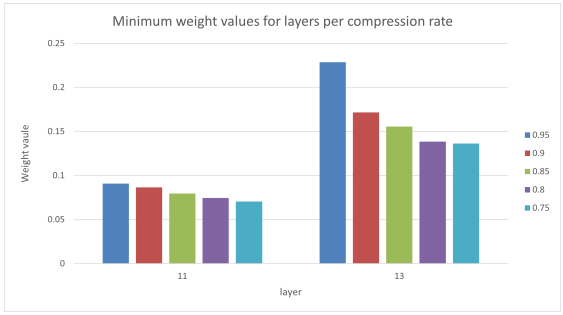
In this phase, the insignificant weights are removed from the network. As described, the first layer is left untouched. Layer 12 of the PointNet++ network does not have any weights as this is a dropout layer. As all layers are pruned at the same rate, all layers, except the first, should have a linear increase in the number of weights for the different compression rates. However, only layers 11 and 13 are influenced by the pruning. The weights in the other layers are all untouched by the pruning. The results of pruning layers 11 and 13 are shown in figure 5.5. In figure 5.5a the relative weights reduction for these layers is shown, these layers do show the expected linear increase of weights for lower compression rates. In figure 5.5b, the minimum weight value for layers 11 and 13 are shown. An exponential decrease in value for both layers is seen in this graph.

5.3.2 Retraining

In figure 5.6, the relation between the compression rate and average relative accuracy reduction (RAR) can be seen. It can be seen that the pruning stage of the compression loses half of the mIoU. The best RAR statistic is at a compression rate of 0.8. The retraining



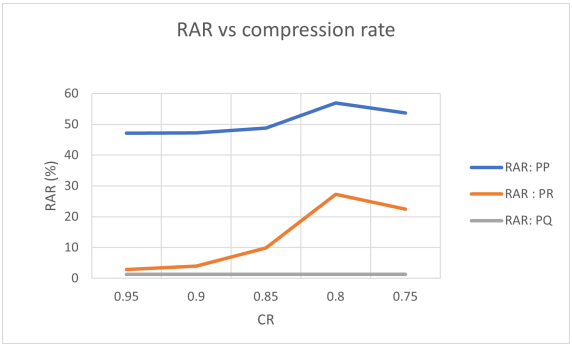
(A) Relative Weight Reduction



(B) Minimum weight values

FIGURE 5.5: Graphs showing the influence of different pruning rates on the weights in layers 11 and 13.

part of the algorithm has a large negative impact as it loses between 30% and 40% of the original mIoU. The retraining step has the least negative impact using a compression rate of 0.8. When quantizing the weights left in the network, there is very little left of the mIoU statistic (less than two per cent of the original mIoU). As the retraining phase of the algorithm is done to regain accuracy lost during pruning, it works counterproductive on the PointNet++ network.

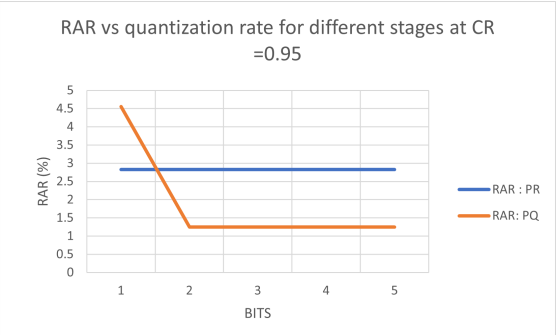


z

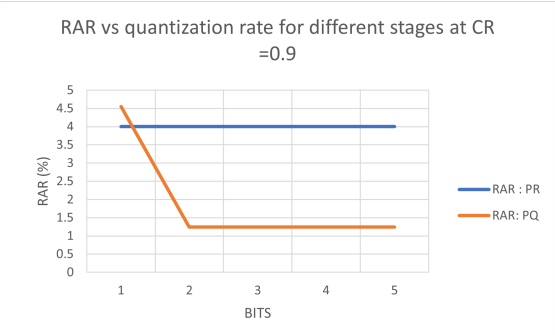
FIGURE 5.6: Plots of RAR at different stages versus the compression rates.

5.3.3 Quantization

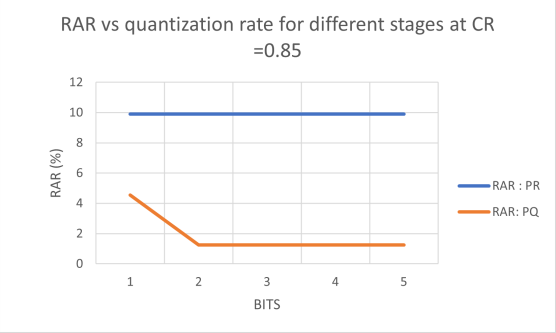
In figure 5.7, the average RAR at different moments (PR and PQ) versus the quantization can be seen for different compression rates. Due to the implementation used for the retraining of the PointNet++ model, the PR mIoU is expected to be the same for all iterations per compression rate. Generally speaking, retraining has a certain amount of randomness to it. However, this is not present in this implementation. In all of the graphs in figure 5.7, the RAR for a quantization level of 1 bit is higher than the RAR for the other quantization levels. For compression rates of 0.95 and 0.9, the relative PW RAR values are even higher than the relative PR RAR values. To add to the previously mentioned consistency, the RAR is consistent among the compression rates for all other quantization levels (2 to 5). In general, quantizing the weights of the PointNet++ network has a large detrimental effect on the RAR statistic.



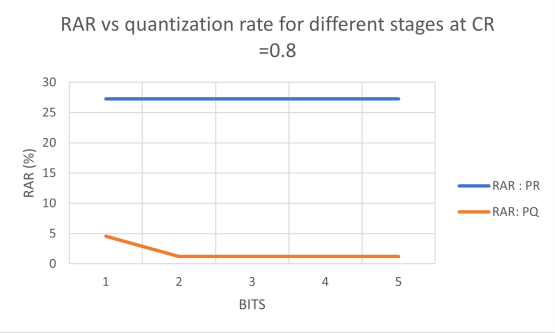
(A) COMPRESSION_RATE: 0.95



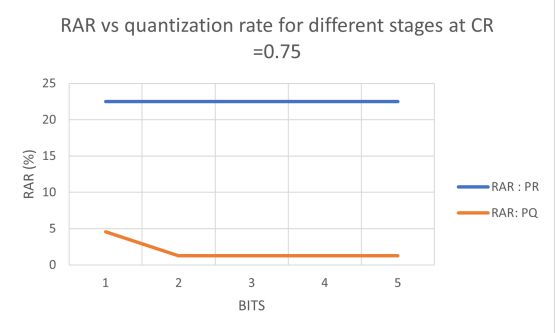
(B) COMPRESSION_RATE: 0.9



(C) COMPRESSION_RATE: 0.85



(D) COMPRESSION_RATE: 0.8

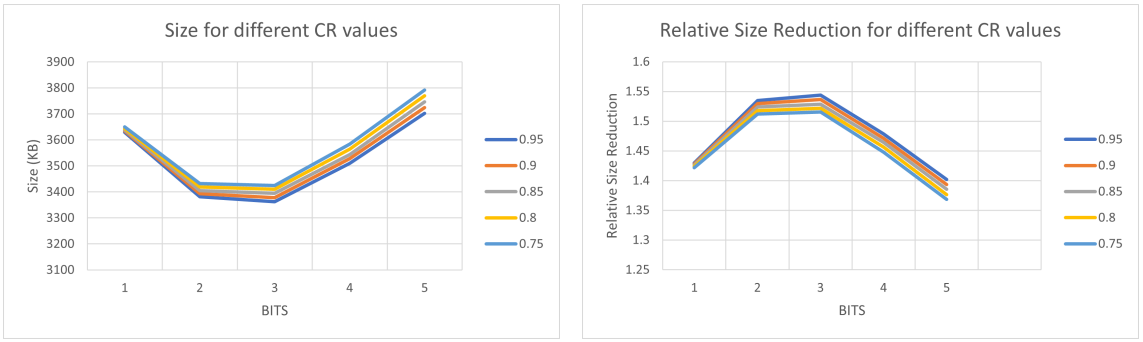


(E) COMPRESSION_RATE: 0.75

FIGURE 5.7: Graphs displaying relation between accuracy PR and PQ for different values of CR.

5.3.4 Size compression

The absolute and relative size compression are plotted in figure 5.8a. It can be seen that the effect of quantization is larger for lower compression rates. This difference is small due to the limited increase in amount of weights. In figure 5.8b, the RSR is plotted for each compression rate versus the quantization levels. It is visible that the RSR is largest at three bits, for larger quantization rates, the RSR decreases roughly linearly. This is a different pivot point than for the RAR, as described in the previous section 5.3.3. It is expected that the RSR decreases of lower quantization rates. Between one and two bits, the RSR does decrease. However as the respective RSR values are lower than the RSR value of a quantization level of three bits, it is not behaving as expected.



(A) Size vs Q

(B) RSR vs Q

FIGURE 5.8: Graphs of the reduction in size with respect to quantization at different CR values.

5.4 Evaluation: comparison of compression results

In the following section, a comparison between the compression results of the MNIST and PointNet++ model will be made. It will give an overview of the similarities and differences between the respective results. In terms of weight pruning, the results look very similar. In the layers the pruning is applied to show a linear increase in the number of weights for smaller compression rates for both models, as seen in figures 5.1 and 5.5. Next to this, the minimum weight value per layer decreases exponentially for both models. This is an indication that the weight distribution between both models might be similar. However, the pruning is not (correctly) applied to all layers that should be pruned. The correctly pruned layers from the PointNet++ model are the only layers with weights that use a type of layer directly from the Keras API, the dense layer [38]. All other layers are types of layers based on layers defined in the Keras API.

In terms of retraining, the results are very different. Although the two different implementations were used, both were expected to improve the respective accuracy statistic of both models. The results are not in line with this expectation. When compression the PointNet++ model, the retraining hurts the accuracy statistic. This is the opposite of the results from retraining the MNIST model as these show an accuracy statistic very close to the original.

In terms of quantization, the results are also very different between both models. For the MNIST model, the quantization has an insignificant impact on the accuracy for most compression rates at larger bit values. If there is a significant impact present for larger bit values, quantization results in better accuracy. One similarity that can be observed is

that there exists a pivot point in terms of quantization level for both models after which the relative accuracy statistic seems to be constant.

In terms of size reduction, the results show both similarity and difference. The absolute reduction in size is much smaller for the PointNet++ model as compared to the MNIST model. The MNIST model is reduced in size by 3,029 KB, which is a compression rate of 8.5. The PointNet++ model is reduced by 1,828 KB, which is a compression rate of 1.54. From a quantization level of 3 bits, the RSR statistic decreases linearly for both models. However, for smaller quantization levels, the RSR statistic shows different behaviour. The RSR for the PointNet++ model has a maximum at a quantization level of 3 bits, where this is expected at 1 bit.

Chapter 6

Conclusion and Future Work

In this chapter, conclusions will be drawn from the results shown in the previous sections. An answer to the research question will be given. The research question is stated as: How can a trained PointNet++ model be compressed to fit on a microcontroller while preserving performance? Following, possible avenues for future work will be mentioned. These could be explored to either increase the understanding of the compression method or to achieve the goal of deployment on a microcontroller in a different way.

6.1 Conclusion

The short answer to the research question is the following, not with the implementation used in this research. However, some elaborations need to be made. First, it is proven that compression while preserving performance is possible. The size of the MNIST model was reduced at a maximum rate of 8.5. This reduction results in an accuracy loss of less than one per cent. When looking at the short-list of widely used microcontrollers, this compression rate would be more than enough to fit the PointNet++ model on an ESP32. However, this compression rate is much lower than claimed by Song et al. [4]. If it is managed to compress a PointNet++ model without sacrificing performance, it will be possible to reduce the size of the current model to fit on a microcontroller. Secondly, the compression results do show similarities. The pruning stage yield similar results in terms of amount and value of weights per pruned layer. Next to this, the computed relative size reduction statistic shows similar behaviour for larger quantization values. There is a pivot point noticeable at a quantization level of three bits, after which the RSR decreases linearly.

There are two areas in which the MNIST and PointNet++ models differ significantly. Both of these could result in the described differences. First, there are differences in the models on software level. As described before, the PointNet++ model uses custom-created layers that are based on the layers from the Keras API [38]. The model trained on the MNIST dataset solely uses layers directly implemented from the Keras API [38]. This could result in incorrectly pruning the weights. Secondly, PointNet++ models learn features in a conceptually different way as compared to models trained on images. This could result in a significant difference in the resulting neural networks. Extra research needs to be done to investigate which of these is the truth. If it turns out to be the first, it will be possible to compress PointNet++ models using available compression methods. If the second turns out to be the case, new methods should be developed to compress PointNet++ and possibly other types of networks based on point clouds.

6.2 Future Work

To increase the understanding of compression, the size reduction is not investigated between the different stages of compression. If it turns out to be possible to compress a PointNet++ model while preserving performance, this could be investigated. The following results would give insight into which stages are the most efficient in terms of least accuracy decrease and largest size reduction. From this, an optimal combination of pruning and quantization parameters could be computed given the desired compression rate.

The first avenue to deploy a model on a microcontroller could be to sparsify the input set. If a model can be trained that can accurately make predictions on a point cloud with a lower density, the complexity of the model could be reduced. A lower density results in less complex features that need to be learned by a network. A less complex network with fewer weights and connections could result in a lower storage requirement to deploy the network on a microcontroller.

A second avenue that could be investigated is hardware acceleration. Hardware accelerators increase the speed at which a neural network can be run. However, hardware acceleration does not decrease the storage requirement of a network, it positively influences its execution once deployed. When it turns out that the compressed network runs too slow on a microcontroller, this can be a way of speeding up the runtime of the network without altering the network itself [8][10].

Bibliography

[1] N. Berchane and N. Berchane, "Artificial intelligence, machine learning, and deep learning: Same context, different concepts - master intelligence economique et stratégies compétitives," https://master-iesc-angers.com/artificial-intelligence-machine-learning-and-deep-learning-same-context-different-concepts/, Apr. 2018, accessed: 2022-5-16.

[2] S. Huber, H. Wiemer, D. Schneider, and S. Ihlenfeldt, "Dmme: Data mining methodology for engineering applications – a holistic extension to the crisp-dm model," *Procedia CIRP*, vol. 79, p. 403–408, 2019.

[3] A. C. Aras, "Explaining what learned models predict: In which cases can we trust machine learning models and when is caution required ?" https://www.researchgate.net/publication/350487701_Explaining_what_learned_models_predict_In_which_cases_can_we_trust_machine_learning_models_and_when_is_caution_required, 2020, accessed: 2022-7-11.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015. [Online]. Available: https://arxiv.org/abs/1510.00149

[5] B. Ton, F. Ahmed, and J. Linssen, "Semantic segmentation of terrestrial laser scans of railway catenary arches: a use case perspective," 2022.

[6] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *CoRR*, vol. abs/1706.02413, 2017. [Online]. Available: http://arxiv.org/abs/1706.02413

[7] H. van der Burgt, "De afstand tussen masten en portalen," Oct 2017. [Online]. Available: https://encyclopedie.beneluxspoor.net/index.php/De_afstand_tussen_masten_en_portalen

[8] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *CoRR*, vol. abs/1710.09282, 2017. [Online]. Available: http://arxiv.org/abs/1710.09282

[9] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani, "A comprehensive survey on model compression and acceleration," *Artif. Intell. Rev.*, vol. 53, no. 7, pp. 5113–5155, 2020.

[10] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE Inst. Electr. Electron. Eng.*, vol. 108, no. 4, pp. 485–532, 2020.

[11] R. S. Williams, "What's next? the end of moore's law," *Computing in Science Engineering*, vol. 19, no. 2, pp. 7–13, 2017.

[12] N. O. US Department of Commerce and A. Administration, "What is lidar?" Oct 2012. [Online]. Available: https://oceanservice.noaa.gov/facts/lidar.html

[13] E. Kavlakoglu, "Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference?" 2020. [Online]. Available: https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks

[14] T. Wood, "Backpropagation," Sep 2020. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/backpropagation

[15] "Google scholar." [Online]. Available: https://scholar.google.com/

[16] [Online]. Available: https://dl.acm.org/

[17] D. Bacciu and D. P. Mandic, "Tensor decompositions in deep learning," *CoRR*, vol. abs/2002.11835, 2020. [Online]. Available: https://arxiv.org/abs/2002.11835

[18] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[19] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014.

[20] J. Liu, B. Zhuang, Z. Zhuang, Y. Guo, J. Huang, J. Zhu, and M. Tan, "Discrimination-aware network pruning for deep model compression," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PP, pp. 1–1, 2021.

[21] P. Singh, V. Kumar Verma, P. Rai, and V. P. Namboodiri, "Play and prune: Adaptive filter pruning for deep model compression," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. California: International Joint Conferences on Artificial Intelligence Organization, 2019.

[22] K. Tan and D. Wang, "Towards model compression for deep learning based speech enhancement," *IEEE ACM Trans. Audio Speech Lang. Process.*, vol. 29, pp. 1785–1794, 2021.

[23] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," 2016. [Online]. Available: https://arxiv.org/abs/1612.00593

[24] R. Keim, "What is a microcontroller? the defining characteristics and architecture of a common component - technical articles," Mar 2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/what-is-a-microcontroller-introduction-component-characteristics-component/

[25] B. Lutkevich, "What is a microcontroller and how does it work?" Nov 2019. [Online]. Available: https://www.techtarget.com/iotagenda/definition/microcontroller

[26] E. S. Brown and Rodney, "Flash memory vs. ram: What's the difference?" Mar 2019. [Online]. Available: https://www.techtarget.com/searchstorage/feature/Flash-memory-vs-RAM-Whats-the-difference#:~:text=Flash

[27] R. Pi, "Raspberry pi 4 model b specifications." [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[28] "Arduino uno rev3." [Online]. Available: https://store.arduino.cc/products/arduino-uno-rev3

[29] "The internet of things with esp32." [Online]. Available: http://esp32.net/

[30] N. Hotz, Apr 2022. [Online]. Available: https://www.datascience-pm.com/crisp-dm-2/

[31] "Tf.keras.metrics.meaniou." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/MeanIoU

[32] A. Rosebrock, "Intersection over union (iou) for object detection," Nov 2016. [Online]. Available: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

[33] "Tensorflow." [Online]. Available: https://www.tensorflow.org/

[34] "Hoe hard rijden treinen?" [Online]. Available: https://www.prorail.nl/veelgestelde-vragen/hoe-hard-rijden-treinen

[35] B. Ton, Mar 2022. [Online]. Available: https://data.4tu.nl/articles/dataset/Labelled_high_resolution_point_cloud_dataset_of_15_catenary_arches_in_the_Netherlands/17048816

[36] "Papers with code - the latest in machine learning." [Online]. Available: https://paperswithcode.com/

[37] Simon-lsy, "Deep compression," https://github.com/Simon-lsy/Deep_Compression, 2019.

[38] fchollet, qlzh727, taehoonlee, and the moliver, "Keras: Deep learning for humans," https://github.com/keras-team/keras/tree/v2.9.0, 2015.

[39] [Online]. Available: https://numpy.org/

[40] "2.3. clustering." [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html#k-means

[41] "Sklearn.cluster.kmeans." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[42] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, p. 1098–1101, Sep 1952.

[43] "Las, laz and lidar." [Online]. Available: https://manifold.net/doc/mfd9/las,_laz_lidar.htm

[44] NilsRutgers, "Point net compression," https://github.com/NilsRutgers/PointNetCompression, 2019.